Model Search: Formalizing and Automating Constraint Solving in MDE Platforms

Mathias Kleiner[1], Marcos Didonet Del Fabro[2], Patrick Albert[2]

[1] Arts et Metiers ParisTech, CNRS, LSIS Laboratory, France
mathias.kleiner@ensam.eu,
[2] IBM Software Group, France
{marcos.ddf, albertpa}@fr.ibm.com

Abstract. Model Driven Engineering (MDE) and constraint programming (CP) have been widely used and combined in different applications. However, existing results are either ad-hoc, not fully integrated or manually executed. In this article, we present a formalization and an approach for automating constraint-based solving in a MDE platform. Our approach generalizes existing work by combining known MDE concepts with CP techniques into a single operation called model search. We present the theoretical basis for model search, as well as an automated process that details the involved operations. We validate our approach by comparing two implemented solutions (one based on Alloy/SAT, the other on OPL/CP), and by executing them over an academic use-case.

1 Introduction

The combination of models and constraints is well-known and widely used in software engineering. On the one hand, the model-driven engineering (MDE) approaches have been using constraint languages (like OCL[21]) to further specify metamodels. Many constraint-based tools such as [15,10] have been developed, mainly for model checking and animation. However, in most of these approaches, constraint-solving is an external operation that can hardly be automated (in terms of input generation and output retrieval), and usually relies on solver-dependent tasks. On the other hand, part of the constraint programming (CP) approaches have aimed at extending the search engines with higher-level language support, either to obtain solver-independent languages [19], or to solve object-oriented/relational problem definitions[5,2,27].

Typical MDE solutions require chaining operations of different nature, such as extractions, injections or transformations. However, the explicit scope of CP-based operations remains vague. We believe they can be seen as model operations with combinatorial properties (see [17] for an application scenario). The CP-solving operation thus needs to be model-driven, fully automated and integrated with existing MDE tools.

In this paper, we therefore present a formalization of CP-solving tasks within a solver-independent MDE process chain. We define it as a first-class model-driven operation called *model search*. Our solution generalizes existing approaches

with an identified set of elementary operations and model-based inputs and outputs. The operations cover the whole model search chain, from the data definition to the solver execution and data re-injection.

We validate our approach by implementing it based on two well know solvers: Alloy/SAT [5] and OPL/CP [8]. We also apply both chains on an academic example of software product lines and discuss the results.

This paper is organized as follows. In section 2, we introduce model-driven engineering and constraint programming. In Section 3, we describe the model search approach and we present formal definitions for it. In Section 4, we present a solver-independent MDE integration for model search. In Section 5, we present two implementations of the presented chain using known solvers. Experiments on an application use case are provided in Section 6, and Section 7 discusses related and future work.

2 Context

2.1 Introduction to MDE and model transformation

Model Driven Engineering considers models, through multiple abstract representation levels, as a unifying concept. The central concepts that have been introduced are terminal model, metamodel, and metametamodel. A terminal model is a representation of a system. It captures some characteristics of the system and provides knowledge about it. MDE tools act on terminal models expressed in precise modeling languages. The abstract syntax of a modeling language, when expressed as a model, is called a metamodel. The relation between a model and the metamodel of its language is called $conformsTo$. Metamodels are in turn expressed in a modeling language for which conceptual foundations are captured in an auto-descriptive model called metametamodel.

The main way to automate MDE is by executing operations on models. For instance, the production of a model Mb from a model Ma by a transformation Mt is called a model transformation. The OMG's Query View Transform (QVT) [20] defines a set of useful model operations and proposes clues on how they should be implemented. As a mean to provide interoperability with tools from non-MDE environments (often referred to as *technological spaces*), special model operations (often called *injection/extraction*) allow for data exchange (usually through serializing/parsing) [14].

We use in this article the model definitions introduced in [6]:

Definition 1 (model). *A model M is a triple* $< G, \omega, \mu >$ *where:*

- G *is a directed multigraph,*
- ω *is a model (called the reference model of M) associated to a graph* G_ω
- μ *is a function associating nodes and edges of G to nodes of* G_ω

Definition 2 (conformsTo). *The relation between a model and its reference model is called conformance and noted* $conformsTo$ *(or abbreviated C2).*

Definition 3 (metametamodel). *A metametamodel is a model that is its own reference model (i.e.it conforms to itself).*

Definition 4 (metamodel). *A metamodel is a model such that its reference model is a metametamodel.*

Definition 5 (terminal model). *A terminal model is a model such that its reference model is a metamodel.*

As stated by the previous definitions, the notion of reference model is independent from the absolute modeling levels. For instance, both the MOF metametamodel and the UML metamodel are reference models (respectively of the UML metamodel and of a UML (terminal) model). Therefore, the conformance relation is also level-independent and can simply be checked by the existence of a function μ between the graphs of a model and its reference model.

