Automatic Role-based Constraint Solving for Real-Time and Embedded Systems: An Approach to Modeling Guidance

Abstract

Emerging distributed real-time and embedded (DRE) systems, such as automotive infotainment, industrial automation, and medical imaging, are increasingly complex. Without effective tool support to check the correctness of the models, this complexity makes it infeasible to model different aspects, such as various deployments and configurations of hardware and software components for specific products. One way to ensure model correctness is to define constraints using OCL or other languages. Even with these constraint languages, however, it is still hard to create correct models manually due to the combinatorial nature of the constraints and their interdependence.

This paper provides two contributions to work on improving model correctness in domains where either model size or constraint combinatorial complexity limits manual approaches. First, we present our approach of Automatic Role-based Constraint Solving (ARCS) and show how the application of declarative constraint programming techniques can ensure model correctness, as well as automatically suggest (deduce) correct modeling steps. Second, we show how model-driven engineering tools can help guide modelers towards correct solutions based on the defined constraints. The results of applying our declarative constraint programming techniques to a case study of a deployment problem in the automotive domain show that this approach can dramatically reduce complexity and improve productivity when modeling DRE systems.

1 Introduction

There are many application domains where the constraints are so restrictive and the solution spaces so large that it is extremely hard for modelers to produce correct solutions manually. Graphical modeling languages such as UML can help to visualize certain aspects of the system under development and automate particular development steps using code-generation. Domain-Specific Modeling Languages (DSMLs) are another graphical modeling approach that combine high-level visual abstractions that are specific to a domain with constraint checking and code-generation to simplify the development of a large class of systems [20]. In these complex domains, however, any type of graphical modeling tool that merely provides solution-correctness checking via constraints provides few benefits compared with handcrafted techniques.

Regardless of the modeling language and notation used, the key challenge in complex domains is their combinatorial nature, not code construction. For example, specifying the deployment of only a few tens of model entities that map software components to Electronic Control Units (ECUs) in an automobile can easily generate solution spaces with millions or more *possible* deployments but few *correct* ones due to the complexity of configuration and resource constraints. For these combinatorially-complex modeling problems, it is impractical to create a complete and valid model manually. As the number of model elements grows to hundreds or thousands, the scalability challenges of large and globally constrained domains clearly become intractable to handle manually.

In general, respecting configuration and resource constraints implies constraint checking and associated revisions to handle any invalid component→ECU assignments. Applying constraint checking alone, however, is often insufficient. For instance, constraint checking might reveal that a component cannot be assigned to any ECU due to the lack of remaining CPU capacity on each ECU. Unfortunately, this information does not provide modelers with insights as to whether redoing several previous assignment might lead to a valid assignment of all currently considered ECUs, *i.e.*, there is no indication on *where* to reassign the mapping.

To address the challenges of modeling combinatorially complex domains, techniques and tools are needed to simplify the combination of graphical modeling environments with declarative constraint programming techniques that can automatically leverage constraint solvers to guide users' modeling steps and assist them in producing correct models. This combined approach should respect the domain-specificity of the modeling tool and provide a flexible mechanism for specifying solvers using domain-specific notations.

This paper describes an approach called *automatic role-based constraint solvers* that provides the following three contributions to address scalability problems arising when modeling complex application domains: (1) how to combine automatic constraint solving with domain-specific modeling to offer automated guidance to modelers and apply constraint-aware batch solving processes to automatically complete models of large systems, (2) how modelers can specify repair operations that can be leveraged by a constraint solver to automatically fix models that violate the domain constraints, and (3) a case-study based on a tool for automatically producing deployment plans to map software components to Electronic Control Units (ECUs) (which are CPUs in an automobile) in the context of the EAST-EEA Embedded Electronic Architecture [3] defined in a European Union research project as a predecessor of the emerging AUTOSAR [4] middleware and modeling standard.

The paper extends our previous experience with AUTODeploy [9] and focuses on how to address the challenges related to the complexities of aspect weaving identified there. Our prior work focused on applying the relationship between automated role-based constraint solving and aspect weaving. This paper generalizes our earlier approach to a broader range of modeling-related activities, including applying batch processes to create connections and automatically adding model elements to satisfy domain constraints. This paper also presents a case study that shows how we use our automatic role-based constraint solving techniques to map deployment problems to configuration and bin-packing solvers.

