GRAPH TRANSFORMATION BASED CONSTRAINT SOLVING

Ákos HORVÁTH
Advisor: Dániel VARRÓ

I. Introduction

Constraint satisfaction is a fundamental Artificial Intelligence technique for knowledge representation and reasoning. It has, however, become clear that in many cases the original formulation of static constraint problems (CSP) [1] with imperative constraints is insufficient to model real problems (dynamic allocation, change in the constraint set, etc.) Research has been already carried out to address these shortcomings in the form of two separate extensions known as flexible [2] and dynamic CSP [3] and recent work is focusing on the combination of these separate extension [4].

In this paper we introduce our novel approach, which uses graph patterns and graph transformation (GT) [5] rules to extend the definition of constraint satisfaction problems. The idea is to define the constraints and the labeling rules as graph patterns and transformation rules, respectively treat the model modified by the rules as the only variable of the CSP, where a result is achieved when no matches are found for any constraint pattern. This approach allows to (i) dynamically add/remove constraints from the problem domain, (ii) modify the domain of the variables and (iii) define structural constraints in a more natural way.

The rest of the paper is structured as follows. In Sec. II. we briefly introduce the concept of metamodeling, graph transformation and constraint satisfaction problems. Sec. III. proposes our graph pattern and transformation based constraint solver. Finally, Sec. IV. concludes the paper.

II. Background

In order to introduce our approach this section briefly outlines the basics of metamodeling, graph transformation and constraint programming.

A. Models and Metamodels

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e., abstract syntax) of modeling languages. More precisely, for the definition of a modeling language the followings shall be given (i) the abstract syntax defining the concepts of the given domain and their relations, (ii) the concrete syntax defining the textual or graphical notations of the concepts, (iii) wellformness rules defining further constraints for the concepts, (iv) the formal semantics defining the dynamic behaviour of the models.

In our approach, we use a unified directed graph representation as the underlying model of the VIATRA2 [6] framework. This way, graph nodes (called entities in VIATRA2) uniformly represent MOF packages, classes, or objects on different metalevels, while graph edges with identities (called relations in VIATRA2) denote MOF association ends, attributes, link ends, and slots in a uniform way. Essentially, nodes represent basic concepts of a (modeling) domain, while edges represent the relationships between model elements. An example metamodel is depicted in Fig. 1 representing a simple allocation problem domain, where `Simple` and `Complex` jobs can be allocated to `Nodes`.

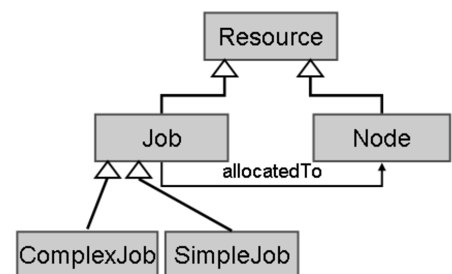
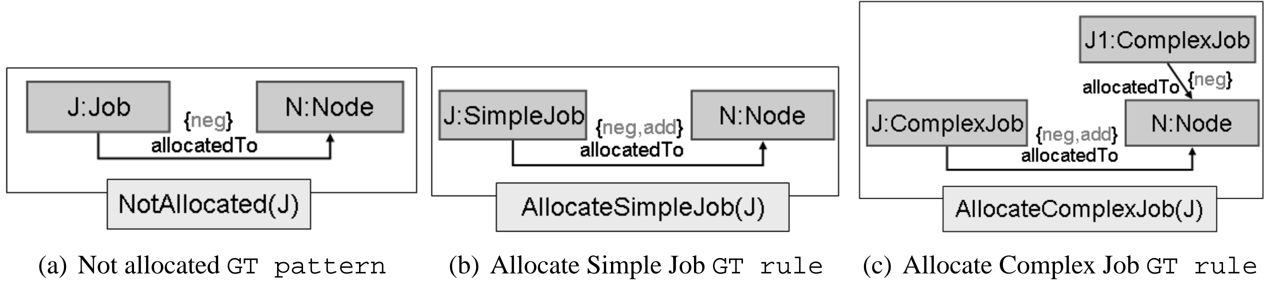


