CSP(M): Constraint Satisfaction Problem over Models ⋆

Ákos Horváth and Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{ahorvath, varro}@mit.bme.hu

Abstract. Constraint satisfaction programming (CSP) has been successfully used in model-driven development (MDD) for solving a wide range of (combinatorial) problems. In CSP, declarative constraints capture restrictions over variables with finite domains where both the number of variables and their domains are required to be a priori finite. However, the existing formulation of constraint satisfaction problems can be too restrictive to support dynamically evolving domains and constraints necessitated in many MDD applications as the graph nature of the underlying models needs to be encoded with variables of finite domain. In the paper, we reformulate the constraint satisfaction problem directly on the model-level by using graph patterns as constraints and graph transformation rules as labeling operations. This allows expressing problems composed of dynamic model manipulation and complex graph structural constraints in an intuitive way. Furthermore, we present a prototype constraint solver for the domain of graph models built upon the VIATRA2 model transformation framework, and provide an initial evaluation of its performance.

Keywords: **constraint satisfaction programming, graph transformation**

1 Introduction

In artificial intelligence, the constraint satisfaction problem (CSP) is to find a solution to a set of constraints that impose conditions which has to be satisfied by a set of variables. Each variable typically takes its value from a finite domain. A solution is one (or all) assignment of variables which satisfy each constraint.

Constraint satisfaction techniques have been successfully applied for various problems of model-driven engineering for applying design patterns [1], to support domain-specific modeling [2] or in the context of model transformations [3]. As a commonality, all these approaches translate high-level models to an existing (off-the-shelf) constraint solver (like e.g. [4, 5]) to provide embedded design intelligence for modeling.

However, advanced constraint solvers typically apply certain restrictions for the CSP problem. For instance, the domains of variables are frequently required to be (a priori) finite, moreover, many approaches disallow the dynamical addition or retraction of constraints. Furthermore, mapping graph-like models obtained in model-driven engineering to variables with finite domain can be a non-trivial task, especially, when considering the evolution of models. While recent research initiatives in CSP have started

⋆ This work was partially supported by the EC FP6 DIANA (AERO1-030985) European Project.

to better address dynamic constraints [6], no efficient solvers are available for structural constraints over graph-like models.

In this paper, we investigate how advanced model transformation technology can contribute to solving dynamic constraint satisfaction problems with global constraints over the domain of model graphs. We extend the definition of constraint satisfaction problems by using *graph patterns to define structural (first-order logic) constraints*, and *graph transformation rules [7] as labeling operations*. Informally, all graph pattern constraints need to be satisfied by the underlying model when searching for a specific goal. However, instead of simple variable substitution, the labeling phase applies graph transformation rules to carry out model manipulations on the underlying graph domain.

As an analogy, our approach allows to (i) dynamically add/remove constraints from the problem domain, (ii) modify the domain of the variables during search and (iii) define structural constraints in a more natural way.

Furthermore, we developed a prototype constraint solver on top of the VIATRA2 [8] model transformation framework by using incremental constraint evaluation and various search strategies and heuristics. An initial evaluation of the solver is carried out using an allocation problem taken from critical systems.

The rest of the paper is structured as follows. In Sec. 2 we briefly introduce the concept of metamodeling, graph transformation and constraint satisfaction problems. Sec. 3 proposes our graph pattern and transformation based constraint solver. Related work is assessed in Sec. 5, and finally, Sec. 6 concludes the paper.

2 Background

In order to introduce our approach this section briefly outlines the basics of graph transformation and gives a motivating example from the avionics domain.

2.1 Running Example: Allocation of an IMA system

As a motivating example, let us assume an integrated modular avionics (IMA) system composed of *Jobs* (also referred as applications), *Partitions*, *Modules* and *Cabinets*. *Jobs* are the atomic software blocks of the system defined by their memory requirement. Based on their criticality level jobs are separated into two sets: *critical* and *simple* (non-critical). For critical jobs double or triple modular redundancy is applied while for simple ones only one instance is allowed. *Partitions* are complex software components composed of jobs with a predefined free memory space. Jobs can be allocated to the partition as long as they fit into its memory space. *Modules* are SW components capable of hosting partitions. Finally, *Cabinets* are storages for maximum

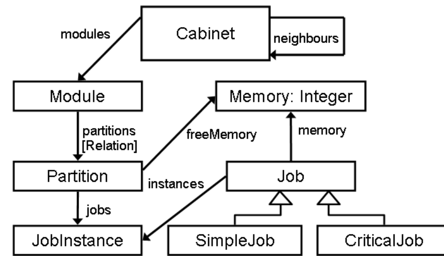
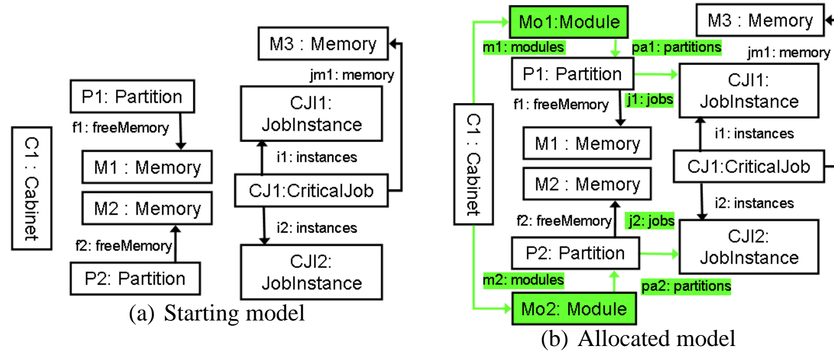