The rest of the paper is organized as follows: Section 2 addresses these considerations in the context of the EAST-EEA automotive architecture, using the deployment of software functions to ECUs as a motivating example; Section 3 shows how our automatic role-based constraints solving technique addresses the scalability challenges of modeling DRE domains; Section 4 describes the results of applying our constraint solvers to model EAST-EEA component deployments; Section 5 presents related work; and Section 6 presents concluding remarks.

2 Motivating Example

EAST-EEA defines the embedded electronic architecture and structure of automotive middleware to help standardize solutions to problems that arise when developing large-scale, distributed real-time and embedded (DRE) systems for the automotive domain. For instance, it is complicated to (re)deploy software components in ECU computers and micro-controllers running software components within a car. Key complexities of (re)deployment include: (1) components often have many constraints that must be met by the target ECU and (2) there are many possible mappings of components→ECUs in a car and finding the optimal one(s) is hard.

For example, finding a set of interconnected ECUs able to run a group of components that communicate via a bus is hard to do manually. Modelers must determine whether the available communication channels between the target ECUs meet the bandwidth, latency, and framing constraints of the components communicating through them. It is also important in automotive domain to optimize the overall solution costs, *e.g.*, by finding deployments that use as few ECUs as possible or by minimizing bandwidth requirements and using cheaper buses. It is hard, if not impossible, to answer these questions manually for production system models.

To illustrate the practical benefits of integrating automatic role-based constraint solvers with a DSML, we present a case study based on AUTODeploy [13], which is a model-driven tool we developed to solve EAST-EEA-like constraints that ensure correct deployment of software components to ECUs. The following is a summary of the aspects of EAST-EEA that are relevant to our example, starting with the two primary viewpoints on EAST-EEA-based systems:

- **Logical collaboration structure**, which specifies components or functions that should communicate with each other and through what interfaces.
- **Physical deployment structure**, which captures the capabilities of each ECU, their interconnecting buses, and their available resources.

A typical task for an EAST-EEA developer is to specify on which ECU each component should run. This mapping from logical collaboration structure to physical deployment structure is generally specified with a graphical tool, as shown in Figure 1.

Modern cars (such as the BMW 7 Series, Mercedes S-class, and Audi D3) are typically equipped with 80 or more ECUs and several hundred or more software components. Simply drawing arrows from 160 components to 80 ECUs is tedious. Adding to the difficulty of a manual approach, moreover, are requirements governing which ECUs can host a component, which involves assessing the amount of memory required to run, CPU power, operating system type and version (if any), etc. These constraints must be considered carefully when deciding where to deploy a particular component. The problem is further exacerbated when considering physical communication paths and aspects, such as available bandwidth in conjunction with periodical real-time messaging.

The work presented in this paper uses the AUTODeploy tool to specify the mapping of components to ECUs for EAST-EEA models as our case study. The goal of this case study was to create *(semi)automated mechanisms to map software components to ECUs without violating the known constraints.* In addition, when there is no way to deploy all the components in the model (*e.g.*, if certain components

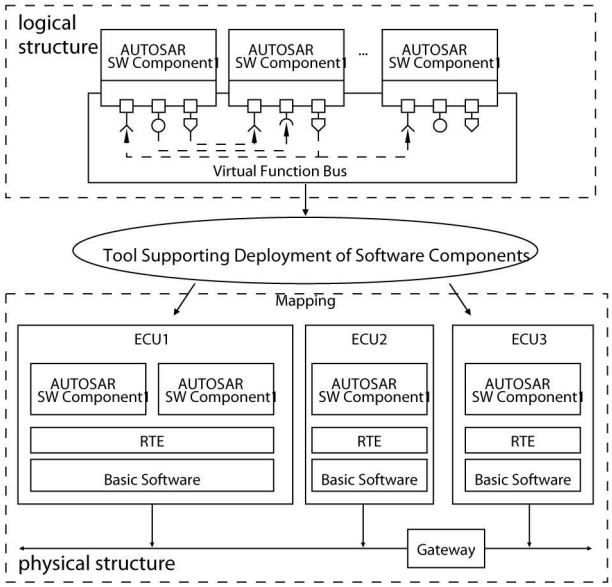

Figure 1. Mapping from the Logical Collaboration to the Physical Deployment Structure

need more memory then available on the ECUs), rather than saying "a deployment is impossible," we want to *provide suggestions on how to change the system to make a full deployment possible*, *e.g.*, the tool could suggest trying a different deployment strategy, relaxing certain constraints, increasing the memory on a certain ECU by a designated amount, or adding a higher speed bus. The following sections describe AUTODeploy and show how it can significantly reduce the time and effort needed to creating EAST-EEA deployment models.

