Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation

Philippe Dhaussy[1], Pierre-Yves Pillain[1], Stephen Creff[1], Amine Raji[1], Yves Le Traon[2] and Benoit Baudry[3]

[1] UEB, Laboratoire LISyC, ENSIETA, BREST, F-29806 cedex 9
{dhaussy, pillaipi, creffst, rajiam}@ensieta.fr
[2] Université du Luxembourg, Campus Kirchberg
yves.letraon@uni.lu
[3] Equipe Triskell, IRISA, RENNES, F-35042
bbaudry@irisa.fr

Abstract. A well known challenge in the formal methods domain is to improve their integration with practical engineering methods. In the context of embedded systems, model checking requires first to model the system to be validated, then to formalize the properties to be satisfied, and finally to describe the behavior of the environment. This last point which we name as the proof context is often neglected. It could, however, be of great importance in order to reduce the complexity of the proof. The question is then how to formalize such a proof context. We experiment a language, named CDL (Context Description Language), for describing a system environment using actors and sequence diagrams, together with the properties to be checked. The properties are specified with textual patterns and attached to specific regions in the context. Our contribution is a report on several industrial embedded system applications.

Keywords: Formal methods, context description, property patterns, observers, timed automata, model checking.

1 Introduction

In the field of embedded systems, software architectures must be designed to ensure increasingly critical functions subjected to strong reliability and real time constraints. Due to these constraints, embedded software architectures often have to go through certification which requires a rigorous design process based on tight rules. However, due to the increasing complexity of systems, there is no guarantee that such a design process leads to error free systems. Formal methods offer rigorous and powerful solutions for helping embedded system designers analyze, validate, or transform systems in a provable sound way. For that purpose, behavior checking methods have been explored for several years by many research teams [2, 8], but also by major companies.

Nevertheless, integration of formal methods in the engineering process is still too weak comparatively to the huge need for reliability in critical systems. This contradiction partly finds its causes in the actual difficulty to handle theoretical concepts within an industrial framework. Besides, formal verification techniques suffer from the combinatorial explosion induced by the internal complexity of the software to be verified. This is particularly recurrent when dealing with real-time embedded systems, interacting with a large number of actors. Additionally, formally checking properties on system models requires the expression of these properties in the form of temporal logic formula such as LTL [18] or CTL [16]. While these languages have a high expressiveness they are not easily readable and easy to handle by the engineers in industrial projects. To overcome this problem, some approaches [5, 12, 10] propose to formulate temporal properties using textual definition patterns.
One way to circumvent the problem of combinatorial explosion consists of specifying/restricting the system environment behavior or the context in which the system will be used. The system is then tightly synchronized with its environment. This context corresponds to well-defined operational phases, such as, for example, initialization, reconfiguration, degraded modes, etc. Moreover, properties are often related to specific use cases of the system. So, it is not necessary to verify them over all the environment scenarios. To the best of our knowledge, no approach currently provides such feature dedicated to an industrial use. In the case of an environment composed of several parallel actors, describing the environmental context can be a difficult task. To address these problems, we proposed [21, 22] the Context Description Language (CDL). This DSL allows specifying the context with scenarios and temporal properties using property patterns. Moreover, CDL provides the ability to link each expressed property to a limited scope of the system behavior.
In this paper, we provide a two years experience feedback on applying our formal verification approach on several aeronautic and military case studies. This paper presents the approach and discusses the results on an exercise in bringing engineers to use a formal method. First, we show that specifying more precisely the context in which the system will be used can reduce the problem of state explosion. Second, we show how to formalize, with CDL, specifications of an execution context, how to formalize properties and how to attach these properties to specific regions in this context.
For better understanding, this approach is illustrated with one industrial case study: the software part of an anti-aircraft system (S_CP[1]), shown Fig.1. It controls the internal modes of the system, its physical devices (radars, sensors, actuators…) and their actions in response to incoming signals from the environment. Due to page limitation, only one requirement (Listing 1) and one sequence diagram are considered to illustrate our approach along the paper.
The paper is organized as follows: Section 2 sets the scope of our work in current formal verification practices and presents related work. Section 3 describes our DSL for contexts and properties specification. Section 4 presents the proposed methodology used for the experiments, as well as the framework supporting it. In section 5 we give selected results on several industrial case studies. Finally, section 6 discusses our approach and future work and concludes.

