Progress in Flight Dynamics, GNC, and Avionics **6** (2013) 455-468
DOI: 10.1051/eucass/201306455
© Owned by the authors, published by EDP Sciences, 2013

A VERIFICATION AND VALIDATION PROCESS FOR MODEL-DRIVEN ENGINEERING

R. Delmas, A. F. Pires, and T. Polacsek

ONERA — The French Aerospace Laboratory
Toulouse F-31055, France

Model Driven Engineering practitioners already benefit from many well established verification tools, for Object Constraint Language (OCL), for instance. Recently, constraint satisfaction techniques have been brought to Model-Driven Engineering (MDE) and have shown promising results on model verification tasks. With all these tools, it becomes possible to provide users with formal support from early model design phases to model instantiation phases. In this paper, a selection of such tools and methods is presented, and an attempt is made to define a verification and validation process for model design and instance creation centered on UML (Unified Modeling Language) class diagrams and declarative constraints, and involving the selected tools. The suggested process is illustrated with a simple example.

1 INTRODUCTION

The importance of embedded software in aerospace systems has grown substantially in recent years, and the aeronautics industry now has to plan and support system maintenance and evolution over very long time horizons, over 70 years in some cases. With the spread of MDE approaches, the models are used to support various activities (design, verification and validation, system evolution, etc.) and should be regarded as critical assets in a long term perspective. This observation led to the creation of the OPEES project, an European Projects and French Competitively Poles Consortium for the definition, elaboration, and deployment of an Open Platform for the Engineering of Embedded Systems. One of its aims is to ensure long-term availability of engineering technologies and tools used for embedded systems development.

In recent years, MDE has become an important part of computer science. Modeling languages like UML or SysML are currently used in many industrial projects, especially in aerospace engineering. Although graphical notations can help designers master the system complexity by conveying intelligible visual cues which allow several people to cooperate on designs, these notations lack the expressiveness needed to capture the finer semantic details of a specification. So, it is often necessary to express additional semantic information through textual notations accompanying the model, such as the OCL [1] supported by the Object Management Group (OMG).

In spite of all these notations, the growth of complexity, in aerospace systems in particular, makes it sometimes very difficult to create correct models and correct model instances. However, more and more tools continue to appear, many from academia, which allow to verify models and their instances. Recent works like [2,3] and [4] propose to introduce formal methods in the MDE process at early stages of model design. The proposed approaches mostly consist in using constraint satisfaction techniques to perform various generic verification and validation tasks on a given model augmented with a set of semantic constraints. These approaches bring the possibility of using constraint satisfaction techniques in MDE to perform efficient verification tasks on models.

Out of all the existing modeling tools, very few offer a complete support (that is, both tools and associated methodologies) for both model design and model instantiation tasks.

The purpose of this paper is not to evaluate the features or performance of such tools in isolation, but rather to propose a generic design process, in which these tools are used in the best possible way considering their strengths or weaknesses, in order to provide a formal support to the user in each phase of the design, from early model design to model instantiation. In section 2, a selection of existing verification tools is presented. Section 3 addresses recent works on the use of constraint satisfaction techniques in MDE. Section 4 introduces the process itself and the description of all its phases. Section 5 illustrates this process with a simple example. In conclusion, section 6 presents some perspectives.

2 VERIFICATION OF INSTANCES

Today, the use of modeling language such as UML and SysML is widespread in the computer science and systems engineering. But while these notation languages allow to express many concepts, they lack the expressiveness needed to cover all the specification and modeling needs for typical systems. It is often necessary to define constraints on the model objects. It can seem attractive to write these constraints in the natural language, but it will just make them ambiguous. This is the reason why formal constraints specification languages were first introduced. Generally, these languages used complex notations which required mathematical knowledge from users. This can become a major problem if models and constraints are supposed to be understood and used by actors from different backgrounds. The OCL has been developed by the OMG to deal with this problem. Object constraint language is a declarative language and, consequently, evaluating an OCL expression has no side effects on the model (object creation is, however, possible in OCL2). It is typically used to specify constraints on class diagrams. With OCL, the goal of the OMG was to define a constraint specification language; yet, the question of its formal semantics was not precisely addressed by the standard, and in the early days, different implementations of OCL could have different behaviors.