3 Automatic Role-based Constraint Solving

Based on the motivation and the example in Sections 1 and 2, the goals of our AUTODeploy project were to (a) create a model-driven tool to guide EAST-EEA modelers by suggesting valid modeling steps to create a correct model, (2) provide mechanisms that perform domain constraint-compliant batch processes to complete a partially specified model, and (3) allow users to express valid repair operations that can be utilized by a constraint solver to fix invalid models.

In previous work [11, 12, 10], we illustrated how a DSML can improve the modeling experience and bridge the gap between the problem and solution domain by the introduction of domain-specific abstractions. As a result of these efforts, the Generic Eclipse Modeling System (GEMS) was created and integrated with the Eclipse Generative Modeling Technologies (GMT) project. GEMS provides a convenient way to define the metamodel, *i.e.*, the visual syntax of the DSML. Based on the metamodel, GEMS can automatically generate a graphical editor that enforces the grammar specified in the DSML. In addition, to facilitate code-generation, GEMS provides some convenient infrastructure (such as built-in support for a Visitor pattern [15] implementation) to simplify model traversal. We used GEMS as the basis for AUTODeploy and our work on automatic role-based constraint solving.

3.1 Domain Constraints as the Basis for Automatic Solvers

A key research challenge was to determine how to specify model constraints in such a way that they could be used not only to check models for correctness, but also *to enable constraints solvers to derive solutions to the constraints*. We considered using Java, the Object Constraint Language (OCL), and Prolog for our constraint specification language. Initially, we implemented our EAST-EEA deployment constraints in each of the three languages to evaluate their pros and cons. As a result of the evaluation we conducted, we came to the conclusion that Prolog was the most appropriate language for providing both constraint checking and model suggestions.

Prolog is a declarative programming language that allows programmers to define the set of rules in terms of known facts or a knowledge base (KB). Prolog can then evaluate these rules and determine if they can be satisfied by the known facts. Prolog provides a unique degree of flexibility for writing constraints in that it not only replies with whether or not a rule can be satisfied but also with what the valid fact combinations are that will satisfy it. If users completely specify the input variables values, Prolog merely checks whether the rule holds for the facts from KB. If, however, some of the input variables are not bound, Prolog has the unique ability to return the set of possible facts from the KB that lead the rule to evaluate to "true".

In the context of AUTODeploy and EAST-EEA, the declarative nature of the Prolog reduces the number of lines of code needed to transform an instance of a DSML into a knowledge base and create constraints (its roughly comparable to OCL for writing constraints). Moreover, Prolog enables AUTODeploy to derive sequences of modeling actions that will take the model from an invalid or incomplete state to a valid one. As discussed in Section 2, this capability is crucial for domains, such as EAST-EEA-based deployment, where completely manual model specification is infeasible or extremely tedious and error-prone.

The remainder of this section describes how we applied Prolog and GEMS to create the automatic role-based constraint solvers used in AUTODeploy. Our research focused on providing modeling guidance with respect to single modeling steps, automatic model completion via batch processing, and automated model repair guided by a constraint

solver.

3.2 Local Modeling Guidance on-the-fly

To provide automatic role-based constraint solving, AUTODeploy must capture the current state of the model and reason about how to guide the model through a series of modifications so that it satisfies the domain constraints. For our EAST-EEA deployment example, when AUTODeploy is given a set of components, ECUs, and their resources and constraints, it must suggest a valid assignment of components to ECUs. Our work provides this reasoning capability to GEMS by automatically generating a Prolog representation of each model as a knowledge base (KB), allowing users to specify Prolog constraints, and then querying the constraints for valid sequences of model changes to bring the model to a valid state.

GEMS metamodels represent a set of model entities and role-based relationships between them. For each model, the generated DSML editor populates a Prolog KB using these metamodel-specified entities and roles. For each entity, we generate a unique id and a predicate statement specifying the type associated with it. For example, each component in our EAST-EEA model is transformed into the predicate statement `component(id)`, where id is the unique id for the component.

For each instance of a role-based relationship in the model, a predicate statement is generated that takes the id of the first participating entity and the id of the entity the first one is relating it to. For example, if a component, with id 23, has a *TargetECU* relationship with a ECU, with id 25, the predicate statement `targetECU(23,25)` is generated. This predicate statement specifies that the entity with id 25 is a TargetECU of the entity with id 23. Each KB provides a domain-specific set of predicate statements.

