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

Dynamic Constraint Satisfaction Problems over Models [⋆]

Ákos Horváth, Dániel Varró

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

The date of receipt and acceptance will be inserted by the editor

Abstract In early phases of designing complex systems, models are not sufficiently detailed to serve as an input for automated synthesis tools. Instead a design space is constituted by multiple models representing different valid design candidates. Design space exploration aims at searching through these candidates defined in the design space to find solutions that satisfy the structural and numeric design constraints and provide a balanced choice with respect to various quality metrics. Design space exploration in an MDE context is frequently tackled as specific sort of constraint satisfaction problem (CSP).

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 capture design space exploration in many MDE applications with complex structural constraints expressed over the underlying models.

In this paper, we interpret flexible, and dynamic constraint satisfaction problems directly in the context of models. These extensions allow the relaxation of constraints during a solving process and address problems that are subject to change and require incremental re-evaluation. Furthermore, we present our prototype constraint solver for the domain of graph models built upon the VIATRA2 model transformation framework, and provide an evaluation of its performance with comparison to related tools.

Key words constraint satisfaction programming, graph transformation, dynamic constraint satisfaction programming, flexible constraint satisfaction problem

[⋆] This work was partially supported by the EC FP6 DIANA (AERO1-030985) , the SecureChange (ICT-FET-231101) European Projects, the Hungarian CERTIMOT (ERC_HU_09) project and the Janos Bolyai Scholarship.

1 Introduction

Evolutionary design space exploration Design space exploration is a process to analyze several "functionally equivalent" implementation alternatives, which meets all design constraints in order to identify the most suitable design chosen based on various quality metrics such as performance, cost, power, and dependability. Typically, the best solution is *flexible* in the sense that it provides a trade-off between the optimal solutions with respect to a single quality metrics. Design space exploration is thus a challenging problem in many application areas including critical embedded systems and IT systems management or cloud computing, where model-driven engineering (MDE) techniques have already been quite popular. Design space exploration in an MDE context is frequently tackled as specific sort of constraint satisfaction problem [1].

Traditionally, most of these constraints and quality attributes were numeric in nature to express time, throughput, budget, memory limits, etc. However, the birth of modular software architectures in critical systems (like AUTOSAR [2] in the automotive or IMA in the avionics domain) introduced a novel type of *complex structural constraints*, which express connectivity restrictions for the graph-based model of the system under design. Complex structural constraints may include restrictions on allocation (e.g. separate critical components from non-critical ones), communication (e.g. use a secure communication channel between two channels), etc.

In addition, in many practical scenarios (like IT systems management or cloud computing), design space exploration is further complicated by the *continuous evoluation of the system*, which imposes further constraints and quality metrics. For instance, in IT systems management and service-oriented architecture, both the actual system, the quality of service requirements and measured parameters, and reconfiguration policies may change quite frequently. Moreover, design space exploration also needs to incorporate the "distance" between the current and the designated configuration, as a reconfiguration to the mathematically "optimal" system configuration may be too complex or costly to implement.

In the paper, we aim to tackle *evolutionary design space exploration* to flexibly identify the most suitable design meeting complex structural constraints and numeric constraints where the underlying constraints may evolve in time, and the evolution of the best design is also restricted by allowed operations and/or quality metrics.

Solving the constraint satisfaction problem over models The aim of the constraint satisfaction problem (CSP) is to find a solution to a set of constraints that impose conditions which have to be satisfied by a set of variables. Each variable takes its value from a predefined 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 such as to apply design patterns [3], to support domain-specific modeling [4] or model transformations [5]. As a commonality, all these approaches translate high-level models to an existing, off-the-shelf constraint solver (like e.g. [6,7]) 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, most approaches disallow the dynamical addition or retraction of constraints [8]. Furthermore, mapping graph models obtained in model-driven engineering to variables with finite domain can be a non-trivial task, especially when considering the evolution of models. As a summary, existing constraint solvers fail to adaquately handle flexible and dynamic structural constraints over graph-like models, which is necessitated for evolutionary design space exploration.

Model driven techniques for solving the CSP over models Since model driven engineering techniques are widely used in our designated application areas, it is worth evaluating how existing model or graph-based techniques could be used to solve dynamic and flexible constraint satisfaction problems with complex structural constraints.

Unfortunately, traditional model transformation tools (like ATL [9]) do not support backtracking when executing a model transformation for performance reasons, and thus they cannot traverse alternate transformation paths. Rare exceptions (like PROGRES [10] which support backtracking) need complex control structures to drive the transformation, lack support for the efficient exploration of an alternate path after backtracking, and fail to handle dynamic changes of constraints or rules.

Sophisticated model or graph based verification tools (like GROOVE [11] or Alloy [12]) need to store the entire state space during traversal, which is very resource consuming. Furthermore, they usually use generic bounded state space traversal strategies, which makes it difficult to fine tune and effectively control how the most promising next candidate should be selected with respect to the CSP problem itself.

In [13], we first introduced constraint satisfaction problem over graph-based models (abbreviated as CSP(M)) to capture traditional design space exploration using *graph patterns* to define structural (first-order logic) constraints, and *graph transformation rules* [14] as labeling operations. Furthermore, we provided a prototype solver capable of solving complex structural constraints built upon advanced incremental model transformation technology to efficiently continue search upon backtracking.

Contributions of the paper In the current paper, we extend [13] in the following way. First, we define two extensions to the CSP(M) formalism to address evolutionary design space exploration by introducing *flexible CSP*, and *dynamic CSP* [8] directly in the context of models. (1) Flexible CSP supports the relaxation of constraints (referred to as soft constraints) to accept solutions that do not satisfy all given constraints. Our flexible CSP(M) approach uses a numeric weight function to capture the satisfiability criteria of the solution state, thus allowing the relaxation of constraints on a fine-grained state-by-state basis. (2) Dynamic CSP addresses the case when the original problem definition itself is changed (e.g. a constraint, or operation is added or removed), and our intention is to find a new solution in an incremental way, i.e. without restarting the solving process from scratch.

As a summary, our approach now allows to solve dynamic constraint satisfaction problems over models where dynamic changes include to (i) add/remove constraints to the CSP problem while partially reusing solution from the original problem, (ii) modify the domain of the variables during search and (iii) define constraint problems with relaxable soft constraints.

Additionally, we enhanced our prototype constraint solver based on the VIATRA2 [15] model transformation framework to support flexible and dynamic constraints. A first experimental evaluation of the prototype solver is also carried both on classical and flexible/dynamic CSP(M) using two dynamic allocation problems taken from the avionics and the cloud computing domains. We also compare the performance of our CSP solver with existing (industrial and academic) tools.[1]

The relevance of the paper to model-driven engineering is three-fold. (1) First, it defines dynamic and flexible constraint satisfaction problem with complex structural and numeric constraints over graph-based models as means to formalize evolutionary design space exploration problems. (2) It provides an intuitive way to capture evolutionary design space exploration problems using techniques (e.g. graph patterns and graph transformation rules) which are closely related to MDE best practices. (3) It proposes actual solving strategies using incremental model transformation techniques, which are especially suitable for automating dynamic and flexible constraint solving for complex structural constraints.

The rest of the paper is structured as follows. In Sec. 3 we briefly introduce the concept of metamodeling, graph trans-

[1] Compared to [13], the current paper thus defines dynamic and flexible CSP(M) problems, it provides a new case study, a comparitive performance evaluation, a more extensive evaluation of related work, and more implementation details.

formation and constraint satisfaction problems. Section 4 proposes our graph pattern and transformation based constraint solver, while Sec. 5 extends the formalism with support for flexible and dynamic constraint problem definition. Section 6 introduces optimization and implementation details of our solver and performance measurements are evaluated in Sec. 7. Finally, related work is assessed in Sec. 8 and Sec. 9 concludes the paper.

2 Motivation

System modeling and *design space exploration* are key issues in the design and synthesis of complex embedded and IT systems. Model Driven Engineering has already contributed languages and tools for capturing high-level system models and design constraints using graph-based models. However, in early phases of design, models are not sufficiently detailed to serve as an input for automated synthesis tools. In fact, in practice, the design space is constituted by multiple models representing different valid design candidates. The design space exploration process aims at searching through these candidates defined in the design space to find solutions that satisfy the requirements (constraints) and provide a balanced choice with respect to (a combination of) quality metrics. These complex exploration processes involve both critical design decisions made by the system architect and semi-automated techniques.

To introduce our CSP(M) formalism and demonstrate how it can help in solving various design space exploration problems, we selected two motivating allocation case studies from the mission critical embedded system in Sec. 2.1 and the cloud infrastructure Sec. 2.2 domains, derived from our ongoing research projects. The embedded system case study describes a typical design space exploration problem with static, non-flexible constraints, while the cloud case study represents evolutionary design space exploration, where the requirements of system evolves and the solver needs to modify its constraint set according to the changes. Throughout the paper, we will use these motivating case studies as our running examples and benchmarks.

2.1 Case Study: Allocation of an IMA system

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

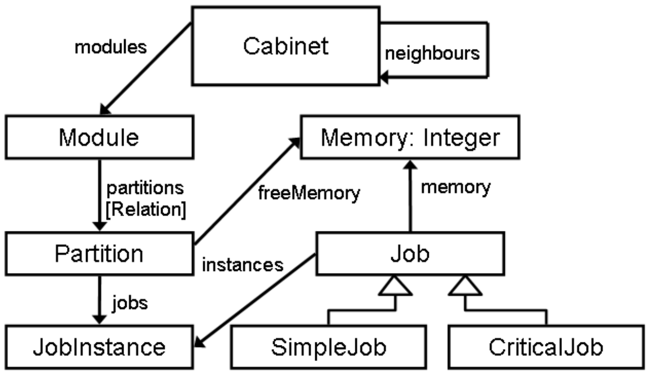


Figure 1 Metamodel of an IMA architecture

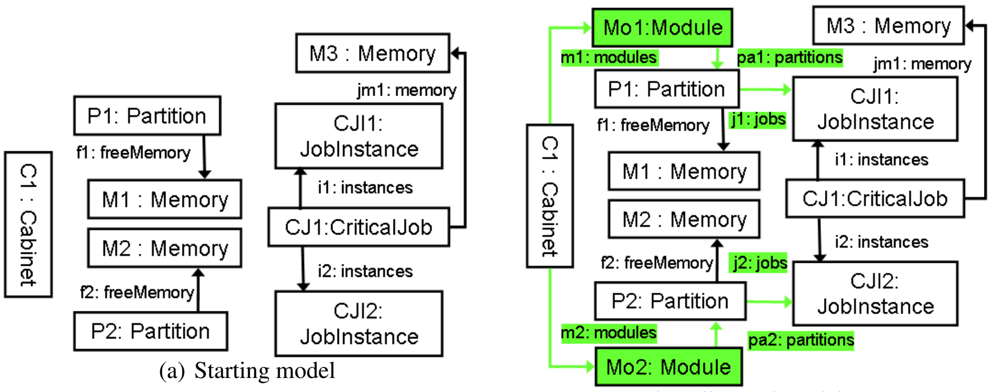
for a maximum (in our example) two modules used to physically distribute elements of the system. Additionally a certain number of safety related requirements will also have to be satisfied: (i) a partition can only host jobs of one criticality level and (ii) different instances of a certain critical job cannot 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 [16] in Fig. 1. Newly created elements are highlighted in grey.

