Coupling solvers with model transformations to generate explorable model sets

Théo Le Calvar, Fabien Chhel, Frédéric Jouault, Frédéric Saubion

▶ To cite this version:

Théo Le Calvar, Fabien Chhel, Frédéric Jouault, Frédéric Saubion. Coupling solvers with model transformations to generate explorable model sets. Software and Systems Modeling, 2021, 20 (5), pp.1633-1652. 10.1007/s10270-021-00867-0 . hal-03594336

HAL Id: hal-03594336
https://hal.science/hal-03594336

Submitted on 9 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Software & Systems Modeling manuscript No.
(will be inserted by the editor)

Coupling Solvers with Model Transformations to Generate Explorable Model Sets

**Théo Le Calvar · Fabien Chhel ·
Frédéric Jouault · Frédéric Saubion**

Received: 23 April 2020 / Revised: 8 January 2021 / Accepted: 26 January 2021

This is a post-peer-review, pre-copyedit version of an article published in Software and Systems Modeling. The final authenticated version is available online at: `https://doi.org/10.1007/s10270-021-00867-0`

Abstract Model transformation is an effective technique to produce target models from source models. Most transformation approaches focus on generating a single target model from a given source model. However there are situations where a collection of possible target models is preferred over a single one. Such situations arise when some choices cannot be encoded in the transformation. Then, search techniques can be used to help find a target model having specific properties.

In this paper, we present an approach that combines model transformation and constraint programming to generate explorable sets of models. We extend previous work by adding support for multiple solvers, as well as extending ATL, a declarative transformation language used to write such transformations. We evaluate our approach and language on a task scheduling case study including both scheduling constraints, and schedule visualization.

Théo Le Calvar
Univ Angers, LERIA, SFR MATHSTIC, F-49000 Angers, France
DIRO, Université de Montréal
E-mail: firstname.lastname@univ-angers.fr

Frédéric Saubion
Univ Angers, LERIA, SFR MATHSTIC, F-49000 Angers, France
E-mail: firstname.lastname@univ-angers.fr

Fabien Chhel · Frédéric Jouault
ERIS Team, ESEO Group, Angers, France
E-mail: firstname.lastname@eseo.fr

Keywords Model Transformation, Constraint Solving, Model Set Exploration

1 Introduction

Developers have to make many choices when they write a model transformation. Actually, they generally have to make *all* necessary choices to create specific target models. The reason is that classical model transformation techniques typically specify target models extensionally, by assigning specific values to all their elements' properties. They generally do not support intensional specifications, which define sets of possible target models according to constraints that must be satisfied. However, in some cases, all choices cannot be encoded into the transformation. For instance, in decision support systems, such as product line configuration tools, users need to explore the set of available solutions. Scheduling problems [6] where resources (e.g., teachers) must be assigned non-conflicting time slots in specific rooms constitute another example of such cases. Automatically solving such problems is hard, notably because this means that all constraints must be precisely expressed. A tool allowing users to explore the solution space would make it possible to consider a human in the loop.

Let us mention two additional examples closer to the realm of model-driven engineering :

- The first one is forward engineering, in which many different implementations of a design model generally exist. Code generation transformation chains often have to integrate design choices. However they generally do not let developers explore the solution space.
- The second example is model visualization: creating diagrams from models. A model visualizer may be implemented as a transformation from the source model into a view model consisting of graphical shapes. Nevertheless, computing a suitable layout is hard. Although an automatic tool may provide a useful starting point, users often have to improve diagram layout manually. In such cases, transformations implementing model visualization should not compute the whole target model, including full position of every shape. Instead, transformations should specify which diagrams are valid with respect to the source model, and let the users explore the space of possible layouts.

Contributions The approach presented in this paper consists in combining constraint programming techniques [28, 2] with model transformation to address such applications. This combination is ensured by a new mechanism

called *bridge variable*. From a source model, we use a model transformation to create a partial target model in which some properties are not assigned. This transformation also generates constraints that apply to the target model. Instead of specifying a full target model, we can thus specify part of the target model plus constraints on some of its model element properties. Users can then be presented with a target model satisfying these constraints. From there they can explore the set of valid target models. A user may change the target model in a breaking way, making it no longer valid for the given source model. In this case, we use a constraint solver to repair the target model in such a way that it is as close as possible to what the user specified, while still remaining a valid target model. This paper is an extended version of our initial preliminary paper [26]. The main new contribution is an extension of the approach to multiple solvers, each solver being in charge of a specific part of the problem. This makes it possible to leverage solvers with different capabilities. This also implied a change of motivating example to better illustrate this new feature. Another significant change is that the approach has now been integrated in the ATL declarative model transformation language [21], whereas [26] was using an embedded DSL (Domain Specific Language) inside of a general purpose language.

We evaluate our approach by adapting it to a task scheduling case study involving two aspects: schedule validity, and schedule visualization. Our approach allows us to specify these two problems as model transformation plus constraint programming, each problem targeting a different solver. Users can see and modify a diagram representing the computed schedule (i.e., a solution of the underlying combinatorial problem). Constraint solvers oversee the modifications and can repair the schedule if breaking changes are introduced by users.

Outline of the paper The paper is organized as follows. Section 2 presents other approaches related to ours. The notion of model set exploration is presented in Section 3 along with the case study. Section 4 presents our approach to model set exploration using constraints and how properties of elements of the model can be used as decision variables of some constraints. A model transformation based method to build explorable model is proposed in Section 5. In Section 6 an implementation of the approach is presented. Then, the approach is evaluated on the case study in Section 7. Finally, Section 8 outlines possible extensions not explored in this paper, while Section 9 concludes.

2 Related Work

Constraint programming [28, 2] has many possible applications in the context of Model-Driven Engineering (MDE). One of these applications is model generation for testing or verification purposes. The rationale being that generating conforming models is not too hard, but metamodels alone cannot necessarily encode all requirements, which are often specified as additional constraints. Thus it is interesting to generate models that are well-formed according to these constraints. In [17, 16, 39, 38] constraints are used to generate and explore a design space. The authors built their approach on top of the VIATRA2 graph pattern matching mechanism. Constraints are encoded as graph patterns and a set of construction rules is used to build a model satisfying these constraints. Exploring models conforming to an evolving metamodel can help find errors. Approaches like [30] use Alloy [18] to generate such models. In [37], the authors propose an approach to generate explorable design spaces based on Prolog. Also based on Prolog, [41] presents an approach to complete partially defined model. In [40], authors present another approach based on Cartier (which used Alloy) to generate models for model transformation testing.

These approaches focus on generating models that conform to a metamodel and a set of constraints, usually for testing purposes. Our approach also aims at generating models that are valid with respect to a set of constraints. However we differ in two main ways. First, these approaches generate sample models from a metamodel and a set of constraints. Whereas our approach takes a source model, apply a model transformation on it and adds constraints on the resulting model to compute properties that cannot be computed by regular model transformation techniques. Second, our approach is incremental and allows the user to suggests changes on the resulting model that are then processed by a constraint solver. While other approaches only focus on using constraints to generate a model and do try to maintain its correctness when modified.

[35] proposes an extension of QVT-R in order to handle constraints. This extension enables users to write fully declarative bindings between source and target models mixing traditional transformation and constraints. The constraints are used to compute attribute values.

This approach is similar to our approach. Instead of extending QTV-R, our approach extends ATL, which is another well known declarative transformation language. There are three important differences between their approach and our approach. First, in [35] the entire transformation and constraints are encoded as a constraint problem and thus solved entirely by the solver, which can greatly limit the scalability of the approach. However, our

approach uses an hybrid method, part of transformation is executed by a normal model transformation engine and then a constraint solver is used to compute constrained part of the target model. This ensures the constraint problem stays small and manageable. Second, our approach leverages the incrementality of our transformation engine and thus offers incremental constrained transformations. Third, in our approach the user can suggest changes to apply on the target model which are then considered by the solver.

In [22], an approach is proposed to fully integrate constraint programming into existing MDE workflows. This approach is used in [23] to build bidirectional transformations. In [12], authors present another approach using Answer Set Programming to enable bidirectional model transformation.

These works use constraint solver to perform bidirectional model transformation and synchronization between models. This goes beyond the capabilities of our current implementation, which does not support bidirectional transformation. However we support unidirectional synchronization.

In [13], the authors use Kodkod [42], a solver for relational logic, in order to repair models with minimal changes. A detailed overview of model reparation approaches is presented in [31]. A recent work [29] presents an approach based on reinforcement learning to repair models. It uses reinforcement learning to find a good sequence of actions to apply to repair a broken model. The method relies on the user specifying a quality metric for a given model.

These works are related to the repairing process that we present in this paper and show that other approaches such as reinforcement learning can be used as backends for reparation. These other solving backends have different capabilities and may be relevant for specific use cases. These approaches however do not really support user incrementality suggesting changes to the model.

Constraint programming can also be used to verify the correctness of model transformations. An approach is proposed in [8] to verify transformations written in ATL. In [9], one can verify the correctness of UML annotated class diagrams using CSP. In [1], the authors verify the correctness of model transformations by representing them in Alloy.

Some of these works are related to our main goal. Using a similar point of view, our purpose is to define a framework for defining and handling transformations and constraints to represent explorable sets of possible target models. Our approach is focused on human interaction with the model. Users actively explore target model sets by modifying a selected target model, which is then repaired by constraint solvers when necessary. Moreover, in order to reduce latency during user interaction, our approach is incremental. Target models are not only incrementally updated upon source model changes, but the

constraints are also updated in their corresponding solver. Unlike the other works mentioned above, we do not try to check the validity of the process or find a suitable transformation. Moreover, unlike other approaches, our approach tries to reuse existing solvers and create bridges between existing tools.

3 Model Set Exploration: Definitions and Examples

This section introduces the concept of explorable model set along with a motivating example: a task scheduling tool. This tool involves two complementary aspects of model set exploration: 1) exploring possible schedules, and 2) exploring possible visualizations of the schedules. The presence of two complementary model set exploration aspects in the motivating example makes it slightly more complex to understand than a simpler case with only one such aspect. However, such a situation often arises in concrete contexts, and constitutes a characteristic of the problems that we want to address. It is therefore necessary to include both aspects in the example, so as to show how our approach works. Section 3.1 gives some definitions. Then, the task scheduling problem is presented in Section 3.2. Finally, diagram visualization is introduced in Section 3.3 before being applied to task scheduling in Section 3.4.

3.1 Definitions

Firstly, let us recall the definition of *model space* presented in [14]. A *model space* is a directed graph $M = (M_\bullet, M_\Delta)$ where $M_\bullet$ is a set of nodes called models and M_Δ is a set of arcs, called deltas or updates, between models. We call *exploration of a model space* the application of deltas on models belonging to $M_\bullet$. The user only sees one model from $M_\bullet$ at any given time. To start exploring, the user makes a change to the model, which corresponds to following an arc from M_Δ.