The verification of OCL constraints has become a common practice, and there are many tools which allow to write and evaluate OCL constraints, most of them developed and supported by academia. Existing tools can be grouped in two categories depending on their underlying operation principle: either by interpretation of constraints, or by generation of executable code dedicated to the evaluation of constraints.

In the interpreter family, one finds tools such as USE (UML-based Specification Environment) [5], MDT (Eclipse Model Development Tools) [6], or Topcased (Toolkit in Open Source for Critical Applications and Systems Development) [7]. USE [8] is a standalone application developed at the university of Bremen. It operates on a subset of UML. It can be used from the command line or from a graphical user interface. It offers, in a single environment, a way for the user: (*i*) to specify both a model and a population of instances in a language based on UML; (*ii*) to specify OCL constraints as invariants or pre/postconditions; and (*iii*) to verify these constraints on instances. MDT is an Eclipse plug-in, which offers a variety of tools, mostly for the modeling and evaluation of OCL constraints on models based on the Eclipse Modeling Framework (EMF) [9] norm. For example, the implementation of its OCL evaluator allows, with the help of a Java API (Application Programming Interface), to create, interpret, and validate an OCL constraint on an instance of a meta-model expressed in the Ecore Language defined in EMF. The *Topcased* project was initiated in Toulouse in 2004 by a consortium of about thirty partners. Topcased is an Integrated Developement Environment presented as an Eclipse Rich Client Platform. It allows to evaluate OCL constraints on Ecore instances and to return results of this evaluation.

In the code generation family, one finds tools such as the Dresden OCL2 Toolkit [10] which allows to transform an UML2 model with its OCL constraints to Aspect Java. This method yields a better evaluation performance than interpretation. As shown in [11], the gain becomes really noticeable on instances with thousands of objects.

3 VALIDATION OF MODEL DESIGN

This section provides an overview of the current state of the art of tools and methods for model validation and verification, with a focus on constraint based approaches.

Works such as [2–4, 12] introduced the use of constraint satisfaction techniques to support analysis tasks on design models. For instance, in [4], the authors transform UML specifications into Constraint Satisfaction Problems in order to study model consistency. The authors of [2] make similar analyses, by transforming an UML specification into an Alloy [13] specification. Finally, the authors of [3] present an analysis of properties of a specification by compiling an UML class diagram in a series of Boolean satisfiability problems. While these works are only experimental today, still they look promising for constraint solvers in MDE.

The verification of formal models in the large is a very hard problem. The computational complexity of reasoning on UML class diagrams alone (with a restricted class OCL constraints) has shown to be EXP-time hard in [14]. If these theoretical results seem discouraging at first glance, many design problems happen to belong to simpler categories of problems with a particular structure and can be efficiently solved using constraint solvers like *Boolean satisfiability solvers* (SAT), *Satisfiability Modulo Theories* (SMT) [15] solvers, or *Constraint Satisfaction Problem* (CSP) solvers.

3.1 Unified Programming Language to Alloy

Unified programming language to Alloy (UML2Alloy) [2] transforms UML class diagram and OCL constraints to a code expressed in the Alloy [16] language, in order to run an analysis using to the Alloy Analyzer [17].

Alloy is a textual modeling language for software conceptions, based on first-order logic. Similarities exist between UML and Alloy but fundamental differences between them make the transformation of one language to another less trivial. For instance, Alloy does not directly support aggregation or composition exactly as they are defined in UML. In UML2Alloy, these difficulties were overcome by creating an Alloy metamodel in Meta-Object Facility (MOF) [18] format, by creating partial UML and OCL metamodels which are sufficient for the purpose of the supported analyzes and by specifying appropriate transformation rules between metamodels. Yet, restrictions still exist and not all of UML can be translated to Alloy by UML2Alloy.