2.2 Case Study: Cloud Allocation

Let us assume a synthetic cloud platform providing a database service. The system is composed of *virtual* and *physical servers* running a heterogeneous database infrastructure. Virtual servers are hosted by physical ones, where each physical server can host a predefined number of virtual ones. In the current configuration the cloud uses three different types of fictitious databases to provide its service, namely: *DB_P*, *DB_C* and *DB_V*. A server can host at most one database at once and a physical server can either hold virtual servers or a database. Each database has different performance characteristic with regard to its underlying server captured by the following rules: (i) in general, DBs on virtual servers are performing almost half as fast as on physical, (ii) DB_V is slightly faster than the other two on virtual servers and also DB_C performs better than DB_P, (iii) however, DB_P is almost twice as fast on a physical server than the others and finally (iv) DB_C supports rapid clustering, where two instances can form a cluster-pair that counts as an additional virtual instance in their overall performance. The VPM metamodel of the cloud case study is depicted in Fig. 3.

The task is to allocate databases to produce the required overall performance over a static physical server infrastructure, where both the number of licences for each database types and the required overall performance are predefined.



(a) Starting model (b) Allocated model

Figure 2 Example IMA system

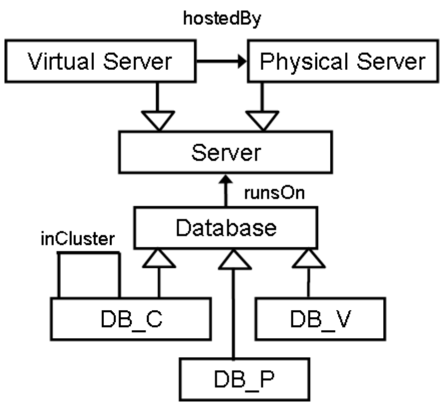


Figure 3 Metamodel of the Cloud example

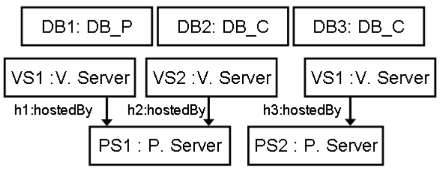
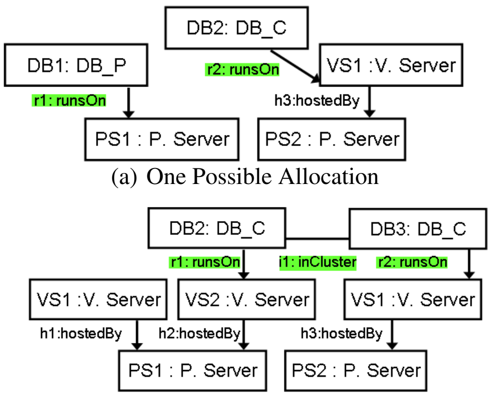


Figure 4 Starting Model of a Cloud System

The main difference between this and the IMA allocation is that in the current case not necessarily all databases are needed to be allocated to achieve a solution state. However, as business needs require, changes may occur in the problem definition over time that would require the reallocation of the databases.

A simple cloud configuration composed of two physical servers with one and two hosted virtual servers along with one DB_P and two DB_C databases is depicted in Fig. 4. Two possible solutions with a required overall performance of 15 are also depicted in Fig. 5(a) and Fig. 5(b). The solution in Fig. 5(a) uses the DB_P and a single DB_C database running over a physical and virtual server, respectively, producing an overall performance of 19. The other solution uses the two DB_C databases in a cluster-pair producing exactly the required performance of 15. For easier readability non-



(a) One Possible Allocation
(b) An Other Possible Allocation

Figure 5 Allocated example Clouds

allocated databases are not shown in the solution figures and newly created elements are highlighted in grey.

3 Background

In order to introduce our approach this section briefly outlines the basics of graph transformation.

3.1 Graph Patterns and Graph Transformation

Graph patterns (GP) are frequently considered as the atomic units of model transformations [15]. They represent conditions that have to be fulfilled by a part of the underlying instance model. The VIATRA2 notation in particular, describes them as a disjunction of pattern bodies $GP = \vee_{i \in I} PB_i$, where a pattern is fulfilled if at least one of its pattern body is fulfilled. *Pattern bodies* $PB = (SC, AC, \vee_{j \in J} NAC_j)$ consist of

– *structural conditions* SC prescribing the existence of type conformant nodes and edges. These conditions describe a graph that needs to be matched to a subgraph of the underlying model in order to be fulfilled.

– *attribute conditions* (*AC*) prescribe boolean conditions over the attributes of the matched elements (marked by the check keyword). Check conditions are similar to terms in traditional programming languages and usually describe conditions over integer and string values.
– 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. For the satisfaction of a negative application condition there should not be a match extending the match of the parent pattern. A graph pattern can have arbitrary number of negative application conditions. Additionally, negative conditions can be embedded into each other to an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [17].

A *match* m for a graph pattern $GP = \vee_{i \in I} PB_i$ in an instance model M denoted by $m : GP \longrightarrow M$ means that there exists a pattern body $PB_i = (SC_i, AC_i, NAC_{i,j})$ where: (i) $\exists m : SC_i \mapsto M$ there exists an injective, type conformant total morphism m from the graph defined by its structural conditions to the instance model, (ii) $\nexists j \in J; m' : NAC_{i,j} \mapsto M$ there is no morphism for any of its embedded NACs that extends the match of the pattern body PB_i and (iii) all attribute conditions AC_i are fulfilled by m.

Graph transformation [14] 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 actions *AMA*. The *RHS* is a simple graph, ie., a restricted pattern that can have only one pattern body that prescribes only structural conditions and has no embedded NACs. Similarly, to the concept of attribute conditions in graph patterns, graph transformation rules can manipulate attributes, where manipulation actions are described by the *AMA*. These actions are usually, simple attribute value manipulation operations like assignments, integer addition, etc.

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 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 $G \overset{r,m}{\Longrightarrow} H$, where H is the resulting model; r and m denote the applied rule and the matching, respectively.

The complete formal description of the VIATRA2 graph transformation notation is described in [15].

Example. Sample graph patterns and transformation rules are depicted in Fig. 7. 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.

4 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.

4.1 Constraint Satisfaction Problem specification

To introduce the basis of our approach Sec. 4.1.1 introduces finite domain constraint satisfaction problems

4.1.1 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.

4.2 CSP(M): Constraint Satisfaction Problem over Models

An overview of the input and output artifacts of our CSP(M) formalism is depicted in Fig. 6. A CSP(M) problem consists of:

– An *initial model* representing the starting point of the problem. With the initial model the user can put additional knowledge into the system to give hint (e.g., in the form of a partial solution) to the solving process. This is a typical use case in design space exploration of embedded systems, where the system architect either reuses earlier solutions or use standard architecture patterns to start the evaluation from. Please note, that the initial model can also be empty.

– The *goal* representing the conditions that need to hold in a valid solution of the problem. For example, in model-based modular embedded software design this can mean a certain level of redundancy that the system needs to implement or a connectivity restriction on the communication network of the system.
– A set of *global constraints* representing a special subset of constraints that needs to be satisfied by all models (states) traversed during the search for a solution. The use of global constraint is not mandatory but they can effectively prune the search space by early detection of invalid models. For example, when allocating software components a global constraint can define the maximum number of allowed components on a CPU, pruning out all invalid models where too many components were allocated to a CPU.
– A set of *labeling rules* capturing the permitted operations. These labeling rules are conceptually similar to operations in planner algorithms [18], which aim to restrict the possible transitions in the search space. For example, in case of software component allocation a labeling rule can describe the underlying model manipulations required to allocate a free (non-allocated) SW component to a CPU.

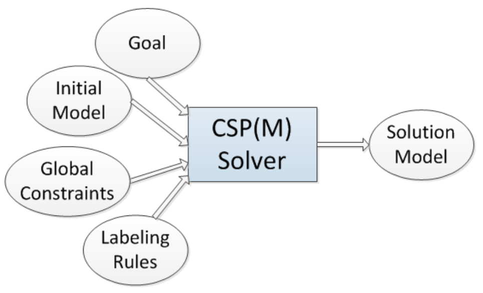


Figure 6 Overview of CSP(M) Solver

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:

1. $M_0 \rightsquigarrow M_s$; there exists a trajectory $T : M_o \xrightarrow{l_1} M_1 \xrightarrow{l_2} .. \xrightarrow{l_n} M_s$ where $i = 1..s : l_i \in L$. Informally, M_s is reachable from M_0 through a sequence of applied labeling rules in trajectory T.
2. $\forall G_i \in G : M_s \models G_i$; M_s satisfies all subgoals G_i
3. $\forall C_i \in C : M_s \models C_i$; M_s also satisfies all global constraints C_i
4. $\forall M_i \in T, \forall C_j \in C : M_i \models C_j$; along the trajectory T from the initial to the solution model all visited model M_i satisfies each global constraint.

As models in MDE are usually described as graphs we instantiate our formalism on graph transformation a well-known model transformation language. In our instantiation both the initial and solution models are defined by typed graphs over a given metamodel. Based on this metamodel we use graph patterns to declaratively define both goals and global constraints. This way constraints are directly defined over the problem domain and no mapping to other formalisms (e.g., finite domain constraint logic programming) is required. Finally, model manipulation operations described by the labeling rules are captured by graph transformation rules. Altogether, the complete problem can be defined in a declarative manner using model driven techniques making the whole formalism intuitive, especially for complex structural constraints.

Additionally, this instantiation allows to directly apply the GT defined labeling rules on the underlying (graph) models, giving way to a better insight of the solving process with potential feedback on (i) valid and invalid goals and global constrains and (ii) applicable labeling rules in each states, allowing easier traceability of the solving process.

For the concrete definition of CSP(M) problems we used the VIATRA2 [15] transformation language. However, this formalism can also be incorporated into other modeling approaches such as MOF models, OCL constraints and QVT rules.