Fig. 1. Metamodel of an IMA architecture



(a) Starting model (b) Allocated model

Fig. 2. Example IMA system

(in our example) up to two modules used to physically distribute elements of the system. Additionally a certain number of safety related requirements will also have to be satisfied: (i) a partition can only host jobs of one criticality level and (ii) instances of a certain critical job can not be allocated to the same partition and module. The task is to allocate an IMA system defined by its jobs and partitions over a predefined cabinet structure and to minimize the number of *modules* used.

A sample system composed of a critical job with two instances and two partitions with a single cabinet is shown in Fig. 2(a) with a possible allocation depicted in Fig. 2(b) defined over the metamodel captured in the VPM formalism [9] in Fig. 1. Newly created elements are highlighted in grey. Throughout the paper we will use this example to demonstrate the technicalities of our constraint satisfaction technique over models.

2.2 Graph Patterns and Graph Transformation

Graph patterns (GP) are frequently considered as the atomic units of model transformations [8]. They represent conditions that have to be fulfilled by a part of the instance model. The VIATRA2 notation in particular, describes them as a disjunction of pattern bodies $GP = \vee PB_i$, where a pattern is fulfilled if at least one of its pattern body is fulfilled. *Pattern bodies* $PB = (SC, AC, NAC_j)$ consist of (i) *structural conditions* SC prescribing the existence of nodes and edges of a given type, (ii) *attribute conditions* (AC) allowing term evaluation over the attributes of the matched elements (marked by the check keyword) and (iii) arbitrary number of negative application conditions. A *negative application condition* $NAC = \neg GP$, defined by a negative subpattern, prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [10].

A *match* m for a graph pattern GP in a instance model M denoted by $m : GP \longrightarrow M$ means that (i) $m : PB_i \mapsto M, (\exists PB_i \in GP)$ there exists an injective, type conformant total morphism m from one of its pattern bodies $PB_i = (SC_i, AC_i, NAC_{i,j})$ to the instance

model; (ii) $m' : NAC_{i,j} \mapsto M, (\nexists NAC_{i,j})$ if no matches exist for any embedded NACs of that pattern body PB_i and (iii) all attribute conditions AC_i are fulfilled by m.

Graph transformation [7] provides a high-level rule and pattern-based manipulation language for graph models. Graph transformation $GT = (LHS, RHS, AMA)$ rules can be specified by using a left-hand side – LHS (or precondition) pattern determining the applicability of the rule, a right-hand side – RHS (postcondition) pattern which declaratively specifies the result model after rule application, and additional attribute manipulation AMA actions .

The *application* of a GT rule to a host model G alters the model by replacing the pattern defined by LHS with the pattern defined by RHS. This is performed by (i) finding a matching $m : LHS \longrightarrow G$ of the LHS pattern in model graph G; (ii) removing a part of the model graph M that can be mapped to LHS but not to RHS; (iii) adding new elements to the which exist in RHS but not in LHS and finally (iv) performing the attribute manipulation operations described in AMA. A *graph transformation* step is denoted formally as $G \stackrel{r,m}{\Longrightarrow} H$, where H is the resulting model; r and m denote the applied rule and the matching, respectively.

Example. Sample graph patterns and transformation rules are depicted in Fig. 3. The jobInstancewithoutPartition pattern matches an input parameter JobInstance JIns which is not already allocated to a Partition P by the j1 jobs relation (elements of the NAC are encapsulated by the NEG rectangle).

The allocateJobInstance GT rule allocates the JobInstance JI to the Partition P1 (by the jobs j1 relation) if it is not already allocated to the P2 Partition and decreases the MP free memory attribute of the P1 partition by the memory requirement of Job J captured in MJ. We use a combined representation that jointly defines the left hand side (LHS) of the graph transformation rule, and the model manipulation operations to be carried out where newly created elements and attribute manipulation operations are tagged with an add and set keywords, respectively.

3 Constraint Satisfaction Programming

In this section, we provide a detailed description of our constraint satisfaction framework and its conceptual foundations and demonstrate how to apply it on the IMA system allocation problem introduced in Sec. 2.1.

3.1 Constraint Problem specification

Constraint Satisfaction Problem for Variables of Finite Domain A CSP(FD) 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 $x_1, x_2, ..., x_n$; D is a function which maps every variable in Z to a set of objects of arbitrary type; and C is a finite (possibly empty) set of constraints on an arbitrary subset of variables in Z. The task is to assign a value to each variable satisfying all the constraints. Solutions to CSPs are usually found by (i)

constraint propagation a reasoning technique to explicitly forbid values or domains for variables by predicting future subsequent constraint violations and (ii) *variable labeling* searching through the possible assignments of values to variables already restricted by the (propagated) constraints.