Secondly, only some models in a model space are typically of interest. To be valid, a model must conform to its metamodel, but this is usually not enough. There are often *rules*, also called *constraints*, that dictate which models are valid. A valid model therefore needs to both conform to its metamodel and satisfy this set of rules, which filter out invalid models. For instance, when drawing diagrams made of simple shapes, a geometric metamodel is not sufficient. There are rules to follow in order to correctly place shapes so that they form a valid diagram. In order to take correctness of models into account, we introduce the concept of *model set*. A *model set* is a subset M_V of

$M_\bullet$ only containing valid models. As defined above, exploring a model space involves following deltas in M_Δ to move from a model in $M_\bullet$ to another one. *Exploring a model set* follows the same principle but the first and last models of a sequence of deltas have to be in subset M_V (i.e., they have to be valid).

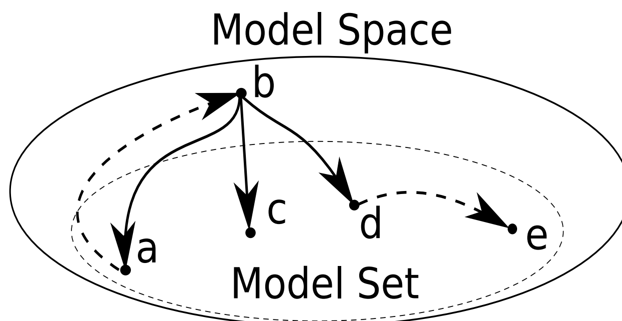

Fig. 1: Exploring some solutions

Finally, a mechanism is needed to complete any sequence of deltas so that it reaches a valid model. We call this mechanism *model repairing*. Figure 1 gives an overview of the situation. The ellipse with the plain outline represents the whole model space, whereas the one with the dashed outline represents the model set. Each dot represents a model, and arrows represent deltas. Dashed arrows represent deltas applied by a user, while plain ones represent possible ways to repair the model into a valid one. After a user update, the current model may be in one of two possible states:

1. **Valid:** the user modified the model and all rules are satisfied (e.g., model `e` in Figure 1). Since the model is already valid, no further action is required.
2. **Invalid:** the user applied a delta that results in a model outside of the model set (e.g., model `b` in Figure 1). The model still conforms to its metamodel but does not satisfy all the associated rules. It is therefore necessary to repair the model by following a repairing delta (i.e., a plain arrow on Figure 1). The first possibility, which is always available, is to block the user change and roll back (e.g., to model `a`). Other possibilities are to follow deltas to other valid models (e.g., models `c` and `d`).

When repairing models, multiple sequences of deltas leading to valid models typically exist but only one is chosen. Consequently, model repairing is a kind of search problem. We call *behavior* the strategy that is used to select which sequence will be used to repair models. Useful strategies will typically avoid systematic rollback, and rather reach a model that is the closest (according to some definition of closeness) to the model reached by the user.

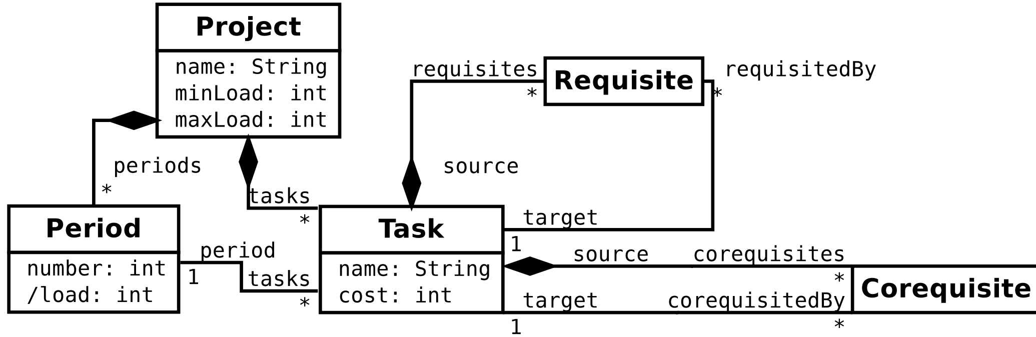

Fig. 2: Scheduling metamodel

3.2 The Task Scheduling Problem

The previous section introduced the notion of model exploration. This section presents a first example application: task scheduling. This is indeed a basic version of scheduling problems [6]. The overall idea is to assign tasks to periods so that specific domain constraints are satisfied. This problem notably arises in several software engineering activities, such as when scheduling development effort, or when planning runtime executions. For the sake of understandability, we simplified this example by removing unnecessary concepts.

The problem is modeled by the metamodel given in Figure 2. It consists of a named `Project`, which contains `Period`s and named `Task`s. Each `Task` has a `cost`, a list `requisites` of `Requisite`s, and a list `corequisites` of `Corequisite`s. The `Corequisite` and `Requisite` dependencies both have a `source` and `target` `Task`. The target `Task` of a `Requisite` (resp. `Corequisite`) must be in a `Period` that comes after (resp. or be the same as) the `Period` of its source. Each `Period` has a `number` used to specify the order in which `Period`s take place in time: all `Period`s with a lower `number` than the one of a given `Period` happen before it. Each `Period` also has a derived property called `load`, which corresponds to the sum of the costs of the `Task`s it contains. Each `Project` has a `name`, an upper-bound (`maxLoad`), and a lower-bound (`minLoad`) corresponding to minimum and maximum `load` allowed for each `Period` it contains.

While simple, this metamodel is complex enough to make it possible to introduce interesting constraints. A *schedule* is an assignment of all `Task`s of a `Project` to `Period`s of the said `Project`. There are many possible assignments, of which many are not relevant, such as assigning all `Task`s to the first `Period`. Constraints can be used to restrict the set of possible assignments to the ones that are relevant. For instance, the `load` of a `Period`

should be between the `minLoad` and `maxLoad` of its `Project`. Valid schedules not only have to conform to their metamodel, but also verify a set of constraints specific to this case study. Based on the task scheduling metamodel we can define several rules that can be used to select valid assignments from all possible ones:

Rule 1: Appropriate Load. The `load` of each `Period` of a `Project` must stay between the `minLoad` and `maxLoad` of said `Project`. Or, written in OCL:

```
context Project inv: self.periods->forAll(p | p.load >= self.minLoad &&
                                              p.load <= self.maxLoad)
```

Rule 2: Precedence of `Tasks`. All requisites of a `Task` must be assigned to earlier `Periods`. Each corequisite of a `Task` must be assigned to the same `Period` or an earlier one. Or, written as an OCL invariant:

```
context Task inv:
        self.requisites->forAll(t | self.period.number > t.period.number) &&
        self.corequisites->forAll(t | self.period.number == t.period.number)
```

The assignment of `Tasks` to `Periods` is important. Thus, finding a solution consists of modifications to the `Period-Task` relation instead of property modifications. After an initial schedule is found, alternative solutions can be explored by following delta arcs. Each time a delta is applied, if the resulting schedule is invalid, a repair must be computed and applied.

3.3 Diagrams as a Model Set Exploration Problem

In previous section, we have presented an abstract task scheduling problem as a model exploration problem without connection to any user interface. In order to be able to fully demonstrate the approach presented in this paper, we need to complement this problem with another model exploration aspect. For this purpose, we consider the integration of the task scheduling problem into a visual scheduling tool. Before looking into the specific details of task scheduling visualization, the present section gives an overview of how a diagramming tool can be modeled as a model exploration problem.

Diagrams consist of simple geometric shapes. For the sake of simplicity, we only consider rectangles, lines, arrows, and texts in this paper. We therefore consider models conforming to the metamodel given in Figure 3. This is a simplified view metamodel inspired by JavaFX[1]. Like the actual JavaFX API, it contains simple graphical shapes that can be displayed on a canvas.

[1] JavaFX is a graphical Java framework, which API documentation can be accessed at `https://docs.oracle.com/javase/8/javafx/api/`

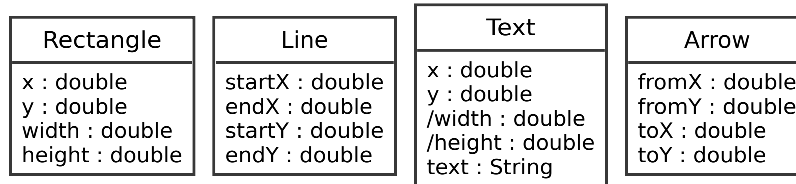

Fig. 3: Simplified diagram metamodel

`Rectangle`s and `Text`s have `x` and `y` coordinates along with a `width` and a `height`. Each `Text` additionally has a `text` attribute, which contains the `String` value to be written. Moreover, the `width` and `height` properties of a `Text` are derived from the contents of its `text` and additional information not specified in this simplified metamodel (e.g., specific font, font size). `Line`s have coordinates for their starting point (`startX`, `startY`) as well as for their ending point (`endX`, `endY`). Finally, `Arrow`s follow a similar pattern to `Line`s, with `startX` (resp. `startY`) renamed into `fromX` (resp. `fromY`), and `endX` (resp. `endY`) renamed into `toX` (resp. `toY`). This metamodel is independent of any specific domain that needs to be represented, and only defines shapes that may be used to represent domain concepts.

A list of geometric shapes is not enough to specify valid diagrams. A set of rules is needed to define how shapes can be positioned with respect to each other. For instance, the shapes in Figure 4 do not form a meaningful scheduling diagram. Texts should be contained in the rectangles, and each arrow should connect two rectangles. Although all shapes are present, they lack a proper positioning for the diagram to be valid.

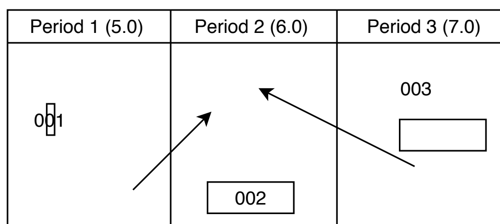

Fig. 4: A diagram conforming to metamodel Figure 3

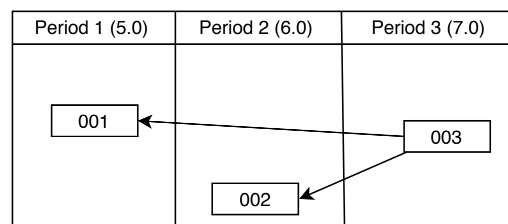

Fig. 5: Valid diagram example

In order to be valid, a diagram not only needs to conform to its metamodel of geometric shapes, but it also needs to follow specific rules imposed by its specific type. These rules are specific to each diagram type, and they are part of its definition. For instance, we consider the diagram in Figure 5 to be valid, but not the diagram in Figure 4. Making the set of valid diagrams

explorable is as simple as letting users move shapes around and then repairing the visualization when an invalid position is reached.