4.2.1 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.

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 $\nexists m : GP \longrightarrow M$. While a positive constraint is satisfied if its representing graph pattern has a match in M; $\exists m : GP \longrightarrow M$. A further restriction on positive constraints can be formulated by stating that they are satisfied iff their representing graph pattern has a *predefined* minimum number of of matches (*Cardinality*), formally $|\{m : GP \longrightarrow M\}| \geq Cardinality$. In our IMA case study all patterns are considered as *negative constraints*.

4.2.2 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 $\exists m : LHS_l \longrightarrow M$. 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 IMA case study 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 IMA case study all labeling rules are using choose type execution mode.

Example. Our IMA case study formalized as a CSP(M) problem is depicted in Fig. 7. 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 imply 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.

4.3 Solving CSP over Models

To traverse the search space of a constraint program introduced in Sec. 4.2, 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; (ii) the *constraint store* holds all constraints the solver has satisfied so far while (iii) the *actual model* represents the underlying actual model and finally (iv) the *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 all 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 IMA case study is in the initial state S_0 depicted in Fig. 8. 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 elements: (allocateJobInstance, CJI1), (allocateJobInstance, CJI2) and (createModule, $\emptyset$). 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 contain the same elements as in S_0, while the labeling store changed to: (allocateJobInstance, CJI2) and (createModule, $\emptyset$). For easier readability, actual models of the states are depicted in Fig. 8 in a simplified way without type information e.g., the element CJI1: JobInstance is denoted as CJI1.

5 Flexible and Dynamic Constraint Satisfaction Problems over Models

Our formalism also supports *dynamic* and *flexible* constraint satisfaction problems. In the current section we briefly introduce how these different CSP definitions are adopted in CSP(M).

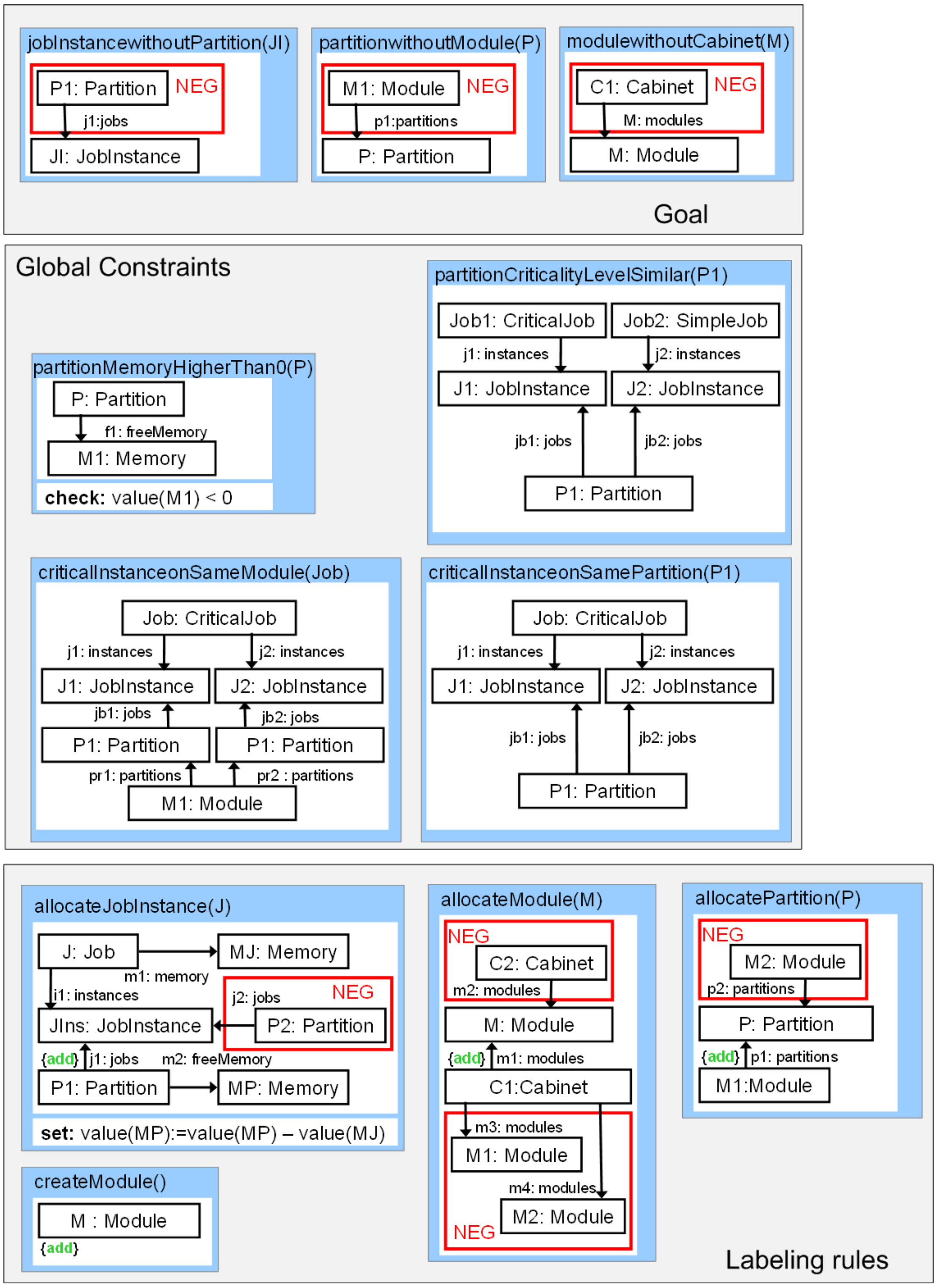


Figure 7 Goal, Labeling rules and Global constraints of the IMA case study

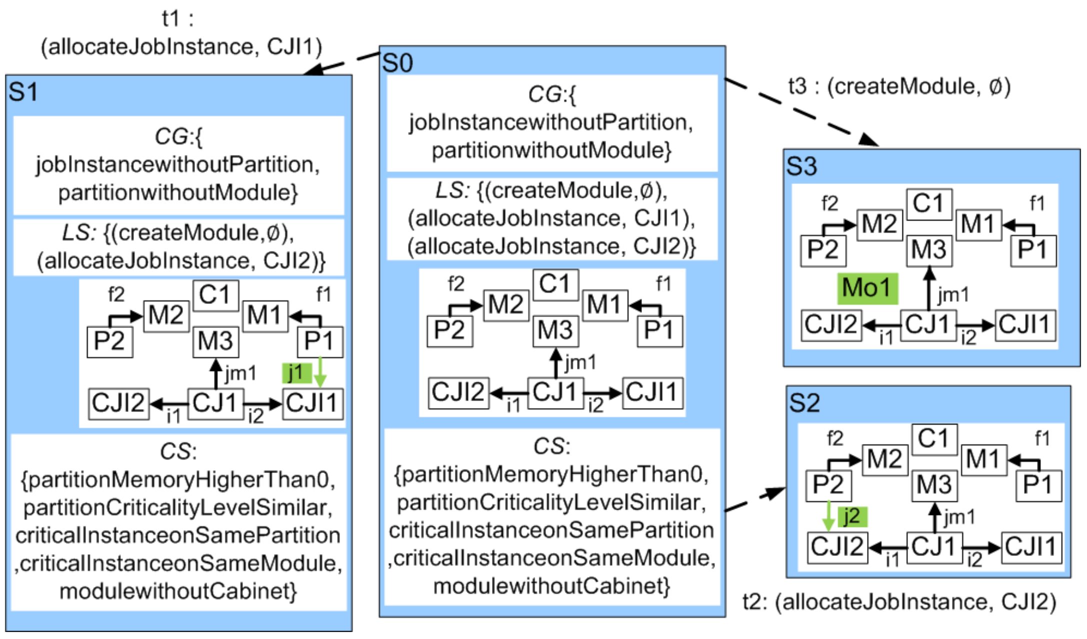

t1 : (allocateJobInstance, CJI1)
S1
CG:{ jobInstancewithoutPartition, partitionwithoutModule}
LS: {(createModule,∅), (allocateJobInstance, CJI2)}
CS: {partitionMemoryHigherThan0, partitionCriticalityLevelSimilar, criticalInstanceonSamePartition ,criticalInstanceonSameModule, modulewithoutCabinet}
S0
CG:{ jobInstancewithoutPartition, partitionwithoutModule}
LS: {(createModule,∅), (allocateJobInstance, CJI1), (allocateJobInstance, CJI2)}
CS: {partitionMemoryHigherThan0, partitionCriticalityLevelSimilar, criticalInstanceonSamePartition ,criticalInstanceonSameModule, modulewithoutCabinet}
t3 : (createModule, ∅)
S3
Mo1
S2
t2: (allocateJobInstance, CJI2)
f2 f1 C1 M1 M2 M3 P1 P2 jm1 j1 j2 CJI1 CJI2 CJ1 i1 i2

Figure 8 Example State Space

5.1 Flexible Constraint Satisfaction Problems

Classical constraint satisfaction techniques support only *hard constraints* specifying exactly the allowed combinations. Hard constraints are *imperative* (a valid solution must satisfy all constraints) and *inflexible* (constraints are either entirely satisfied or violated). In order to overcome these weaknesses flexible CSPs introduced *soft constraints* [19] to relax these assumptions and allow solutions that do not satisfy all constraints.

One well-known approach, called *weighted CSP* [20], introduces the use of weights attached to each constraint indicating its relative importance. A solution is acceptable if the sum of the weight of the satisfied constraints is higher than a predefined value.

By extending the classical CSP(M) formalism (in Sec. 4.2) we define *weighted CSP(M)* as $(M_0, C, G, L, f_w, S_w) : M_s$. S_w is the predefine sum weight required for a solution model to be satisfied and $f_w : G_s, M \to \mathbb{N}$ is a weight function, which takes as input a subgoal $G_s \in G$ and a model M and produces the weight of the subgoal G_s in the model M. The weight function is usually specific to each problem domain and can use the additional attributes of the satisfiability criteria of the subgoals. For example, this can be the number of matches in a specific state or the cardinality value of a *positive* subgoal.

The definition of a solution model M_s changes in the following way:

1. $M_0 \rightsquigarrow M_s$; there exists a trajectory $T : M_o \xrightarrow{l_1} M_1 \xrightarrow{l_2} .. \xrightarrow{l_n} M_s$ where $i = 1..s : l_i \in L$. Informally, M_s is reachable from M_0 through a sequence of applied labeling rules in trajectory T.
2. $$\sum_{\{G_i | G_i \in CS \wedge M_s \models G_i\}} f_w(G_i, M_s) \geq S_w \qquad (1)$$
   In a solution model M_s, the summarized weight of the satisfied subgoals G_i has to be greater or equal to the predefined S_w value.
3. $\forall C_i \in C : M_s \models C_i$; M_s also satisfies all global constraints C_i
4. $\forall M_i \in T, \forall C_j \in C : M_i \models C_j$; along the trajectory T from the initial to the solution model all visited model M_i satisfies each global constraint.