2.2 Constrained metamodels

The notion of constraints is closely tight to MDE. Engineers have been using constraints to complete the definition of metamodels for a long time, as illustrated by the popular combination UML/OCL. Constraints can be, for instance, checked against one given model in order to validate it. In our approach we will always consider that the metamodels on which we wish to conduct CP solving potentially have constraints attached. We propose the following to formally define such combination:

Definition 6. *A constrained metamodel CMM is a pair* $< MM, C >$ *where* MM *is a metamodel and* C *is a set (a conjunction) of predicates over elements of the graph associated to* MM*. C is an oracle that, given a model* $M =< G, MM, \mu >$*, returns true (noted* $C(M)$*) iff M satisfies all the predicates.*

Definition 7. *A model M* conformsTo *a constrained metamodel CMM if and only if* $C(M)$*.*

Many languages can be used to define predicates (i.e. constraints), with different levels of expressiveness. OCL supports operators on sets and relations as well as quantifiers (universal and existential) and iterators. In this article, we will be using an OCL-compatible extension (OCL+ [9]) that focuses on metamodel static constraints. OCL+ is itself defined by a metamodel (available as KM3 [6]) and a parser (generated with TCS [7]).

2.3 Introduction to constraint programming

Constraint programming (CP) is a declarative programming technique to solve combinatorial (usually NP-hard) problems. A constraint, in its wider sense, is a predicate on elements (represented by variables). A CP problem is thus defined by a set of elements and a set of constraints. The objective of a CP solver

is to find an assignment (i.e a set of values for the variables) that satisfy all the constraints. There are several CP formalisms and techniques which differ by their expressiveness, the abstractness of the language and the solving algorithms. In this article we will focus on the language part, i.e what kind of elements and constraints can be represented and reasoned about. In order to narrow the scope, we introduce two important CP formalisms: SAT (boolean SATisfiability problem) and CSP (Constraint Satisfaction Problem). Associated solvers and their (higher-level) language will be presented in Section 5.

The SAT formalism SAT problem is to decide if, for a given boolean formula, each boolean variable can be given an assignment such that the formula evaluates to true. SAT is known as being a NP-complete problem[1].

Definition 8 (SAT instance). *A SAT instance $\mathcal{S}$ is defined by $\mathcal{S} = (\mathcal{X}, \mathcal{C})$ where $\mathcal{X}$ is a set of boolean variables and $\mathcal{C}$ is a set of clauses. A clause is a finite disjunction of literals and a literal is either a variable or its negation.*

The CSP formalism CSP extends SAT in that it does not restrict variable domains to binary values.

Definition 9 (CSP instance). *A CSP instance is well-defined by a triplet $< X, D, C >$:*

- *X is a finite set of variables $X_1, ..., X_n$*
- *D is a finite set of domains $D_1, ..., D_n$ where D_i is a set of possible values for X_i*
- *C is a finite set of constraints where each constraint is an assertion on a subset of $X = X_j, ...X_k$ defined by a subset of $D_j, ..., D_k$*

Solving a CSP consists in assigning a value V_i of the domain D_i to each variable X_i such that it satisfies all the constraints in C.

3 Model search

Deterministic rule-based model transformations are not sufficient for different MDE scenarios, such as model animation or model automatic generation, because they cannot handle combinatorial parts of operations chain. For instance, in [17], the MDE scenario uses a CP-based technique for a part of the process in which the input model needs to be automatically completed. In this section, we present *model search* as a first-class MDE operation for handling such combinatorial tasks.

3.1 Relaxed metamodels and partial models

In order to formally define model search, we first define a set of notions that relate to constrained metamodels.

Definition 10 (Relaxed metamodel). *Let* $CMM =< MM, C >$ *be a constrained metamodel.* $CMM_r =< MM_r, C_r >$ *is a relaxed metamodel of CMM (noted* $CMM_r \in Rx(CMM)$*) if and only if* $G_{MM_r} \subseteq G_{MM}$ *and* $C_r \subseteq C$*.*

In other words, a (minimal) relaxed metamodel can be obtained by the removal of all constraints: minimum cardinalities are set to zero, attributes are optionals and predicates are removed. Computing such a relaxed metamodel can obviously be done easily with existing (meta)model transformation techniques. We call this operation *relaxation*.