Because this problem is mostly geometric, we only consider the modification of numerical properties (i.e., shape coordinates) of the model among the many possible interactions. An example would be changing the `y` coordinate of a rectangle. This excludes interactions that results in deltas that modify the structure of the model (e.g., adding or removing elements). Likewise, we only consider reparations that modify these numerical properties. Several behaviors may appear, depending on how the diagram model is repaired. For instance, in Figure 5, if we allow the user to move the label inside its rectangle, then we also need to define what should happen if the user tries to move the label outside. The corresponding behavior must specify whether the attempted move should be ignored, or whether the rectangle should move, or grow in order to keep the text within its bounds.

3.4 Visualization of the Task Scheduling Problem

This section applies the concepts presented in the previous one to the task scheduling tool. A screenshot of the visual task scheduling tool is given in Figure 6. We consider a simple planning with 3 `Task`s named 001, 002, and 003. `Task` 003 requires 001 and 002 to be completed before starting. These `Task`s can be assigned to three `Period`s named P1, P2, and P3. In Figure 5, 001 is assigned to P1, 002 to P2 and 003 to P3.

Section 3.2 presents a list of constraints specifying what a *valid* assignment of `Task`s to `Period`s is. Once such a valid assignment has been computed, it can be displayed to the user as a diagram. In the above description of the scheduling diagram, there is no position but only containment information (e.g., P1 contains 001). Using this information, it is possible to write rules to correctly position corresponding shapes. For instance:

Rule 1: Text Placement The top left point of each `Text` must coincide with the top left point of its corresponding rectangle.

Rule 2: Containing Period Every `Rectangle` corresponding to a contained task must remain inside the rectangle corresponding to its `Period`.

These geometric rules ensure that the diagram remains meaningful whatever the position of the various shapes. The tool can let users move shapes around. It can then leverage the rules to repair the model or diagram if a user performs a move that would break it. For instance, the user may move a `Task` rectangle contained in a `Period` rectangle beyond the boundary of the latter. Then the tool can either move the `Task` rectangle to another `Period` rectangle or forbid

the move. Moving the `Task` to another `Period` corresponds to changing the containing `Period` of the `Task`. Each one of these two actions would make Rule 2 satisfied again.

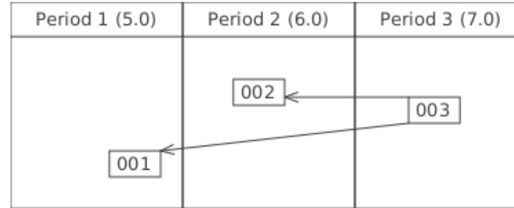

Fig. 6: Screenshot of the final result

We conclude the presentation of the motivating example, which involves two model exploration aspects. The search problem corresponding to the first one, presented in Section 3.2, involves finding valid assignments for a relation. The second one, which has just been presented, involves finding valid assignments for numerical properties. Other kinds of search problems beyond relations and numerical properties can be used in rules but this goes beyond the scope of this paper.

4 An Approach for Model Set Exploration

As presented in Section 3, we focus our approach on providing to users the ability to explore sets of models. Such an exploration enables the user to improve a model by making small changes to it. However, these changes can sometimes make the model invalid. In this case, the model needs to be repaired and this can be done by a constraint solver. For instance, let us consider the following rule applied to a rectangle r: `r.width >= 100`. There is an infinite number of valid solutions for this width, as long as it is greater than 100. In order to make these models explorable we propose an approach that links properties of the model with decision variables, which are handled by constraint solvers.

4.1 Overview

Model sets may be arbitrarily large. Therefore, we need a mechanism to define them intensionally. We propose to use constraint programming and to add constraints to the model element properties. Constraint programming includes generally declarative modeling languages that allow the users to describe problems by means of a constraint satisfaction problem in a declarative fashion instead of implementing a solving process. More formally, a CSP

(Constraint Satisfaction Problem) is a triple $P = (X, D, C)$ where X is the set of decision variables, D is the set of the domains of these variables (these domains may be different for each variable), and C is the set of constraints. Given a set of decision variables $x_i \in X$ and their associated domains of possible values $D_i \in D$, a constraint is a k-ary relation between k variables in X. A solution of P is an assignment $s : X \rightarrow \bigcup_i D_i$ that satisfies all the constraints.

In our approach, models are extended using constraints. These constraints express relations over model element properties and act as the rules described in Section 3. A CSP solver ensures that the models remain consistent and belong to the model set. In order to bind model properties to decision variables we introduce the concept of bridge variable.

4.2 Bridge Variable

A bridge variable binds a property of a model element to a decision variable of a solver. Each property of every model element used in the constraints corresponds to one and only one bridge variable. The same is also true between bridge variables and decision variables in the solver. The bridge variable keeps both its decision variable and property synchronized. That is to say, whenever one of the two is modified, the bridge variable forwards the change to the other. Figure 7 details how modifications are forwarded with a solver that automatically computes a new solution. After the user interacted with the model and one of its properties has been updated[2], the corresponding bridge variable is notified of the update. This update is forwarded to the constraint solver, which then updates the current solution based on the new value. How values are computed by the solver corresponds to the notion of model repairing we presented in Section 3.1. Defining the repair strategy is done through careful constraints declarations, for instance with constraint priorities. After this update, the solver notifies each bridge variable that corresponds to a decision variable that has been changed. The bridge variables then forward the updates to their corresponding model properties. This automatic solving behavior may not always be desired (for instance when modifying coupled variables, such as the x and y coordinates of a point). Considering only one variable at a time can lead to transitory inconsistencies and trigger a repair whereas the model would have been valid with all modifications propagated. Thus this needs to be configurable depending of the situation. Because of the nature of constraint programming, a user update on one property can lead

[2] We consider atomic updates. If the interaction modified several properties then they are treated as a sequence of updates.

to several other updates. Variables have names based on the names of the corresponding `Task` or `Period` that they represent. `Rectangle`s are named by appending `.r` to the element name (e.g., P1.r for P1), `Text`s are named by appending `.t` (e.g., P1.t for P1), and so on for `Line`s and `Arrow`s. For instance, in Figure 6, moving the `Rectangle` corresponding to `Task` 001 to the left only updates property `001.r.x` but the solver also needs to update `001.t.x` to keep the diagram valid.

The bridge variable is somewhat similar to the concept of constraint reification that is common in constraint programming [4] but at a different level. With constraint reification it is possible to handle constraints as Boolean decisions variables. However, in our approach, bridge variables do not result from the truth values of the constraints but from properties available outside of the solver.

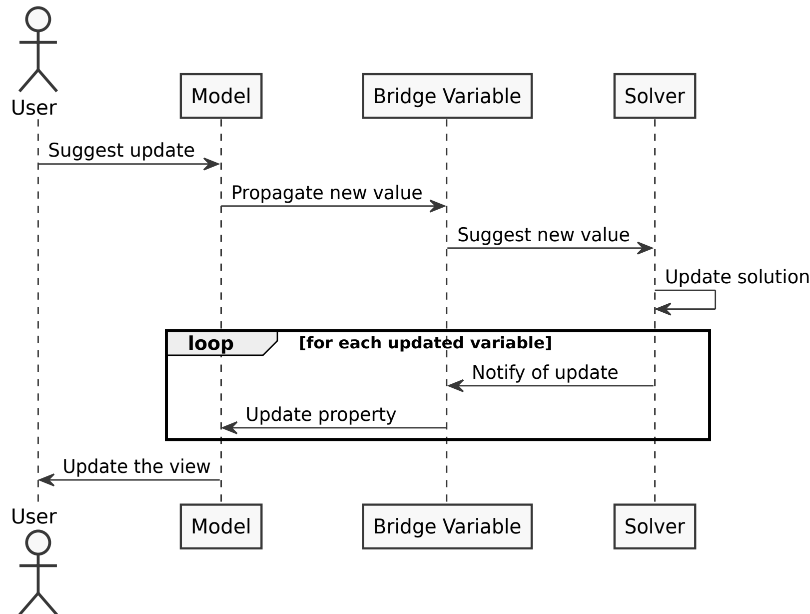

Fig. 7: Propagation of change across bridge variable

In some specific cases it is not desirable to assign a decision variable to a property. Such a case may happen when dealing with read only properties such as derived properties. For instance, the `width` property of a `Text` is read only: it is derived from the length of its `text` and the size of the used font. In other cases, one may simply not want the solver to be allowed to modify a specific property of a model element. It is thus necessary to prevent the solver from updating the corresponding decision variable. There are two main solutions for handling this problem. The first one would be to use a

decision variable but restrict its domain to a single value. The second one would be to represent the model element property using a constant in the solver, rather than as a decision variable. Both solutions work but are not equivalent. Depending on the solver, one may prefer one to the other.

This simple binding works well for numerical values or more broadly types of property that are supported by the chosen constraint solvers. However we express constraints over models and not all property types can be mapped to supported constraint solver types. One important type of property that is often not directly supported by solvers is relations (or associations). Many approaches exist to map relations to numerical values. For instance, it is possible to give objects unique ids and to use predicates to encode the existence of a relation between two objects [41]. Another approach is to directly use graph pattern matching to handle relations [38, 16]. In our case we decided to use an approach similar to [41] but adapted it for CSP solvers where there are no predicates.

Let us consider a constraint stating that `Task` 001 is assigned to one of the three `Period`s would looks like:

$$\forall i \in \{1,2,3\} \quad 001.period = P_i \implies var_i$$

$$\sum_{i \in \{1,2,3\}} var_i = 1$$

With $001.period$ being the variable encoding which `Period` the `Task` 001 is assigned to, P_i being the id of each `Period` and var_i being an intermediate variable. This uses constraint reification to check if the `Task` is assigned to a specific `Period`. It can be expressed in several different ways, for instance by using a Boolean variable for each possible relation. This encoding only works for one-to-one or one-to-many relations, which is the only kind of relation we considered so far. To handle all kind of relations, an encoding closer to the one defined in [41] would be necessary.