Planner Algorithms Planner algorithms [11] are hierarchical problem solving procedures subdividing the original problem into smaller parts. A planner $(I,E,O) \rightarrow P$, is a structure where I is a logic formula of the initial state, E is the logic formula of the goal state, while O is the set of permitted operations. The output P is a sequence of operations (called plan) providing a trajectory from the initial to the goal state. An operation $o = (C,A)$ is a pair where C stands for a precondition defined in first order logic and A for actions. Preconditions must hold before performing its specific operation.

3.2 CSP(M): Constraint Satisfaction Problem over Models

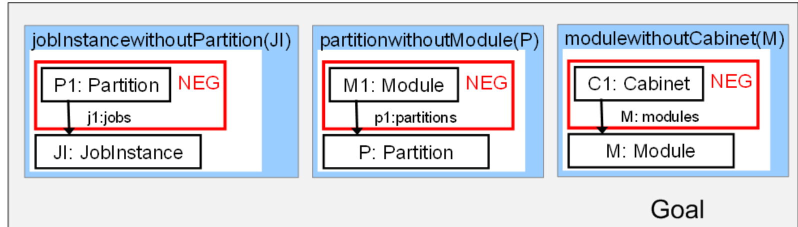
We now define constraint satisfaction problems over models (CSP(M)) by combining *CSP for finite domains* and *planner algorithms* (see in Sec. 3.1). In principle, our approach generalizes planner algorithms with the definition of *global constraints* that can additionally restrict certain trajectories of the search space and extends traditional CSP(FD) by introducing *labeling rules* to define and solve constraint problems over models even with dynamic model manipulation such as element creation and deletion.

A CSP(M) consist of an *initial model*; a *goal* that have to be satisfied by the *solution model* to be searched; *global constraints* that need to be satisfied by all models traversed during the search and finally a set of *labeling rules* capturing the permitted operations. Formally a CSP(M) $(M_0,C,G,L) : M_s$ is a structure where: M_0 is the initial model; C is a set of global constraints; G is a set of subgoals which together in conjunction form the goal and L is a set of labeling rules. The output M_s is the solution model satisfying:

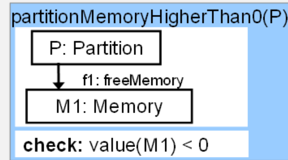
- (i) $M_0 \rightsquigarrow M_s$; there exists a trajectory $M_o \xrightarrow{l_1} M_1 \xrightarrow{l_2} .. \xrightarrow{l_n} M_n$ where $i = 1..n : l_i \in L$. Meaning that M_s is reachable from M_0 through a sequence of applied labeling rules.
- (ii) $\forall G_i \in G : M_s \models G_i$; M_s satisfies all subgoals G_i
- (iii) $\forall C_i \in C : M_s \models C_i$; M_s also satisfies all global constraints C_i
- (iv) $\forall M_i, \forall C_j \in C : M_0 \rightsquigarrow M_i \wedge M_i \rightsquigarrow M_s \wedge M_i \models C_j$; along the trajectory from the initial to the solution model all visited model M_i satisfies each global constraint.

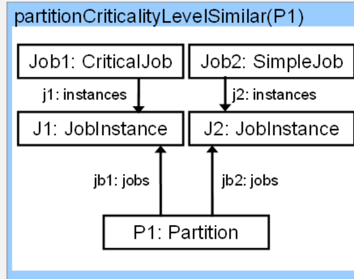
As models in MDD are usually described as graphs we instantiate our formalism on graph transformation using the VIATRA2 [8] language. This way models are captured by typed graphs over a given metamodel while subgoals and global constraints are defined using graph patterns and finally labeling rules are described as graph transformation rules. However, this formalism can also be incorporated into other modeling approaches such as MOF models, OCL constraints and QVT rules.

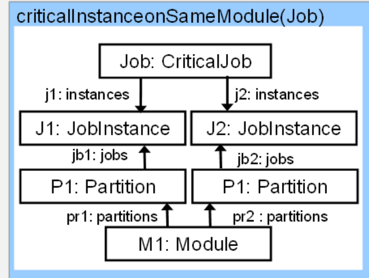
Goal and Global constraints Both subgoals and global constraints are defined by graph patterns. The goal G is the conjunction of subgoals where a subgoal (graph pattern) is a disjunction of alternate pattern bodies.