The following analyzes can be carried using Alloy Analyzer models obtained by this transformation:

(*i*) simulation: the tool will try to find an instance which verifies all constraints in a finite search scope. If an instance is found, it can be visualized like a model; and

(*ii*) verification: the tool will verify each constraint and send back a counterexample if a constraint is violated.

Although the method used in UML2Alloy to implement the transformation of an UML model and its OCL constraints to Alloy allows to surpass the differences between the target and the source languages, there are still particular UML constructs that cannot be transformed to Alloy.

3.2 Unified Modeling Language to Constraint Satisfaction Problem

Unified Modeling Language to CSP (UML2CSP) [4] allows to transform a verification problem of an UML class diagram and its OCL constraints to a CSP. A CSP is represented by a triplet V, D, and C where V is the finite set of variables, D is the set of domains (a domain of values for each variable), and C is the set of constraints applicable on variables. A solution to a CSP is the assignation of values to the variables, such that all the constraints evaluate to true. Today, a wide variety of CSP solvers exists and a yearly international competition allows to benchmark latest innovations [19].

In [4], the authors remind that the verification of UML diagrams with arbitrary OCL constraints is undecidable, i. e., that it is not possible to determine if there exists a satisfying instance in finite time. In UML2CSP, the user specifies finite bounds on the search space. The implemented prototype uses the solver ECLiPSe [20] to solve the CSP instances. Different analyses can be performed on the model, such as:

weak satisfiability search for instances in which some class has a nonempty population of instances;

strong satisfiability search for instances in which all classes have nonempty populations of instances;

nonsubsumption given two constraints C_1 and C_2, search for instances in which C_1 is satisfied and C_2 is violated, to show that C_2 is not a trivial consequence of C_1 and has a real added value in the model;

nonredundancy generalization of the previous analysis, where there is a wish to show that both constraints C_1 and C_2 can be independently satisfied/falsified.

3.3 Unified Modeling Language to Boolean Satisfiability Solvers

The approach introduced in [3] is based on an encoding of UML/OCL model in SAT problems which are solved using MiniSAT [21]. Results of experiments on the verification of the model consistency[*] and the independence of OCL constraints[†] are presented. The authors claim that the SAT approach yields better execution times for the model verification than the enumeration approach of USE or the approach of UML2Alloy.

[*]A model is consistent if there exists a nonempty instance satisfying all constraints.

[†]A constraint is independent of any other constraint if it does not logically imply this constraint.

4 A PROCESS FOR VERIFICATION AND VALIDATION

In the current industrial context, textual specifications tend to be gradually replaced by model-based specifications. Even though these models allow to share information more easily and to deal with systems complexity, it is always difficult for the user to model complex systems efficiently. Indeed, due to the growth of models size and of the complexity of the constraints they are subject to, it is likely that mistakes will be made during the modeling phase. Recent works have shown that approaches based on constraint solvers can offer a performance benefit for verification and help raising the quality of models. In this section, a design, verification, and validation process is presented that will help users to identify the best suited formal techniques depending on the process phase, without getting lost in the plethora of options offered by the wide variety of methods and tools available today (as discussed in the previous section).

There are five phases in the process presented. A model of this process in *Business Process Modeling Notation* (BPMN) language is given in Fig. 1.