4.3 Constraints Management

Bridge variables link properties of model elements to decision variables in the solver. We now need a complete language or metamodel to express constraints that use these bridge variables. There are many generic constraint languages, such as Minizinc [34], but they lack features to efficiently represent bridge variables and integrate into the MDE ecosystem. They are not thus suitable as pivot languages in our approach. Similar remarks hold for solvers such as Alloy [18] or Choco [36] that respectively have a dedicated language, and an API to interact with. To be able to export/express our constraints to

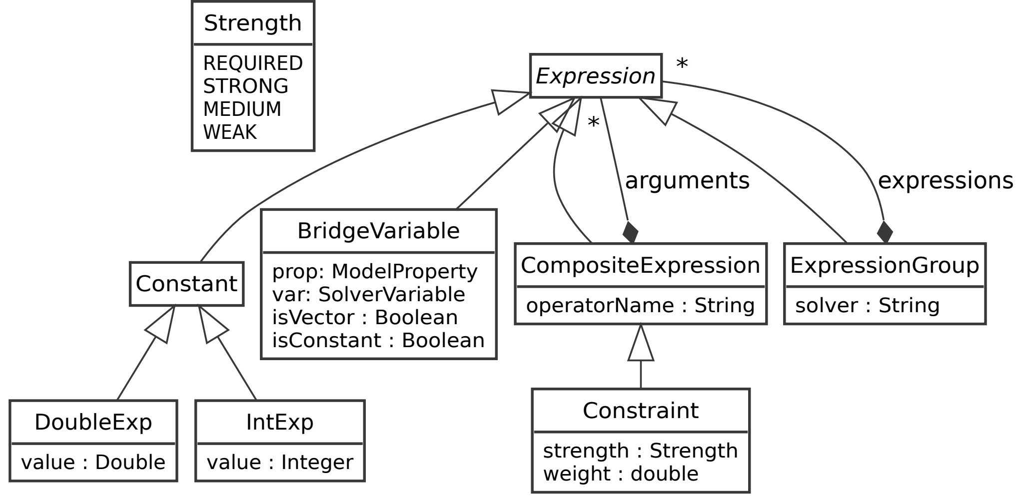

Fig. 8: Metamodel for Constraints

various solvers with different features and expressiveness, we introduce a pivot constraint metamodel to ease the transformation into specific solver languages.

Our proposed constraints metamodel is built around the notion of bridge variable. A simplified version is shown in Figure 8. Because the `BridgeVariable` is responsible for maintaining a model element property and its corresponding solver decision variable synchronized, it needs to keep a reference to both of them. To stay as generic as possible, `CompositeExpression`s may have any number of arguments and use a name to identify which operator is used. `Constraint`s are `CompositeExpression` with a `strength` and a `weight`. Each operator has a specific number of arguments and semantics attached to its name. Concrete sets of predicates depend on the application. For instance, for our case study, there could be geometric predicates like `contains` or `above`.

To be processed by a solver, this intermediate model needs to be transformed into solver-specific constraints. Transforming this pivot model into a solver-specific language may involve constraint rewriting if the target solver does not support specific predicates or operators used in the pivot constraints. For instance, we may rewrite geometric constraints into algebraic ones or encode relations with the method described in Section 4.2. Using such a pivot metamodel adds a layer of abstraction that eases support of new solvers.

Figure 8 also shows that each constraint can have a strength and a weight. These can be used with solvers that support hierarchical constraints [44, 32]. They can for instance help specify behaviors. Hierarchical constraints

can easily be emulated using classical constraints and a carefully crafted objective function. Finally, `BridgeVariable` has a property `isVector` that indicates if the source property is a single value or a collection. We call `VariableVector` a `BridgeVariable` that references a property that is a collection. `VariableVector` may be used to abstract a set of similar constraints into a single one. It regroups multiple variables into a single vector, which can be used as a regular variable. Thus, constraints with `VariableVector`s can result in multiple constraints once translated to a specific solver. `ExpressionGroups` are used to gather constraints into a single model element. Their use is detailed in Section 6.1.

Having an intermediary metamodel for constraints makes it possible to define this series of rewrittings as a series of relatively simple model transformations that are applied one after the other. For instance, flattening constraints that contain `VariableVector`s into a list of constraints that only contain simple variables is one of these transformations. Having these steps defined as solver agnostic transformations helps reusing them when the same intermediary transformations are necessary for different solvers. For instance, all solvers of our current implementation use the flattening step presented above.

We saw earlier that `BridgeVariable`s are used to monitor changes made on properties so that these changes can be propagated to a solver. This works great for solvers that have the following features:

Feature 1: Adding variables and constraints. Suggesting a new value for a variable corresponds to the addition of a new constraint. It also adds a new variable used to save the difference between the suggested value and the actual value.

Feature 2: Removing variables and constraints. After a suggestion has been posted, it may be necessary to remove constraints that have been added in order to suggest a new value. Otherwise it would not be possible to undo suggestions.

Feature 3: Preventing variables from changing too much. Solvers do not necessary ensure that solving the same problem twice will result in the same solution, or a close one. This is especially true when the problem itself is modified. Thus, it may be necessary to specify that some variables are assigned to values that should stay stable across multiple solving attempts. This corresponds to the `stay` constraint in incremental solvers such as Cassowary [3].

Feature 4: Minimizing/maximizing a variable. Once a value has been suggested the problem is no longer a satisfaction problem rather becomes an

optimization problem. The solver then has to find a solution that minimizes the distance between all variables and their suggested values.

However, not all solvers present these features. With some solvers it may be necessary to compute a new solution from scratch every time an update is posted, which can be costly. Thus it may be preferable to disable auto-solving and only solve a problem when the user asks for a new solution. However, this behavior alone is not sufficient.

Without additional information the solver is free to choose any solution, it does not care about previous ones. Therefore, it is necessary to change the constraint problem so that the satisfaction problem becomes an optimization problem. A system similar to Cassowary can be implemented on top of solvers that do not have any concept of incrementality. Each time a new solution is asked, additional constraints are added so that the new solution stays *close* to the old one. The idea is that two solutions are *close* if the differences between the values of their variables are small. With hierarchical constraints, it is also possible to specify that some variables are allowed to change more than others. Finding a new solution that is *close* to another one is an optimization problem where the solver tries to minimize the (possibly weighted) sum of differences between the old and new values. When implementing such a system on top of solvers that do not support dynamic addition of removal of constraints, it is necessary to re-serialize the problem each time a new solution is asked, thus emulating an incremental behavior on a non incremental solver.

4.4 Solver Collaboration

As mentioned earlier, each solver has its own set of features depending on its method of resolution. For instance, Cassowary is an incremental solver built to efficiently solve changing linear constraints on floating point values. On the other hand, Choco [36] is able to handle non linear constraints over integers but cannot efficiently handle changing problems. Thus it is interesting to make different solvers collaborate so that each solver can be used to solve problems it is built for. Also, giving each solver a smaller problem is beneficial in itself. CSP is an NP-complete problem, thus solving several smaller problems can be faster than solving a single big one. Making solver collaborate has been discussed for a long time [5,33], and strategies have emerged to make this collaboration more efficient.

However, when considering independent problems, these methods are not necessary. Two problems are said to be independent from each other if they do not share decision variables, or in our case model element properties. If

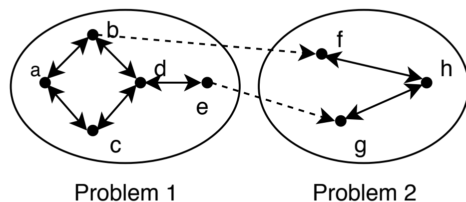

Fig. 9: Problems with shared variables

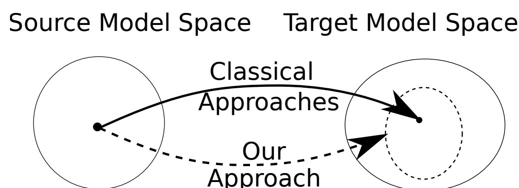

Fig. 10: Diagram of the model to model set approach

two problems are independent, then an update on one problem cannot have any impact on the second problem. In this ideal case, each solver can act on its problem without any sort of coordination.

In our case study we have two sub-problems. One consists of business constraints to find a valid schedule, while the other consists of graphical constraints to properly display said schedule. These two problems are almost independent. They share only few variables, those that encode which `Period` each `Task` is assigned to. In such a case, a mechanism is needed to ensure these shared variables are handled correctly.

One simple strategy is to give the right to update shared variables to only one solver, when this is possible. Then, other solvers can only access these variables as read-only values (i.e., constants). To work correctly, this strategy requires that all shared variables between two problems are writable by the same solver. This mechanism of read-write and read-only variables can be represented as a directed graph. Variables are the nodes, and edges represent constraints. An example with two non independent problems is shown in Figure 9. In this example, variables of problem 1 are accessed without restriction (plain double arrow). Variables f, g, h in problem 2 are accessed without restriction whereas variables b and e are accessed as read-only values (simple dashed arrow). This example works correctly because all arrows between problems 1 and 2 go in the same direction. In the general case, with more than two problems, this strategy works as long as there is no cycle between problems. With this representation it is possible to easily find an order in which solvers can be executed by using a topological sort. A similar approach to the one used in [25] could be used to determine better orderings.

Figure 11 shows a variant of Figure 7 adapted to a scenario with two solvers. Solver 1 has write access to shared variables, thus it is notified of changes first. Then, the new found solution is propagated to Solver 2 as read-only values. Solver 2 can then compute a new solution.

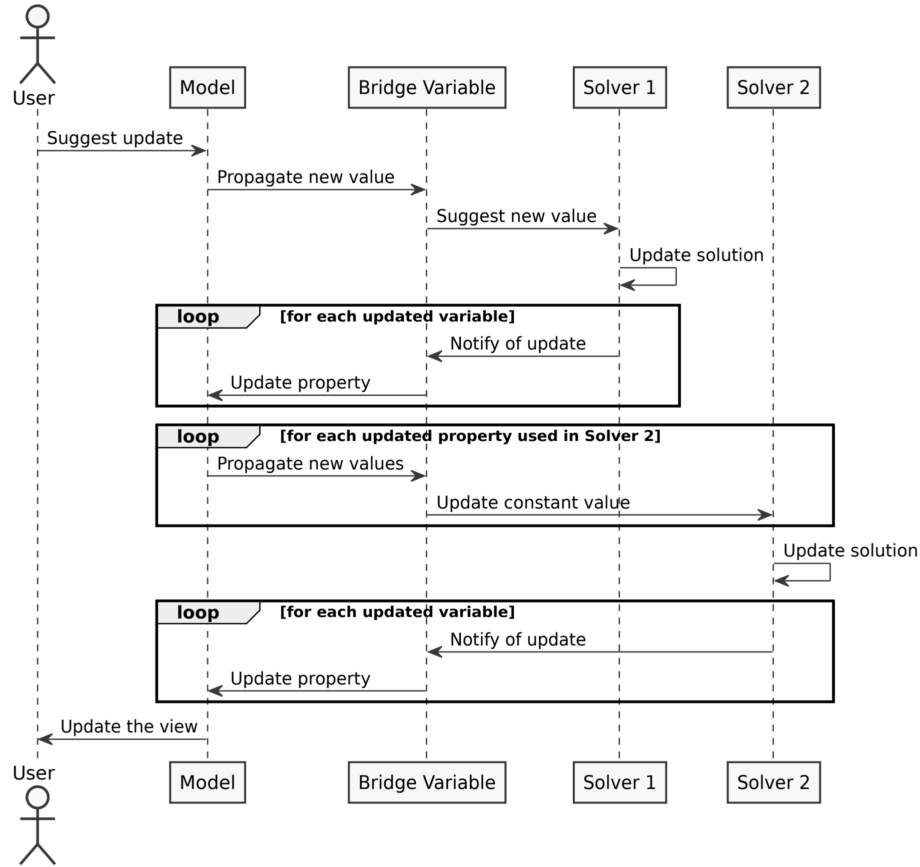

Fig. 11: Sequence diagram of interactions with multiple solvers

5 Generation of Explorable Model Sets

In Section 4 we presented our approach to explore model sets based on constraints. Theses constraints use bridge variables to forward updates from the model to the solver and back. We need a convenient way to setup constraints while building the target model. In Section 5.1 we describe how this can be achieved. In Section 5.2 we give a few necessary requirements for such a tool.