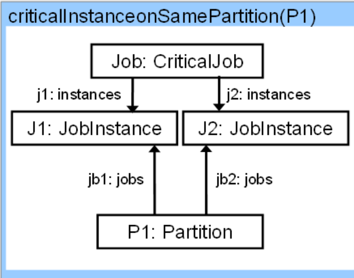


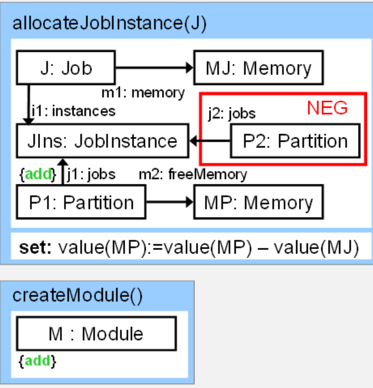
Global Constraints

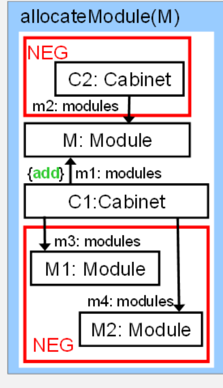












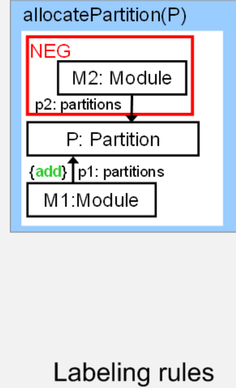


Fig. 3. Goals, Labeling rules and Global constraints of the running example

A subgoal or global constraint C described by the graph pattern GP is either a *positive* or *negative* constraint. A negative constraint is satisfied by a model ($M \models C$) if it does not have a match in M, formally $m : GP \longrightarrow M$, ($\nexists m$). While a positive constraint is satisfied if its representing graph pattern has a match in M; $m : GP \longrightarrow M$, ($\exists m$). A further restriction on positive constraints can be formulated by stating that they are satisfied iff their representing graph pattern has a *predefined* positive number ($Cardinality$) of matches, formally $|\{m : GP \longrightarrow M\}| = Cardinality$. In our running example all patterns are considered as *negative constraints*.

Labeling rules Labeling rules are described as graph transformation rules. A labeling rule l is enabled when the precondition LHS_l of its representing graph transformation rule is applicable to the underlying model M, formally $m : LHS_l \longrightarrow M, (\exists m)$. However, additional properties are used to refine the execution order and semantics of an enabled rule application:

- *Priority (integer: 0..100)*: Defines a precedence relation on labeling rules. It organizes the labeling rules into sets based on their priorities. In each state the solver selects its next step from the set with the highest priority. In our running example we use the same priority for all labeling literals.
- *Execution mode* (*forall* — *choose*): Defines whether a rule is simultaneous applied at all possible matches (forall) (as a single transition) or only once on a randomly selected single matching (choose). In the running example all labeling rules are using choose type execution mode.

Example. Our running example formalized as a CSP(M) problem is depicted in Fig. 3. The jobInstancewithoutPartition, partitionwithoutModule and modulewithoutCabinet subgoals formulating the *goal* describe that in a solution model each JobInstance, Partition and Module is allocated to a corresponding Partition, Module and Cabinet, respectively. For example, the jobInstancewithoutPartition subgoal captures its requirement using a double negation (NAC and negative constraint) stating that there are *no unallocated* job instance JI in the solution model. Similar double negation is used in case of the other two subgoals.

Global constraints formulate the safety and memory requirements. The partitionMemoryHigherThan0 pattern captures the simple memory constraint that all partitions must have higher than zero free memory. The safety requirement stating that a partition can only host jobs of one criticality level is captured by the partitionCriticalityLevelSimilar pattern. As it is a *negative constraint* it describes the (positive) case where the P1 partition holds two job instances J1 and J2 of a simple and a critical job Job1 and Job2, respectively. The criticalInstanceonSamePartition and criticalInstanceonSameModule patterns restrict in a similar way that no job instances J1 and J2 of a critical job Job can be allocated to the same partition P1 or module M1.

Finally, *labeling rules* describe the allocation operations. The allocatePartition graph transformation rule defines how a partition P can be allocated to a module M1. As a common technique in graph transformation based approaches, a negative application condition stating that the partition is not already allocated is used to indicate that the rule should only be used for unallocated partitions. On top of that the allocateModule rule

uses an additional NAC to forbid allocation of module M to cabinet C1 when two other modules M1 and M2 are already presented on C1, while the allocateJobInstance defines an additional attribute operation to decrease the free memory value MP of partition P1 by the required memory MJ of the allocated job J. The createModule rule simply creates a module M without any precondition.

Although not demonstrating in our ongoing example, our constraint framework is able to dynamically add and remove subgoals and labeling rules during the traversal of the state space in response to changes made in the original formulation of the problem. This allows to address problems which can change over time and solutions are relying on already made decisions such as reconfiguration of system components. In this case the requirement of the provided QoS (e.g., at least three service nodes must be running) by the system can vary over time and reconfiguration needs to be applied on the actual system state.

3.3 Solving CSP over Models

To traverse the search space of a constraint program introduced in Sec. 3.2, we define the solver as a virtual machine that maintains a 4-tuple (CG, CS, AM, LS) as a state. CG is called the *current goal*; CS is the *constraint store*; AM is the *actual model*; and finally LS is the *labeling store*. The (i) *current goal* stores the subgoals that still need to be satisfied; the (ii) *constraint store* holds all constraints the solver has satisfied so far while the (iii) *actual model* represents the underlying actual model and finally the (iv) *labeling store* contains all enabled labeling rules. An element in the labeling store is a pair (l, m), where l is a labeling rule and m is a valid match of its precondition LHS_l in AM; formally $m : LHS_l \longrightarrow AM$.