Definition 11 (Partial model, p-conformsTo). *Let* $CMM =< MM, C >$ *be a constrained metamodel and M a model.* M_r *p-conformsTo CMM if and only if it conforms to a metamodel* CMM_r *such that* CMM_r *is a relaxed metamodel of CMM (*$CMM_r \in Rx(CMM)$*).* M_r *is called a partial model of CMM.*

3.2 Model search

Definition 12 (Model search). *Let* $CMM =< MM, C >$ *be a constrained metamodel, and* $M_r =< G_r, MM_r, \mu_r >$ *a partial model of* CMM*. Model search is the operation of finding a (finite) model* $M =< G, MM, \mu >$ *such that* $G_r \subseteq G$*,* $\mu_r \subseteq \mu$ *(embedding i.e* $\forall x \in Gr, \mu(x) = \mu_r(x)$*), and* M $conformsTo$ CMM*.*

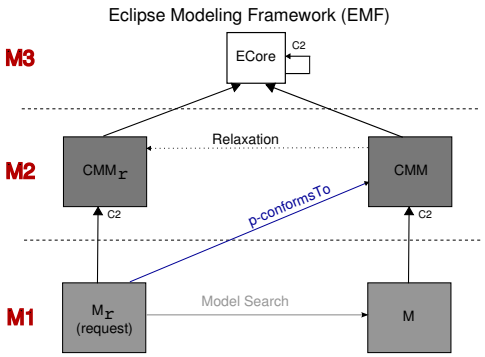

Fig. 1. Model search

This MDE operation is illustrated in Figure 1. We consider model search as a model transformation where the source (metamodel and model) is an instance of a non-deterministic (combinatorial) problem and the target model is a solution (if any exists). From the CP point of view, the target metamodel acts as the constraint model whereas the source model (the request) is a given partial assignment that needs to be extended.

4 A solver-independent MDE integration

As introduced in the previous Section, the goal of model search is to generate a complete and valid model M of a constrained metamodel CMM out of a partial (possibly empty) model M_r (the *request*). Figure 2 illustrates the whole process in a model-driven engineering framework. This process is composed of 5 main tasks.

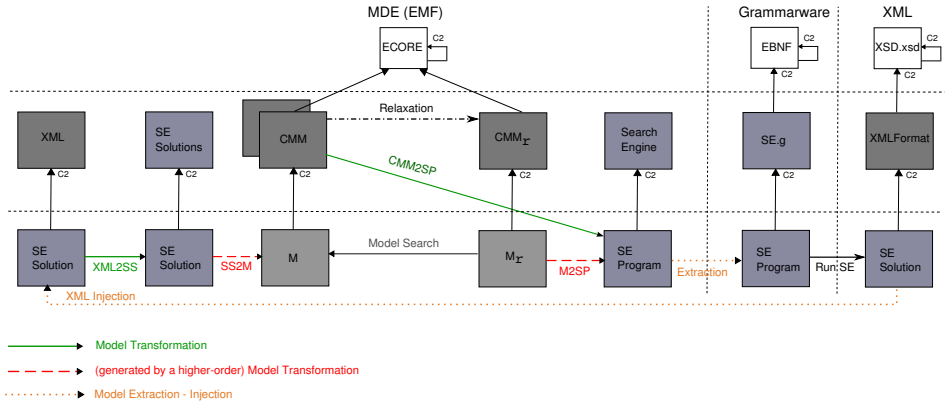

Fig. 2. Model search integration

1) Search problem generation: this task, illustrated by the $CMM2SP$ transformation, expresses the constrained metamodel as a model conforming to the search engine metamodel. However, the $CMM2SP$ arrow in Figure 2 is a simplified view of the process, since there are actually two source models (represented by the doubled square) as input to the transformation. Figure 3 illustrates the complete generation process. The metamodel MM contains the structural constraints, such as cardinality and lower bounds. However, typical model search applications require more complex domain constraints. These domain constraints, which are not part of standard ECORE, are expressed in the constraint model C, which conforms to the OCL metamodel.

The difficulty of expressing a constrained metamodel in the search engine language is highly dependent on the abstraction level and the basic elements offered by the language. Differences between search engines and implementation issues will be thoroughly discussed in Section 5.

2) Search data generation: this task is illustrated by the $M2SP$ transformation. It takes the request model M_r as input and generates the input data that will be used by the search engine. However, this transformation is metamodel-specific, i.e., there is one $M2SP$ per input metamodel. To avoid writing one transformation for every metamodel MM, a higher-order transformation (HOT) is defined. The HOT takes MM as input and produces the transformation $M2SP$, using the transformation language as target metamodel.

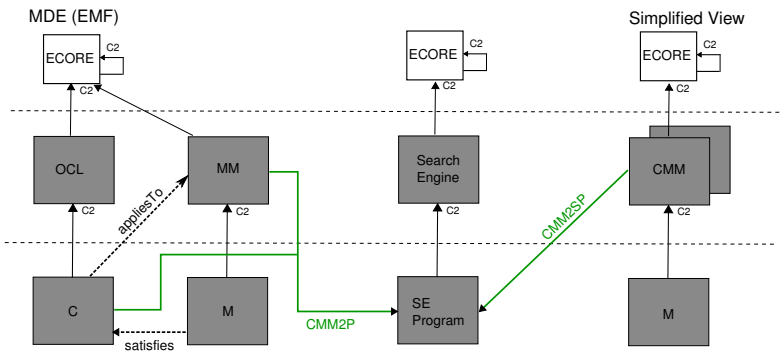

Fig. 3. Generation of the problem definition

It is important to note that most of the search engines do not separate the problem and the data definition: they are expressed all together using the same language. For that reason, the input data and the problem definition are merged. This is a straightforward task, since there are no overlapping elements.