5.1 Overview

Classical model transformation approaches generally output a single target model from a single source model. However, there may be many correct target models. Figure 10 summarizes the differences between classical approaches and our approach. The two ellipses with solid outlines correspond to the source and target model spaces. The dots represent models in the

corresponding $M_\bullet$ sets. The ellipse with a dashed outline represents the target model set that is generated by our approach. With our approach we can provide an explorable target model set by using constraints to complement a model. Model transformation can, from a source model, generate both the target model and the constraints. The idea is depicted in Figure 12. Both the target and constraint models are generated by the transformation from a source model. In our context, the target model is displayed by JavaFX but this is not the case for other kinds of target models. Constraint, target, and source models are synchronized in order to ensure changes are properly propagated. This synchronization structure is automatically setup by a transformation between models, and bridge variables between models and solvers. Then the constraints are dispatched to their target solver wrapper. There they are rewritten by a series of transformations in order to adapt them to the specific solver used.

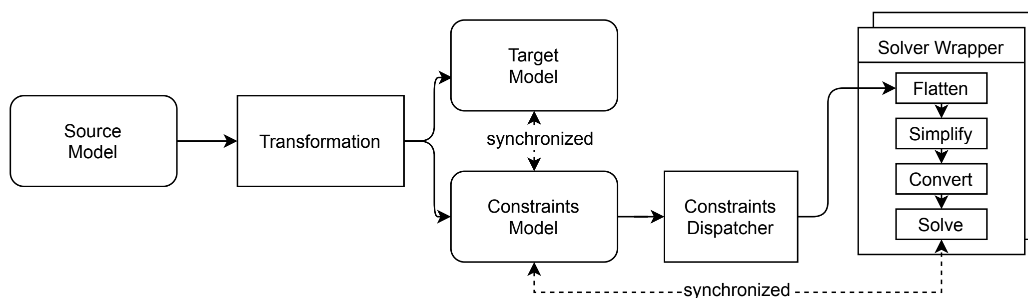

Fig. 12: Full diagram of the approach

As explained in previous sections, in our approach, constraints are used to specify a model set intentionally. By adding constraints to the generated transformation, the target model structure along with its associated constraints are generated in a consistent way. In a single unified step, the developer can write a transformation that describes how to create a target model set from a source model. This description also precises which properties can be modified and how they relate to each other.

5.2 Requirements

From the various properties discussed earlier in this paper we derive the following four requirements.

Requirement 1: Declarative Rule-based. Constraint programming is a declarative way to specify relations between variables. It makes sense to also

opt for a declarative transformation language. Several rule-based transformation approaches [43, 27, 19] already exist and have proven to be relevant. Moreover, rules are an abstraction that encapsulates source and target elements along with how they relate. They can relatively easily be extended to also encapsulate constraints.

Requirement 2: Specifying Constraints. Any implementation of the approach must provide means to specify which constraints apply to target element properties. In our case study, the constraints needed for the visualization part are geometric by nature (see Constraints 1 and 2 from Section 3.4). Like many similar frameworks, JavaFX uses a Cartesian coordinate system with double precision floating point values. Therefore, many fine-grained geometric constraints, like specifying minimum lengths, are actually arithmetic constraints on doubles. The front end must consequently provide means to express geometric and arithmetic constraints.

Requirement 3: Prioritizing Constraints. In order to capture complex behaviors, there must be a mechanism to specify that some constraints may be violated. Moreover, when either one of two constraints must be violated, it must be possible to specify which one should be violated first. This is a key mechanism to be able to express fine-grained behaviors.

Requirement 4: Incremental Transformation. By incrementality we mean that if the source model changes, the target should be updated, along with the set of constraints that apply to it. This should happen without having to recompute the solutions from scratch to be more reactive. This requirement makes it possible to visualize changing models in real time. Thanks to incrementality it is possible to use the transformation to rewrite parts of the model that the solver would not be able to deal with.

Given these requirements, and the fact that our team has been working on an incremental declarative ATL engine, we decided to extend ATL to support constraint specification and prioritizing. The incremental ATL engine, ATOL, is presented in [10]. The next section gives an overview of the current implementation of the approach.

Remark: although guided by our choice of case study, some of these requirements would likely apply to other implementations of the approach, but this is beyond the scope of this paper.

6 Implementation Overview

In this section, we detail how we leveraged ATOL's capabilities and extensibility in order to implement the approach presented in this paper. All

tools used in this paper are available at `https://github.com/ESEO-Tech/ATL-Tools-Library` and are licensed under EPL-2.0 license.

ATOL is an ATL compiler that generates Java code that uses AOF (**A**ctive **O**perations **F**ramework, a Java implementation of Active Operations [19]), thus enabling incremental execution of ATL transformations. Active operations constitute a set of operations that can be applied on collections (ordered or not, such as singleton, set or list) to build complex incremental expressions. There are such incremental variants of well-known OCL operations such as: `select`, `collect`, `concat`, or `size`. Each of these operations has one or more inputs, as well as an output, and is equipped with propagation algorithms. These algorithms make it possible for the operation to react to changes applied on its input, by computing consistent modifications of its output. After computing the changes to perform on the output, it notifies the other operations which use it as input. Thus a modification of an input can be propagated from operation to operation until all intermediary elements of the incremental expression are updated.

ATOL compiles ATL bindings into AOF expressions, so that changes on source properties are propagated to target properties. Right now, ATOL is restricted to a subset of ATL, which is sufficient to demonstrate that active operations can be used as a back-end for ATL transformations. We currently only support *unique lazy rules*, and not classical ATL rules, because the current implementation of the incremental rule matcher is limited. ATL transformations are compiled into Java classes with one method for each ATL rule. Users can then directly call specific rules on specific source elements.

While not fully compatible with ATL, ATOL has a plugin mechanism that can be used to implement extensions. More precisely, each plugin can match bindings in order to compile them differently. These bindings are not compiled by ATOL itself, but are directly forwarded to the plugin instead. The plugin is then in charge of generating the corresponding Java code. This process will be described more precisely in Section 6.2.

6.1 Structure of a Constrained Transformation Rule

Including constraints in classical ATL files is not directly possible. However, the plugin mechanism described above makes it possible to target specific bindings and compile them differently from the rest of the transformation. The generation of constraints can be achieved as described in Section 4.3.

An example ATL rule containing constraints is given in Listing 1. This rule is quite similar to a classical ATL rule. It includes the same `from` (l. 2-3) and `to` (l. 4-14) sections. Classical target element declarations are not

```
rule ExampleRule {
    from
        s : SourceMetaModel!SourceClass
    to
        t: TargetMetaModel!TargetModel (
            exampleProperty <- 'something'
        ),
        constraints: Constraints!ExpressionGroup (
            solver <- 'choco',
            expressions <- Sequence {
                t.x + t.y < 42,
                t.x.stay('MEDIUM')
            }
        )
}
```

Listing 1: Example ATL rule containing constraints

modified (l. 5-7). Even our constraint target element uses the classical ATL syntax (l. 8-14). For instance, the `solver` property of `ExpressionGroup` is set to `choco` (l. 9).

6.2 Compiling a Transformation

The constraints plugin of ATOL intercepts only the compilation of bindings targeting the `expressions` property of `ExpressionGroups`. Instead of compiling the content of the `Sequence` into classical ATL, this plugin compiles it to Java code that instantiates the corresponding constraints metamodel.

Constraints written in this sequence can use specific operations that are not supported by ATL (and only a limited subset of ATL is supported here). For instance, at line 12, the `stay` operation is a constraint that does not correspond to any ATL operation.

Part of the generated code for the constraint `t.x + t.y < 42` from Listing 1 at line 11 is shown in Listing 2. The code is indented and variables have been renamed in order to help the reader. As we can see, the code includes model elements that correspond to the constraint written in the ATL transformation. For instance, l. 7-11 (resp. l. 14-18) correspond to the instantiation of variable `x` (resp. `y`) and l. 23-24 to the creation of the constant 42. Each reference to a property of the target model in a constraint is compiled into a `BridgeVariable`. Hence, there is nothing special to write in the ATL code in order to indicate that a property is a variable. However, it is possible to use the `toConstant()` operation to indicate that a property should be considered as a constant. It would then be updated upon model changes, but the solver would not be allowed to change it.

```
1 Constraint cstr_inf = Constraints.Constraint.newInstance();
2 cstr_inf.setOperatorName("<");
3
4     CompositeExp operation_plus = Constraints.CompositeExp.newInstance();
5     operation_plus.setOperatorName("+");
6
7         BridgeVariable var_x = Constraints.BridgeVariable.newInstance();
8         var_x.setIsVector(false);
9         var_x.setIsConstant(false);
10        var_x.setSource(t);
11        var_x.setPropertyName("x");
12    operation_plus.getArguments().add(var_x);
13
14        BridgeVariable var_y = Constraints.BridgeVariable.newInstance();
15        var_y.setIsVector(false);
16        var_y.setIsConstant(false);
17        var_y.setSource(t);
18        var_y.setPropertyName("y");
19    operation_plus.getArguments().add(var_y);
20
21 cstr_inf.getArguments().add(operation_plus);
22
23    IntExp cst_42 = Constraints.IntExp.newInstance();
24    cst_42.setValue(42);
25 cstr_inf.getArguments().add(cst_42);
```

Listing 2: Part of the generated Java code corresponding to line 11 of Listing 1

Then these variables and constants are used as operands for the various operators (l. 12 and l. 19) and constraints (l. 21 and l. 25). Finally, it is possible to specify the strength of a constraint by adding a comment containing the desired strength level (*weak*, *medium*, *strong*, or *required*). The default strength of a constraint is *required*. The strength levels are used to derive the priority of the constraints in the problem. This concept of strength level is directly inherited from Cassowary.