Initially, the CG, CS and LS are initialized with the *goal*, *global constraints* and the enabled *labeling rules* of the CSP(M) problem, respectively, while AM is set to the initial model. The solver proceeds by selecting an enabled *labeling rule* (l, m) and applies it to AM resulting in AM'. After each *labeling rule* application (and after initialization) CS is checked for consistency. In principle, whenever (i) a *global constraint* in CS is violated the solver backtracks, (ii) a *subgoal* in CG is satisfied by M it is moved to CS and (iii) vica-versa moved from CS to CG if it becomes unsatisfied and finally (iv) a successful termination is reached when CG becomes empty.

Formally, a transition in the search space is a pair of 4-tuples of $(CG, CS, AM, LS) \rightarrow (CG', CS', AM', LS')$, which describes a step between the two states. A transition is possible iff $\exists (l, m) \in LS$ where $AM \overset{l,m}{\Longrightarrow} AM'$ i.e., a labeling rule can be applied on the actual model for a certain match. A goal G can be proved if there exists a trajectory of individual steps $(CG, CS, M_0, LS) \rightsquigarrow (\emptyset, CS', M_s, LS)$ for a satisfiable constraint store CS. In other words, a solution model is found if there exists a sequence of labeling rule applications, that lead to an empty CG and satisfiable CS.

Example. Let us consider that our running example is in the initial state S_0 depicted in Fig. 4. The actual model is the initial model M_0 (detailed in Fig. 2(a)); the current goal CG contains the jobInstancewithoutPartition and the partitionwithoutModule subgoals; the constraint store CS holds all global constraints and the modulewithoutCabinet subgoal while the labeling store LS holds the following pairs: (allocateJobInstance,

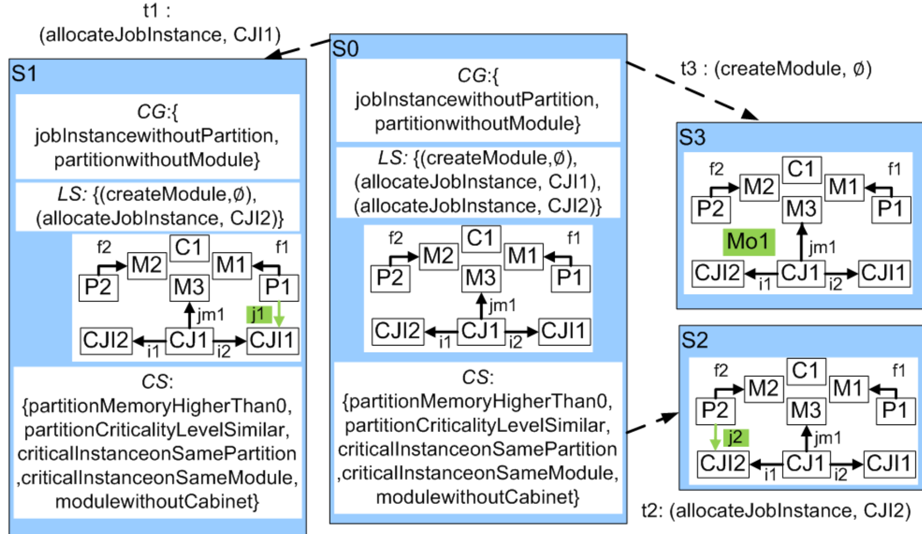


Fig. 4. Example State Space

CJI1), (allocateJobInstance, CJI2) and (createModule, ∅). The solver has three enabled labeling rules (transitions) $t1$, $t2$, $t3$ resulting in states $S1$, $S2$, $S3$. For example, $S1$ is traversed by applying the allocateJobInstance labeling rule on the critical job instance CJI1. In S_1 the actual model changed with an additional j1 jobs relation (highlighted in grey) between partition P1 and job instance CJI1; the current goal and constraint store did not change and contains the same elements as in S_0 while the labeling store changed to: (allocateJobInstance, CJI2) and (createModule, ∅). For easier readability, actual models of the states are depicted in Fig. 4 in a simplified way without type information e.g., the element CJI1: JobInstance is denoted as CJI1.

3.4 Search Strategies

Most algorithms for solving CSPs systematically traverse the possible search space. Such algorithms are guaranteed (in case of finite search space) to find a solution, if one exists, or to prove that the problem is irresolvable.

The most common algorithm for performing systematic search is *backtracking* based on depth-first search. Backtracking incrementally builds candidates to the solutions, and abandons each partial candidate (”backtracks”) as soon as it determines that it cannot possibly be completed to a valid solution. In our case it means that in the actual state a global constraint is violated or its labeling store is empty, thus backtracks the last applied step and continue with a different one. One of the main drawbacks of the simple backtracking algorithm is *thrashing*; i.e. repeated failure due to the same reason. Thrashing occurs because the backtracking algorithm does not identify the root cause of a conflict, i.e., the unsatisfiable global constraint or subgoal leading to a dead-end. Therefore, search in different parts of the search space keeps failing for the same reason.

