0

Context Aware Model-Checking for Embedded Software

Philippe Dhaussy[1], Jean-Charles Roger[1] and Frédéric Boniol[2]
[1]*Ensta-Bretagne*
[2]*ONERA*
France

1. 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 (Berthomieu et al., 2004; Holzmann, 1997; Larsen et al., 1997), 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 (Clarke et al., 1986; Holzmann & Peled, 1994; Park & Kwon, 2006). 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 chapter 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. Their 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 on a restriction of the system behavior for verification of specific properties instead on exploring the global system automaton.

In this chapter, we describe the formalism called CDL (Context Description Language), such as DSL[1]. 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 chapter is organized as follows: Section 2 presents related work on the techniques to improve model checking by state reduction and property specification. Section 3 presents the principles of our approach for context aware formal verification. Section 4 describes the CDL language for context specification. Our toolset used for the experiments is presented section 5. In Section 6, we give results of industrial case studies. Section 7 discusses our approach and presents future work.

2. Related works

Several model checkers such as SPIN (Holzmann, 1997), Uppaal (Larsen et al., 1997), TINA-SELT (Berthomieu et al., 2004), have been developed to assist in the verification of concurrent asynchronous systems. For example, the SPIN model-checker based on the formal language Promela allows the verification of LTL (Pnueli, 1977) 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 (Godefroid, 1995). In (Bosnacki & Holzmann, 2005) 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 (Godefroid, 1995; Peled, 1994; Valmari, 1991) 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. (Alfaro & Henzinger, 2001; Clarke et al., 1999; Flanagan & Qadeer, 2003; Tkachuk & Dwyer, 2003) 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. We propose to formally specify the context behavior of components in a way that allows a fully automatic divide-and-conquer algorithm. We choose to explicit contexts separately from the model to be validated. However, our approach can be used in conjunction with design by contract process. It is about using the knowledge of the environment of a whole system (or model) to conduct a verification to the end.

Another difficulty is about requirement specification. Embedded software systems integrate more and more advanced features, such as complex data structures, recursion,

[1] Domain Specific Language

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 (Clarke et al., 1986)) 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 (Dwyer et al., 1999; Konrad & Cheng, 2005; Smith et al., 2002) 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.

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. We present the results for a part of the S_CP model. 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 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 $context$.

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.

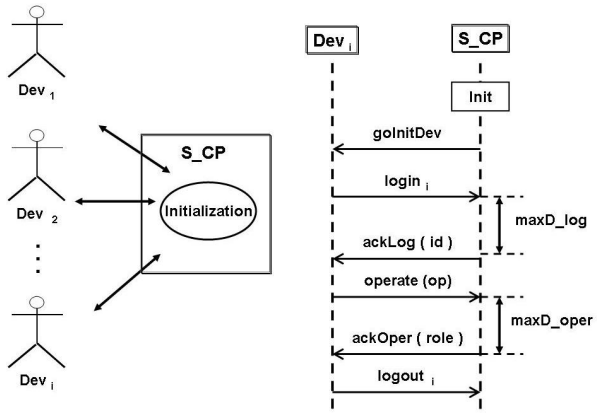

Fig. 1. *S_CP* system: partial description during the initialization phase.

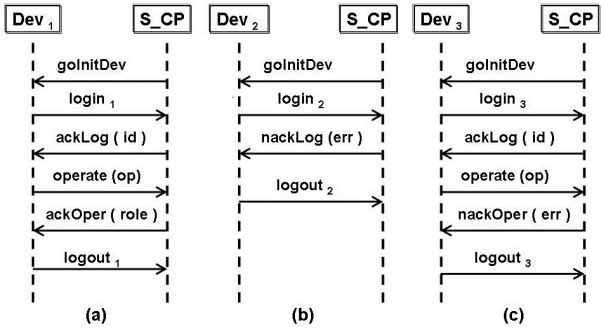

Fig. 2. An example of *S_CP* context scenario with 3 devices.

3.2 Model-checking results

To verify requirements on the system model[2], we used the TINA-SELT model checker. To do so, the system model is translated into FIACRE format (Farail et al., 2008) 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 1 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.

3.3 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

[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.

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	–	–

Table 1. Table highlighting the verification complexity for an industrial case study (*S_CP*).

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 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..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 to check 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.