6.3 Processing the Resulting Constraints

At runtime, this code generates a constraint for each source element that was matched by the rule. We call this resulting constraint the *constraint model*. However, no solver can directly use this constraint model. Each solver has its own metamodel. We call these solver-specific constraints *concrete constraints* in order to distinguish them from the (abstract) *constraint model*. Several transformations are necessary to generate concrete constraints from the constraint model. These transformations correspond to the right-most part of Figure 12. At this time, our tool supports several solvers such as Cassowary [3], Choco [36], MiniCP [24], and XCSP3 [7].

The corresponding transformations are written in Xtend, and directly use AOF in order to be incremental. New constraints added to the constraint model generated by the ATL transformation are automatically fed to this sequence of incremental transformations, resulting in new concrete constraints that are added to the solver. Similarly, when constraints are removed from the constraint model, the corresponding concrete constraints are removed from the solver. This ensures that the constraint model is synchronized with the concrete constraints, and that they represent the same constraint problem. Solutions found by the solver can thus be translated to solutions in the constraint model. We are currently working on replacing these hand-written Xtend transformations by ATL transformations. However the ATOL compiler does not, at this time, include all necessary feature to efficiently write these transformations.

Each of these transformations performs a different task depending on the target solver. For instance, with the constraint metamodel presented in Figure 8 (Section 4.3), it is possible to nest an `ExpressionGroup` inside another `ExpressionGroup`. This allows for easier processing of the constraints generated by the transformation. However, these `ExpressionGroup`s need to be flattened so that the result is only composed of `CompositeExpressions`.

Another example is the simplification transformation shown in Figure 12. `BridgeVariable`s can reference properties that are collections. For instance, let us consider that properties `x` and `y` of the `SourceClass` used in Listing 1 are both lists of values and that we have an instance `s` of this class with `s.x = [a,b,c]` and `s.y = [4,5]` , where `[1,2,3]` denotes an ordered list containing elements 1, 2 and 3 in that order. Then the corresponding constraint would be $[a, b, c] + [4, 5] < 42$, which is not directly usable by most solvers. This constraint needs to be *simplified*. Simplifying a constraint expands the single constraint containing `VariableVector`s into several constraints containing only simple values.

Depending on how the simplification is performed, this constraint could be simplified into the six following constraints, $a + 4 < 42$, $b + 4 < 42$, $c + 4 < 42$, $a + 5 < 42$, $b + 5 < 42$, and $c + 5 < 42$. In this example, when an operator is surrounded by two `VariableVector`s its expansion is the Cartesian product of elements of the surrounding `VariableVector`s.

It could also be simplified into the two following ones, $a + 4 < 42$ and $b + 5 < 42$. In this example, instead of using a Cartesian product, elements of surrounding `VariableVector`s are matched pairwise. We call this scalar expansion.

Depending on the situation, either Cartesian or scalar expansion can be useful. We decided to support them both in our constraint language. All

classical arithmetic operators use Cartesian expansion, and operator prefixed with a "." (such as .+ or . =) use scalar expansion.

6.4 Interacting with the Solver

Wrappers, shown at the right of Figure 12, encapsulate solvers so that the user can interact more easily with them. These wrappers take the constraint model as input and apply the transformations presented in Section 6.3.

After applying all these transformations, concrete constraints can be generated. This final transformation translates the constraint model into a solver-specific model, or direct calls to a solver's API. Depending on the considered solver and its features, this last step can be trivial or may involve complex data management in order to emulate missing features.

Solvers present various sets of features. For instance, Choco does not support adding or removing constraints during solving. In order to be able to use it in a context where constraints or variables can be added or removed at any time, the final transformation is much more complex than with the Cassowary solver, which supports adding or removing constraints and variables. Therefore, when using Choco, every change in the constraint model results in a completely new set of concrete constraints. Rebuilding this set of concrete constraints, and asking for a new resolution from scratch, can be costly if the problem is complex.

Hence, we propose two ways to interact with solvers:

1. the solver is automatically called whenever a change occurs in the concrete model. This mode is suitable for solvers that support fast incremental resolution, like Cassowary. We refer to this mode as *autosolving*.

2. the user has full control over the solver and triggers resolution manually. This second mode can be useful when changes between coupled variables occur. This is the preferred mode for solvers that do not support incremental solving or take a significant time to solve a problem[3], like Choco.

The wrapper is used to suggest changes on properties of the model as shown in Figure 7 (Section 4.2). A new constraint is then added to the constraint model in order to reflect the suggestion. Then, whether triggered automatically or called by the user, the solver can compute a new solution and propagate new values. Finally, the suggestion constraint is marked for removal for the next solving.

[3] In the context of graphical interfaces that we use as a case study, we consider a solving time to be significant if it is noticeable to the user.

Wrappers also take an important part in the synchronization between model properties and solver variables. As presented earlier in Section 4.2, each `BridgeVariable` references a specific property of a specific model element. Actual synchronization between the model element property and the corresponding variable in the solver is performed by the wrapper, which can read or write the value of the property. For instance, after a call to Choco, values of variables in the solution are compared to their corresponding property in the target model. If they differ, the wrapper updates the corresponding model element properties with the values found in the solution. This update is made possible by the `BridgeVariable` that targets a specific property of a model element.

If a model element property is used as a variable in one solver and as a constant in another, updating the value of said property will trigger an update in the constraint model of the second solver. This update will then be propagated to the concrete constraints of the second solver, and can trigger a new resolution if the solver is set to autosolve whenever an update occurs. This is the mechanism used to perform basic solver collaboration. It has the advantage of requiring no additional synchronization mechanisms but is relatively limited in terms of possible collaboration scenarios. For this mechanism to work properly, the user has to be careful during transformation development, and must ensure that no property is used by multiple solvers as a variable. Only one solver can use a given model element property as a variable, others have to use it as a constant. Moreover, if multiple properties are shared between two solvers, one must ensure that the one solver only has write permission to these variables, and the other only has read permission.

7 Evaluation of the Approach

In Section 5 we presented how our model set exploration approach can be deployed in an automatic way by using a transformation language to generate both target models and constraints. In Section 7.1 we present the constraint problems generated by our approach on the example detailed in Section 3.4. Then, in Section 7.2 we present some ATL rules used to generate the target and constraints presented in Section 7.1.

7.1 Explorable Task Scheduling Models

In Sections 3.2 and 3.4 we have introduced a simple example of a task scheduling tool. We briefly described how shapes should be placed in order to form

a valid diagram, and which constraints apply when creating a schedule. Listing 3 lists part of the constraints needed to ensure the diagram shown in Figure 6 is valid. We omitted strengths from constraints in order to keep them readable. Variables are named after the property they represent. For instance, `p1.r.x` correspond to the property `x` of element `p1.r`.

```
1  p1.r.y = 0, p1.r.x = 0.0*p1.r.width, p1.l.startX = p1.r.x,
2  p1.l.endX = p1.r.x + p1.r.width, p1.l.endY = p1.l.startY,
3  p1.l.startY = p1.t.y + 15.1 + 5,
4  p1.r.width = 125, p1.r.height = 200,
5  p1.t.x = p1.r.x + p1.r.width/2 - 88.3/2, p1.t.y = p1.r.y+5,
6
7  t000.r.x >= 0,
8  t000.r.width = 1.2 * 24.8,
9  t000.r.height = 1.2 * 15.1,
10 t000.t.x = t000.r.x + 5, tt000.t.y = t000.r.y + 2,
11 t000.r.x >= p1.r.x, t000.r.y >= p1.l.startY,
12 t000.r.x + t000.r.width <= p1.r.x + p1.r.width,
13 t000.r.y + t000.r.height <= p1.r.y + p1.r.height,
14
15 rq_t002_t000.l.fromX = 0, rq_t002_t000.l.fromY = 0,
16 rq_t002_t000.l.fromX - rq_t002_t000.l.toX = 0,
17 rq_t002_t000.l.fromY - rq_t002_t000.l.toY = 0,
18 rq_t002_t000.l.fromX >= t002.r.x,
19 rq_t002_t000.l.fromX <= t002.r.x + t002.r.width,
20 rq_t002_t000.l.fromY >= t002.r.y,
21 rq_t002_t000.l.fromY <= t002.r.y + t002.r.height,
22 rq_t002_t000.l.toX >= t000.r.x,
23 rq_t002_t000.l.toX <= t000.r.x + t000.r.width,
24 rq_t002_t000.l.toY >= t000.r.y,
25 rq_t002_t000.l.toY <= t000.r.y + t000.r.height
26
27 ...
```

Listing 3: Part of graphical constraints of Figure 6

```
1  000.{period}.number < 3, 000.{period}.number >= 0,
2  001.{period}.number < 3, 001.{period}.number >= 0,
3  002.{period}.number < 3, 002.{period}.number >= 0,
4  002.{period}.number > 000.{period}.number,
5  002.{period}.number > 001.{period}.number,
6
7  sum(5*reify(000.period=0),6*reify(001.period=0),7*reify(002.period=0))>=0,
8  sum(5*reify(000.period=0),6*reify(001.period=0),7*reify(002.period=0))<=15,
9  sum(5*reify(000.period=1),6*reify(001.period=1),7*reify(002.period=1))>=0,
10 sum(5*reify(000.period=1),6*reify(001.period=1),7*reify(002.period=1))<=15,
11 sum(5*reify(000.period=2),6*reify(001.period=2),7*reify(002.period=2))>=0,
12 sum(5*reify(000.period=2),6*reify(001.period=2),7*reify(002.period=2))<=15
```

Listing 4: Business constraints of Figure 6

In this example we decided to use Cassowary [3], a linear constraint solver. Therefore, constraints shown in Listing 3 only consist of linear equalities and

inequalities. We mentioned earlier that geometric constraints were well suited for diagram definition. However, when dealing with simple shapes such as rectangles or lines, most of these geometric constraints can be expressed as linear inequalities between coordinates or values. For instance, Constraint 1, that places the top left corner of a `Text` relative to its `Rectangle`, becomes two constraints, one for each coordinate (l. 10).

Listing 4, lists all constraints needed to compute a correct schedule. We decided to use the constraint solver Choco [36] to solve these constraints. We used the same notation for variables than the one used in Listing 3. We add only new operators (e.g., `reify`, `sum`) and the braces to denote variable encoding relations (l. 1-5). The `sum` operator is self explanatory. The `reify` operator converts the truth value of a constraint into an integer with value 0 if the constraint is not satisfied and 1 otherwise. For instance, Constraint 1, that bounds the load of a `Period` corresponds to two constraints (e.g., l. 7-8).

Patterns can be observed in constraints: those relating to model elements of the same type share a similar structure. `Period`s involve two constraints, one to ensure `load` is above `minLoad` and another one to ensure it is below `maxLoad`. `Task`s involve three constraints, two to specify the valid ids of `Periods` (l. 1, 2, and 3) and one to place `requisites` `Task`s in `Period`s with a lower `number` (l. 4-5). But not all `Task`s have all these constraints. Only `Task` 003 has the constraints corresponding to `requisites`. Differences appear because 003 requires 001 and 002, whereas neither 001 nor 002 have requisites.