In order to overcome trashing we implemented two additional search strategies:

Random Backjumping is a backtracking strategy based on the assumption that a traversal might be in a dead-end if no solution was found within a certain amount of time (deadline). When the solver exceeds this deadline, it jumps back to a state at least as high as the half of the actual depth of the search space tree. This way the solver can restart the traversal from an earlier state and continue on different random transitions. However, to keep the completeness of the traversal we implemented a simple policy introduced in [12] that is to increase the height of the backjump each time it is used. This approach is obviously not effective to prove unsatisfiability because all the runs except the last are wasted but has a good average performance in certain real scenarios.

Guided traversal by Petri net abstraction is a state space traversal strategy which conducts search towards the most promising candidate paths calculated according to a Petri net abstraction of graph transformation systems introduced in [13]. A marking of the derived (cardinality) Petri net abstracts from the actual structure of the corresponding model, and stores only the number of elements of each type (in the metamodel). This way, we solve an integer linear programming problem of the derived Petri net to obtain an optimal transition occurrence vector (storing only how many times a labeling rule needs to be applied) leading to a designated target state (formulated as a target submarking). Then the search strategy first explores those branches (i.e. labeling rule applications) which are consistent with this hint. If no solution is found on the level of CSP(M), then the next optimal transition occurrence vector candidate is derived, and the exploration of the CSP(M) problem continues.

Note that due to the abstraction, the transition occurrence vector might not represent a feasible trajectory in the search space of the CSP(M) problem. However, it provides a good lower bound on the minimal number of labeling rule applications required to reach a solution model if its corresponding solution submarking can be precisely estimated or calculated. The first transition occurrence vector calculated for our running example is $(2,1,1,1)$ meaning that to achieve a solution submarking derived from a solution model where all job instances and partitions are allocated, the allocateJobInstance rule has to be applied twice while the other three only once.

3.5 Optimization

To further reduce the size of the traversed state space we introduce two additional optimization techniques that complement our search strategies described in Sec. 3.4.

Look-ahead pattern Additional restrictions on the applicability of labeling rules can be formulated by incorporating a subset of global constraints called *look-ahead* constraints into the precondition (LHS) of rules. These constraints are validated in the precondition of labeling rules to prevent unnecessary steps which would violate these constraints. Currently, this is a manual hint by the designer, but in the future, we plan to automate this task by applying critical pair analysis [14] or transformations of graph constraints to preconditions [15].

In our running example the *allocateJobInstance* rule can be further restricted regarding the memory consumption of the JIns job instance making the *partitionsMemoryHigherThan* global (look-ahead) constraint obsolete. Its modified version with the extra check condition on the required and available memory is depicted in Fig. 5.

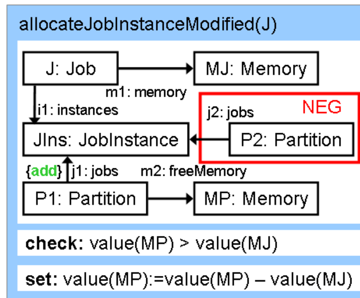


Fig. 5. Modified allocateJobInstance rule

Exception priority In order to explicitly restrict the number of application of labeling rules along a trajectory we introduced a priority class called *exception*. *Exception* rules have the lowest priority and will only be selected when no other labeling rules are enabled. In any trajectory if the number of applications of an exception rule exceeds its predefined value the solver backtracks and continues along another transition. Exception rules are used as hints by the solver to avoid state space explosion especially, when the Petri net based abstraction cannot predict the number of labeling rule applications for *element creation* rules without preconditions such as the createModule rule in the running example.

3.6 Implementation

We implemented an experimental solver for CSP(M) including all the techniques above on top the VIATRA2 model transformation framework, which offers efficient rule- and pattern-based manipulation of graph models by the means of graph transformation. In order to implement the solver using graph based state representation we had to address the problems of *constraint evaluation*, *typed graph comparison* and *backtracking*.

For effective *evaluation* of constraint satisfiability we rely upon the incremental pattern matcher component [16] of the framework. In case of incremental pattern matching, the matches of a pattern are stored to be readily available in constant time, and they are incrementally updated when the model changes. As matches of patterns are cached, this reduces the evaluation of constraints and preconditions of labeling rules to a simple check. This way, the solver has an incrementally maintained up-to-date view of its constraint store and enabled labeling rules. Furthermore, incrementality provides an efficient *constraint propagation* technique to immediately detect constraints violations after a labeling rule is fired.

As for *backtracking* between states, we implemented 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 neighbour states. This feature is also useful in problems that require solutions that are ”nearest” to a given initial model (e.g. for reconfiguration rules).

For *comparison of graphs* we adapted the DSMDIFF [17] 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. It is also important to mention that to keep the memory consumption low, we serialized *already visited states* as strings and applied the algorithm directly on them.