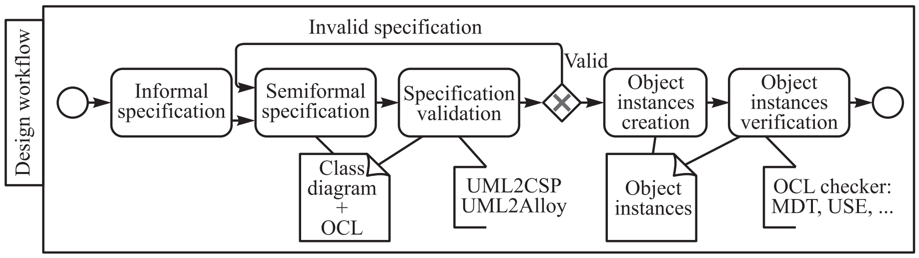

Figure 1 Process for models and instances design

Informal specification phase. This first phase consists in creating an informal specification in the natural language. This phase tends to disappear nowadays because of the use of modeling languages, but it is still presented in many projects. Indeed, it is a very quick and easy way to express the main concepts of a specification in a form close to one's natural own understanding of the problem.

Semiformal specification phase. In a context where the concept of supply chain and extended enterprise is more and more important (especially, in the aerospace industry with its multiple layers of subcontractors), it becomes necessary to go through a formalization phase in order to share information and to think on common concepts while minimizing ambiguity. This formalization is often performed using modeling languages such as UML. Moreover, it is possible to extend this specification by adding semantic constraints expressed in constraint languages like OCL.

Specification validation phase. Here, it is assumed that a specification is characterized by at least a model and collection of formal constraints. The goal of this phase is to ensure that the specification correctly captures the original intuition and has desirable formal properties like weak or strong satisfiability, absence of redundancy or subsumption, etc. For this purpose, tools like UML2CSP and UML2ALLOY, or the SAT approach introduced in [3], are best suited.

Instance creation phase. Once the generic formal properties of the specification have been assessed, the user is free to create an instance of the specification. An instance of the specification is seen as an UML Object diagram of the specification.

Instance verification phase. If instances have been manually created during the previous phase, it is necessary to verify that they are conformed to the specification, i. e., they are conformed to the OCL constraints expressed with the model. In that case, any verification tool such as Dresden or USE can be used.

5 SUPPORTING THE INSTANCE CREATION PHASE

So far, a design process has been suggested that offers an assistance in model design and that uses constraint satisfaction tools and methods. However, in [11], the authors suggest to go further in the use of constraints solvers, always in the scope of MDE: they do not limit themselves to the verification of models and instances, but they suggest to use the solvers to automatize a large part of the instance creation process thanks to constraint solvers.

Indeed, either the complexity of constraints or the size of the instances to be created can be so great that it can become difficult, when not practically impossible, to manually create correct instances of a model. In [11], the authors show on example that, with the help of the Java constraint solver CHOCO, it is possible to generate extended instances from partial instances not necessarily correct, which satisfy a given specification by construction.

The possibility to extend this process to include the instance synthesis phase has been studied, in order to propose a tool-supported process covering the whole spectrum of design activities from the specification to the instantiation. The method chosen for this phase consists in providing a constraint solver based tool with the specification as an UML class diagram and a set of declarative constraints, together with a partial instance containing all class instances but where some of the relation between objects are not yet fully defined. The tool then extends this partially defined instance in order to meet the specification.