The `contains` constraint is a geometric constraint that can be translated into four linear inequalities[4]. The four constraints encoding the fact that p1 contains task 001 are on l. 11-13 of Listing 3. Similar patterns appear for the `Requisite` arrow, the two ends are contained into the `Rectangle`s of the corresponding `Task` (l. 18-21 and l. 22-25).

As described in Section 4.2, once everything has been initialized, updates on the diagrams are propagated to the solver, which updates the solution, which is then propagated back to the diagram.

7.2 Generation of Explorable Task Scheduling Models

In Sections 3.2 and 3.3 we presented how our approach can be used to generate a valid schedule along with an interactive diagram visualization. This section describes how to implement this example with two model transfor-

[4] For simplicity we consider that every shape can be assimilated to a rectangle aligned with the x and y axis. This corresponds to the notion of bounding box.

mations that build the constraints to generate this schedule and the view for it.

In Section 3.2 we detailed a small scheduling problem and a visualization shown in Figure 6. It consists of a `Project` with 3 `Periods` and 3 `Tasks`. `Task` 003 requires `Tasks` 001 and 002 to be completed before it can start.

```
1  rule Period {
2    from
3      s : Scheduling!Period
4    to
5      constraints: Constraints!ExpressionGroup (
6        solver <- 'choco',
7        expressions <- Sequence {
8          (s.project.tasks.cost.toConstant() *
9            (s.project.tasks.period."="(s.toConstant())).reify()
10         ).sum() >= s.project.minLoad.toConstant(),
11         (s.project.tasks.cost.toConstant() *
12           (s.project.tasks.period."="(s.toConstant())).reify()
13         ).sum() <= s.project.maxLoad.toConstant()
14       }
15     )
16 }
```

Listing 5: Sample rule that generate business constraints for a Period

Sources of the transformations used in this paper are available at `https://github.com/TheoLeCalvar/scheduling-example`.

Listing 5 contains an ATL rule responsible for generating the constraints used to generate a proper schedule. From a `Period` (l. 3) it generates a `ExpressionGroup` (l. 5-15) containing the two constraints related to scheduling. The specific solver to use can be specified in the `ExpressionGroup` (e.g., Choco in this rule at l. 6). These two constraints ensure that the sum of the costs of `Tasks` assigned to a specific `Period` is between `minLoad` and `maxLoad`. Checking if a `Task` is assigned to the current `Period` is done through constraint reification (l. 9 and 12). It corresponds to constraints from l. 7-12 of Listing 4. Note that constraints on l.9 and 12 result in multiple constraints in Listing 4. This is because `s.project.task` references a collection, which is translated into a `VariableVector`. During translation, constraints containing `VariableVectors` are expanded to simple constraints. Thus, a simple constraint on a `VariableVector` can be translated into multiple constraints.

Listing 6 contains the rule that generates part of the diagram representing a `Task`. A single `Task` generates one `Rectangle`, one `Text`, and constraints in a `ExpressionGroup`. Unlike the rule in Listing 5, this one contains other output elements that are specified in plain old ATL (l. 5-11). It contains standard bindings between source and target elements. The `movable` property of `Rectangle` (l. 6) is a helper added to specify that an element can be

```
unique lazy rule Task {
  from
    c: Scheduling!Task
  to
    r : JFX!Rectangle (
      movable <- true
    ),
    t : JFX!Text (
      text <- c.code, textOrigin <- #TOP,
      mouseTransparent <- true
    ),
    constraints: Constraints!ExpressionGroup (
      solver <- 'cassowary',
      expressions <- Sequence {
        r.width = 1.2 * t.width.toConstant()
        ,r.height = 1.2 * t.height.toConstant()
        ,r.x.stay('MEDIUM'), r.y.stay('MEDIUM')
        ,t.x = r.x + 5, t.y = r.y + 2
        ,r.x >= 0, r.y >= thisModule.Period(c.period).l.startY
        ,r.x >= thisModule.Period(c.period).r.x -- strong
        ,r.x + r.width <= thisModule.Period(c.period).r.x
         + thisModule.Period(c.period).r.width --strong
        ,r.y+r.height <= thisModule.Period(c.period).r.y
                + thisModule.Period(c.period).r.height
      }
    )
}
```

Listing 6: Sample rule that generates view and graphical constraints for a `Period`

moved. Constraints of this rule target the Cassowary solver (l. 13). The first two constraints give the dimensions of the `Rectangle` (l. 15-16). Note the `toConstant()` operation, used to specify that a constant should be used instead of a variable. Corresponding constraints are on l. 8-9 of Listing 3. Other constraints specify the position of the `Text` (l. 18) and the position of the `Rectangle` (l. 19-24). It is possible to attach strengths to constraints by specifying it in a comment after the constraint (l. 20 and 22).

`Stay` constraints are special constraints that minimize changes to the variables they are applied to across multiple solves. That is to say, a *stayed* variable tends to keep its previous value. For instance, without `stay`, the position of a rectangle is only constrained by its container position and dimensions meaning that the solver is allowed to choose any value verifying the constraint. Moreover, there is no warranty that the solver will always choose the same value leading to unpredictable changes.

Remark: the ATL code listed in this section is slightly simplified from the original code, which can be consulted on GitHub using the link given above. Notably, the rule of Listing 5 is actually used in ATL refining mode, the JavaFX elements in Listing 6 also have identifiers, and an additional `Figure`

element should also be created in Listing 6 in order to encapsulate both JavaFX elements along with their constraints. These elements have been removed to simplify the listings here. Thus, we can focus on the new support for constraints, rather than on classical ATL mechanisms. Otherwise, these missing elements do not add much complexity.

7.3 Performance overview

Actual models used in MDE may be large, and performance may become critical. In our approach, we have identified several limiting factors.

The first one is due to AOF itself, the incremental backend that is used to perform incremental transformations and synchronizations. We have already shown that AOF can be as efficient as other state of the art approaches [11, 19]. Even if AOF still has performance issues with some structures of expressions [20, 25], we are working on solutions to limit these effects.

The other main limiting factor of our approach is related to the chosen constraint solver. We have scaled our different case studies and test projects to assess the performance of solvers. We confirmed that Cassowary [3], an incremental linear solver originally designed for graphical interfaces, is not a limiting factor, even for transformations including thousands of variables and constraints. We were surprised to observe that the JavaFX rendering algorithm was indeed the limiting factor for large interactive diagrams. Incremental solvers are not common but new approaches, such as [45], add incrementality to existing solvers.

On the other hand, using Choco, we quickly reached a point where the solving time was noticeable to the user. Therefore, in order to use Choco on larger inputs, the way the user interacts with the diagrams should be adapted. For instance, it is possible to let the user manually trigger a resolution instead of automatically solving each time a `Task` is moved from one `Period` to another.

One important point is that the complexity of the constraint problem depends on the number of input elements, but also on the constraints defined in the transformation. Indeed, these elements are instantiated for each input element matched by a rule. For instance, for the graphical constraints of the scheduling case study we have:

- 10 constraints per `Period`,
- 11 constraints per `Task`,
- 12 constraints per `Requisite` and `Corequisite`.

For the scheduling part we have:

- 2 constraints per `Period`,
- 5 constraints per `Task`.

However, due to `VariableVectors`, constraints contained into other constraints (via the `reify` operator) and their encoding in the solver, the actual number of constraints in the solver can be different.

For the smallest example (3 `Periods`, 3 `Tasks`, and 2 `Requisites`) we have 87 constraints for the graphical part and 78 constraints for the scheduling part. With the larger example (5 `Periods`, 20 `Tasks`, 15 `Requisites`, and 3 `Corequisites`), the number of graphical constraints grows to 487, and the number of scheduling constraints grows to 466. Cassowary had no issue dealing with this number of constraints, however Choco solving time started to become noticeable. Solving times are around 20ms but generating the constraint problem and propagating new values take around 80ms. Propagation of new value can trigger several calls to Cassowary if properties shared between Cassowary and Choco are updated.

However, constraint solving constitutes an active field of research. There are several international competitions such as the SAT Competition[5], the XCSP3 Competition[6], the MiniZinc Challenge[7], or the International Timetabling Competition[8]. Solvers are improved each year and they are now able to handle relatively complex problems.

8 Discussion

In this paper we only considered a weak integration of constraints in model transformation. The constraint solver is only used to complete small parts of the model: numerical values of properties and relations. However, other types of solver, more powerful, could handle more complex situations like cases in which the solver can decide that new elements should be added to the model. This is a limitation imposed by the kind of solver used in our prototype, not by the overall approach. The Cassowary solver is limited to linear constraints on double precision floating point values. However, thanks to this limitation the solver is able to deal with dynamic problems, which significantly reduces latency. For instance, in our diagram visualization example, while moving

[5] `http://www.satcompetition.org/`
[6] `http://xcsp.org/competition`
[7] `https://www.minizinc.org/challenge.html`
[8] `https://www.itc2019.org/home`

an element, each mouse move requires solving the problem while taking into account the new position of the moved element. Choco on the other hand is able to solve non linear constraints on integers but is not capable of efficiently solving dynamic problems. Whilst not built for solving dynamic problems, Choco proved to be responsive enough to be used in this use case. Other solvers may support more complex deltas that involve modifications of the structure of the model, such as adding or removing elements. It is unlikely that they would be able to solve problems in the time it takes to move the mouse, but this may be compatible with other applications of our approach. However, this goes beyond the scope of this paper.

We considered both satisfaction and optimization problems in Section 7.2 with the `stay` constraint because it is often necessary to ensure stability when updating existing solutions. Cassowary is a linear hierarchical constraint solver especially designed to be used in dynamic contexts such as graphical interfaces (see [32, 44, 15]). With Cassowary it is possible to add and remove constraints at any time, it is also possible to suggest new values for decision variables. After any modification, it can recompute a new solution based on the previous one. This makes it relatively efficient with dynamic problems such as graphical interfaces. During solving, Cassowary minimizes the error associated with its constraints. It also associates a weight to every constraint. These weights are used to determine which constraints should be satisfied in priority. This feature is used to define behaviors by assigning a strength and a weight to each constraint. We showed in Section 4.3 that while not present in all solvers, hierarchical and stay constraints can easily be implemented in classical solvers. Our current implementation wraps Choco in way that makes it appear as a dynamic solver. However, this does not mean that any solver can be used to solve any dynamic problem when there are performance considerations.

Currently the constraint dialect supported by the ATOL constraint plugin can be a limiting factor for the user. We are working on a geometric abstraction that would ease the development complexity, especially for users without constraint programming knowledge. This geometric abstraction would use a similar technique to the constraint plugin of ATOL. The user would be able to write geometric constraints that would be compiled into a geometric metamodel. This metamodel would then be transformed into constraints and graphical elements.