Figure 1: Sample Metamodel



(a) Not allocated `GT pattern` (b) Allocate Simple Job `GT rule` (c) Allocate Complex Job `GT rule`

Figure 2: Sample graph pattern and transformation

B. *Graph Transformation Rules and Patterns*

Graph transformation (GT) is a rule and pattern-based paradigm frequently used for describing model transformation. A graph transformation rule contains a left-hand side graph LHS, a right-hand side graph RHS, and (one or more) negative application condition graphs NAC connected to the LHS. A negative application condition is a graph morphism, which maps the LHS pattern to a NAC pattern. In other terms, the LHS and NAC graphs together denote the precondition (also referred to as GT pattern) while the RHS denotes the postcondition of a rule.

The application of a rule to a host (instance) model M replaces a matching of the LHS – provided it is not invalidated by a matching of the NAC, which prohibits the presence of certain nodes and edges – in M by an image of the RHS.

Sample GT pattern and rules defined over the metamodel described in Fig. 1 are depicted in Fig. 2, using a combined representation that jointly defines the left hand side (LHS) of the graph transformation rule, and the actions to be carried out. The `NotAllocated` pattern matches an input parameter `Job J` which is not already `allocatedTo` to a `Node N` (the NAC is highlighted by the `neg` adornment). That the `AllocateSimpleJob` and `AllocateComplexJob` represent GT rules that allocate a `Simple` or `Complex` job to a `Node`, respectively. The additional NAC guarantees that only a one `Complex` job can be allocated to a single `Node`. In both cases the `add` adornment defines the element to be created.

C. *Constraint Satisfaction*

Basically, a CSP is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. In a more precise way a constraint satisfaction problem is a triple: (Z, D, C) where

- Z is a finite set of variables $x1, x2, ..., xn$;
- D is a function which maps every variable in Z to a set of objects of arbitrary type.
- C is a finite (possibly empty) set of constraints on an arbitrary subset of variables in Z. In other words, C is a set of sets of compound labels.

The task is to assign a value to each variable satisfying all the constraints.

III. Overview of the Approach

This section first, gives an overview of the approach using our running resource allocation example, and then summarizes the advantages and disadvantages

A. *Resource Allocation*

In general the aim of a resource allocation problem is to determine the allocation of a fixed amount of resources to a given number of activities in order to achieve an effective results. Considering the metamodel described in Sec. A. our task is to allocate `Complex` and `Simple` jobs to `Nodes` with a restriction that one `Node` can have arbitrary number of `Simple` but only one `Complex` job. Without

going into details in order to formulate the problem using GT rules and patterns we have to define the following artifacts:

- A typed graph *model* – conforming to a given metamodel – that represents the initial state of the CSP. During the solution process it is modified using *labeling* literals to achieve a concrete state that satisfies all given `goals`, where a *state* of the solution process is a concrete state of the *model*.
- *Goals* (constraints) are represented by GT patterns over the given metamodel. A *goal* is satisfied if no match is found to its representing GT pattern. For a given problem arbitrary number of *goals* can be defined.
- *Labeling* literals are GT rules defined over the given metamodel used to manipulate the *model*. During the solution processa following *state* is achieved by a random selection from the applicable GT rules and its invocation on the *model*. A solution process ends if a solution is found or if all possible *states* are traversed or the depth of the state space tree is higher than an initially given number. This last restriction is necessary to ensure that the process will definitely terminate as in general termination of graph rewriting is undecidable.

We formalized the running example constraint problem with the GT pattern and rules depicted in Fig. 2. The `NotAllocated` pattern defines the *goal* of the problem, namely, that all `Jobs` have to be allocated to a `Node`, which is equivalent to that there is no match found for the pattern. The *labeling* literals are captured by the `AllocateSimpleJob` and the `AllocateCompleyJob`. For an initial model we use the one depicted in Fig. 3. A possible solution is that `CJ1` and `CJ2` `ComplexJobs` are allocated to the `N1` and `N2` `Nodes`, respectively, while both `SimpleJobs` are allocated to one of the `Nodes`.