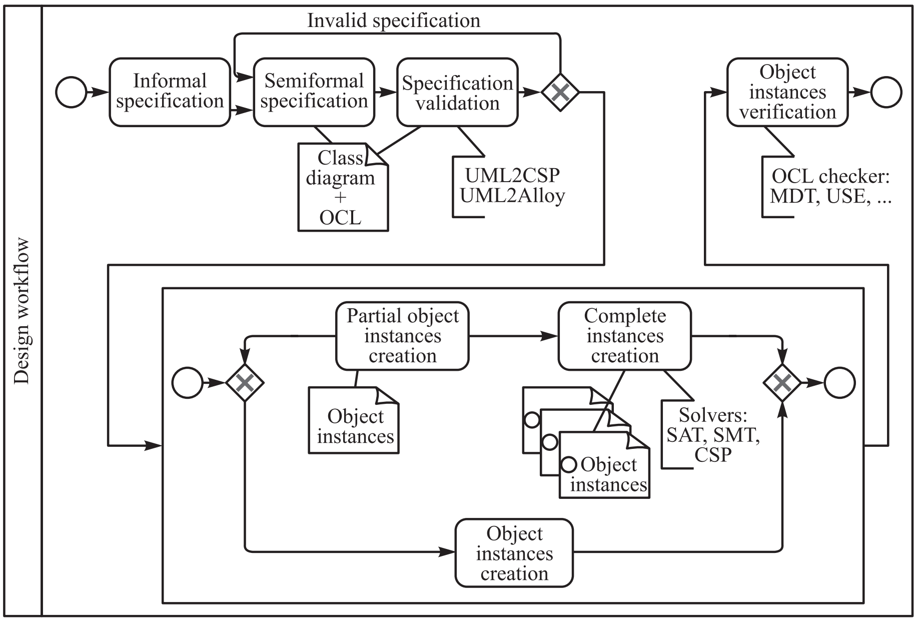

Figure 2 Extended process

Such design problems can have more than one solution, i. e., more than one possible complete instance for each partial instance provided as input. Luckily, as there will be seen on example in section 6, quantitative optimization criteria can be given to constraint solvers in order to guide their search towards optimal solutions, in a domain oriented sense. An extension of the process suggested with the synthesis phase is shown in Fig. 2.

6 APPLICATION ON AN EXAMPLE

6.1 Informal Specification Phase

Here, an example used to illustrate the suggested process is presented. It has been kept simple yet sufficiently interesting to illustrate the main concepts. It is a constrained allocation problem, in which various tasks must be allocated to agents in order to be processed, while respecting some limitations of agents but still ensuring that all tasks will be fulfilled. There are a lot of real-world examples of constrained allocation ranging from human resources management to critical embedded architecture design. Three entities are involved in the problem:

(*i*) topic: represents various fields of competence;

(*ii*) task: represents tasks to be taken care of by agents. Each task is linked to a single topic and must be assigned to only one agent. Moreover, each task has an attribute representing its workload, i. e., the amount of work needed to complete the task; and

(*iii*) agent: represents agents of the system. Each agent can do multiple tasks and can be competent in multiple topics. Last, each agent has a maximal workload that he can tolerate, which practically limits the number of tasks he can be assigned.

This system must verify the following constraints:

– `allAssigned`: each task is assigned to exactly one agent;

– `competentOnly`: each agent is only assigned tasks matching its topics of competence; and

– `noOverload`: no agents should be overloaded, i. e., the sum of the workload of all tasks assigned to an agent will not exceed its maximum workload.

6.2 Semiformal Specification Phase

In the first phase, a desirable system is modeled. The UML class diagram is available in Fig. 3*a*. In addition, the semantic constraints describing a proper allocation are formalized using OCL, as shown in Listing 1:

Listing 1

```
// Each task is allocated
context Task inv allAssigned :
Task.allInstances
  ->forAll(t:Task|t.allocatedAgent
    ->notEmpty())
// Agents are not overloaded with work
context Agent inv noOverload :
Agent.allInstances
  ->forAll (a:Agent | (a.taskToDo
    ->collect (t:Task | t.workload)
      ->sum < a.maxWorkload))
// Agents are only given tasks whose
// topics matches their skills
context Task inv competentOnly :
Task.allInstances
  ->forAll(t:Task |
    t.allocatedAgent.competence
      ->includes(t.taskTopic))
```