[1] For confidential reasons, company and system names are not mentioned in this paper.

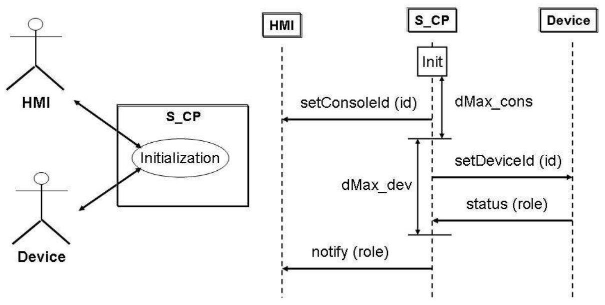


Fig. 1. S_CP system: partial use case and sequence diagram describing the behavior of the system during the initialization phase

Requirement: "During initialization procedure, the S_CP shall associate a generic device identifier to one or several roles in the system (Device), before dMax_dev time units. It shall also associate an identifier to each console (HMI), before dMax_cons time units. The S_CP shall send a notifyRole message for each connected generic device, to each connected console. Initialization procedure shall end successfully, when the S_CP has set all the generic device identifiers and all console identifiers and all notifyRole messages have been sent."
End Requirement

Listing 1: Initialization requirement for the S_CP system

2 Context and related work

These days, embedded software systems integrate more and more advanced features, such as complex data structures, recursion, multithreading. These features pose challenging theoretical and practical problems to developers of automatic analysis and verification methods. 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 most of the time informal. This fact is more challenging than it appears because of the difficulty to write logic formula correctly without some expertise in the idioms of the specification languages. While temporal logic based languages allow a great expressivity for the properties, these languages are not adapted to practically describe most of the requirements expressed in industrial analysis documents. First, a requirement can refer to many events related to the execution of the model or environment (cf. Listing 1). Then, it depends on an execution history that has to be taken into account when checking it. As a result, the logical formulas are of great complexity and become difficult to read and to handle by engineers. It is thus necessary to facilitate the requirement expression with adequate languages: abstracting some details in the property description, at a price of reducing the expressivity. This conclusion has been done by many authors a long time ago and some [5, 12, 10] proposed to formulate the properties using definition patterns. Patterns are textual templates that capture common temporal properties and that can be instantiated in a specific context to express application-specific properties.

Specification patterns [5, 10] have been proposed to assist engineers in expressing system requirements directly in a formal specification language, such as linear-time temporal logic (LTL). These patterns represent commonly occurring types of real-time properties found in several requirement documents for appliances and automotive embedded systems applications.

In addition to the ease of writing real time properties, the patterns proposed by Dwyer [5] and Cheng [10] have been defined to deal with high-level specifications. Providing high-quality requirements is important since they serve as a baseline between multiple teams working on the model under study (MUS[2]). Besides, Hassine et al. [17] suggest an abstract high level pattern-based approach to the description of property specifications based on Use Case Maps (UCM). They propose to build property pattern systems that consider architectural aspects. Smith et al. developed *Propel* [12], in which they extended the specification patterns of Dwyer et al. [5] to address important aspects about properties. They extend the patterns with options that can be used explicitly on these patterns.

In this paper, we reuse the categories of Dwyer's patterns and extend them to deal with more specific temporal properties which appear when high-level specifications are refined. Furthermore, in several industrial projects, intended requirements are not associated to the entire lifecycle of software, but only to specific steps in its lifecycle. In the system specification documents, requirements are often expressed in a context of the system execution. For that reason, in addition to the use of property patterns, we propose to link formalized properties to a specific execution context and thus to limit the scope of the property. Hassine et al.[17] consider applying patterns to architectural aspects; we focus on applying them to specific functional contexts, which refer to system use cases. The benefit is to explicitly specify the conditions under which is its meaningful to check the validity of a given property. So, according to this feature, properties will be checked only in a specific execution context. Consequently, the number of states over which the property is checked considerably decreases. In this paper, we address the problem of applying property patterns in industrial practices and provide concrete statistical results.