3) Engine program extraction: this task extracts the search engine model into its executable format.

4) CP search: the generated search program is executed in the search engine. When the search succeeds (i.e there is at least one solution), we obtain a solution model in the search engine output format. The most common formats are XML or plain-text files.

5) Solution injection: this last task is to inject the resulting solution produced by the search engine as a model of the original metamodel MM. We have illustrated this tasks by two transformations, $XML2SS$ and $SS2M$. Although those transformations could be merged, we have considered that the engine generates an XML file. As a result, it is natural to decompose the operation into two tasks: expressing the XML model as a model of a metamodel of the search engine solutions, then transform it to a model of MM. For the same reasons as the $M2SP$ transformation, $SS2M$ is generated using a HOT, which takes MM as source and generates a transformation from SS to MM.

ECORE vs KM3 The presented process assumes the use of EMF's Ecore (from the Eclipse modeling project) as the metametamodel. Effective implementations, as the ones described later in this article, define the metamodels using the KM3 language[6]. Since KM3 offers automatic translation to Ecore, the conversion from one framework to another does not introduce any difficulty.

5 Implementation alternatives

The presented chain is solver-independent. We materialize such chain using two technologies. However a number of difficulties arise with implementations, because of the solvers languages lack of expressiveness or the formalisms inherent

limitations. We discuss in this section the difficulties that we identified for each of the two formalisms presented in Section 2.3 combined with state-of-the-art solvers: OPL/CSP and Alloy/SAT.

5.1 Implementation with Alloy/SAT solver

The SAT paradigm has clear limitations: it requires a finite set of boolean variables and only offers a low-level predicate language (only negation, disjunction and conjunction are supported). However, [5] introduced an expressive relational language with a built-in compilation that allows the use of many recent SAT solvers. We will thus use Alloy as our target search engine language in order to ease the transformation definition.

Alloy, which can be seen as a subset of the Z language [13], allows for expressing complex predicates using atoms (indivisible elements), sets (of atoms), relations, quantifiers (universal or existential), operators for relations traversal, etc. However, due to the properties of SAT problems, Alloy cannot be considered as a true first-order logic solver. Indeed, to be able to translate the problem into SAT, a *scope* needs to be given to each set, that limits the number of atoms that can be contained in the set.

In Alloy, every element is either an atom or a relation but the language is exclusively based on relations. Indeed, a set is itself a relation from an atom to the contents of that set (which in turn are also atoms). The main artifacts that we will manipulate in the Alloy language are:

- *Signatures*, declarations of sets, for which the body may contain fields as *relations* to other signatures. Attributes are treated the same as any relation. Scalars, as for signatures, are treated as sets of atoms. Signatures also support a form of inheritance.
- *Facts*, declarations of predicates, with quantifiers and an important number of logical, scalar and set operators available.

Generic expression of constrained KM3 metamodels We developed a metamodel of the Alloy language containing the necessary constructs to represent KM3 metamodels and OCL+ constraints. Figure 4 shows an overview of the metamodel. The complete metamodel is written in KM3. We also developed a TCS parser generator allowing to inject/extract between the textual version of the language and our metamodel. Both the metamodel and the TCS are freely available, submitted as a TCS use case (under the form of an Eclipse project), and can be downloaded from [4]. The OCL+ metamodel is also written in KM3, its metamodel and TCS are freely available and can be downloaded from [9]. An overview is presented in Figure 5.

On this basis, we defined a mapping from KM3 to Alloy and developed the corresponding transformation, using ATL (AtlanMod Transformation Language), a QVT-like model transformation language and tool [11]. An excerpt of the mapping is presented in Table 1. In short, KM3 classes are mapped to Alloy signatures, KM3 attributes and references are mapped to Alloy fields, references