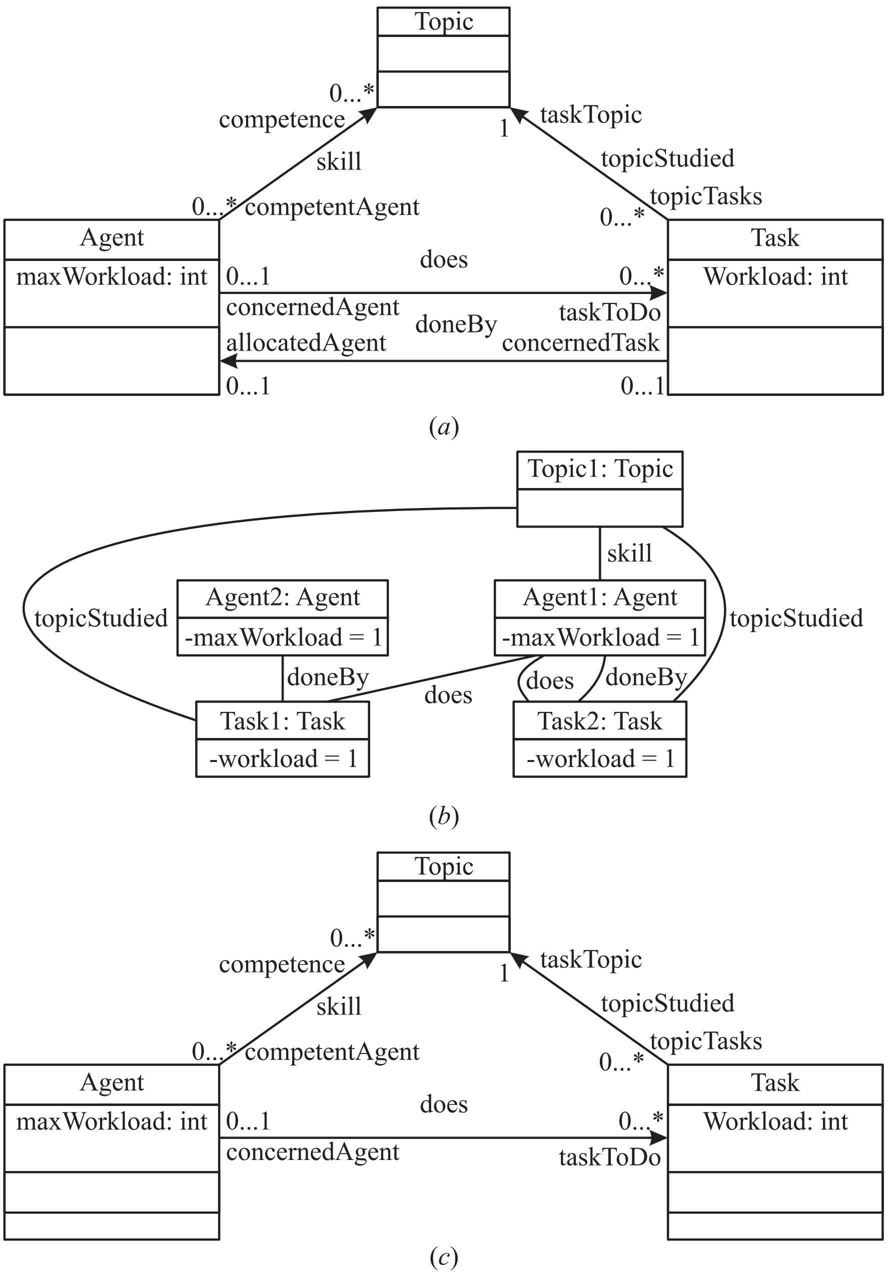

Figure 3 Example: (*a*) UML Class Diagram of the example; (*b*) instance generated by UML2CSP; and (*c*) corrected UML Class Diagram

6.3 Specification Validation Phase

For the experimentation, let verify the strong satisfiability property on the model for the first constraint (`allAssigned`), with a search space of two instances for `Agent` and `Task` classes, and one instance for the `Topic` class. In order to validate the specification, the following tools were used: UML2CSP [4] and UML2Alloy [2]. The tool UML2CSP is able to find a simple model instance, depicted in Fig. 3*b*.