This way the solving process described in Sec. 4.3 is slightly modified to; a solution model is found if there exists a sequence of labeling rule applications, that leads to a constraint store that fulfills the inequality defined in 2 and contains all global constraints.

5.2 Dynamic Constraint Satisfaction Problem

A further limitation of classical CSP is in its assumption of a *static* problem. This means that once the constraints have been defined they are fixed for the duration of the solving process. However, in certain cases problems are subject to change either as a solution is being constructed or while the constructed solution is in use. Classical CSP usually can deal with this situation by considering the changed problem as an entirely new problem which needs to be solved from scratch.

Dynamic constraint satisfaction [21] addresses this kind of problems and allows to add and remove constraints from the actual problem definition as necessary. However, to utilize the advantage of dynamic constraint manipulation and re-use partial solutions obtained for a problem before it changes, additional techniques [22] are required.

In our case, it is possible to dynamically add or remove global constraints, labeling rules and goals from a problem definition in a solution state. However, not all combinations are worth to be carried out as a dynamic constraint satisfaction problem with respect to solution re-use:

Global constraint

- In case a global constraint C_r is *removed* from the constraint store then all previously visited states remain valid except those leaf states, that were invalidated by the constraint, need to be recalculated as potentially valid states. The original problem is redefined as $(M_0, C, G, L) : M_s \rightarrow (M_0, C \setminus \{C_r\},\ G, L) : M'_s$. In this case all already visited states are left as valid states of the new problem.
- If a global constraint C_a is *added* then all already visited states need to be re-evaluated with the new constraint, which is almost identical to a fresh state space exploration from the initial state of the original problem. This means that the new problem is $(M_0, C \cup \{C_a\}, G, L) : M'_s$. Assuming that VS is the set of the already visited states of the original problem, the invalidated visited states are $\{S_i | S_i \in VS\}$.

Labeling rule

- If a labeling rule L_r is *removed*, then all transitions that used this rule are invalid. It means that all visited states after these transitions are also invalid and must be deleted from the already visited states. Depending on the actual traversal this might affect the entire visited state space or no states at all. Informally, the new problem is $(M_0, C, G, L \setminus \{L_r\}) : M'_s$, where the invalidated states by the removed labeling rule are $\{S_i | S_i, S_{0..n} \in VS \wedge S_0 \rightsquigarrow S_j \xrightarrow{L_r} S_{j+1} \rightsquigarrow S_n \wedge j < i \leq n\}$.
- In case a labeling rule L_a is *added*, then similarly to the global constraints all previously visited states need to be re-evaluated with the new rule as it can potentially create new branches for the exploration. However, these states are not invalid, thus they can re-evaluated on demand only when the solver algorithm revisits these states. In this way the new problem is $(M_0, C, G, L \cup \{L_a\}) : M'_s$, where the states to be re-visited are $\{S_i | S_i \in VS\}$.

Labeling rules and global constraints can be treated similarly in case of classical and flexible CSP(M) problems. However, as the definition of a satisfying solution is different in a both cases, different actions needed to be carried out when a subgoal is dynamically added or removed:

Goal Classical CSP(M)

- If a subgoal G_r is *removed*, then the problem definition changes to $(M_0, C, G \setminus \{G_r\}, L) : M'_s$ and all visited states have to be re-evaluated, $\{S_i | S_i \in VS\}$. However, these updates are rather simple as only the subgoal G_r needs to be removed from either the current goal or the constraint store and this does not involve constraint evaluation (pattern matching). Additionally, solution states remain valid and states S_j where G_r is the only unsatisfied subgoal becomes solution states ($G_r \in CG_j \wedge |CG_j| = 1$).
- In case a subgoal G_a is *added* ($(M_0, C, G \cup \{G_r\}, L) : M'_s$), then all visited states have to be updated with constraint evaluation in each state. Similarly to an addition of a global constraint the problem becomes identical with a fresh state space exploration of the original problem.

Flexible CSP(M)

- If a subgoal G_r is *removed* $(M_0, C, G \setminus \{G_r\},\ L, f_w, S_w) : M'_s$, then similarly to the classical case all already visited states have to be updated, $\{S_i | S_i \in VS\}$. The complexity of the update mainly depends on the weight function f_w as it have to be recalculated on each already visited states along with the deletion of G_r from the constraint store or the current goal.
- Similarly to the case in classical CSP(M) all already visited states have to be updated with complete constraint evaluation and weight calculation when a subgoal G_a is *added* to a flexible constraint definition $(M_0, C, G \cup \{G_a\}, L, f_w, S_w) : M'_s$.
- Additionally, a flexible CSP(M) problem $(M_0, C, G, L, f_w, S_w) : M_s$ can be changed through its weight function f_w and solution weight S_w. A change in the weight function f_w cannot be treated as a dynamic manipulation in the problem definition as it requires a complete recalculation of all visited states, which is identical to a fresh state space exploration of the changed problem. However, if the solution weight S_w is changed, all already visited states remain valid and the state space exploration can continue from the solution state of the original problem. Formally, the new problem becomes $(M_0, C, G, L, f_w, S'_w) : M'_s$.

In case more than one constraint, goal or labeling rule is added or removed from the problem definition, then the union of the effects described has to be carried out.

In overall, dynamic CSP(M) can effectively and incrementally solved by reusing the previous solution in the following cases: (i) elements are *removed* from the problem definition, (ii) the solution weight is *modified* in a flexible CSP(M) definition or (iii) depending on the solver algorithm in cases where labeling rules are *added*.

5.3 The Cloud Case Study as a Flexible Constraint Satisfaction Problem

Our cloud example formalized (see in Sec. 2.2 as a flexible CSP(M) problem is depicted in Fig. 9. Similar to the IMA example the *labeling rules* capture the operations of the allocation.

- The allocateDatabase rule allocates the database DB to a server S1 if it is not already allocated.

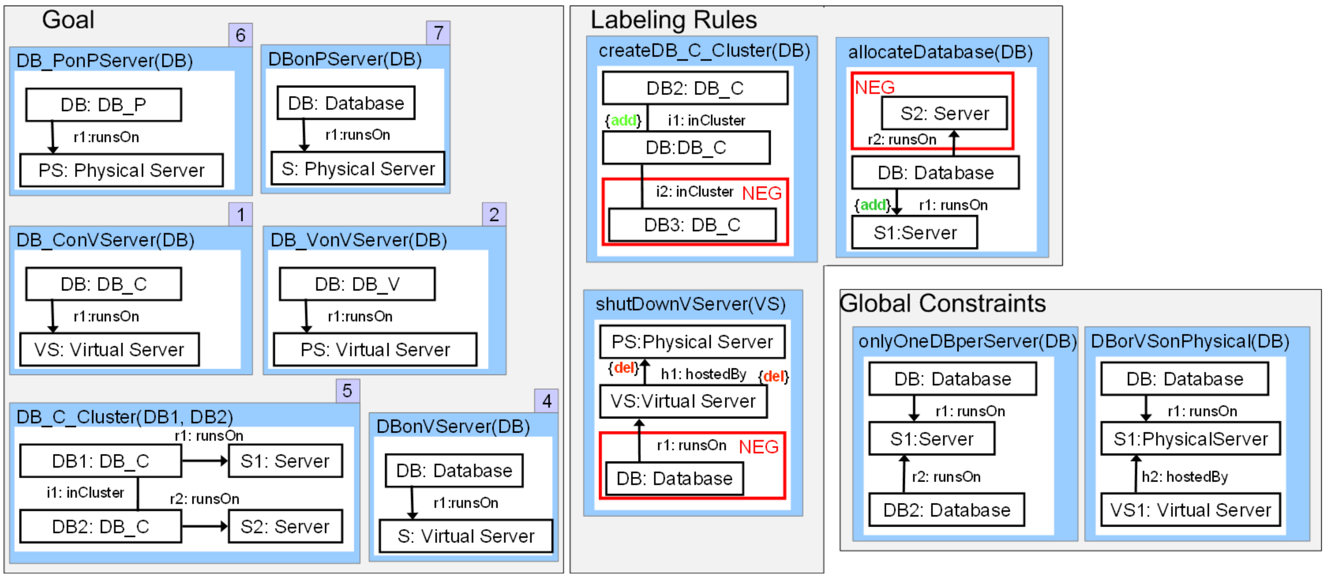


Figure 9 Global Constraints, Goals and Labeling Rules of the Cloud Case Study

– The createDB_C_Cluster rule simply creates a cluster-pair through the inCLuster relation between the DB and DB2 databases of type DB_C if DB is not already in a cluster.
– The last shutDownVServers labeling rule is used to turn off a virtual server VS hosted by the physical server PS if no database DB is running on VS. Only this simple rule is required to model the server infrastructure as our initial model will represent the state where each physical server is hosting its maximum allowed number of virtual servers.

As mentioned in Sec. 5.1 a solution state is defined by its structural goal G, its weight function f_w and the required solution weight S_w. In this example the weight function is $f_w(G_i,M_j) = perf_{G_i} * |\{m : G_i \longrightarrow M_j\}|$. It means that the weight of a subgoal G_i in state M_j is equal to the number of its matches in M_j multiplied by a predefined constant performance indicator $perf_{G_i}$. The performance indicator is a relative value derived from the requirements to capture the performance characteristic of the different database types.

The *goal* is captured by six *positive* subgoals each with its own performance indicator depicted by the number in their top right corner.

– The DBonPServer and the DBonVServer patterns with performance indicators of 7 and 4 set the average performance of a server running on a physical or a virtual server, respectively. Compared to these average values, the other four patterns formulate the *relative* performance difference defined in the problem specification.
– The DB_VonVServer and the DB_ConVServer patterns capture the requirement that the database type DB_V is faster, with a performance indicator of 2, than the other types.
– Additionally, the DB_PonPServer pattern describes that the database type DB_P performs almost twice as fast on a physical server than the other types.
– Finally, the DB_C_Cluster pattern defines that if two DB_Cs are running on different servers and form a cluster-pair then they produce an additional performance of 5.

The negative global constraints onlyOneDBperServer and DB0-or0-VS0-onPhysical specify that no server can hold more than one database and a physical server can hold either a virtual server or a database, respectively.