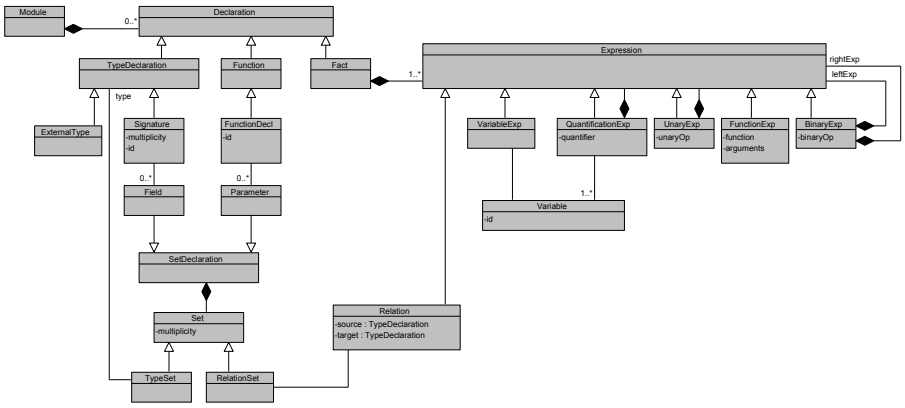

Fig. 4. Overview of the Alloy metamodel

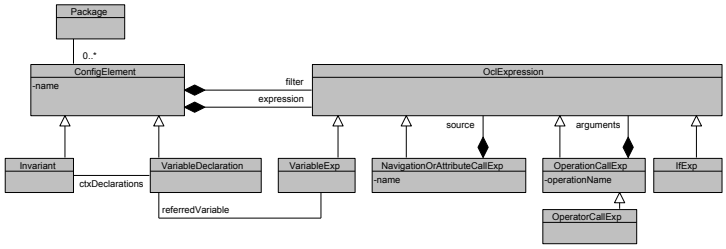

Fig. 5. Overview of the OCL+ metamodel

properties are turned into facts. We also developed an ATL transformation from OCL+ to Alloy so as to express metamodel constraints. An informal excerpt of these mappings are presented in Table 1. Both the transformations are merged into a unique transformation using two source models and able to resolve the links between the constraints and the metamodel elements on which they apply. This combined transformation corresponds to the *CMM2SP* of Figure 2. It is freely available, submitted as an ATL use case (under the form of an Eclipse project), and can be downloaded from [18].

The whole project is a partial implementation (all but the two high-order transformations) of the model search process presented in Section 4, using Alloy as the search engine. Thanks to its language expressiveness, Alloy bridges the gap between constrained metamodels and low-level languages. As an open-source tool, it is a viable alternative with only few drawbacks.

5.2 Implementation with OPL/CP solver

OPL (Optimization Programming Language) [28] is a language part of the IBM ILOG $^{\mathrm{TM}}$OPL-CPLEX$^{\mathrm{TM}}$development bundle [8], which is an IDE for developing CP and optimization models. The OPL programs are executed by the IBM

KM3 concept	Alloy concept
Metamodel	Module
DataType	ExternalType
Class	Signature
Attribute	Field
Reference	Field
StructuralFeature multiplicity	Quantifier or Fact
Reference containment	Fact
Reference opposite	Fact
OCL+ concept	**Alloy concept**
Invariant	Fact and QuantificationExpression
Invariant declarations	QuantificationExpression variables
VariableDeclaration	Variable
VariableExp	VariableExpression
IfExp	ImpliesExpression
NavigationOrAttributeCallExp	NavigationExpression
OperatorCall	BinaryExpression
OperationCall (size)	SetCardinalityExpression
OperationCall (isIn)	ComparisonExpression
OperationCall (others)	ExternalFunction

Table 1. Excerpt of the mapping from KM3 and OCL+ to Alloy concepts

ILOG^{TM}CP Optimizer engine. The OPL language has a clear separation between the input data (booleans, integers, sets, strings, tuples, and others) and the decision variables (integers and arrays of integers). It offers as well a set of logical and arithmetic expressions on those elements. These features - together with the possibility of defining universal quantifiers over variables - enables the reutilization of the optimization models over different data.

We have developed an OPL metamodel based on the definition from [8] (see an extract on Figure 6). The main structures used are the following:

- *Expressions*: combination of logical and arithmetical expressions, functions, aggregates (sum, union, max and min) and (indexed) variables.
- *Input parameters*: the input (fixed) data. A parameter may be initialized from a data set or it can be calculated using any kind of expression.
- *Decision variables*: the decision variables are scalars or arrays of integers and doubles. The decision variables may considered the "output data", i.e., the values of the decision variables are assigned based on a set of constraints and on the input data.
- *Constraints*: constraints are logical expressions that are written as an arbitrary composition of expressions, input parameters and decision variables. These constraints must be respected during the solver execution.

Generic expression of constrained KM3 metamodels We have applied the same approach as for the Alloy/SAT tool: we have developed an OPL metamodel in KM3 and a TCS parser generator for the injection/extraction between the OPL textual version and the model[3]. The mapping from constrained KM3 into OPL has a higher conceptual mismatch (model-based vs integer-based) than the one into Alloy. An excerpt of the mapping is presented in Table 2.