The model is such that each task is assigned to an agent; yet, there is clearly something wrong in this model: `Agent2` does `Task1`, but task `Task1` is `doneBy` `Agent1`. So the `does` and `doneBy` relationships are not the exact inverse of one another. This comes from an error in the class diagram, which can be easily fixed by specifying that a unique bidirectional association name `does` between `Agent` and `Task` and adapting the OCL constraints with the new association. Corrections are visible in Fig. 3*c* and Listing 2:

Listing 2

```
// Each task is allocated
context Task inv allAssigned :
Task.allInstances
  ->forAll(t:Task|t.concernedAgent
    ->notEmpty())
// Agents are only given tasks whose
// topics matches their skills
context Task inv competentOnly :
Task.allInstances
  ->forAll(t:Task |
    t.concernedAgent.competence
      ->includes(t.taskTopic))
```

In the same way, there have been used UML2Alloy, limiting the analysis to the two structural constraints, and omitting the **noOverload** constraint (arithmetic expressions like the `sum()` used to express that no agent is overloaded are not supported). The analysis leads to the same conclusion: the specification is not contradictory, but some of the generated instances reveal the problem with the `does` and `doneBy` relations not being inverse of each other.

The conclusion is that despite the scope of action can be limited with the presented tools, it allows to find bugs in the problem specification and provides for a genuine improvement of the correctness of the specification.

6.4 Supporting the Instance Creation Phase

Now, when the problem has been formalized and confidence has been gained in its formal specification, formal methods can be used to actually solve the

Table 1 The CPU run times (in seconds) for instance synthesis

Topic	Agent	Task	MiniSat+	USE
3	5	5	0.004	0.597
3	5	8	0.004	1.972
3	5	10	0.004	—
5	10	50	0.045	—
15	25	100	0.258	—
20	50	200	2.677	—

problem. For example, a partial instance of the models is created where all class instances are known but where the `does` relation is unknown. In order to complete these instances, the constraints solver MiniSat+ was used and the results were compared with one more tool named USE [5], both introduced in [3]. For example, a translation of the specification and the instance to the different tools in which the objects were stipulated need to be completed. The goal here was not to implement a tool, but to experiment the process.

The experimentation was performed on a computer with an Intel Core2Duo processor clocked at 2.13 GHz with 2 GB of main memory. The results are shown in Table 1. The first three columns give the number of class instances for the `Topic`, the `Agent`, and the `Task`, respectively. The last two columns give the CPU run times for both approaches. The results show that the MiniSat approach is more interesting than the USE approach. It can be explained by the fact that USE is based on an iterative and exhaustive model enumeration method which does not scale up to large problems. It can be seen that on the last four experimented instances, USE was not able to find a solution in an acceptable time.

It was previously explained that many different instances can be built to the specification. Some models could be more favorable than others in some respects that are not fully captured by the semantic constraints. With a solver like MiniSat+, it is possible to add a criterion in the search of the solution in order to obtain the optimal one. So, the solver is asked to produce a solution which uses the fewest possible agents to perform all tasks. For this experimentation, execution times are more important than in a standard solutions search (14 s for a partial instance with 20 Topics, 50 Agents, and 200 Tasks, instead of 2.677 s previously), but the generated solution is optimal.

Although this phase was not fully implemented, the experimentation results for complete instances generation from partial instances with the help of constraint solvers seemed promising and it will be interesting to think about a tool which would fully automatize this phase.

6.5 Instance Verification Phase

In the experimentation made, two methods for OCL checking were compared: one based on USE and the other based on MiniSat+ as back-end solver. Even though it is not the main goal of these tools to proceed to instance verification, their functionality allows to deal with this task. The two methods deliver more or less the same performance, about 10 ms for hundreds of objects, and no noticeable blowup was observed with any of the tools. This observation is not really surprising, given the fact that the constraints used in the toy example are flat, universally quantified conditions ranging over all class instances of the model. These particular constraints cannot benefit from the powerful space pruning and conflict learning mechanisms offered by constraint solvers.