The introduced solver is already in use in the context of the DIANA [18] European project as its underlying allocation engine for a system-level integration scenario.

4 Evaluation

To evaluate the performance of our CSP(M) solver, we carried out experiments[1] based on our running IMA allocation example. We assume that we have to allocate different software workloads (functionalities) on a system of three modules (which corresponds to the avionics architecture used in the DIANA project).

Each row in Table 4 defines a software workload allocation test case. The *Simple Job*, *Critical Job*, and *Partition* rows define the actual number of software components to be allocated where critical jobs are separated based on their redundancy scheme into double (DMR) and triple (TMR) modular redundancy. *All Job Instances* represents the total number of job instances to be allocated. For our initial measurement (denoted by *ATTR*) we assume that each job requires the same amount of memory (30 units) and each partition offers the same free memory (300 units).

Due to the random strategy of our solver we considered an allocation *completed* if a solution was found within 200 seconds. In each case we executed the solver ten times and present the number of *Completed Allocations*. *Runtime* performance and traversed *State Space* size of the completed allocations are also presented by their minimum (*min*), maximum (*max*) and average (*avg*) values for each test case.

During the analysis and profiling of our implementation we have discovered that the performance bottleneck in our system is mainly related to the model management component of the underlying VIATRA2 transformation framework (which is obviously not optimized for constraint solving purposes). In almost all cases we have observed that core attribute manipulation functions (e.g., setValue) are the most time consuming. This is due to the low-level notification mechanism that keeps the incremental pattern

Table 1. Runtime performance of the IMA allocation problem

	Simple job #	Critical Job		Partition #	All Job instances	Completed Allocation (out of 10)	Runtime [msec]			Traversed States #		
		DMR	TMR				min	max	avg	min	max	avg
ATTR.	3	2	4	4	19	10	1078	196469	66991	64	13984	4802
	5	2	5	5	24	6	4212	145823	81689	146	12632	8296
	16	2	5	5	35	1	-	50678	-	-	237	-
NON ATTR.	5	2	5	5	24	10	879	156322	41274	72	17639	4278
	16	2	5	5	35	4	12023	195672	57837	174	1311	404
	20	5	7	7	51	1	-	102452	-	-	276	-

matcher up-to-date after changes in the model space, which is more effective for graph manipulations rather than attribute changes.

Therefore we also evaluated our approach without attribute manipulation (i.e., memory requirements) on the running example denoted by *NON ATTR.*. In order to solve a conceptually similar problem we defined an additional global constraint stating that a partition can not affiliate more than ten job instances. Results show that (i) in both cases solutions were found traversing only a small number of states compared to the size of the problem, (ii) the *NON ATTR.* implementation scales almost up to twice the size in the number of job instances to allocate and (iii) due to the heuristic character of the state space traversal the runtime performances can vary up to two order of magnitudes.

Our measurement shows that our constraint solver based upon incremental pattern matching is able to solve non-trivial problems of model oriented constraints. On the other hand further investigations have to be directed to combine them with constraints over regular attributes.

5 Related Work

Applications of CSP in MDD. While constraint satisfaction techniques have been successfully applied in the context of MDD. [19] proposes an approach for partial model completion based on constraint logic programming. [2] support efficient domain specific modeling by transforming constraints to a Prolog representation. In [1], poor design patterns are detected by using off-the-shelf CSP techniques and tools. [20] defines an interactive guided derivation algorithm to assist model designers by providing hints about valid editing operations that maintain global correctness of models.

In the context of model transformations, [21] proposes constraint solving as a graph pattern matching strategy. [3] proposes Constraint Relation Transformation an extension of QVT Relations with numerical constraints by integrating local numerical constraint solving (over attributes of model elements).

Recent approaches like [22–24] aim at automatically creating instance models, which conform to a given metamodel and a set of constraints. This model generation problem is solved by existing back-end tools like Alloy, or by a dedicated theorem prover for Horn-like clauses as in [24]. This problem can also be interpreted as a special (restricted) CSP problem without numeric constraints on attributes.

In all these papers, constraint satisfaction techniques are used to assist model-driven development. The main innovation of our work is just the opposite: it investigates how model transformation techniques can contribute to solve complex constraint satisfaction problems over complex structural constraints and dynamic labeling rules.

State Space Exploration for GT. There are several state space exploration approaches [25, 26] to analyze graph transformation systems. Common in these solutions that they store system states as graphs and directly apply transformation rules to explore the state space similar to our approach. Their main difference is that they use an exhaustive state space exploration to verify certain conditions in the graph transformation system, while our approach rely on guided traversals.

[1] For our experiments, we used a average PC with Core Duo@1.8 GHz and 2GB RAM running Windows XP and Java SDK 1.6

CSP-specific Research in the field of constraint satisfaction programming has been conducted towards flexible and dynamic constraints [6, 27]. Our approach shows similarities with both approaches as (i) it also allows to add (or remove) additional constraints during the solution process as defined in the dynamic extension, and (ii) can give support for cost based optimization defined over the constraint (flexible) even in the case of complex structural constraints.