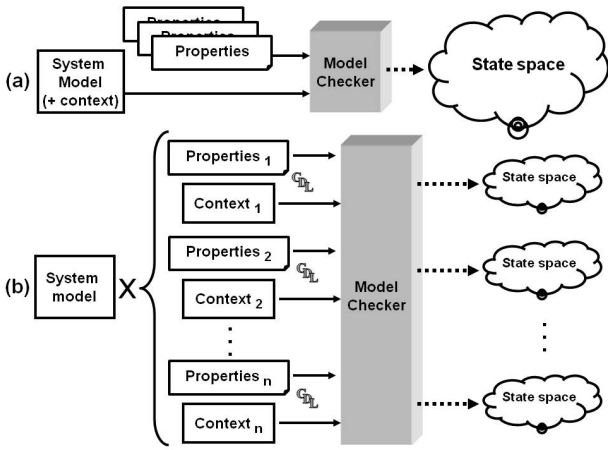

Fig. 3. Traditional model checking (a) vs. context-aware model checking (b).

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 chapter, 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.

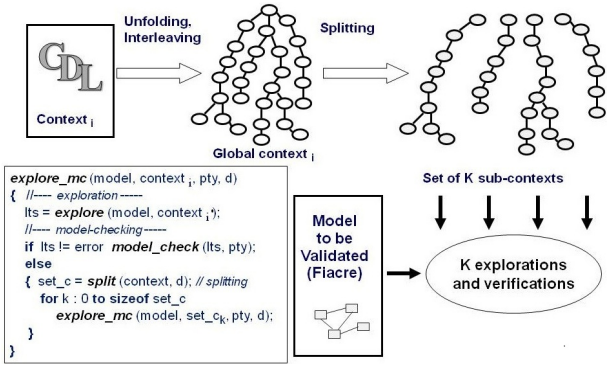

Fig. 4. Context splitting and verification for each partition (sub-context).

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 chapter).

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, we proposed (Dhaussy et al., 2009) 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 (Whittle, 2006) 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 3.3.

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 that we do not describe here but can be found in (Dhaussy & Roger, 2011). 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

[4] For the detailed syntax, see (Dhaussy & Roger, 2011) available (currently in french) on http://www.obpcdl.org.

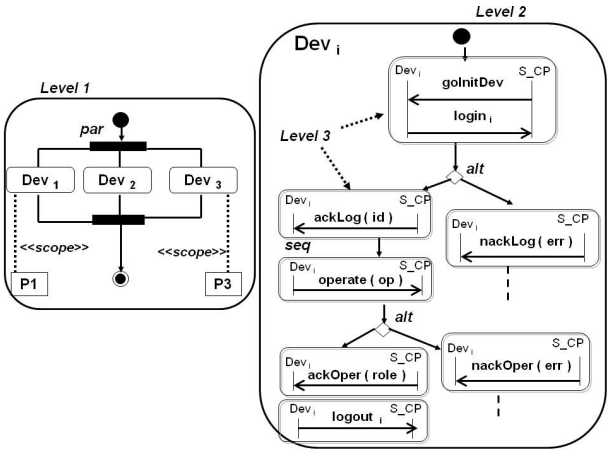

Fig. 5. *S_CP* case study: partial representation of the context.

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 5. We will now present the CDL formal syntax and semantics.

4.2 Formal syntax

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

$$C ::= M \mid C_1; C_2 \mid C_1 + C_2 \mid C_1 \| C_2$$
$$M ::= \mathbf{0} \mid a!; M \mid a?; M$$

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].

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

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

4.3 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:

$$Wait\ (\mathbf{0}) \stackrel{\text{def}}{=} \emptyset \qquad Wait\ (a!;M) \stackrel{\text{def}}{=} \emptyset \qquad Wait\ (a?;M) \stackrel{\text{def}}{=} \{a\}$$
$$Wait\ (C_1 + C_2) \stackrel{\text{def}}{=} Wait\ (C_1) \cup Wait\ (C_2) \qquad Wait\ (C_1;C_2) \stackrel{\text{def}}{=} Wait\ (C_1)\ if\ C_1 \neq \mathbf{0}$$
$$Wait\ (\mathbf{0};C_2) \stackrel{\text{def}}{=} Wait\ (C_2) \qquad Wait\ (C_1 \| C_2) \stackrel{\text{def}}{=} Wait\ (C_1) \cup Wait\ (C_2)$$

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 $(C,B) \stackrel{a}{\rightarrow} (C',B')$ 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 **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).