Based on this domain-specific knowledge base, modelers can specify user-defined constraints in the form of Prolog rules for each type of metamodel relationship. These constraints are a semantical enrichment of the model that indicate the requirements of a correct model. They are also used by constraint solvers to automatically deduce the sets of valid model changes to create a correct model. For example, consider the following constraint to check whether a ECU is a valid ECU of a component:

```
is_a_valid_component_targetECU(Component,
                               ECU).
```

This constraint can be used to (1) check a Component-ECU combination, *i.e.*:

```
is_a_valid_component_targetECU(23,[25]).
```

and (2) to find valid ECUs that can play the `TargetECU` role for a particular component using the Prolog's ability to deduce the correct solution:

```
is_a_valid_component_targetECU(23,
                               ECUs).
```

In this example, the `ECUs` variable will be assigned the list of all constraint-valid ECUs for the `TargetECU` role of the specified component. This example shows how the constraint predicate can be used as an automatic role-based constraint solver to check *and* generate the solution.

Figure 2 depicts what the dynamic suggestions mechanism looks like for modelers and how it simultaneously supports the visual modeling steps. The upper part of the

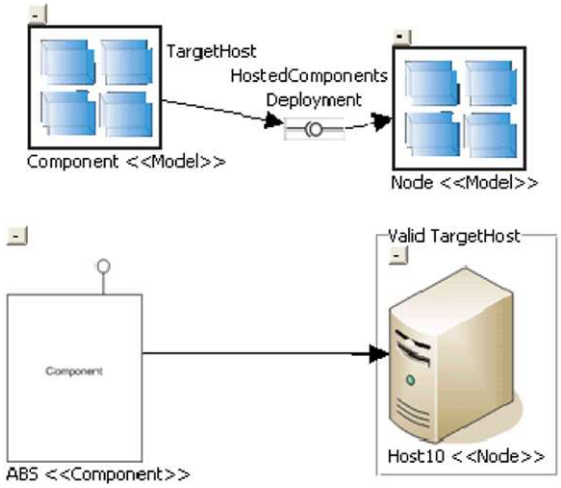

Figure 2. Highlighting valid target ECU

figure shows the fragment of the metamodel that describes the "Deployment" relationship between "Component" and "ECU" model entities. The lower part of the picture illustrates how the generated editor shows "ABS" and "ECU10" as instances of the "Component" and "ECU" types respectively. This screenshot was made at the moment when the modeler starts dragging the connection line using "ABS" as starting point. The rectangle around "ECU10" labeled "Valid TargetECU" is drawn automatically as a result of triggering the corresponding solver rule and receiving a valid solution as feedback.

3.3 Constraint Solver Integration With GEMS

We have integrated a tool infrastructure into the GEMS-generated DSML editors that listens for user-initiated actions, finds all valid relation roles that the source entity may participate in, and then finds the set of all valid values for these connection relationships. These valid connection endpoints are then suggested to the user by highlighting valid model entities. This feedback mechanism is automatically incorporated into GEMS. The DSML editors generated by GEMS can leverage multiple Prolog implementations, including SWI Prolog [21] and XSB Prolog [18], and can

use the Java Prolog Library (JPL) [2] or Interprolog [6] libraries to invoke Prolog queries from within Java. The KB is synchronized with the model by issuing Prolog assert/1 and retract/1 statements when new information is added or removed from the model.

Using the tools and techniques described above, modelers only need to specify the constraints and the generated DSML editor can automatically find valid solutions for many types of problems. The feedback mechanism and the respective automatic role-base constraint solvers are also dynamic, *i.e.*, users can add or modify role-based constraint (solvers) at modeling time and immediately begin receiving step-wise guidance. Having all these mechanisms, modelers can receive on-the-fly suggestions for each modeling step, *e.g.*, while dragging the mouse to create a connection relation between component and ECU, the valid drop targets are highlighted.

There are also mechanisms to trigger arbitrary Prolog rules from the modeling tool and incorporate their results back into a model. In our EAST-EEA case study, this mechanism can be used to resolve complete component to ECU deployments and add deployment relationships based on a model of partially specified component-ECU assignments. User-defined solvers usually apply generic rules that can be derived from the metamodel and often incorporate certain heuristics to simplify the solution-finding process. These solvers encapsulate *formalized experience* gained while developing similar systems and can greatly improve the quality of the model.

An example of a situation where user-defined rules are helpful is the deployment problem described above. It is very hard to define a correct global deployment manually if there are 300 components and 80 ECUs available. In contrast, we were able to solve this scale of problem in $\sim$3 seconds using AUTODeploy.