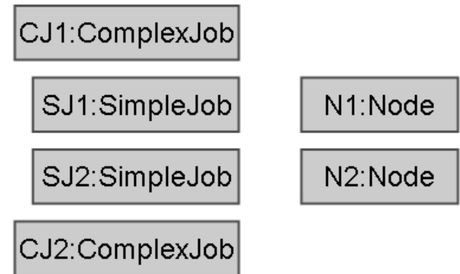


Figure 3: Sample instance model

B. Advantages and Drawbacks

The proposed approach promises a number of benefits but also has drawbacks that have to be carefully analyzed in order to use it in its full potential. In the following we try to address some of these issues:

- One of the main advantage of using GT rules for the labeling is that it allows to define **dynamic creation/deletion** rules on any element (defined in the metamodel) of the problem domain. Considering our running example the GT rule depicted in Fig. 4 gives dynamic `Node` creation for a `ComplexJob` that is not allocated. However this kind of dynamic behavior can only be effectively used if proper priorities are defined to the labeling literals and handled during the solution process as these kind of rules can easily run into nonterminational graph rewriting sequences.
- Our approach not only allows to dynamically create and delete elements of the problem domain, but also allows to **dynamically add/remove** *goals* and *labeling* literals to alter the constraint problem. This feature helps to address problems defined in DCSP [7].
- **Flexible constraint** satisfaction can also be easily adapted by defining weights for each goal. In this case a solution is achieved if the accumulated weight of the unsatisfied goals is lower than a predefined limit (weighted CSP [4]). Similar weighting is required for adapting the MAX-CSP [4] approach, where the *quality* of the solution is related to the sum of the unsatisfied goals.
- By using graph based state representation we had to address the problem of **typed graph comparison**. For this we adapted the DSMDIFF [8] algorithm, which relies on (i) signatures (for nodes and edges) composed of type and name information and (ii) containment relation between nodes of the graph. Initial research showed that this approach works only on small scale problems and some cases require problem specific comparators as DSMDIFF can not handle intensive node creation and deletion.

- It is also important to mention that saving **already visited states** effectively is also crucial on overall performance. For which we serialized the models using a containment based traversal combined with a signature driven ordering algorithm.
- In order to support **backtracking** between states of the model, we apply a simple transaction mechanism that saves the atomic model manipulation operations applied on the model in an undo stack. This stack not only allows us to backtrack the manipulations but also ease the computation of difference between the states of the model. This unique ability is useful in problems that require solutions that are "nearest" to a given initial model (e.g., reconfiguration).

IV. Conclusion and Future Work

In the current paper, we have presented a novel approach defining constraint problems using a combination of graph transformation rules and patterns and also listed some of the concerns of advantages and disadvantages.

As for the future, we plan to address some of the questions raised in Sec. III. B., especially focusing on dynamic behavioral extensions.

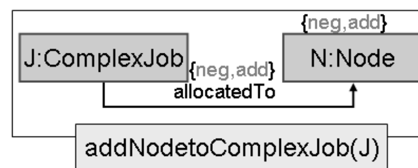


Figure 4: AddNode GT rule

References

[1] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1. edition, August 1993.

[2] R. J. Wallace, "Enhancements of branch and bound methods for the maximal constraint satisfaction problem," in *Proc. of AAAI-96*, pp. 188–195, 1996.

[3] T. Schiex, "Solution reuse in dynamic constraint satisfaction problems," in *In Proceedings of the 12th National Conference on Artificial Intelligence*, pp. 307–312. AAAI Press, 1994.

[4] I. Miguel and Q. Shen, "Dynamic flexible constraint satisfaction," *Applied Intelligence, 13(3)*, pp. 231–245, 2000.

[5] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations*, World Scientific, 1997.

[6] A. Balogh and D. Varró, "Advanced model transformation language constructs in the VIATRA2 framework," in *Proc. of the 21st ACM Symposium on Applied Computing*, pp. 1280–1287, Dijon, France, April 2006. ACM Press.

[7] R. Dechter and A. Dechter, "Believe maintenance in dynamic constraint network," in *Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 37–42, 1988.

[8] Y. Lin, J. Gray, , and F. Jouault, "Dsmdiff: A differentiation tool for domain-specific models," *European Journal of Information Systems, Special Issue on Model-Driven Systems Development 16(4)*, pp. 349–361, 2007.