4.4 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):

$$< (C,B_1)|(s,\mathcal{S},B_2) > \stackrel{a}{\stackrel{\sigma}{\Rightarrow}} < (C',B_1')|(s',\mathcal{S},B_2') > (1)$$

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):

$$< (C,B_1)|(s,\mathcal{S},B_2) > \stackrel{t}{\stackrel{\sigma}{\Rightarrow}} < (C,B_1)|(s',\mathcal{S},B_2') > (2)$$

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 ($null_\sigma$) (to the context). Note that in the case of timed

$$\frac{}{(a!;M,B) \xrightarrow{a!} (M,B)}\ [\text{pref1}] \qquad \frac{}{(a?;M,a.B) \xrightarrow{a?} (M,B)}\ [\text{pref2}]$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1,B) \xrightarrow{a} (C_1',B')}{(C_1.C_2,B) \xrightarrow{a} (C_1'.C_2,B')}\ [\text{seq1}] \qquad \frac{(C_1,B) \xrightarrow{a} (\mathbf{0},B')}{(C_1.C_2,B) \xrightarrow{a} (C_2,B')}\ [\text{seq2}]$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1,B) \xrightarrow{a} (C_1',B')}{\begin{array}{l}(C_1\|C_2,B) \xrightarrow{a} (C_1'\|C_2,B') \\ (C_2\|C_1,B) \xrightarrow{a} (C_2\|C_1',B')\end{array}}\ [\text{par1}] \qquad \frac{(C_1,B) \xrightarrow{a} (\mathbf{0},B')}{\begin{array}{l}(C_1\|C_2,B) \xrightarrow{a} (C_2,B') \\ (C_2\|C_1,B) \xrightarrow{a} (C_2,B')\end{array}}\ [\text{par2}]$$

$$\frac{(C_1,B) \xrightarrow{a} (C_1',B')}{\begin{array}{l}(C_1+C_2,B) \xrightarrow{a} (C_1',B') \\ (C_2+C_1,B) \xrightarrow{a} (C_1',B')\end{array}}\ [\text{alt}] \qquad \frac{a \notin wait(C)}{(C,a.B) \xrightarrow{null_\sigma} (C,B)}\ [\text{discard}_C]$$

Fig. 6. Context semantics.

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 $< (C,B)|(s,\mathcal{S},B') > \not\mapsto$ to express that the system and its context cannot evolve (the system is blocked 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:

$$\begin{array}{c}[\![C \mid (s,\mathcal{S})]\!] \stackrel{\text{def}}{=} \{a_1 \cdot \sigma_1 \cdot \ \ldots\ a_n \cdot \sigma_n \cdot end_C \mid \\ < (C,null_\sigma) \mid (s,null_\sigma) > \xrightarrow[\sigma_1]{a_1} \ < (C_1,B_1) \mid (s_1,\mathcal{S},B_1') > \\ \xrightarrow[\sigma_2]{a_2} \ \ldots \ \xrightarrow[\sigma_n]{a_n} \ < (C_n,B_n) \mid (s_n,\mathcal{S},B_n') > \ \not\to \}\end{array}$$

$[\![C|(s,\mathcal{S})]\!]$ 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 accessibility properties to be checked.

$$\frac{(s,\mathcal{S},B_2) \xrightarrow[\sigma']{} (s',\mathcal{S},B_2')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[\sigma]{null_\Phi} < (C,B_1.\sigma)|(s',\mathcal{S},B_2') >} \text{[cp1]}$$

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

$$\frac{(C,B_1) \xrightarrow{a?} (C',B_1')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[null_\sigma]{null_\Phi} < (C',B_1')|(s,\mathcal{S},B_2) >} \text{[cp3]}$$

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

Fig. 7. CDL context and system composition semantics.

4.5 Property specification patterns

Property specifying needs to use powerful yet easy mechanisms for expressing temporal requirements of software source code. As example, let's see a requirement of the *S_CP* system described in section 3.1. This requirement was found in a document of our partner and is shown in Listing 1. It refers to many events related to the execution of the model or environment. It also depends on an execution history that has to be taken into account as a constraint or pre-condition.

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

Listing 1. Initialization requirement for the *S_CP* system described in section 3.