3.4 Layered Constraint Solving

We initially used a single-solver approach to deduce solutions to the deployment constraints. The single-solver approach used local rules to find valid deployment targets for each component based on the configuration constraints. When resource and other global constraints began being added, we had to adopt a multi-solver approach to solve deployments.

As can be seen in Figure 3, local constraint solving is the initial step of our automatic constraint-aware deployment. We used AUTODeploy to check the feasibility of our multi-solver approach. The local constraints in AUTODeploy correspond to the configuration constraints, such as required OS, that impact only the valid target ECU sites for a single component. The solution space initially contained many millions or more possible deployment combinations, as shown in step 1 of Figure 3. Any global constraint-aware deployment solving began by iterating over each unassigned component and considering only valid ECUs respecting the configuration constraints. After pruning the solution space,

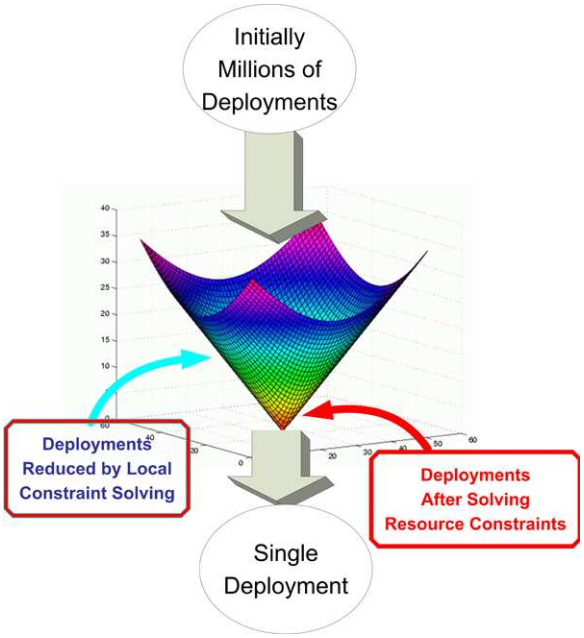

Figure 3. Layered Deployment Solving

global constraints, such as resource requirements, are considered, as shown in step 2 of Figure 3. After solving the global constraints, AUTODeploy is left with a drastically reduced number of deployment solutions to select from. At this point, depending on the number of solutions available, optimization algorithms can be applied to select a solution that optimizes a particular criteria, such as the number of ECUs used.

In many cases, we observed that domain experts could not formally specify certain types of constraints, such as political, legacy, or vendor-specific constraints. For these types of situations, we found it was essential to allow modelers to fix certain modeling decisions so that the automated role-based constraint solvers could not override them. We therefore introduced a mechanism into AUTODeploy so that developers could first fix the deployment locations of certain components with un-specifiable constraints an then use the constraint solver to complete the partially specified deployment.

3.5 Model Completion Solvers

Global constraint solvers and their respective suggestions are necessary for solving complex problems involving multiple model entities and roles. For example, in AUTODeploy, we developed a suggestive solver that not only checked configuration constraints, but also checked resource constraints for all components and derived a valid

TargetECU for every component. Nevertheless, automatic global constraint solvers should integrate into the modeling experience and not be separate from visual modeling. Automated role-based constraint solving therefore uses the role-based constraint rules to provide the glue between the visual modeling experience and user-defined global constraint solvers.

For example, the following Prolog rule

```
batch_assign_component_targetECU(Component,
  ECUs, Best, MaxSolutions, FitnessFunc) :-
  ...
```

represents a role-based constraint solver for finding valid connections between a set of components and ECUs. The modeling step of connecting a component to a valid ECU is accompanied by a menu item offering to perform such a step globally, *i.e.*, assign a target ECU for each component. After deploying several critical components to some ECUs by using the step-wise guidance, modelers can trigger the constraint-aware batch processor to request a complete deployment. The domain-specific GEMS editor will then attempt to calculate a deployment structure and if one is found, the corresponding connections will appear in the model. This type of batch processing can aim for an optimal deployment structure by using constraint-based Prolog programs, such as CPL(X) [16]. It can also integrate domain-specific heuristics known to domain modelers, as described in Section 3.2.

There are many existing algorithms and libraries developed by the artificial intelligence community and other Prolog experts that can be used and adapted while performing complex model analysis. In other work, we have used the automated role-based constraint solving capabilities of GEMS to integrate an existing Prolog Qualitative Differential Equation simulator, based on the QSim algorithm, into one of our DSMLs. The integration was straightforward and required less than 100 lines of Prolog code.