[3] The complete OPL metamodel/TCS and the transformation from KM3/OCL+ are not freely available

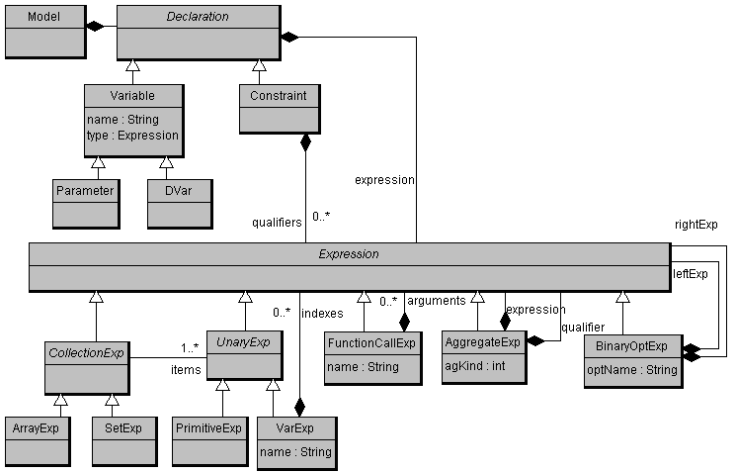

Fig. 6. Extract of the OPL metamodel

The transformation has three major set of rules. First, the KM3 model is transformed into the input parameters. Second, the KM3 metamodel is transformed into the output decision variables. The KM3 metamodel is transformed twice because of the difference of expressiveness between the OPL input parameters and the decision variables. The decision variables are restricted to integers and array of integers. Finally, the constraints are transformed into OPL constraints compatible with the input and output variables. The arithmetical, logical and comparison expressions are translated into their equivalent counterparts in OPL. The navigation expressions are translated into indexed decision variables, where the index is the calling expression. Then, the collection expressions are transformed into aggregates.

6 Application and Experiments

In this section, we describe an effective application of the approach on a Software Product Lines (SPL) use case. First, we briefly present the context of SPL and the considered problem. Then, we compare the results of the two implementation alternatives presented in Section 5.

6.1 Search in software product lines

The goal of SPL [22] is to create a shared model for a given application domain, which acts as a basis to generate a set of derived products. The specificities of each product are defined by features satisfying the needs of a particular application. These features contain explicit variation points, which guide the generation of the final products.

KM3 concept	OPL concept
Metamodel	Model
DataType	Type
Class	Integer set
Attribute	PrimitiveType set and integer array
Reference	Integer set and integer bi-dimensional array
StructuralFeature multiplicity	Aggregate expression
Reference containment	Global uniqueness constraint
Reference opposite	Equality constraint
OCL+ concept	**OPL concept**
Invariant	ForAll constraint
Invariant declarations	ForAll qualifiers
VariableDeclaration	Qualifier expression
VariableExp	VarExp
NavigationOrAttributeCallExp	Indexed variable exp
OperatorCall (arithmetic and logical)	BinaryOptExp + operator type
OperationCall (size)	AggreateExp
OperationCall (isIn)	AggregateExp and indexed variable
OperationCall (others)	FunctionCall
CollectionExp	Combination of aggregate expression

Table 2. Excerpt of the mapping from KM3 and OCL+ to OPL

The first step in a SPL chain is to define a model of the shared domain. A domain model contains a set of characteristics and components that are common for a class of applications, plus a set of variation points. The variation points may be expressed in terms of choices of possible values or in terms of user constraints. Each combination of variation point may generate a distinct product (a process called *derivation*). In other words, finding and generating all the possible products satisfying a set of constraints in a SPL is a model search problem. The domain model and the variation points are expressed in terms of a constrained metamodel.

In our example we need to generate classes that handle the execution of watches (this use case is an adaptation from [23]). We want to generate 5 different kinds of watches: 1) one simple watch, 2) one with alarm, 3) one with sound alarm, 4) one with sound and visual alarm and 5) one with visual alarm. We provide below a simple KM3 metamodel for this SPL problem:

```
package watches {
   class Root {
     reference classifiers[1-10] container : Class;
   }
   abstract class Class {
      reference methods[1-15] container : Method oppositeOf class;
   }
   class Watch extends Class {
    reference class : Class oppositeOf methods;
   }
   abstract class Method { }
   class DisplayTime, Start, StartAlarm, StartSoundAlarm,
       StartVisualAlarm, Stop, StopAlarm extends Method {}
}
```

However, not all combinations of methods are allowed. The derived models should respect the following constraints. 1) the *DisplayTime* and *Start* methods

are mandatory; 2) if there is a *Start*, there is a *Stop*; 3) if there is a *StartAlarm*, there is a *StopAlarm*; 4) if there is a *StartSoundAlarm*, there is a *StartAlarm*; 5) if there is a *StartVisualAlarm*, there is a *StartAlarm*. We show below one constraint in OCL+, Alloy and OPL, respectively.

```
context Class inv : methods.exists ( m | m.oclIsTypeOf(Start) ) ;
fact { all c : Class | some m : c.methods | m in Start }
forall(c in classes)(sum(m in methods) (c_m[c][m] > 0 && m in start)) >=1;
```

6.2 Results

We executed an Alloy/SAT and an OPL/CP chain in a Intel Core Duo, 2.53GHz, 3GB of RAM and 32 bit processor, with the same metamodel and the same set of constraints. We used Alloy 4.1.10 with the default SAT4J [26] solver and OPL 6.2 with the CP solver. The CMM2SP for Alloy transformation produced an Alloy program with 107 lines and the CMM2SP for OPL produced an OPL program with 131 lines.