3 Context Description Language

In our approach, CDL aims at formalizing the context with scenarios and temporal properties using property patterns. This DSML[3] is based on UML 2. A CDL 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. 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. For formal validation, CDL associates a formal semantics to UML models, described as a set of traces [7, 13, 22]. The language is designed and tooled to offer a simple and usable context description framework.

[2] In this paper, MUS denotes the component model specified by the industrial in languages such as UML 2, AADL [19], SDL [4], etc.

[3] Domain Specific Modeling Language

The syntax of the CDL language is specified in multiple and complementary ways. One is the metamodel (e.g. the domain ontology) enhanced with OCL constraints. The metamodel is an ECore model (EMF). It is annotated with OCL invariants to enforce its semantics. A diagrammatical concrete syntax is created for the context description and a textual syntax for the property expression. The following paragraphs outline: (i) the proof context formalization, (ii) the property expressions.
In [11], we proposed a context description language using UML 2 diagrams (cf. Fig.2 for case study illustration). It is inspired by Use Case Charts of [13]. We extend this language to allow several entities (as Device and HMI in Fig.1 and Fig.2) to compose the proof context. Those entities are running in parallel. 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 by alternatives. Each scenario is fully described at Level-3 by UML 2 sequence diagrams. These diagrams are composed of two lifelines, one for the proof context and another for the MUS. Delayable interaction event occurrences are specified on these lifelines. Counters limit the iterations of diagram executions. It ensures the generation of finite context automata, as described in [11]. Transitions at Level-1 and Level-2 are enabled according to the values of some un-timed guards or timed guards. As mentioned in the introduction, the approach links the context description (Level-1 or Level-2) to the specification of the properties (as P1 and P2 in Fig.2) to be checked by stereotyped links property/scope. A property can have several scopes and several properties can refer a single diagram. Semantics of Level-1 and Level-2 is described in terms of traces, inspired by [7]. Level-1 and Level-2 are based on the semantics of the scenarios and expressed by construction rules of sets of traces built using *seq*, *alt* and *par* operators (*par* only for Level-1). At Level-3, the semantics of a scenario is expressed by a set of traces as described in [7] and in accordance with the semantics of UML 2 sequence diagrams. A scenario trace is an ordered events sequence which describes a history of the interactions between the context and the model. A scenario with several interactions is described by a set of traces.

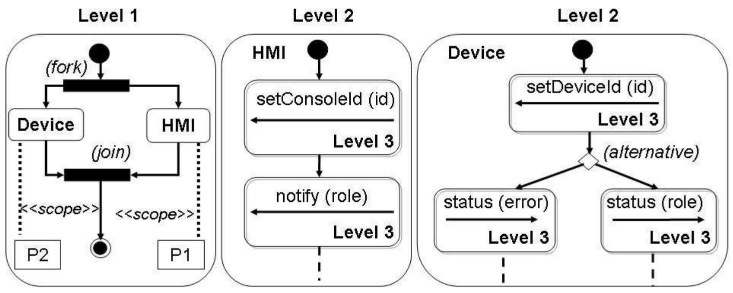


Fig. 2. S_CP case study: partial representation of the context. Initial Use cases and Sequence diagrams (cf. Fig.1) are transformed and completed to create the context model. All context scenarios are represented, combined with parallel and alternative operators, in terms of CDL.

For the property specification, we use a pattern-based approach and integrate property patterns description in the CDL language (we refer the reader to [22] for details). Patterns [5] are classified in basic families, which take into account the timed aspects of the properties to be specified. The patterns identified allow 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. These basic forms are enriched by options (*Pre-arity, Post-arity, Immediacy, Precedence, Nullity, Repeatability*) using annotations [10]. The property must be taken into account during all the model execution, before, after or between occurrences of events. Patterns have the possibility of expressing guards on the occurrences of events expressed in the properties [22]. Guards refer to variables declared in the context model. This mechanism adds precision to the property/scope reference introduced in the previous section. Another extension of the patterns is the possibility of handling sets of events, ordered or not ordered similar to the proposal of [9]. 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. Illustrating with our case study, Fig.3 depicts one bounded liveness property (P1) obtained from the *R1* requirement decomposition as explained in section 4:

> ***R1:*** *During initialization procedure, the S_CP shall associate an identifier to NC console (HMI), before dMax_cons time units.*

R1 is linked to the communication sequence between the S_CP and consoles (HMI). According to the sequence diagram of Fig.1, the association to other devices has no effect on *R1*.

```
Property P1 ;
    exactly  one  occurence of   S_CP_hasReachState_Init
    eventually  leads-to  [0..dMax_cons]
    ALL Ordered
          exactly  one  occurence of   sendSetConsoleIdToHMI1
          exactly  one  occurence of   sendSetConsoleIdToHMI2
    end
    S_CP_hasReachState_Init  may never occurs
    one of sendSetConsoleIdToHMI1  cannot  occur before  S_CP_hasReachState_Init
    one of sendSetConsoleIdToHMI2  cannot  occur before  S_CP_hasReachState_Init
    repeatibility : true
```

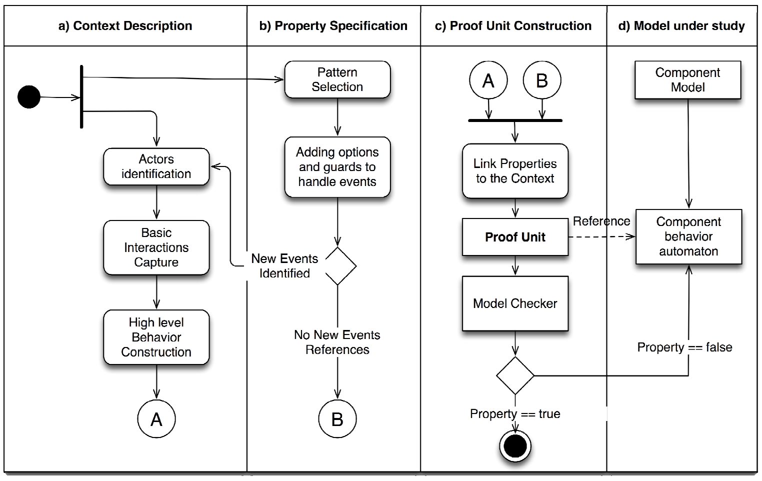
Fig. 3. S_CP case study: A response pattern from R1 requirement.

In the illustrated case study, the number of consoles (*HMI)* considered is two (*NC*=2). R1 specifies an observation of event occurrences. *S_CP_hasReachState_Init* refers a state change in a MUS process. *sendSetConsoleIdToHMI1* and *sendSetConsoleIdToHMI2* refer to the ones described in the CDL model (Fig.2). As mentioned in section 4, our OBP toolset transforms each property into an observer automaton [6], including a *reject* node. 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. This *reject* node is reached after detecting event "*S_CP_hasReachState_Init*" if the sequence "*sendSetConsoleIdToHMI1*" and "*sendSetConsoleIdToHMI2*" is not produced in that order before *dMax_cons* time units. Conversely, the *reject* node is not reached either if event "*S_CP_hasReachState_Init*" is never received, or if the sequence of the two events above is correctly produced (in the right order and with the right delay).

Consequently, such a property can be verified by using reachability analysis implemented in a formal model checker.

4 Methodology and OBP toolset

Our proposed specification and analysis process is based on checking a set of requirements on the system interacting with its environment. To perform such checking, we suppose that the set of properties can be formalized into a logic form, that the environment interactions are also formally modeled as well as the possibility to simulate the MUS in order to use a formal verification tool. With this hypothesis, the process is decomposed into the following steps:
- *Context Description* (Fig.4.a): the environment interactions are formally modeled with CDL activities diagrams (as illustrated Fig.2). This activity produces a set of CDL context diagrams.
- *Property Specification* (Fig.4.b): the set of properties are formalized with property patterns (as illustrated Listing 1). This activity produces a set of CDL pattern-based properties.



a) Context Description
b) Property Specification
c) Proof Unit Construction
d) Model under study
Actors identification
Basic Interactions Capture
High level Behavior Construction
A
Pattern Selection
Adding options and guards to handle events
New Events Identified
No New Events References
B
A
B
Link Properties to the Context
Proof Unit
Reference
Model Checker
Property == true
Component Model
Component behavior automaton
Property == false

Fig. 4. Activity diagram overviewing our specification and analysis process.