The drawbacks of allowing modelers to fix certain modeling decisions becomes apparent when batch processing is applied. Modelers can build models with no valid global deployment that satisfies the supplied constraints and the set of available ECUs. For these situations, we further developed our automated role-based constraint solving approach to allow modelers to provide model repair operations that could be leveraged by the constraint solver, as discussed next.

3.6 Model Repair

It quickly became apparent that an AUTODeploy model could be defined with various errors, such as conflicting constraints or insufficient resources, that would make deployment impossible. With numerous complex composition rules guiding the deployment process, it was hard for modelers to figure out *why* there was no valid way of deploying the components and *how* to repair the model to overcome the problem. Simply failing to deploy the components and not providing an explanation would leave the reasoning of the underlying cause to modelers, without any hints on possible modifications (such as resource expansions) to make it work. In these situations, deducing the errors in the model could be as hard as finding a valid concern composition manually.

A key question we needed to address, therefore, was what type of feedback should be provided to modelers. One approach we evaluated was marking model elements, such as components, that could not satisfy their domain constraints. For example, we considered marking components with resource requirements exceeding the available resources of any available ECU. We found this approach unsatisfactory for the following reasons:

- For global constraints, such as resource constraints, the overall state of the system determines whether or not the constraint succeeds. In the automotive domain, if the ECUs do not provide sufficient resources to ECU all of the components, more than a single component may be causing the problem. Marking the first component that could not be placed would therefore make no sense since different packing orders could result in different components marked as the cause of failure.
- Even if the cause of the failure was marked in some manner, modelers would still need to manually determine how to modify the model from its present state to make it compliant with its constraints. Although fixing the model might appear trivial when the failing constraint was identified, changing the model could have unforseen affects on the other domain constraints. Again, manual approaches do not scale for these types of constraint satisfaction problems.

We adopted a strategy in automatic role-based constraint solving of allowing modelers to express a set of legal model modifications that could be performed, which we call "repair operations." We then used the role-based constraint solvers to apply these repair operations to the model to make deployment possible. Repair operations can be any valid modeling action, such as changing attribute values, adding or removing modeling elements, or creating connection and containment relationships. For example, a repair operator *IncreaseCPUPower* could be used to allow the constraint solver to place a component on a ECU or if no suitable ECU was found to increase the power (a suggestion to improve the hardware) of an upgradeable ECU. By specifying a series of repair operations, such as *IncreaseCPUPower*, *IncreaseRAM*, and *AddECU*, the constraint solver could first try to upgrade an existing ECU or if none could be upgraded, add an ECU to the merged model.

Suggesting corrective model changes can be applied to both failed local constraints or global constraints. For in-

stance, modelers could try to deploy a component manually and find that the automatic deployment guidance does not provide any valid ECUs. This failure might occur if no ECU matching the configuration requirements of the component (*e.g.*, the required operating system or target ECU type). Another reason for failure could be that all resources of valid ECUs have been exhausted by previous component assignments. Corresponding suggestions could therefore be to create a compliant ECU or to increase respective resources of a single ECU. Conversely, a global solver could use the repair operations to apply a batch of corrections to the model to make deployment possible.

The key concept enabling repair operations was our extension of the automatic role-based constraint solvers by adding additional parameters for the repair operations. For example, consider the format of the component-targetECU constraint again:

```
is_a_valid_component_targetECU(
  Component, ECU,
  RepairOperations,
  DoneModificationOperatorL).
```

This constraint can be used to pass the following operator to a call of `is_a_valid_component_targetECU`:

```
modify_resource_increment_by_factor(
  Component, ECU,
  ApplicationMode,
  InputArgs, OutputArgs).
```

This rule uses the same pair of component/ECU variables. In addition, there are some input arguments and output arguments, *e.g.*, the third application mode parameter specifies whether

- The correction operator should check the repair operation's applicability (Mode = try) to the current model,
- Perform the repair operation and record them in the Prolog record database (Mode = do), or
- Undo a repair operation that has already been performed (Mode = undo) by removing the respective previous repair recordings.

Distinguishing these three modes is essential to keep the modularity of all correction related activities.