As the specification does not precisely define the performance differences between the databases the current definition of the problem can be a subject to change. Possible changes to the problem definition are discussed in Sec. 5.3.1 along with the required dynamic manipulation to model them in our formalism.

5.3.1 Dynamic Problem Extensions However, it is possible that the imprecise assumptions on performance, newer versions of databases or a change in business rules can slightly modify the problem definition and it requires changes in its CSP(M) definition. These changes can be treated as separate dynamic constraint satisfaction problems of our cloud example. To simulate such modifications we defined three different changes. These three modifications represent the practically relevant cases, where dynamic reevaluation does not require a fresh state space exploration and previous solutions can be partially reused.

– Let us assume that the additional plus 1 performance indicator defined by the DB_ConVServer pattern for the DB_C database is no longer required and needs to be removed from the problem definition.
– It is also possible that a newer version of the *DB_C* database supports not only cluster-pairs but also cluster-triplets, where the performance output is doubled compared to three single instances. This modification can be captured by the createDB_C_ClusterTriplet labeling rule (depicted

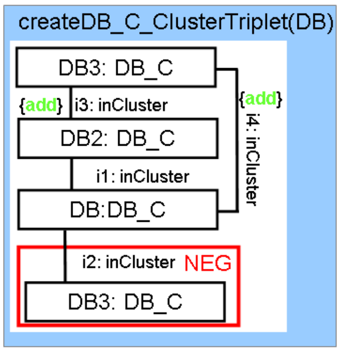


Figure 10 Dynamically added Labeling rule

in Fig. 10). The double performance is calculated by the fact that the DB_C_Cluster pattern matches three times on a single cluster-triplet.

– Finally, a third variant of dynamic change can be that business reconfiguration is no longer available due to other services provided by the cloud. This case can easily be handled by removing the shutdDownVServer labeling rule from the definition.

To assess the performance aspects of dynamic CSP(M) problem changes, Sec. 7.2 gives a first experimental evaluation of the introduced dynamic capabilities based on the cloud case study implemented in our CSP(M) solver.

6 Optimization Strategies and Implementation Details

The current section describes several optimization and implementation considerations built into our prototype CSP(M) solver. Section 6.1 briefly introduces the different search strategies applied for the guided state space traversal, while Sec. 6.2 details optimization techniques to reduce the travelled state space and finally, Sec. 6.3 focuses on concrete implementation details.

6.1 Search (Labeling) Strategies

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

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 the system backtracks to 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:

6.1.1 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 [23] 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 world scenarios.

6.1.2 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 [24]. It introduces temporal numerical *cuts* to guide the state space exploration by temporally pruning the state space to postpone the unpromising paths. By formulating the solution state configuration as submarking of the Petri net, we can solve an integer linear programming problem of the derived Petri net using its incidence matrix to obtain an optimal *transition occurrence vector* leading to a designated target state (formulated as a target submarking). A transition occurrence vector prescribes how many times a labeling rule needs to be applied in order to reach the derived submarking of a solution model. Then the search strategy first explores those branches (i.e. labeling rule applications) which are consistent with this hint. This means that if a graph transformation (labeling) rule is applied more than prescribed in the vector, then the exploration of its branch is postponed. 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 IMA 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.

It is important to mention that in case of flexible CSP(M) problems the estimation of the solution occurrence vector heavily depends on the weight function. Additionally, in case of

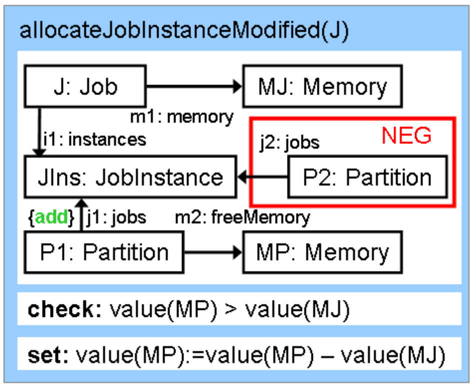


Figure 11 Modified allocateJobInstance rule

dynamic CSP(M) problems, in each case the problem changes the abstraction needed to be updated and recalculated. This traversal technique becomes less useful in these cases.

6.2 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. 6.1.

6.2.1 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 [25] or transformations of graph constraints to preconditions [26].

In our IMA 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. 11. Similarly, the global constraint onlyOneDBpreServer can be integrated as part of the allocateDatabase labeling rule in the cloud example.

6.2.2 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 search strategy to avoid state 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 IMA example.

6.3 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*, *backtracking* and *typed graph comparison*.

- For effective **evaluation of constraint** satisfiability we rely upon the incremental pattern matcher component [27] 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.
- 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 eases the computation of difference between neighbour states. However, the undo stack based implementation also has a drawback as backtracking is only possible from the actual state upward to the root and no jumping is supported between different paths of the state space. This means that traversal algorithms in the state space needs to follow a depth-first strategy.
- To be able to detect already visited states we needed to store and compare states represented by graphs as whenever the solver traverse a new state it also checks that it have not already visited this state.
  For fast **graph comparison** we adapted the DSMDIFF [28] algorithm, which relies on (i) signatures (for nodes and edges) composed of type and name information and (ii) containment relations between nodes of the graph, both supported by VIATRA2. However, the general algorithm did not scale well with large models, especially when a significant part of the model is static and cannot change during evaluation but is always compared between states. To overcome this problem, we defined a domain specific model comparator based on the general DSMDIFF algorithm. This new algorithm (i) compares only non-static parts of the model and (ii) the user can restrict elements (from the metamodel) to be used for the model comparison. In the current implementation these comparators are hand coded for each domain (meta)model.
- Finally, to keep the memory consumption low, we stored **already visited states** in a serialized form using a simple breadth-first algorithm and applied our graph comparison algorithm directly on this representation. Additionally, to

reduce the number of candidates for comparison we also applied a hash function based on the number of elements on each level of the model containment hierarchy. However, to further reduce the number of comparisons the use of domain specific hash functions are also supported by our implementation. Note that these domain specific hash functions are also have to satisfy that if two models are equal then their hash values are also equal.

The introduced solver is already in use in the context of the DIANA [29] European project as its underlying allocation engine for a system-level integration scenario for avionics software allocation.

7 Evaluation

To evaluate the performance of our CSP(M) solver, we carried out experiments both on our IMA (in Sec. 7.1) and cloud (in Sec. 7.2) allocation case studies for classical and dynamic/flexible CSP(M), respectively. Moreover, in order to compare our results with other available tools , we selected three from closely related fields:

– *Standard CSP* tools over finite integer domains are the most widely used and general purpose constraint solvers available. For our evaluation we selected the commercial SICStus Prolog [30] CLP(FD) library version 4.1.2.
– *Structural constraint solvers* similarly to our CSP(M) aim to find object graphs satisfying a given set of structural constraints. For our measurements we selected the original KORAT [31] framework based on bounded exhaustive testing.
– Finally, we used the GROOVE 4.0.1 [11] a *model checker for graph transformation system* as our third tool due to its very close problem definition language. Note that only the IMA case study was implemented in GROOVE as it does not support flexible constraints.

For all of our measurements, we used an average PC with Mobile Core Duo@1.8 GHz and 3GB RAM running Windows XP and Java SDK 1.6.13. Prior to the actual experiments we expected that:

– The SICStus CLP(FD) library will outperform our approach in all cases by orders of magnitude.
– The KORAT constraint solver will be faster especially on large models where huge traversals are expected.
– Finally, the GROOVE model checker will have a comparable performance with our implementation on small problems and we would outperform it on larger problem sizes due to the exhaustive search algorithm of GROOVE.

7.1 The IMA Case Study

We assume that we have to allocate different software workloads (functionalities) on a system with three cabinets (which corresponds to the avionics architecture used in the DIANA project).

[1] the source code of the case studies is available from http://home.mit.bme.hu/∼ahorvath/papers/sosymHVSource.zip

Size	Simple job #	Critical Job		Partition #	All Job instances
		DMR	TMR		
Small	3	2	4	4	19
Medium	5	2	5	5	24
Large	16	2	5	5	35
XLarge	20	5	7	7	51

Table 1 IMA Test Cases

7.1.1 CSP(M) Solution Each row in Table 1 defines a software workload allocation test case of different *Size*. The *Simple Job*, *Critical Job*, and *Partition* columns 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).

Runtime results of the four test cases are captured in Table 2. 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 *Finished Allocations*. *Runtime* performance and the size of the traversed *State Space* for the completed allocations are also presented by their minimum (*min*), maximum (*max*) and average (*avg*) values for each test case.

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

Therefore we also evaluated our approach without attribute manipulation (i.e., memory requirements) on the IMA case study denoted by *NON ATTR.*. In order to solve a conceptually similar problem we defined an additional global constraint stating that a partition cannot host 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 orders of magnitude.

		ATTR.			NON ATTR.			
Size		Small	Medium	Large	Small	Medium	Large	XLarge
Finished Allocations (out of 10)		10	6	1	10	10	4	1
Runtime [sec]	min	1,1	4,2	50,6	0,9	0,8	12,1	102,4
	max	196,4	145,8	50,6	89,3	156,3	195,3	102,4
	avg	66,9	81,6	50,6	32,2	41,2	57,8	102,4
Traversed States #	min	64	146	237	83	72	174	276
	max	13 984	12 632	237	9 367	17 639	1 311	276
	avg	4 802	8 296	237	3 167	4 278	404	276

Table 2 Runtime Characteristic of the CSP(M) solution on the IMA Allocation Problem

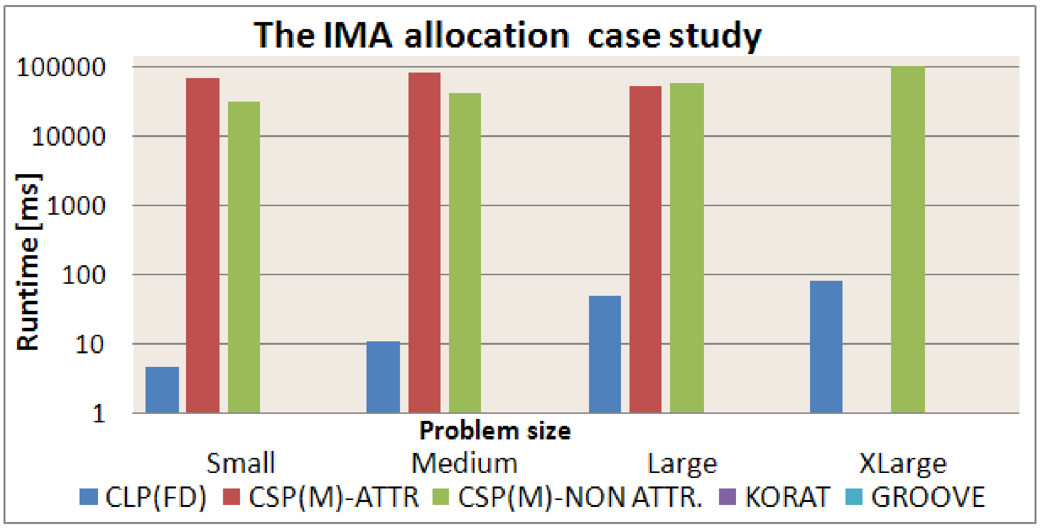


Figure 12 Runtime Results of all Approaches on the IMA Case Study

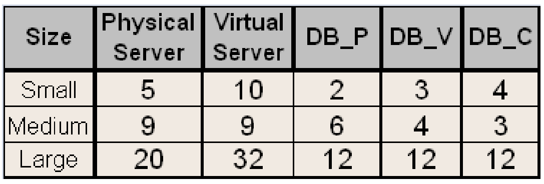
7.1.2 Other Approaches We implemented the IMA case study on additional three different tools. In all three cases the maximum number of modules were explicitly given and any solutions within this given range were accepted.