- *Proof Unit Construction* (Fig.4.c): we proposed in [20] the *Proof Unit* (PU) concept, which gathers all required data to perform proof activities, i.e. a reference to the model to be checked (MUS), the context models and the properties (CDL model) to be verified. The set of constructed PUs represents the set of requirements to be checked on the MUS to prove it is correct (cf. Fig.5).
- *Model Under Study* (Fig.4.d): it has to be simulated in order to use a formal verification tool. For this, OBP produces set of error observers. The observers perform dynamic diagnosis and play the role of probes to locate the cause of an error. When a fault is located, it is necessary to modify the model and create or modify requirement

The prerequisite of the methodology is the organization of the industrial specifications into two sets: (i) the design models that represent the MUS structure and behavior; (ii) the requirements that design models have to fulfill. This organization is necessary to extract useful information about the context execution for a given requirement (conditions under which a requirement has to be fulfilled). Indeed, in industrial requirement documents, this contextual information is very often implicit or disseminated in several documents and long discussions with engineers are usually needed to precisely understand the different contexts for the system and capture them in a model. Considering our case study, the given requirement (Listing 1) can be decomposed and reordered into four sub-requirements, stated as follow:

R1: *During initialization procedure, the S_CP shall associate an identifier to NC console (IHM), before dMax_cons time units.*

R2: *After, the S_CP shall associate a generic device identifier to NE roles in the system (Device), before dMax_dev time units.*

R3: *Each device returns a statusRole message to S_CP before dMax_ack time units.*

R4: *The S_CP shall send an notifyRole message for each connected generic device, to each connected console. Initialization procedure shall end successfully, when the S_CP has set all the generic device identifiers and all console identifiers and all notifyRole messages have been sent.*

After this decomposition, the user can specify more easily these requirements with definition property patterns.

We use the CDL language to represent the context, using actors and sequence diagrams, and all the requirements. The constructed CDL models reference elements of the MUS (events, variables). Elements of CDL models and MUS are at the same abstraction level. Moreover, we extract a formal specification describing the MUS's behavior. This description is generally represented as a timed automaton so that it can be executed by a simulator after model transformations. Property patterns capture, with a textual format, types of properties translated from the requirement documents.

It is obvious that providing all these verification proof units is not a trivial activity. It takes a great part of time and effort within a project. Besides, verification efforts made to check whether an implementation meets the requirements have to be capitalized. This capitalization captures the business logic to be used to redo the proof if the requirements and thus the implementation evolve over the development lifecycle. The definition of a general formal framework for the proof unit concept is out of the scope of this paper and left for future work.

To carry out our experiments, we implemented the *Observer Based Prover* (OBP[4]) tool onto the Eclipse platform through plug-ins. OBP takes as input the MUS behavior model and CDL models. OBP is an implementation of a CDL language translation in terms of formal languages, i.e. IF2 [2] or FIACRE [15] language. IF2 is based on timed automata [1] extended to the asynchronous communicating process context. Work is in progress to finalize the translation into FIACRE language and thus take benefits from the TINA [14] model checker. The essence of a translational approach to semantics is to move to a technological space that has a precise semantics [3] and tools. As depicted in Fig.5, OBP leverages existing academic simulators and model checkers, as TINA, IFx [2] or CADP [8].

[4] OBP is available (version 2.0) under EPL license at : http://gforge.enseeiht.fr/projects/obp

To handle the gap between CDL meta-model and the final DSLs (e.g. IF2 or FIACRE) the translation has several stages. We defined an ad-hoc domain-specified transformation language in terms of ECore metamodel and define a Model to Model transformation chain. From CDL context diagrams, OBP tool generates a set context path automata which represent the set of the environment runs. OBP generates all the possible paths. Each path represents one possible interaction between model and context. The OBP tool generates, with a similar model transformation technique, the observer automata from the properties. Each generated context path is transformed into an IF2 automaton which is composed with the MUS and the generated observer automata by the IFx simulator. To validate the component model, it is necessary to compose each path with the model and the observers. Each property must be verified for all paths. The accessibility analysis is carried out on the result of the composition between a path, a set of observers and the MUS. If there is a *reject* state reached of a property observer for one of paths, then the property is considered as false.