If we want to express this requirement 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 (Dwyer et al., 1999) 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 (Konrad & Cheng, 2005). To improve the expressiveness of these patterns, we enriched them with options (*Pre-arity*, *Post-arity*, *Immediacy*, *Precedence*, *Nullity*, *Repeatability*) using annotations as (Smith et al., 2002). 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 (Janssen et al., 1999). 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 (Listing 1) must be interpreted and can be written with CDL in a property $P1$ as follow (cf. Listing 2). $P1$ is linked to the communication sequence between the S_CP and device (Dev_1). According to the sequence diagram of figure 5, the association to other devices has no effect on $P1$.

$$
\begin{array}{l}
Property\ P1; \\
ALL\ Ordered \\
\quad exactly\ one\ occurence\ of\ S\_CP\_hasReachState\_Init \\
\quad exactly\ one\ occurence\ of\ login1 \\
end \\
eventually\ leads-to\ [0..maxD\_log] \\
AN \\
\quad one\ or\ more\ occurence\ of\ ackLog(id) \\
end \\
S\_CP\_hasReachState\_Init\ may\ never\ occurs \\
login1\ may\ never\ occurs \\
one\ of\ ackLog(id)\ cannot\ occur\ before\ login1 \\
repeatibility:\ true
\end{array}
$$

Listing 2. S_CP case study: A response pattern from R requirement.

$P1$ specifies an observation of event occurrences in accordance with figure 5. $login1$ 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.

For the sake of simplicity, we consider in this chapter that properties are modeled as observers. Our OBP toolset transforms each property into an observer automaton including a reject node. 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 *runs* of $[\![C|(init, \mathcal{S})]\!]$) 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. For that purpose, OBP translates the property into an observer automaton, depicted in figure 8.

4.6 Formalization of observers

The third part of the formalization relies on the expression of the properties to be fulfilled. We consider in the following that an observer is an automaton $\mathcal{O} = \langle \Sigma_o, init_o, T_o, Sig, \{reject\}, Sv_o \rangle$

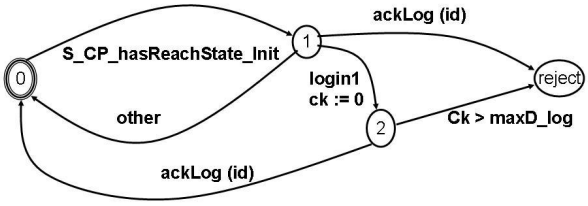

Fig. 8. Observer automaton for the property P1 of Listing 2.

(a) emitting a single output event: *reject*, (b) where *Sig* is the set of matched events by the observer; events produced and received by the system and its context and (c) such that all transitions labelled *reject* arrive in a specific state called "unhappy".

Semantics. We say that $\mathcal{S}$ in the state $s \in \Sigma$. $\mathcal{S}$ closed by C satisfies $\mathcal{O}$, denoted $C|(s,\mathcal{S}) \models \mathcal{O}$, if and only if no execution of $\mathcal{O}$ faced to the runs r of $[\![C|(s,\mathcal{S})]\!]$ produces a *reject* event. This means:

$$C \mid (s,\mathcal{S}) \models \mathcal{O} \iff \forall r \in [\![C \mid (s,\mathcal{S})]\!],$$
$$(init_o, \mathcal{O}, r) \xrightarrow{null_\sigma} (s_1, \mathcal{O}, r_1) \xrightarrow{null_\sigma} \dots \xrightarrow{null_\sigma} (s_n, \mathcal{O}, r_n) \not\rightarrow$$

Remark: executing $\mathcal{O}$ on a run r of $[\![C|(s,\mathcal{S})]\!]$ is equivalent to put r in the input buffer of $\mathcal{O}$ and to execute $\mathcal{O}$ with this buffer. This property is satisfied if and only if only the empty event ($null_\sigma$) is produced (i.e., the *reject* event is never emitted).

5. OBP toolset

To carry out our experiments, we used our OBP[6] tool (Figure 9). OBP is an implementation of a CDL language translation in terms of formal languages, i.e. currently FIACRE (Farail et al., 2008). As depicted in Figure 9, OBP leverages existing academic model checkers such as TINA 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 (Halbwachs et al., 1993) from each property for OBP Explorer, or SELT logic formula (Berthomieu et al., 2004) 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 (Dhaussy et al., 2009). 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 3.3.

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.