SICStus Prolog CLP(FD) The complete IMA problem was translated into a CLP(FD) problem, where both job instances, jobs, partitions, modules and all mappings between them were mapped to CLP variables. It is important to note, that we optimized the labeling strategy to effectively search for the first solution rather than do a breadth-first like traversal to find all solutions. As a personal experience, the implementation of the IMA case study in SICStus CLP(FD) required far more man-hour (approximately, 30 with optimization and debugging) than the other three solutions. At the end the whole implementation consisted of 31 Prolog clauses in 150 lines of code.

KORAT It required three inputs for instance generation: (i) a Java *class hierarchy* of the problem domain that we derived directly from the IMA metamodel (see in Sec. 1) with minor modification as inheritance is not supported by the framework, (ii) a *finitization* statement that explicitly specifies bounds on the number of objects to be used for the instance construction and finally, (iii) an *imperative predicate* that specifies the desired structural constraints of the IMA case study, written as a Java method consisting of approximately 100 lines of code .

GROOVE Due to the similar graph transformation based specification language of GROOVE, we simply adopted the graph patterns and GT rules of the NON-ATTR version of the IMA case study. Additionally, the initial models of the test cases were also easily reused. Note that, we used only the basic constructs of the GROOVE language and did not apply advanced features like nested graph transformation rules.

7.1.3 Evaluation of the Results The results are shown in Fig. 12 with average execution times in a logarithmically scaled *Runtime* axis for all four test cases (see in Table 1). Test cases are identified by their *size*. All test cases were executed ten times. We also applied a 200000 milliseconds (200 seconds) upper limit on the execution times. Results exceeding this upper limit are not shown.



Size	Physical Server	Virtual Server	DB_P	DB_V	DB_C
Small	5	10	2	3	4
Medium	9	9	6	4	3
Large	20	32	12	12	12

Table 3 Cloud Test Cases

Within the 200 seconds limit both the KORAT and the GROOVE framework failed to provide a solution even for the smallest test case. In case of the GROOVE engine it is acceptable, as it had to generate the complete state space of the problem to check if there is a solution state that satisfies all given constraints. However, also KORAT failed to provide a solution and it was parametrized to stop after the first valid solution. During the analysis of KORAT, we have discovered that it always tried to allocate first, job instances to partitions and only after going through all combinations started to allocate partitions to modules. This resulted in a giant state space even for the smallest test case. The SICStus implementation generated results at least two orders of magnitudes faster than our approach with very similar execution times.

The results of this case study show that (i) our approach outperforms the GROOVE model checker that uses an exhaustive state space exploration, (ii) it finds a single solution significantly faster than the well-known KORAT algorithm based on bounded exhaustive testing and (iii) our current implementation is lagging behind classical CLP(FD) libraries with orders of magnitude.

7.2 The Cloud Case Study

We assume that we have to allocate a predefined number of different databases to an infrastructure consisting of virtual and physical servers and reach a predefined overall performance indicator value for the whole system.

7.2.1 CSP(M) results Each row in Table 3 defines a separate cloud allocation test case of predefined *Size* and different *performance indicator* to achieve. The number of servers in the cloud are defined by the *Physical Server* and *Virtual Server* columns. Similarly, the number of database licences are captured by the *DB_P*, *DB_V* and *DB_C* columns, respectively.

In each test case we did four different measurements (see in Table 7.2.1. First, we evaluated the flexible CSP(M) with the defined resources and required overall performance (see in Table 13(a)). Based on this flexible constraint satisfaction problem we assessed three different dynamic changes of the original problem. We evaluated the cases described in Sec. 5.3.1 where

– *SubGoal removal:* The subgoal DB_ConVServer was removed from the problem.
– *Labeling rule removal:* The labeling rule shutDownVServer was removed from the definition.
– *Labeling rule addition:* Finally, the labeling rule createDB_C_ClusterTriplet (depicted in Fig. 10) was added. In the latter two cases we also modified the required overall performance indicator to balance out their effects.

In all three dynamic modifications we followed the considerations discussed in Sec. 5.2:

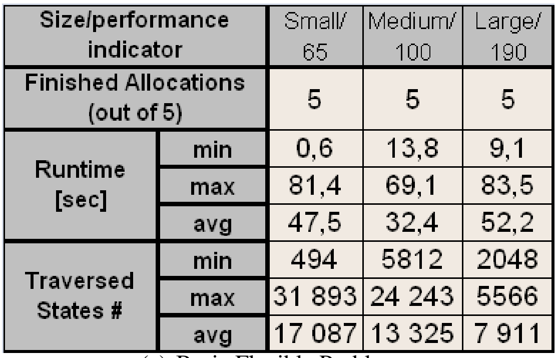
– In case of the *subgoal removal* it means that all already traversed states were updated, but that only required a recalculation of the weight function on each state.
– The *labeling rule removal* required the pruning of the already visited state space after any transaction that applied the shutDownVServer rule.
– Finally, for the *labeling rule addition* we followed the strategy to continue the solving process after the modification without re-evaluating any already visited states. This was mainly used as our transaction mechanism does not effectively support jumping between states belonging to different branches.

All three dynamic changes were made in a solution state from where the evaluation of the modified constraint problem started. Their performance results using our CSP(M) framework are captured in Figure 13(b), 13(c) and 13(d), respectively.

For the flexible CSP(M) we measured the overall *Runtime* of the solving process and the number of *traversed states*. As for the three different dynamic modifications we assessed the number of newly traversed states (*Traversed states*) to solve the dynamically changed problem. Additionally, we measured the overall *Runtime* required for both the reevaluation of the state space and the new solving process. In all four measurements, we executed the solver five times and present the number of *Finished Allocations* using again a 200 seconds upper limit on execution time.

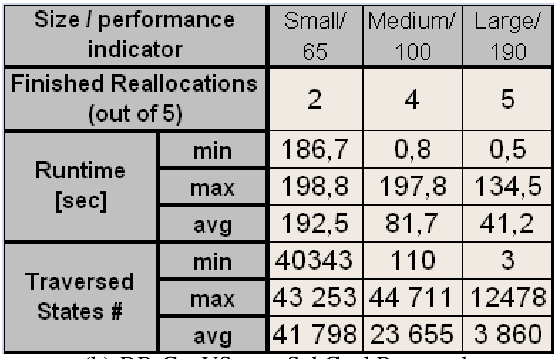
Lessons Learned As a summary, our solver is capable of handling reasonable sized flexible CSP(M)s. However, during the analysis of the traversed state space we have discovered that our search strategies do not always effectively guide traversals of flexible constraint satisfaction problems. Their main drawback is that they do not take into account the weight function when selecting the labeling rules to apply. We believe that effective guidance of flexible CSP(M) should adapt informed search strategies like $A*$ [32] with the estimated cost function directly derived from the weight function as it holds all relevant guidance information.

Similarly, the lack of guidance can be observed in case of the dynamic modifications. After the re-evaluation of the already visited states the traversals acted similarly as an exhaustive search, resulting in runtime performances that vary up to several orders of magnitude. For example, on one hand the addition of a labeling rule resulted in very fast traversals for the new solution of the modified problem. On the other hand, removal of the DB_ConVServer constraint from the problem definition resulted in a state space exploration that exceeded our 200 seconds upper limit. These differences were due to the fact that our engine preferred the use of clusters and the allocation of databases to virtual server rather



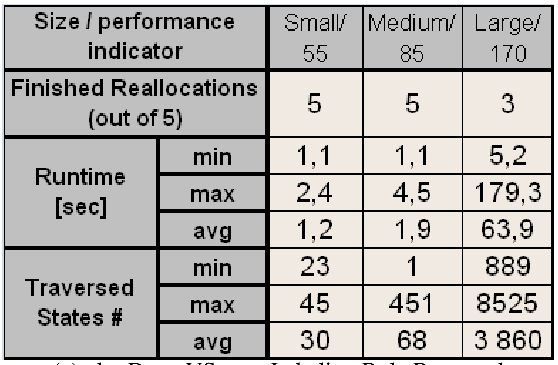
Size/performance indicator		Small/ 65	Medium/ 100	Large/ 190
Finished Allocations (out of 5)		5	5	5
Runtime [sec]	min	0,6	13,8	9,1
	max	81,4	69,1	83,5
	avg	47,5	32,4	52,2
Traversed States #	min	494	5812	2048
	max	31 893	24 243	5566
	avg	17 087	13 325	7 911

(a) Basic Flexible Problems



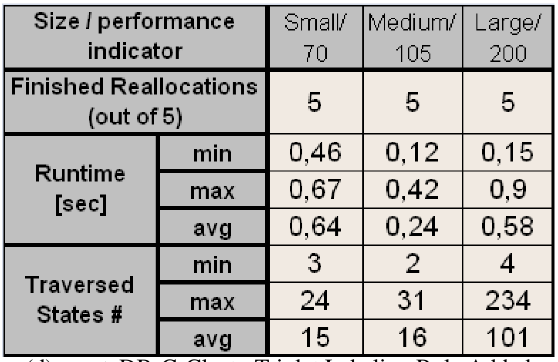
Size / performance indicator		Small/ 65	Medium/ 100	Large/ 190
Finished Reallocations (out of 5)		2	4	5
Runtime [sec]	min	186,7	0,8	0,5
	max	198,8	197,8	134,5
	avg	192,5	81,7	41,2
Traversed States #	min	40343	110	3
	max	43 253	44 711	12478
	avg	41 798	23 655	3 860

(b) DB_ConVServer SubGoal Removed



Size / performance indicator		Small/ 55	Medium/ 85	Large/ 170
Finished Reallocations (out of 5)		5	5	3
Runtime [sec]	min	1,1	1,1	5,2
	max	2,4	4,5	179,3
	avg	1,2	1,9	63,9
Traversed States #	min	23	1	889
	max	45	451	8525
	avg	30	68	3 860

(c) shutDownVServer Labeling Rule Removed



Size / performance indicator		Small/ 70	Medium/ 105	Large/ 200
Finished Reallocations (out of 5)		5	5	5
Runtime [sec]	min	0,46	0,12	0,15
	max	0,67	0,42	0,9
	avg	0,64	0,24	0,58
Traversed States #	min	3	2	4
	max	24	31	234
	avg	15	16	101

(d) createDB_C_ClusterTriplet Labeling Rule Added

Figure 13 Runtime Characteristic of the CSP(M) solution of the Cloud Case Study

than physical ones. In case of the addition of the createDB_C_ClusterTriplet labeling rule, it was able to easily produce the required cluster triplets from the already allocated cluster pairs. Moreover, the retraction of the shutDownVServer did not have any effect when a solution mainly allocated to virtual servers and extensively created clusters (like in case of our small and medium sized test cases). However, when solutions could only be found, which heavily relied on allocation to physical server, our approach had to traversed large state spaces.