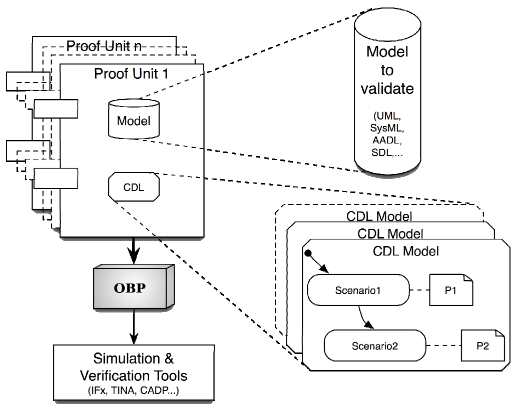


Fig. 5. Proof Units transformation with OBP

At present time, the input MUS of OBP (Fig.5) are imported currently with IF2 format. To import models with standard format as UML 2, AADL [19] or SDL [4], it is necessary to implement adequate translators as studied in projects such as TopCased[5] or Omega[6]. The model driven developed tool OBP set out in this paragraph was used in several case studies which are summed up in the experiment following section.

5 Experiments and results

Our approach was applied to several embedded systems applications in avionic or electronic industrial domain. These experiments are carried out with our french industrial partners. This section reports on six case studies (CS_1 to CS_6). Four of the

[5] http://www.topcased.org

[6] http://www-omega.imag.fr

software components come from an industrial A and two from a B. 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 1 evaluates the complexity of the component. To validate these models, we follow the methodology described in section 4. So, we describe the following phases: property specification, context description and proof unit construction.

Table 1. Industrial case study classification.

	CS_1	CS_2	CS_3	CS_4	CS_5[7]	CS_6
Modeling language	SDL	SDL	SDL	SDL	UML2	UML2
Number of code lines	4 000	15 000	30 000	15 000	38 000[8]	25 000[9]
Number of requirements	49	94	136	85	188	151

5.1 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 can be translated into observer automata. Firstly, we note that most of requirements had to be rewritten into a set of several properties (as shown in the S_CP case study along the paper). 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 2 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.

[7] CS_5 corresponds to the S_CP case study described partially in section 2.

[8] The UML model is implemented by 38 000 lines ADA program.

[9] The UML model is implemented by 25 000 lines JAVA program.

Table 2. Table highlighting the number of expressible properties in 6 industrial case studies.

	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%)

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.

5.2 Context description

After property definition, we had to link each property to environment scenarios. Here, the work consisted in transforming use cases into context with our CDL language. One or several CDL contexts have been created according to the complexity of behavior contexts and to the environment actor number. Table 3 shows the number of paths obtained for different CDL models for the case study CS_1. This number depends on alternative and parallel operators, actors, interactions used in the CDL model. We linked a set of properties related to a specific phase or scenarios at each CDL model. We note that the verification time can be long (for example, 20 minutes for CDL_4 and CS_1) because the compilation time for state graphs IFx generation for each context path. In the future work, we focus on path reduction and evaluating how paths can be equivalent with respect to a particular property.

Table 3. Table highlighting the number of CDL and paths generated for CS_1.

	CDL_1	CDL_2	CDL_3	CDL_4	CDL_5
Number of actors	1	3	3	5	3
Number of path	3	128	82	612	96
Time of verification (sec)	6	256	164	1224	192

5.3 Proof unit exploitation

In the case studies, for each CDL model, one proof unit is created. A proof unit enables to organize a set of observers and one context. For each path generated by OBP, one accessibility graph is generated and represents the set of all possible model executions. A property is not verified by the tool if a "*reject*" observer automata state exists. For this, OBP produces set of error observers. During simulation execution, combinatorial explosion may appear. We do not resolve this point, but we propose this partial solution. It is necessary to create specific contexts in order to restrict the behaviors of the model. The solution is to initialize the system in specific configurations and to create specific CDL models which restrict scenario spaces with counters, actors, message parameters. So, partial verification is made on restricted scenario spaces.