A modification solver capable of increasing a resource capacity of a ECU must first check whether the currently considered invalid component/ECU pair is caused by a lack of resources on the ECU and whether or not the insufficient resources can be increased. Once the repair operation is deemed appropriate, it is applied to the model using the 'do' mode. Calling the modification solver with the 'undo' mode allows it to remove a suggested modification from the Prolog recording database, which allows it to undo all the repairs performed by an operator. This mechanism is essential since the Prolog constraint solver may discover that the repair operations it has performed must be undone to allow it to backtrack and undo some component to ECU assignments it has made.

The role-based constraint solving model repair capabilities described above go beyond standard Prolog tracing. Standard Prolog tracing would track execution down to the point of any assignment problem and force the modeler to figure out the reason behind a failure to find proper solution. In contrast, these model repairs raise the level of abstraction by specifying possible domain-specific corrections within the underlying domain structure and domain entities.

4 Case Study: Applying AUTODeploy to EAST-EEA Deployments

To validate our automatic role-based constraint solving paradigm, we created a DSML called AUTODeploy that model EAST-EEA deployment problems. AUTODeploy enables users to specify partial solutions as sets of components, requirements, ECUs, and resources. It can produce both valid assignments for a single component's TargetECU role and global assignments for the TargetECU role of all components.

It is often the case in the automotive domain that certain software components cannot be moved between ECUs from one model car to the next due to manufacturing, quality assurance, or other concerns. In these situations, developers must be able to fix the TargetECU role of certain components and allow the tool to solve for valid assignments of the remaining unassigned component TargetECU roles. There are also cases where it is infeasible to specify formal rules, so instead the knowledge of domain engineers must be used to assign critical components to ECUs. There are typically only a few of these cases and the rest of the assignments can be calculated automatically. AUTODeploy can therefore complete a partially specified deployment of components to ECUs, if a valid deployment exists.

The first step in developing AUTODeploy was to create a deployment metamodel that defines a DSML that users can apply to model components with arbitrary configuration and resource requirements and ECUs with arbitrary sets of provided resources. Each component configuration requirement is specified as an assertion on the value of a resource of the assigned TargetECU. For example, *OSVersion* $>$ *3.2* would be a valid configuration constraint.

We then created resource constraints by specifying a resource name and the amount of that resource consumed by the component. Each ECU was not allowed to have more components deployed to it than its resources could support. Each ECU in turn could provide an arbitrary number of resources. Figure 4 shows an screenshot from AUTODeploy.

Typical resource requirements were the RAM usage and CPU usage. Resource requirements were specified using $<$, $>$, - and = signs to denote that the value of the resource with the same name and type (for example OS version) must

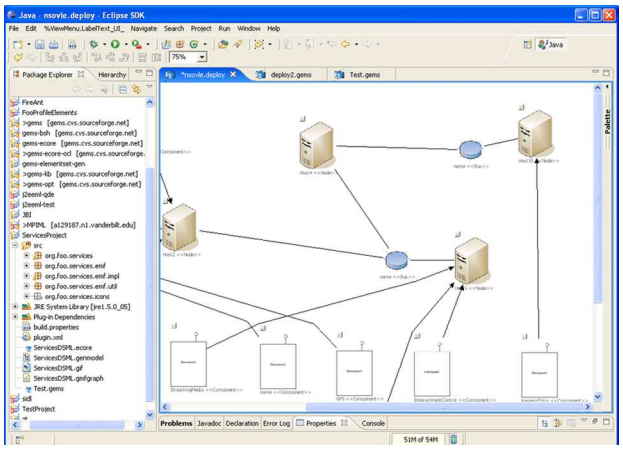

Figure 4. An AUTODeploy Deployment Diagram Showing Component to ECU Mappings

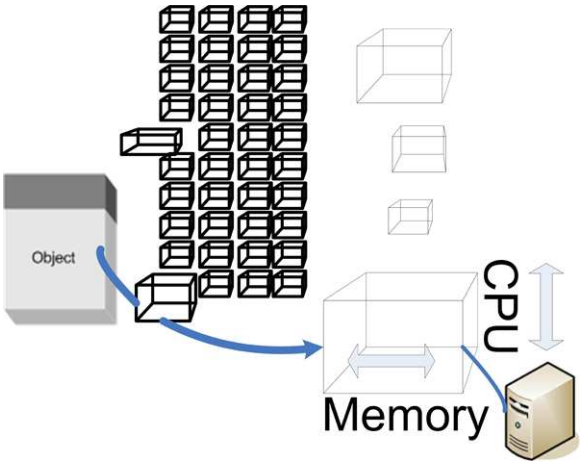

Figure 5. Resource Constraint Mapping to Bin-Packing

be less, greater or equal to the value specified in requirement. The "-" relationship indicates a summation constraint or that the total value of the demands on a resource, by the components deployed to the providing ECU, does not exceed the amount present on the ECU.