7.2.2 Other Approaches We implemented the cloud case study using both SICStus and KORAT. As these approaches do not support dynamic manipulation of constraints, we separately evaluated the modified constraint problems starting from the original initial state.

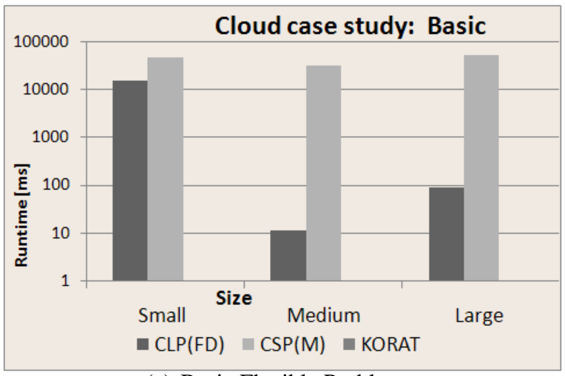
SICStus Prolog CLP(FD) Similarly to the IMA approach we translated the servers to CSP variables and modelled the available databases with the integer domain of these variables. Mapping of virtual serves to physical ones were implemented as a set of constraint over their CSP variables. Additionally, auxiliary CSP variables were used for the definition of clusters and for the evaluation of the weight function. The implementation consists of 28 Prolog clauses in approximately 170 lines of code. For the modified constraint problems, only small modifications were required on few clauses of the original code. Again this implementation took considerable more time than any other.

KORAT KORAT cannot define constraints for a dedicated instance of a class (only for the class itself, to our best knowledge). We had to modify the problem definition that all servers can host the *same amount* of virtual servers. As a consequence, the case study where the shutDownVServer labeling rule were removed could not be effectively defined in the *imperative predicate* and therefore we omitted it from from the measurements. Similarly, the Java classes were derived directly from the Cloud metamodel (see in Sec. 3) and the *imperative predicate* were also given as a Java method consisting of approximately 140 lines of code.

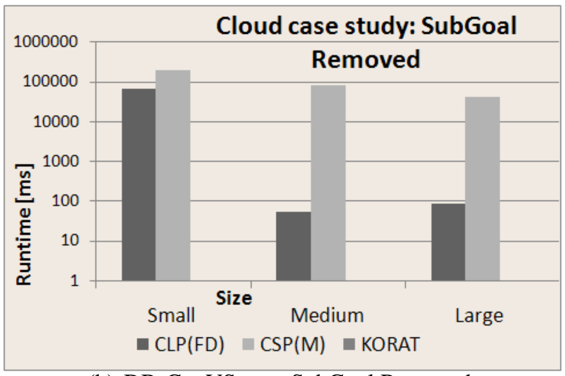
7.2.3 Evaluation of the Results The results are shown in Fig. 14 with separate figures for the basic flexible problem and its three modified version. Average execution times in *milliseconds* are presented in a logarithmically scaled *Runtime* axis. Each measurements were executed five times. We again applied a 200000 milliseconds (200 seconds) upper limit on the execution times and results exceeding this upper limit are not shown in Fig. 14.

Again, within the 200 seconds limit the KORAT framework failed to provide a solution even for the smallest test case for the original problem or its two modified versions. The main reason is that KORAT always preferred to allocate databases to physical servers rather than to virtual ones. This resulted in extremely large search spaces.

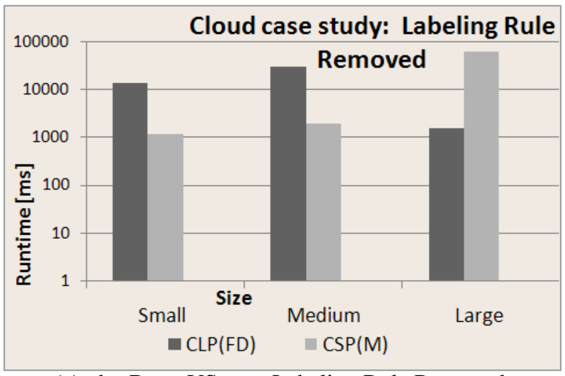
The SICStus implementations again produced very consistent execution times and in certain cases orders of magni-



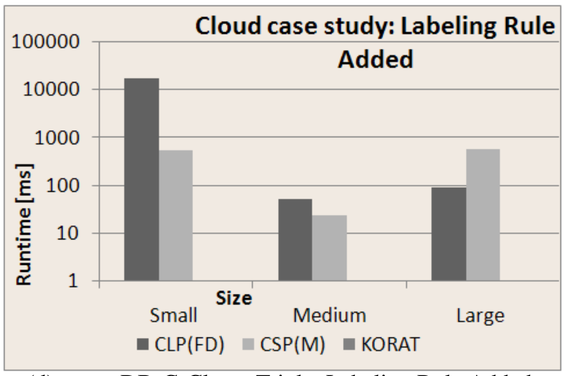
(a) Basic Flexible Problems



(b) DB_ConVServer SubGoal Removed



(c) shutDownVServer Labeling Rule Removed



(d) createDB_C_ClusterTriplet Labeling Rule Added

Figure 14 Runtime Results of all Approaches on the Cloud Case Study

tude faster than any other approach. However, for the *Small* sized test case in average our engine produced solutions within a comparable range. This was due to the fact that in our implementation the labeling algorithm of the SICStus engine always tried to allocate databases to physical servers and only slowly found solutions were both clusters and virtual servers were required. This is one of the main difference between the two solutions and reason for the diverse runtime performances.

Altogether, these measurements demonstrated that in four out of nine dynamic cases partially reusing the solution obtained from a previous traversal of the original problem is a competitive alternative. Additionally, in case of complex structural constraints, the way how the search space is traversed has a significant impact on performance and effective solutions require explicit problem specific fine-tuning or hints to achieve acceptable performance.

7.3 Summary

Our measurements show that our constraint solver based upon incremental pattern matching is able to solve non-trivial classical and flexible problems of model oriented constraints. We also demonstrated that certain dynamic changes of constraint definitions can be effectively handled with a good level of solution reuse. More specifically:

– Constraint satisfaction problems with complex structural constraints can be intuitively captured by our proposed formalism combining graph patterns and graph transformation rules. In contrast, expressing structural constraints in the traditional CLP(FD) formalism requires significant modeling workaround.
– Our approach outperformed in all cases the well-known academic KORAT structural constraint solver. We believe that this is a combined effect of using (i) incremental pattern matching to efficiently detect possible continuations and (ii) explicit labeling rules to guide the traversal.
– Unsurprisingly, exhaustive generation of the state space (like in case of GROOVE) is not a feasible solution for constraint satisfaction problems without further support. Ongoing research in the GROOVE framework aims to restrict state space travels by adding a conjunction of global constraints.
– As we expected, the industrial SICStus CLP(FD) library outperformed our engine in the static cases by orders of magnitude. However, in case of dynamic constraint satisfaction problems our approach resulted in comparable (in certain cases even better) runtime thanks to good level of solution reuse.
– Additionally, in almost all cases to achieve acceptable performance problem specific hints or fine-tuning is advantageous. However, these fine tuning hints would increase the complexity of the problem definition. We be-

lieve that our graph transformation based CSP formalism gives a good trade-off between easy declarative problem definition and fine-tuning.
- Due to the nondeterministic nature of our traversal strategy, execution times may vary significantly. For this reason we plan to better exploit adaptive search algorithms.

It is also important to note that these measurements were carried out on specific problems derived directly from our ongoing research projects. Despite the large set of predefined synthetic case studies used mainly for performance measurements in model transformation tool contests ([33, 34]), very few cases (e.g., the live contest at [35]) address related challenges (like backtracking or flexible problems). We plan to submit our case studies to future editions of these tool contests.

Our further investigations have to be directed to (i) combine our constraint definitions with constraints over regular attributes, (ii) develop specific informed search strategies for traversals of flexible CSP(M)s and (iii) further examine the effects of dynamic constraint changes to enhance solution reuse.

8 Related Work

Applications of CSP in MDE. Constraint satisfaction techniques have been successfully applied in the context of MDD. [36] proposes an approach for partial model completion based on constraint logic programming. [4] support efficient domain specific modeling by transforming constraints to a Prolog representation. In [3], poor design patterns are detected by using off-the-shelf CSP techniques and tools. [37] 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, [38] proposes constraint solving as a graph pattern matching strategy. [5] 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 [39–41] 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 as in [39], or by a dedicated theorem prover for Horn-like clauses as in [41]. This problem can also be interpreted as a special (restricted) CSP problem without numeric constraints on attributes. Additionally, UMLtoCSP [42] verifies certain correctness properties of OCL adorned UML model by translating them into the ECLiPSe CSP solver and executes a bounded instantiation search.

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.

Structural constraint solving allows finding object graphs that satisfy given constraints both on attributes and (object) structures for systematic testing by exploring a (usually) bounded number of possible object graphs. Many promising approaches exist like the CUTE [43] framework that uses a combination of symbolic and concrete execution to derive path constraints for each separate execution paths, the Java PathFinder [44] that is based on Generalized Symbolic Execution [45] that first introduced the idea to use model checkers for solving structural constraints, Alloy [12] a lightweight object modelling framework using a simplified Z notation that is translated to boolean formulas for SAT based evaluation or KORAT [46] that performs specification based testing by using a predicate representing the properties (constraints) of the desired output structures and explores the input state space of the predicate using bounded exhaustive testing.