The initial setting for executing the solvers is the standard setting of both tools. However, in the Alloy case, the bit-width is increased to 6 (the default is 4), to be able to represent the cardinalities of the references $classifiers$ and *methods*. OPL/CP has a largest integer default of $2^{31} - 1$. We used the same input, i.e., one element per class, (1 Root, 1 Watch, 1 Start, 1 Stop, 1 StopAlarm, 1 StartAlarm, 1 DisplayTime, 1 StartVisualAlarm and 1 StartSoundAlarm).

Both engines produced the combination of methods shown in Figure 7. Despite being simple, this example enables the visualization of all the solutions produced and the implication of the input constraints in the output models.

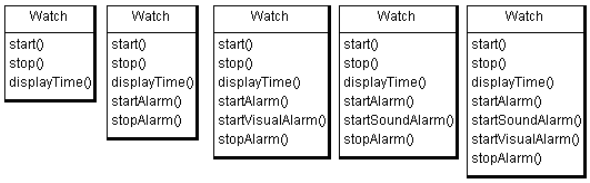

Fig. 7. Produced watches

Both transformation chains produced equivalent solutions from the same specification. The operations are called automatically (sequentially) by an Apache-Ant script. The Alloy program is then translated by the built-in compiler into a SAT predicate (923 lines) with 753 vars, 79 primary vars and 1271 clauses. The extraction plus the execution were executed in 0,19 seconds. The OPL program is transformed and executed by the CP engine in-memory, so the number of lines of the problem is not accessible. The problem definition has 8 variables and 300 constraints. It was executed in 0,11 seconds. The high difference in the

number of variables is due to the expressiveness of the outputs: boolean-based vs integer-based. The conceptual mismatch between OCL+ and OPL is higher than from OCL+ and Alloy, which provides an expressive language based on relations and logic.

The clear separation of the problem definition from the input data in OPL - and the existence of universal quantifiers - enables defining a $CMM2SP$ for OPL transformation independent of the input data. In particular, it is not necessary to unfold all the variables and loops.

The metamodels of both tools have been designed to work directly with the TCS parser. Therefore, some syntactical constructs would deserve a semantical analysis to completely check the validity of a textual input during model injections. This 2-step parsing process is left for future work.

To summarize, we were able to execute model search on both tools by implementing the modeling operations of the chain. Despite differences on the solver capabilities, the tools produced the expected results, showing the applicability of our approach. Automatic generation of the problem did not here generate an extra performance overhead compared to a manual definition. Considering the complexity of constrained search, this should however be validated on larger problems.

7 Related and future work

To the best of our knowledge, this article presents a first formal definition of *model search* as a deep integration of constraint programming in the MDE conceptual framework. However, this work can be linked to other recent developments that apply optimization techniques to solve MDE problems. For example, [16] describes transformation as an "Optimization Problem", this is very close to our approach, though in this case the optimization engine is not used during the transformation. The goal is rather to find a good transformation starting from a small set of available examples. A large share of the work of Jules White relates to our approach. The CURE system - for Configuration Understanding and REmedy [12] - for example, transforms a configuration into a set of constraints, automating the diagnosis of invalid configurations or the adapting of existing configurations to fulfill new requirements. Our approach also differs from SPL-dedicated solutions, such as [24], because we do not target a specific application domain. Based on some preliminary work, we believe that the SPL as a whole will benefit from the model search approach.

More generally, the many bridges that have been built between CP and MDE in the past years can be divided in two categories:

The CP community that works on modeling has started focusing on the DSL (Domain Specific Languages) approach for providing specific modeling languages, see [19] or [25], while preserving the so-called "solver independence", or supporting object-oriented or relational problem definitions [5,2,27]. Although they usually do not provide MDE integration, these languages have a higher

expressiveness and adapted engine support, therefore easing the transition from metamodeling languages.

The MDE community has been using constraint languages to further specify metamodels or transformation rules, while constraint-based tools such as [15,10] were developed mainly for model checking and validation. More recently, constraints have been considered to specify transformations or extend their capabilities [3]. Most of these tools depend on a specific solver/language. Moreover, the MDE integration is incomplete either because the inputs/outputs for the engine cannot be directly generated/retrieved or because they do not use model-driven transformations. Finally, partial assignments (i.e non-empty input models) are usually not taken into account.

In this respect, this article presents both a formalization and generalization of these approaches. Existing tools can be seen either as partial implementations, components or goal-specific usage of model search. By formalizing model search as a first-class model operation, we allow for comparison and integration into MDE platforms.