7 CONCLUDING REMARKS

The purpose of the paper was to illustrate how formal methods can be used throughout the different phases of a typical MDE process, from the problem formalization to the solution synthesis, and solution verification. It is argued that even though the formal verification and synthesis problem on UML models lies within the limit of the computationally tractable world, there are lots of practical design problems that are relatively easy to solve for the state of the art constraint solvers, due to their bounded and regular structure, and, hence, can benefit from constraint solving techniques.

Constrained allocation is a recurring problem in embedded systems design and the proposed process has been really fruitful in the authors' experience. The possibility to use widely accepted MDE notations to communicate about the problem to solve, to use formal methods to disambiguate the problem specification, and to generate and present cost-optimal solutions in the same MDE idioms to the designers has helped speeding up the problem solving process and the quality of the results.

The perspectives targets are both the methodological and the technical aspects of the proposed process. On the methodological side, the experiments will be continued to refine the approach and to try to identify what is the generic portion of the methodological experience on embedded systems that could translate to other fields of applications. The dynamic aspects of systems is also a very important topic to address and this will lead to study the possible gateways between MDE and temporal model checking techniques. On the technical side, and to better support the methodology, the current tool chain for verification and synthesis is made of a collection of ad-hoc translators, and the present authors would like to make it more generic and better integrated in the existing MDE platform.

REFERENCES

1. Group, O. M. 2006. Object constraint language object constraint language. OMG Available Specification. Version 2.0.
2. Anastasakis, K., B. Bordbar, G. Georg, and I. Ray. 2007. Uml2alloy: A challenging model transformation. *ACM/IEEE 10th Conference (International) on Model Driven Engineering Languages and Systems*. 436–50.
3. Soeken, M., R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. 2010. Verifying UML/OCL models using Boolean satisability. *Design Automation Test Europe*. 2010.
4. Cabot, J., R. Claris, and D. Riera. 2008. Verification of UML/OCL class diagrams using constraint programming. *2008 IEEE Conference (International) on Software Testing Verication and Validation Workshop*. 73–80.
5. http://www.db.informatik.uni-bremen.de/projects/USE/.
6. http://www.eclipse.org/modeling/mdt/.
7. http://www.topcased.org/.
8. Gogolla, M., F. Büttner, and M. Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* 69:27–34.
9. http://www.eclipse.org/modeling/emf/.
10. Wilke, C. 2009. Java code generation for Dresden OCL2 for Eclipse. Ph.D. Thesis. Dresden: Technische Universitt.
11. De Roquemaurel, M., T. Polacsek, J. F. Rolland, J. P. Bodeveix, and M. Filali. 2010. Assistance la conception de modèles l'aide de contraintes. *AFADL 2010 Proceedings*. 121–36.
12. Delmas, R., D. Doose, A. Fernandes Pires, and T. Polacsek. 2011. Supporting model based design. In: *Model and data engineering*. Lecture notes in computer science ser. 6918:237–48.
13. Jackson, D. 2003. Alloy: A logical modelling language. In: *Formal specification and development in Z and B*. Lecture notes in computer science ser. 2651:629–29.
14. Berardi, D., D. Calvanese, and G. De Giacomo. 2005. Reasoning on UML class diagrams. *Artificial Intelligence* 168(1-2):70–118.
15. Barrett, C., A. Stump, and C. Tinelli. 2010. The SMT-LIB Standard: Version 2.0. *8th Workshop (International) on Satisfiability Modulo Theories*.
16. http://alloy.mit.edu/community/.
17. http://alloy.mit.edu/alloy4/.
18. http://www.omg.org/mof/.
19. http://cpai.ucc.ie/09/.
20. http://eclipseclp.org/.
21. http://minisat.se/.