It is common in these approaches that each solution satisfies all given constraints similar to our approach; however, their main difference is that they cannot define restrictions on how these solutions are achieved from the initial state, meaning that no constraints can be defined to hold on states visited during a solution trajectory, which in our case is supported by the global constraints.

State Space Exploration for GT. There are several state space exploration approaches to analyze graph transformation systems (GTS).

Augur2 [47] is a GTS model checker that tackles the complexity associated with independent rules by condensing the entire state space into a single graph with unfolding semantics. It also provides some approximative techniques to deal with infinitely large state spaces, and counterexample-guided refinement of this abstraction.

GROOVE [11] is a model checker over graph transformation systems. Its main benefit is the ability to verify model transformation and dynamic semantics through applying CTL model checking on the generated state space of the GTS. It is mainly used for modeling and verifying the design-time, compile-time, and run-time structure of object-oriented systems.

It is 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 relies on guided traversals.

Graph constraints were first introduced in the context of negative application condition and later extended as a specification formalism [48, 49] to define constraints associated to visual modelling formalisms and reason about them with a set of sound and complete inference rules. Our graph pattern based constraint specification is based on these foundations, however, we use a different pattern language that allows recursive pattern compositions but more restrictive on formulas and does not support all connectives e.g., implication. However, we believe that CSP(M) can also be instantiated over this graph constraint formalism.

Graph transformation rules are also used in [50] to define a non-restrictive contract specification language by the means of pre- and postconditions. It combines GT rules with OCL in order to be able to capture non-deterministic specifications and overcome the frame-problem [51]. Its language is far more expressive than our, however due to this expressiveness no implementation is available to evaluate its performance as in our case.

Constraint based graphic systems define complex (graph based) drawings and diagrams using constraints on their graphical objects and relationships.

ThingLab [52] is an extensible constraint solver for graphical simulation. In ThingLab, constraints are imperatively defined providing functions to solve individual constraints, and the solver attempts to invoke them in an appropriate order for solving the complete constraint store. It also supports definition of constraints in an object oriented manner, allowing inheritance of constraints along the supertype relationship.

DeltaBlue [53] is a perturbation based constraint hierarchy solver, maintaining solutions incrementally as constraints are dynamically added or removed. Additionally, it minimizes the cost of finding a new solution after each change by exploiting its knowledge of the last solution.

Juno-2 [54] is a constraint-based double-view drawing editor for the definition of interactive graphics. It uses a extensible declarative constraint language including non-linear functions and ordered pairs compiled into effective constraint primitives for interactive feedback.

On one hand, common in these approaches that they support only a limited set of constraints and (except Juno-2) cannot define cyclic constraints (e.g., simultaneous equations and inequalities on the variables). On the other hand, many techniques applied in these approaches for handling large number of constraints such as (i) packing and unpacking constraints into constaint primitives and (ii) propagation of values through predefined constraint hiearchies can be partially adopted to our framework giving space for future research.

CSP-specific Research in the field of constraint satisfaction programming has been conducted towards flexible and dynamic constraints [8, 55]. 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.

Additionally, our state space exploration approach also builds on the idea of random traversals described in [23] to solve large problems.

9 Conclusion and Future Work

In order to address design space exploration in complex embedded and IT systems using MDE techniques, in the current paper, we presented a novel approach (by extending initial work [13]) for defining constraint satisfaction problems directly over models using graph transformation rules and graph patterns. Compared to traditional CSP, we extended labeling by using model manipulation as provided by graph transformation to *dynamically create* and *delete* model elements. Additionally, we introduced dynamic CSP(M) that allows to *dynamically add or remove* global constraints, subgoals and labeling rules to alter the problem definition. Furthermore, we have presented *weighted CSP(M)* an extension to classical CSP(M) that supports flexible constraint satisfaction problems based on relaxable soft constraints.

We have also built 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 strategies (e.g. random backjumping, directed search) to guide the state space traversal.

On top of that, we carried out various comparative measurements to assess the performance of our approach, which demonstrated that our solver based upon incremental pattern matching is able to solve non-trivial classical, flexible and dynamic problems for model-oriented constraints.

As a summary, we argue that model transformation technology can efficiently contribute to formulate and solve constraint satisfaction problems with complex structural constraints and dynamic labeling rules.

However, by analyzing the measurement results, we have also identified some key ares as where the performance of the solver tool could be further improved in the future such as (i) to combine traditional constraint programming techniques to our algorithms to effectively *handle constraints over attributes*, (ii) to adapt *informed search strategies* for effective traversals of flexible problem definitions, (iii) to further examine the effect of dynamic problem definition changes especially, when more than one part of the definition changes and finally (iv) to handle traversed states space in a more space efficient way and use more advanced graph comparison algorithms like [56, 57] that can handle larger graphs.

In addition to improving performance, we plan to support automatic detection of look-ahead pattern based on critical pair analysis for state space optimization.

Acknowledgements We would like to thank Eduardo Zambon and Marteen de Mol their help and useful comments on how to use GROOVE and implement our case studies. Additionally, we would also like to thank Zoltán Ujhelyi for his help with the the SICStus CLP(FD) system.

References

1. Neema, S.: Analysis of matlab simulink and stateflow data model. (March 2001)
2. AUTOSAR Consortium: The AUTOSAR Standard. `http://www.autosar.org/`.
3. 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
4. White, J., Schmidt, D., Nechypurenko, A., Wuchner, E.: Introduction to the generic eclipse modelling system. Eclipse Magazine (6) (2007) 11–18
5. Petter, A., Behring, A., Mühlhäuser, M.: Solving constraints in model transformation. In: ICMT'09: International Conference on Model Transformation, Zurich, Switzerland (2009)
6. Intelligent Systems Laboratory, Swedish Institute of Computer Science: Sicstus User's manual (2009) `http://www.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf`.
7. Official website of ILOG Solver: `http://www.ilog.com/products/cp/`.
8. Miguel, I., Shen, Q.: Dynamic flexible constraint satisfaction. Applied Intelligence, 13(3) (2000) 231–245
9. ATLAS Group: The ATLAS Transformation Language. Available from `http://www.eclipse.org/atl/`.
10. Schürr, A.: Introduction to PROGRES, an attributed graph grammar based specification language. In Nagl, M., ed.: Graph–Theoretic Concepts in Computer Science. Volume 411 of LNCS., Berlin, Springer (1990) 151–165
11. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE). (2004) 479–485
12. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2) (2002) 256–290
13. Horváth, A., Varró, D.: CSP(M): constraint satisfaction programming over models. In: Proc. of MODELS'09, ACM/IEEE 12th International Conference On Model Driven Engineering Languages And Systems. LNCS 5795 (2009) 107–121
14. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Chapter: Algebraic Approaches to Graph Transformation. Volume 1: Foundations. World Scientific (1997)
15. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming **68**(3) (October 2007) 214–234
16. 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
17. Rensink, A.: Representing first-order logic using graphs. In: ICGT 2004: 2nd International Conference on Graph Transformation, Rome, Italy. (2004) 319–335
18. Weld, D.S.: An introduction to least commitment planning. AI Magazine **15**(4) (1994) 27–61
19. Bistarelli, S., Montanari, U., Rossi, F.: Constraint solving over semirings. In: In Proc. IJCAI95, Morgan (1995) 624–630
20. Descotte, Y., Latombe, J.C.: Making compromises among antagonist constraints in a planner. Artif. Intell. **27**(2) (1985) 183–217
21. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In: Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88), St. Paul, MN (1988) 37–42
22. Verfaillie, G., Schiex, T.: Solution reuse in dynamic constraint satisfaction problems. In: AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1), Menlo Park, CA, USA, American Association for Artificial Intelligence (1994) 307–312
23. 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
24. Varró-Gyapay, S., Varró, D.: Optimization in graph transformation systems using Petri net based techniques. Electronic Communications of the EASST **2** (2006)
25. 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
26. 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
27. 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)
28. 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
29. Official website of the Distributed equipment Independent environment for Advanced avioNics Applications (DIANA) European project: `http://diana.skysoft.pt`.
30. The Swedish Institute of Computer Science: SICStus Prolog - home page http://www.sics.se/isl/sicstuswww/site/index.html.
31. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: The Korat structural constraint solver - home page http://korat.sourceforge.net/.
32. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths in graphs. IEEE Transactions On Systems Science And Cybernetics **SSC-4**(2) (1968) 100–107
33. The AGTIVE Tool Contest: official website (2007) `http://www.informatik.uni-marburg.de/~swt/agtive-contest`.
34. The Transformation Tool Contest: official website (2010) `http://www.planet-research20.org/ttc2010`.
35. The Graph-Based Tool Contest: official website (2008) `http://fots.ua.ac.be/events/grabats2008/`.
36. 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, Warzburg, Germany (2007)
37. 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
38. 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
39. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling (2009)
40. 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
41. 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

42. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, New York, NY, USA, ACM (2007) 547–548
43. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. SIGSOFT Softw. Eng. Notes **30** (September 2005) 263–272
44. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. SIGSOFT Softw. Eng. Notes **29**(4) (2004) 97–107
45. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: In Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2003) 553–568
46. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on java predicates. In: International Symposium on Software Testing and Analysis (ISSTA, ACM Press (2002) 123–133
47. 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
48. Orejas, F., Ehrig, H., Prange, U.: A logic of graph constraints. In: Fundamental Approaches to Software Engineering (FASE'08). (2008) 179–198
49. Orejas, F.: Attributed graph constraints. In: ICGT '08: Proceedings of the 4th international conference on Graph Transformations, Berlin, Heidelberg, Springer-Verlag (2008) 274–288
50. Baar, T.: OCL and graph-transformations - a symbiotic alliance to alleviate the frame problem. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of Lecture Notes in Computer Science., Springer (2005) 20–31
51. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Transactions on Software Engineering **21** (1995) 785–798
52. Borning, A.: The programming language aspects of thinglab, a constraint-oriented simulation laboratory. ACM Trans. Program. Lang. Syst. **3**(4) (1981) 353–387
53. Sannella, M., Maloney, J., Freeman-Benson, B., Borning, A.: Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. SOFTWARE PRACTICE AND EXPERIENCE **23** (1993) 529–566
54. Heydon, A., Nelson, G.: The juno-2 constraint-based drawing editor. In: Technical Report 131a, Digital Systems Research. (1994)
55. 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
56. P. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10) (2004) 1367–1372
57. Rensink, A.: Isomorphism checking in groove. In Zündorf, A., Varró, D., eds.: Graph-Based Tools (GraBaTs), Natal, Brazil. Volume 1 of Electronic Communications of the EASST., European Association of Software Science and Technology (September 2007)