6 Discussion and conclusion

CDL is a prototype language to formalize contexts and properties. But CDL concepts can be implemented in another language. For example, context diagrams are easily described using UML 2. CDL permits us to check 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 proof units which take CDL models as input and which generate sets of paths.

6.1 Approach benefits

CDL contributes to overcome the combinatorial explosion by allowing partial verification on restricted scenarios specified by the context automata. CDL permits to formalize contexts and non ambiguous properties. Property can be linked to whole or specific contexts. During experiments, we noted that some requirements were often described in the available documentation in an incomplete way. The collaboration with engineers responsible for developing this documentation has motivated them 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. The set of proof units gather all these data to perform proof activities and validate models. These data have to be "capitalized" if the implementation evolves over the development lifecycle. Proof units formalize proof contexts. It thus appears essential to study a framework to describe and formalize proof contexts as MDA components jointly describing the requirements to be checked and environment behaviors in which the model is plunged at the time of simulations and the formal analysis.

6.2 Using the CDL language

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 are raised. The first one is the lack of complete and coherent description of the environment's behavior. Use cases describing interactions between the MUS (S_CP for instance) and its environment are

often incomplete. For instance, data concerning interaction modes may be implicit. CDL diagrams development thus required discussions with experts who have designed the models under study in order to explicit all context assumptions.
The problem comes from the difficulty to formalize 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 requirements 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.
Furthermore, constraints from CDL can guide developers to construct formal properties to check against their models. As a result, developers can formalize system requirements. Using CDL, they have a means to rigorously check whether requirements are captured appropriately in the models using simulation and model checking techniques. Nevertheless, property patterns will continue to evolve as we receive feedback from academia and industry about possible improvements.

6.3 Property proofs

In the case studies, about forty significant requirements have been formally verified. These requirements were written by using the property language presented section 3, and then was translated automatically into IF2 observer automata. About 13% (non-computable) of the requirements (cf Table 2) required manual translation. They did not match the safety and bounded response time translation pattern,. The 61% (provable) are translated and afterwards verified automatically. For the others 26%, the requirements have to be discussed with the industrial partners to improve their use. Following that approach, we found, in two case studies (CS_1 and CS_5), an execution that didn't meet the requirements. Each case study corresponds to an operational embedded system. The classical simulation techniques could not permit to find these errors.

6.4 Future work

One element highlight, working on embedded software case studies with industrial partners, is the need of 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 proof. For that purpose, we identified MDA components, called *proof units*, referencing all the data, models, meta-models,

etc. necessary to the verification. The definition of such MDA components can take part in a better methodological framework, and afterwards a better integration of validation techniques in model development processes. Indeed, *proof units* themselves are handled as models, and are managed like a product resulting from the specification activities. As a conceptual framework, they allow the activity and the knowledge to be capitalized by gathering the necessary data to the proof. Consequently, the development process must include a step of environmental specification making it possible to generate sets of bounded behaviors in a complete way. This assumption is not formally justified in this article but is based on the essential idea that the designer can correctly develop a software system only if he knows the constraints of use. This must be provided formally by the process analysis of the designed software architecture, using a framework of development process. Although the CDL approach has been shown scalable on several industrial case studies, the approach suffer from a lack of methodology. The handling of contexts, and then the formalization of CDL diagrams, must be done carefully in order to avoid the combinatorial explosion when generating linear context path to be composed with the observer automata. The definition of such a methodology will be addressed by the next step of this work.
One essential point, dealing with model transformations, is the feedback obtained in the formal target technical space into the source one. We take advantages of model driven techniques and transformation traces in tooling to have validation feedbacks on source models. Current and future works are dealing with increasing diagnosis feedbacks to different users, including requirement managers and component model designers.

In addition, work is still in progress at CDL level. It focuses on path reduction, evaluating how paths can be equivalent with respect to a particular property. This optimization aims at reducing the combinatorial explosion, allowing treating larger and larger applications. Otherwise, experiments shown that part of the requirements found in industrial specification documents were not translatable into property patterns proposed by the approach. Several directions are followed to face the problem, one is to extend actual patterns, and another is to create other patterns. Implementation of experimental extended patterns is in progress.