6 Conclusion and Future Work

In the current paper, we have presented a novel approach defining constraint problems directly over models (denoted shortly as *CSP(M)*) using graph transformation rules and graph patterns. As a distinctive feature from a CSP point of view, we extended traditional labeling by using model manipulation as provided by graph transformation to dynamically create and delete model elements. Furthermore, not demonstrated in the current paper but our framework also allows to *dynamically add/remove* subgoals and labeling rules to alter the constraint problem to address problems defined in dynamic constraint satisfaction programming [27].

We have also built (and initially evaluated) a prototype solver implementation on top of the VIATRA2 model transformation framework using incremental pattern matching that provides an efficient *constraint propagation* technique to immediately detect constraint violation. Moreover, the solver integrates various strategy (e.g. random backjumping, directed search) to guide the state space traversal.

As the main innovation, we argued that model transformation technology can efficiently contribute to formulate and solve constraint satisfaction problems with complex structural constraints and dynamic labeling rules.

In the future, we plan to investigate (i) how can traditional constraint programming concepts can be combined with our approach to effectively handle constraints over attributes, (ii) further state space optimization by automatic detection of look-ahead pattern based on critical pair analysis and finally (iii) other structural constraint based frameworks such as Alloy for a detailed comparison.

References

1. El-Boussaidi, G., Mili, H.: Detecting patterns of poor design solutions using constraint propagation. In: MoDELS '08: Int. Conference on Model Driven Engineering Languages and Systems. (2008) 189–203
2. White, J., Schmidt, D., Nechypurenko, A., Wuchner, E.: Introduction to the generic eclipse modelling system. Eclipse Magazine (6) (2007) 11–18
3. Petter, A., Behring, A., Mühlhäuser, M.: Solving constraints in model transformation. In: ICMT'09: International Conference on Model Transformation, Zürich, Switzerland (2009)
4. Intelligent Systems Laboratory, Swedish Institute of Computer Science: Sicstus User's manual (2009) http://www.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf.
5. Official website of ILOG Solver: http://www.ilog.com/products/cp/.
6. Miguel, I., Shen, Q.: Dynamic flexible constraint satisfaction. Applied Intelligence, 13(3) (2000) 231–245

7. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Chapter: Algebraic Approaches to Graph Transformation. Volume 1: Foundations. World Scientific (1997)
8. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming **68**(3) (October 2007) 214–234
9. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2**(3) (October 2003) 187–210
10. Rensink, A.: Representing first-order logic using graphs. In: ICGT 2004: 2nd International Conference on Graph Transformation, Rome, Italy. (2004) 319–335
11. Weld, D.S.: An introduction to least commitment planning. AI Magazine **15**(4) (1994) 27–61
12. Baptista, L., Margues-Silva, J.: Using randomization and learning to solve hard real-world instances of satisfiability. In: CP '00: 6th International Conference on Principles and Practice of Constraint Programming. (September 2000) 489–494
13. Varró-Gyapay, S., Varró, D.: Optimization in graph transformation systems using petri net based techniques. Electronic Communications of the EASST **2** (2006)
14. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: ICGT '02: International Conference on Graph Transformation. (2002) 161–176
15. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Funda. Inf. **74**(1) (2006) 135–166
16. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT'08, 3rd Int. Workshop on Graph and Model Transformation. (2008)
17. Lin, Y., Gray, J., , Jouault, F.: Dsmdiff: A differentiation tool for domain-specific models. European Journal of Information Systems, Special Issue on Model-Driven Systems Development 16(4) (2007) 349–361
18. Official website of the Distributed equipment Independent environment for Advanced avioNics Applications (DIANA) European project: `http://diana.skysoft.pt`.
19. Sen, S., Baudry, B., Precup, D.: Partial model completion in model driven engineering using constraint logic programming. In: INAP'07: International Conference on Applications of Declarative Programming and Knowledge Management, Würzburg, Germany (2007)
20. Janota, M., Kuzina, V., Wasowski, A.: Model construction with external constraints: An interactive journey from semantics to syntax. In: MoDELS '08: Int. Conference on Model Driven Engineering Languages and Systems. (2008) 431–445
21. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: 6th Int. Workshop on Theory and Application of Graph Transformations. (2000) 238–251
22. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling (2009)
23. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted ocl constraints into graph constraints for generating meta model instances by graph grammars. Electron. Notes Theor. Comput. Sci. **211** (2008) 159–170
24. Jackson, E., Sztipanovits, J.: Constructive techniques for meta and model level reasoning. In: MoDELS '07: Int. Conference on Model Driven Engineering Languages and Systems. (October 2007) 405–419
25. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE). (2004) 479–485
26. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: TACAS '06: Tools and Algorithms for the Construction and Analiysis of Systems. (2006) 197–211
27. Schiex, T.: Solution reuse in dynamic constraint satisfaction problems. In: In Proceedings of the 12th National Conference on Artificial Intelligence, AAAI Press (1994) 307–312