Another complementary approach we are considering would be to add support for more OCL operations already supported by ATL, but not in constraints. Ideally, this would enable the user to write constraints with classical ATL expressions. These expressions would then be interpreted as constraints that must be enforced instead of simply being evaluated.

We also presented a simple strategy to make solvers collaborate by imposing that only one solver can write a property or relation. Other solvers only have access to a read-only version of the property or relation. This ensures that no cycle can occur during update propagation. Our implementation currently does not check if there are cycles, and users must make sure of it.

Finally, in our current implementation we only considered cases where the first solver can notify the user that no solution can be found and that rollback is the only option. Thus, solvers down the propagation chain cannot notify solvers upstream that their problem is unsatisfiable and that a rollback is necessary.

9 Conclusion

This paper presents the model set exploration problem along with a framework to write model transformations that generate explorable sets of models. The model set exploration problem arises when the target model of a model transformation contains information that cannot be derived from the source model but also needs to respect a set of constraints in order to be valid.

This paper describes techniques to use constraint solvers to generate an intensionally defined model set from a partially populated target model and a set of constraints. This is enabled by bridge variables that act as synchronization mechanisms between properties of target model elements and variables in a constraint solver.

This approach has been applied to a case study based on schedule generation and visualization. Both generation and visualization are simple and sufficient to illustrate the approach. This case study focuses on interactive diagrams but the approach presented in this paper is not limited to interactive applications. Small examples, as the one presented in this paper, could be solved by any solver but solving time is a real limitation that has to be considered in the solver choice. In fact, many solvers are not considered here since they cannot solve interactive diagram problems fast enough. Only few solvers are supported but we would like to add support for new ones. Especially solvers that could solve constraints over different variable domains.

This case study also illustrates the fact that this approach can leverage multiple solvers to solve independent parts of the problem. For instance, finding a valid schedule and positioning graphical elements of the visualization. This kind of solver collaboration is relatively limited right now but could be improved with a deeper analysis of the constraints network.

Acknowledgment

Work was partially funded by Angers Loire Métropole and RFI Atlanstic 2020. We would like to thank Dániel Varró for his helpful comments.

References

1. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of model transformations via alloy. In: MoDeVVa '07, pp. 47–56 (2007)
2. Apt, K.: Principles of Constraint Programming. Cambridge University Press (2003). DOI 10.1017/CBO9780511615320
3. Badros, G.J., Borning, A., Stuckey, P.J.: The Cassowary linear arithmetic constraint solving algorithm. TOCHI **8**(4), 267–306 (2001)
4. Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J.: On the reification of global constraints. Constraints **18**(1), 1–6 (2013). DOI 10.1007/s10601-012-9132-0. URL `https://doi.org/10.1007/s10601-012-9132-0`
5. Benhamou, F.: Heterogeneous constraint solving. In: M. Hanus, M. Rodríguez-Artalejo (eds.) Algebraic and Logic Programming, pp. 62–76. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
6. Blazewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Weglarz, J.: Handbook on Scheduling: From Theory to Applications. Springer Publishing Company, Incorporated (2014)
7. Boussemart, F., Lecoutre, C., Piette, C.: XCSP3: an integrated format for benchmarking combinatorial constrained problems. CoRR **abs/1611.03398** (2016)
8. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL Transformations Using Transformation Models and Model Finders. In: ICFEM 2012, pp. 198–213 (2012)
9. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: ASE '07, pp. 547–548. ACM (2007)
10. Calvar, T.L., Jouault, F., Chhel, F., Clavreul, M.: Efficient ATL incremental transformations. JOT **18**(3), 2:1 (2019). DOI 10.5381/jot.2019.18.3.a2
11. Calvar, T.L., Jouault, F., Chhel, F., Clavreul, M.: Efficient ATL incremental transformations. J. Object Technol. **18**(3), 2:1–17 (2019). DOI 10.5381/jot.2019.18.3.a2. URL `https://doi.org/10.5381/jot.2019.18.3.a2`

12. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: A bidirectional and change propagating transformation language. In: SLE '10, pp. 183–202 (2011)

13. Cunha, A., Macedo, N., Guimarães, T.: Target oriented relational model finding. In: S. Gnesi, A. Rensink (eds.) Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, *Lecture Notes in Computer Science*, vol. 8411, pp. 17–31. Springer (2014). DOI 10.1007/978-3-642-54804-8_2. URL https://doi.org/10.1007/978-3-642-54804-8_2

14. Diskin, Z., Wider, A., Gholizadeh, H., Czarnecki, K.: Towards a rational taxonomy for increasingly symmetric model synchronization. In: ICMT 2014, pp. 57–73 (2014)

15. Freeman-Benson, B.N., Maloney, J., Borning, A.: An incremental constraint solver. Commun. ACM **33**(1), 54–63 (1990). DOI 10.1145/76372.77531

16. Horváth, Á., Varró, D.: CSP(m): Constraint satisfaction problem over models. In: Model Driven Engineering Languages and Systems, pp. 107–121. Springer Berlin Heidelberg (2009). DOI 10.1007/978-3-642-04425-0_9

17. Horváth, Á., Varró, D.: Dynamic constraint satisfaction problems over models. Software & Systems Modeling **11**(3), 385–408 (2012)

18. Jackson, D.: Alloy: a lightweight object modelling notation. TOSEM **11**(2), 256–290 (2002)

19. Jouault, F., Beaudoux, O.: Efficient OCL-based Incremental Transformations. In: 16th International Workshop in OCL and Textual Modeling, pp. 121–136 (2016)

20. Jouault, F., Beaudoux, O., Brun, M., Chhel, F., Clavreul, M.: Improving incremental and bidirectional evaluation with an explicit propagation graph. In: M. Seidl, S. Zschaler (eds.) Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 10748, pp. 302–316. Springer (2017). DOI 10.1007/978-3-319-74730-9_27. URL https://doi.org/10.1007/978-3-319-74730-9_27

21. Jouault, F., Kurtev, I.: Transforming models with ATL. In: J.M. Bruel (ed.) Satellite Events at the MoDELS 2005 Conference, pp. 128–138. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

22. Kleiner, M., Didonet Del Fabro, M., Albert, P.: Model search: Formalizing and automating constraint solving in MDE platforms. In: ECMFA 2010, pp. 173–188 (2010)

23. Kleiner, M., Didonet Del Fabro, M., De Queiroz Santos, D.: Transformation as search. In: ECMFA 2013, pp. 54–69 (2013)

24. Laurent Michel, Pierre Schaus, Pascal Van Hentenryck: MiniCP: A lightweight solver for constraint programming (2018). Available from `https://minicp.bitbucket.io`

25. Le Calvar, T., Chhel, F., Jouault, F., Saubion, F.: Using process algebra to statically analyze incremental propagation graphs. In: OCL '18, pp. 160–173. Copenhague, Denmark (2018)

26. Le Calvar, T., Chhel, F., Jouault, F., Saubion, F.: Toward a Declarative Language to Generate Explorable Sets of Models. In: SAC '19, pp. 1837–1844. Limassol, Cyprus (2019)

27. Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J.: A comparison of incremental triple graph grammar tools. Electronic Communications of the EASST **67** (2014)

28. Lecoutre, C.: Constraint Networks: Techniques and Algorithms. Wiley-IEEE Press (2009)

29. Ludovico, I., Barriga, A., Rutle, A., Heldal, R.: Model repair with quality-based reinforcement learning. The Journal of Object Technology **19**(2), 17:1 (2020). DOI 10.5381/jot.2020.19.2.a17

30. Macedo, N., Cunha, A., Guimarães, T.: Exploring scenario exploration. In: A. Egyed, I. Schaefer (eds.) Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, *Lecture Notes in Computer Science*, vol. 9033, pp. 301–315. Springer (2015). DOI 10.1007/978-3-662-46675-9_20. URL `https://doi.org/10.1007/978-3-662-46675-9_20`

31. Macedo, N., Jorge, T., Cunha, A.: A feature-based classification of model repair approaches. IEEE Transactions on Software Engineering **43**(7), 615–640 (2017). DOI 10.1109/TSE.2016.2620145

32. Menezes, F., Barahona, P., Codognet, P.: An incremental hierarchical constraint solver. In: PPCP, vol. 93, pp. 190–199 (1993)

33. Monfroy, E., Castro, C.: Basic components for constraint solver cooperations. In: G.B. Lamont, H. Haddad, G.A. Papadopoulos, B. Panda (eds.) Proceedings

of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA, pp. 367–374. ACM (2003). DOI 10.1145/952532.952606

34. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: CP '07, pp. 529–543. Springer (2007)
35. Petter, A., Behring, A., Mühlhäuser, M.: Solving constraints in model transformations. In: ICMT 2009, pp. 132–147 (2009)
36. Prud'homme, C., et al.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017). URL `http://www.choco-solver.org`
37. Schätz, B., Hölzl, F., Lundkvist, T.: Design-space exploration through constraint-based model-transformation. In: ECBS '10, pp. 173–182 (2010). DOI 10.1109/ECBS.2010.25
38. Semeráth, O., Nagy, A.S., Varró, D.: A graph solver for the automated generation of consistent domain-specific models. In: ICSE '18, pp. 969–980. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180186
39. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: Fundamental Approaches to Software Engineering, pp. 87–103 (2016)
40. Sen, S., Baudry, B., Mottu, J.M.: Automatic Model Generation Strategies for Model Transformation Testing. In: R.F. Paige (ed.) ICMT 2009, pp. 148–164 (2009)
41. Sen, S., Baudry, B., Precup, D.: Partial model completion in model driven engineering using constraint logic programming. In: INAP '07, p. 59 (2007)
42. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: O. Grumberg, M. Huth (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4424, pp. 632–647. Springer (2007). DOI 10.1007/978-3-540-71209-1_49. URL `https://doi.org/10.1007/978-3-540-71209-1_49`
43. Varró, D., et al.: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. Software & Systems Modeling **15**(3), 609–629 (2016). DOI 10.1007/s10270-016-0530-4
44. Wilson, M., Borning, A.: Hierarchical constraint logic programming. The Journal of Logic Programming **16**(3), 277 – 318 (1993). DOI 10.1016/0743-1066(93)90046-J

45. Zheng, G., Bagheri, H., Rothermel, G., Wang, J.: Platinum: Reusing constraint solutions in bounded analysis of relational logic. In: H. Wehrheim, J. Cabot (eds.) Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, *Lecture Notes in Computer Science*, vol. 12076, pp. 29–52. Springer (2020). DOI 10.1007/978-3-030-45234-6_2. URL `https://doi.org/10.1007/978-3-030-45234-6_2`