Acknowledgments. This work results from collaboration between the authors and other members of the Ensieta team. We thank Leilde V. for his contribution in the OBP development and Aizier B. for his experiments and results.

References

1. Alur R, Dill D.: A Theory of Timed Automata. In Theoretical computer Science, 126(2), pp. 183-235 (2004)
2. Bozga M., Graf S., Mounier L.: IF2: A validation environment for component-based real-time systems. In Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen, LNCS. Springer Verlag (2002)
3. Clarke T., Evans A., Sammut P., Willians J.: Applied Meamodeling: A foundation for Language Driven Development. Technical report, version 0.1, Xactium (2004)

4. ITU-T. Recommendations Z-100. Specification and Description Language (SDL) (1994)
5. Dwyer M.B., Avrunin G.S., Corbett J.C.: Patterns in property specifications for finite-state verification. In Proc. of the 21st Int. Conf. on Software Engineering, pp. 411-420. IEEE Computer Society Press (1999)
6. Halbwachs N., Lagnier F., Raymond P.: Synchronous observers and the verification of reactive systems. In 3rd int. Conf. on Algebraic Methodology and Software Technology (AMAST'93) (993)
7. Haugen O., Husa K.E., Runde R.K., Stolen K.: Stairs: Towards formal design with sequence diagrams. In journal of Software and System Modeling (2005)
8. Fernandez J-C et al.: « CADP: A Protocol Validation and Verification Toolbox », in Alur R. and Henzinger T.A, editors, Proceedings of CAV'96 (new Brunswick, USA), Vol. 1102 LNCS, August (1996)
9. Janssen W., Mateescu R., Mauw S., Fennema P., Stappen P.: Model Checking for Managers. Conference Spin'99, pp. 92-107 (1999)
10. Konrad S., Cheng B.: Real-Time Specification Patterns. In Proc. Of the 27th Int. Conf. on Software Engineering (ICSE05), St Louis, MO, USA (2005)
11. Roger J.C. Exploitation de contextes et d'observateurs pour la vérification formelle de modèles, Phd report, Univ. of Rennes I (2006)
12. Smith R., Avrunin G.S., Clarke L. and Osterweil L.: Propel: An Approach Supporting Property Elucidation. In Proc. of the 24st Int. Conf. on Software Engineering, ACM Press, pp. 11-21 (2002)
13. Whittle J.: Specifying precise use cases with use case charts. In MoDELS'06, Satellite Events, pp. 290–301 (2005)
14. Berthomieu B., Vernadat F.: Time Petri nets analysis with TINA. 3rd Int. Conf. on the Quantitative Evaluation of Systems (QEST'06), Riverside (USA), pp. 123-124 (2006)
15. Berthomieu B, Bodeveix JP., Filali M., Garavel H., Lang F., Peres F., Saad R, Stoecker J., Vernadat F.: The Syntax and Semantics of FIACRE, Version 1.0 alpha. Technical report projet ANR05RNTL03101 OpenEmbeDD (2007)
16. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 2, pp. 244–263 (1986)
17. Hassine, J.; Rilling, J., Dssouli, R. Use Case Maps as a property specification language, Software System Model, 8, pp. 205-220 (2009)
18. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer, New York (1992)
19. Feiler, P., Gluch D.P., Hudak J.J.: The Architecture Analysis and Design Language (AADL):An introduction.Technical report, Society of Automotive Engineers (SAE) (2006)
20. Dhaussy P., Boniol F.: Mise en œuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués. pp. 133–157 RSTI (2007)
21. Dhaussy P., Auvray J., De Belloy S., Boniol F., Landel E.: Using context descriptions and property definition patterns for software formal verification, Workshop Modevva'08 (hosted by ICST 2008), Lillehammer, Norway (2008)
22. Dhaussy P., Creff S., Pillain P.Y., Leilde V.: CDL language specification (Context Description Language). Technical report version N° DTN/2009/8, ENSIETA (2009)