[6] OBP_t (OBP for TINA) is available on http://www.obpcdl.org.
[7] http://www.topcased.org
[8] http://www-Omega.imag.fr

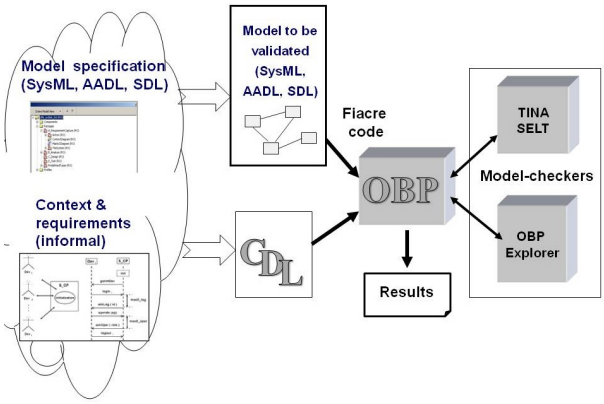

Fig. 9. CDL model transformation with OBP.

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. We reported here the results of these experiments.

6.1 Requirement specification

This section reports on six case studies (CS_1 to CS_6). Four of the software components come from an industrial A and two from a B[9]. For each industrial component, the industrial partner provided requirement documents (use cases, requirements in natural language) and the component executable model. Component executable models are described with UML, completed by ADA or JAVA programs, or with SDL language. The number of requirements in Table 2 evaluates the complexity of the component. To validate these models, we specify properties and contexts.

	CS_1	CS_2	CS_3	CS_4	CS_5	CS_6
Modeling language	SDL	SDL	SDL	SDL	UML2	UML2
Number of code lines	4 000	15 000	30 000	15 000	38 000	25 000
Number of requirements	49	94	136	85	188	151

Table 2. Industrial case study classification.

6.1.1 Property specification

Requirements are inputs of our approach. Here, the work consists in transforming natural language requirements into temporal properties. To create the CDL models with patterns-based properties, we analyzed the software engineering documents of the proposed case studies. We transformed textual requirements. We focused on requirements which

[9] CS_5 corresponds to the case study partially described in section 3.1.

can be translated into observer automata. Firstly, we note that most of requirements had to be rewritten into a set of several properties. Secondly, model requirements of different abstraction levels are mixed. We extracted requirement sets corresponding to the model abstraction level. Finally, we observe that most of the textual requirements are ambiguous. We had to rewrite them consequently to discussion with industrial partners. Table 3 shows the number of properties which are translated from requirements. We consider three categories of requirements. *Provable* requirements correspond to requirements which can be captured with our approach and can be translated into observers. The proof technique can be applied on a given context without combinatorial explosion. *Non-Computable* requirements are requirements which can be interpreted by a pattern but cannot be translated into an observer. For example, liveness properties cannot be translated because they are unbounded. Observers capture only bounded liveness properties. From the interpretation, we could generate another temporal logic formula, which could feed a model checker as TINA. *Non-Provable* requirements are requirements which cannot be interpreted at all with our patterns. It is the case when a property refers to undetectable events for the observer, such as the absence of a signal.

	CS_1	CS_2	CS_3	CS_4	CS_5	CS_6	Average
Provable properties	38/49 (78%)	73/94 (78%)	72/136 (53%)	49/85 (58%)	155/188 (82%)	41/151 27%)	428/703 (61%)
Non-computable properties	0/49 (0%)	2/94 (2%)	24/136 (18%)	2/85 (2%)	18/188 (10%)	48/151 (32%)	94/703 (13%)
Non-Provable properties	11/49 (22%)	19/94 (20%)	40/136 (29%)	34/85 (40%)	15/188 (8%)	62/151 (41%)	181/703 (26%)

Table 3. Table highlighting the number of expressible properties in 6 industrial case studies.

For the CS_5 , we note that the percentage (82%) of provable properties is very high. One reason is that the most of 188 requirements was written with a good property pattern matching. For the CS_6, we note that the percentage (27%) is very low. It was very difficult to re-write the requirements from specification documentation. We should have spent much time to interpret requirements with our industrial partner to formalize them with our patterns.

6.2 Context specification

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.

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 4. Exploration with TINA explorer with context splitting using OBP_t (*S_CP* case study).

Table 4 shows the amount of TINA exploration[10] 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 4 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 1. It is clear that device number limit depends on the memory size of used computer.