As future work, we plan to release the higher-order model transformations needed to complete the whole presented process. One of them allows to transform models to partial instances of the considered problem, and the other to transform the search results into a MDE format (*M2SP* and *SS2M* in Figure 2). We also plan to further validate the two implementations on a set of industrial and academic use cases[4]. At the theoretical level, our model search theory and process can naturally be extended to a general model transformation scheme (i.e with different source and target metamodels), which would expand the scope of transformations through an implementation of Relational-QVT.

8 Conclusion

In this article, we presented and formalized the use of constraint-based search engines in MDE platforms as a novel model operation called *model search*. Besides the presented theoretical foundations, we also described a MDE solver-independant process chain to realize model search, demonstrated its validity with two implementations (resp. Alloy/SAT and OPL/CSP), and discussed the results through experiments on an academic SPL use case. The presented approach generalizes existing work about constraints resolution in MDE, allows the complete automation of the process and provides a basis to develop and compare different alternatives. It simplifies the use of constraint solvers in model-based software engineering at two levels: a shared knowledge representation (constrained graph-based models), and a generic process in that one implemented chain is reusable for any application scenario.

Acknowledgments This article has been partially funded by ANR Idm++ project.

[4] We have successfully applied the approach to parse English sentences[17].

References

1. Cook S. A. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.
2. Felfernig A., Friedrich G., Jannach D., and Zanker M. Configuration knowledge representation using uml/ocl. In *UML*, pages 49–62. Springer, 2002.
3. Petter A., Behring A., and Muhlhauser M. Solving constraints in model transformations. In *ICMT'09*, pages 132–147, 2009.
4. *Alloy usecase: http://www.lsis.org/kleinerm/MS/Alloy_mm.html*, 2010.
5. Jackson D. Automating first-order relational logic. In *FSE*, pages 130–139, 2000.
6. Jouault F. and Bézivin J. Km3: A dsl for metamodel specification. In *FMOODS*, volume 4037 of *LNCS*, pages 171–185. Springer, 2006.
7. Jouault F., Bézivin J., and Kurtev I. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE*, pages 249–254. ACM, 2006.
8. *IBM ILOG CPLEX Development Bundle: http://www-01.ibm.com/software/integration/optimization/cplex-dev-bundles/*, Dec 2009.
9. *OCL+ usecase: http://www.lsis.org/kleinerm/MS/OCLP_mm.html*, 2010.
10. Cabot J., Clarisó R., and Riera D. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *ASE*, pages 547–548, 2007.
11. Jouault J. and Kurtev I. Transforming Models with ATL. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.
12. White J., Schmidt D. C., Benavides D., Trinidad P., and Ruiz-Cortez A. Automated diagnosis of product-line configuration errors in feature models. In *Software Product Lines Conference (SPLC 2008) Limmerick, Ireland*, 2008.
13. Spivey J.M. *The Z Notation : a reference manual.*, 2001.
14. I. Kurtev, J. Bezivin, and M. Aksit. Technological spaces: An initial appraisal. In *International Symposium on Distributed Objects and Applications*, 2002.
15. Gogolla M., Büttner F., and Richters M. Use: A uml-based specification environment for validating uml and ocl. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.
16. Kessentini M., Sahraoui H. A., and Boukadoum M. Model transformation as an optimization problem. In *MoDELS*, pages 159–173, 2008.
17. Kleiner M., Albert P., and Bezivin J. Parsing sbvr-based controlled languages. In *Models'09*, pages 122–136, 2009.
18. *Model search: http://www.lsis.org/kleinerm/MS/ModelSearch-Alloy.html*, 2010.
19. Nethercote N., Stuckey P. J., Becket R., Brand S., Duck G. J., and Tack G. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th CP. 3(4). LNCS, vol. 4741*, pages 529–543, 2007.
20. Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0*, 2008.
21. *OCL 2.0 specification: http://www.omg.org/spec/OCL/2.0/*, 2008.
22. Clements P. and Northrop L. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 1st edition, 2001.
23. Tessier P, Servat D, and Gerard S. Variability management on behavioral models. In *VaMoS Workshop*, pages 121–130, 2008.
24. Trinidad P., Benavides D., Cortés A. R., Segura S., and Alberto Jimenez. Fama framework. In *SPLC*, page 359. IEEE Computer Society, 2008.
25. Chenouard R., Granvilliers L., and Soto R. Model-driven constraint programming. In *10th ACM SIGPLAN PPDP, Valence, Spain*, 2008.
26. *SAT4J. A SATisfiability libray for Java: http://www.sat4j.org*, 2010.
27. Junker u. and Mailharro D. The logic of (j)configurator : Combining constraint programming with a description logic. In *IJCAI'03*. Springer, 2003.
28. Hentenryck P. V. *The Optimization Programming Language*. MIT Press, 1999.