7. 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.

CDL contributes to overcoming the combinatorial explosion by allowing partial verification on restricted scenarios specified by the context automata. CDL permits contexts and non ambiguous properties to be formalized. Property can be linked to whole or specific contexts. During experiments, we noted that some contexts and requirements were often described in the available documentation in an incomplete way. With the collaboration between engineers responsible for developing this documentation and ourselves, these engineers were motivated to consider a more formal approach to express their requirements, which is certainly a positive improvement.

In some case study, 70% textual requirements can be rewritten more easily with pattern property. So, CDL permits a better formal verification appropriation by industrial partners. Contexts and properties are verification data useful to perform proof activities and to validate models. These data have to be capitalized if the implementation evolves over the development life cycle.

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 make explicit 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.

[10] Tests with same computer as for Table 1.

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. Furthermore, constraints from CDL can guide developers to construct formal properties to check against their models. Using CDL, they have a means of rigorously checking whether requirements are captured appropriately in the models using simulation and model checking techniques.

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.

8. References

Alfaro, L. D. & Henzinger, T. A. (2001). Interface automata, *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, Press, pp. 109–120.

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

Bosnacki, D. & Holzmann, G. J. (2005). Improving spin's partial-order reduction for breadth-first search, *SPIN*, pp. 91–105.

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

Clarke, E. M., Long, D. E. & Mcmillan, K. L. (1999). Compositional model checking, MIT Press.

Dhaussy, P., Pillain, P.-Y., Creff, S., Raji, A., Traon, Y. L. & Baudry, B. (2009). Evaluating context descriptions and property definition patterns for software formal validation, *in* B. S. Andy Schuerr (ed.), *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, Vol. LNCS 5795, Springer-Verlag, pp. 438–452.

Dhaussy, P. & Roger, J.-C. (2011). Cdl (context description language) : Syntax and semantics, *Technical report*, ENSTA-Bretagne.

Dwyer, M. B., Avrunin, G. S. & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification, *21st Int. Conf. on Software Engineering*, IEEE Computer Society Press, pp. 411–420.

Farail, P., Gaufillet, P., Peres, F., Bodeveix, J.-P., Filali, M., Berthomieu, B., Rodrigo, S., Vernadat, F., Garavel, H. & Lang, F. (2008). FIACRE: an intermediate language for

model verification in the TOPCASED environment, *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*, SEE.

Flanagan, C. & Qadeer, S. (2003). Thread-modular model checking, *SPIN'03*.

Godefroid, P. (1995). The Ulg partial-order package for SPIN, *SPIN Workshop* .

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

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

Holzmann, G. & Peled, D. (1994). An improvement in formal verification, *Proc. Formal Description Techniques, FORTE94*, Chapman & Hall, Berne, Switzerland, pp. 197–211.

Janssen, W., Mateescu, R., Mauw, S., Fennema, P. & Stappen, P. V. D. (1999). Model checking for managers, *SPIN*, pp. 92–107.

Konrad, S. & Cheng, B. (2005). Real-time specification patterns, *27th Int. Conf. on Software Engineering (ICSE05), St Louis, MO, USA*.

Larsen, K. G., Pettersson, P. & Yi, W. (1997). UPPAAL in a nutshell, *International Journal on Software Tools for Technology Transfer* **1**(1-2): 134–152.
URL: *citeseer.nj.nec.com/larsen97uppaal.html*

Park, S. & Kwon, G. (2006). Avoidance of state explosion using dependency analysis in model checking control flow model, *ICCSA (5)*, pp. 905–911.

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

Pnueli, A. (1977). The temporal logic of programs, *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, pp. 46–57.

Smith, R., Avrunin, G., Clarke, L. & Osterweil, L. (2002). Propel: An approach supporting property elucidation, *24st Int. Conf. on Software Engineering(ICSE02), St Louis, MO, USA*, ACM Press, pp. 11–21.

Tkachuk, O. & Dwyer, M. B. (2003). Automated environment generation for software model checking, *In Proceedings of the 18th International Conference on Automated Software Engineering*, pp. 116–129.

Valmari, A. (1991). Stubborn sets for reduced state space generation, *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, Springer-Verlag, London, UK, pp. 491–515.

Whittle, J. (2006). Specifying precise use cases with use case charts, *MoDELS'06, Satellite Events*, pp. 290–301.