This method of requirements specification blends both the flexibility and intuitiveness of a textual approach with the concrete meaning of a constraint solver format. The Name can be any string and thus modelers can create meaning by providing very descriptive names. The Type provides a clear definition of how the constraint is compared to the resources available on a candidate node. The Type also indicates exactly which constraint solver must be used to analyze the constraint.

4.1 Solver Implementations

After defining the metamodel and generating the graphical editor for deployment DSML using GEMS, we added a set of automatic role-based constraint solvers to enforce the configuration and resource constraint semantics of our models.

Our constraint rules specified that for each child requirement element of a component, a corresponding resource child of the TargetECU has to satisfy the requirement. The resource constraints were mapped to a bin packing problem. A bin-packing problem is defined by a set of items with various dimensions and a set of bins with various capacities that the items must be packed into. With AUTODeploy, items and bins can be of an arbitrary number of dimensions, so we provided rules for transforming a set of ECUs and components into a list of n-dimensional items and n-dimensional bins, as can be seen in Figure 5.

This mapping of resource constraints to bin-packing is created by the following rules that map requirements of type '-' to the various N-dimensions of the bins.

```
requirement_spec(Req, Name, Type) :-
    self_type(Req, requirement),
    self_name(Req, Name),
    self_resourcetype(Req, Type).
resource_spec(Res, Name, Type) :-
    ...
requirement_resource_valid_pair(Req, Res) :-
    ...
requirement_to_resource(Req, ECU, Res) :-
    ...
```

Our solvers and repair operators were written so that they could be applied to any kind of role-based relationship and also reused across metamodels with minor amendments only. We found that the right way to do this was by using the bridge pattern [15] to decouple the actual problem specification implementation from its abstract interface. We defined an abstract bin-packing problem interface that could be re-implemented for different metamodels to change the modeling elements and attributes the bin-packer operated on.

4.2 Repair Operator Implementation

The repair operators we built work in three phases. First the operators collect all bin-packing requirement/resource pairs where the requirement exceeds the remaining resource capacity. These tuples are then passed to a rule which tests each repair operator using the 'try' mode to see if the deficient resource can be fixed with the operator. If the operator can correct the deficiency, the operator is applied with the do mode. By invoking the 'do' mode of the operator, each

exceeded resource is expanded by the minimum necessary to accommodate the failed requirement.

Repair operators can be applied in combination, such as first adding a set of missing configuration resources and then adding missing bin-packing resources. We applied our global solver with two repair operators (adding missing configuration resources and adapting the resource capacities) to an entire 300 component and 80 ECU deployment in ∼4.5 seconds. In this test, the modification operators corrected deficiencies in 25 nodes that prevented the deployment of 80 components.

An interesting result is that we discovered the repair operators could be used to plan for future expansions to the deployment model. For example, designers could take an existing automotive deployment model and specify the new components that would be added in the next year's model. The deployment solver would then use the repair operators to find the most efficient way of upgrading the existing infrastructure to support the new functionality.

5 Related Work

Decision support systems are similar to the automated role-based constraint solving approach proposed in this paper. In [5], Achour and all propose a modeling tool based on the Unified Medical Language System (UMLS), to create KBs for diagnosing and treating diseases. Both their approach and the automated role-based constraint solving approach attempt to glean domain knowledge and constraints from an expert and simplify a users ability to find the correct solution to a partially specified problem. For the UMLS-based decision support system, the goal is to, given a set of patient condition information, find the appropriate diagnosis and course of treatment.

The approach in this paper differs significantly from the research proposed in [5]. First, the research proposed here facilitates the creation of decision support systems for any domain-specific modeling language. Moreover, automated role-based constraint solving is not limited solely to decision tree guidance, but also complex analysis and optimizations specified by users. Finally, the automated role-based constraint solving proposed in this paper is generated from a metamodel and integrated with a graphical modeling tool. GEMS and its solver generation capabilities are a tool for creating graphical modeling tools with integrated modeling decision support for arbitrary domains.

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [17, 8, 19, 7, 14]. These tools provide mechanisms for describing domain-constraints, a set of knowledge, and finding solutions to the constraints. These tools, however, are not designed to generate domain-specific solvers based on a metamodel. These tools also do not support the generation of a DSML graphical environment and integrated graphical suggestions. Finally, as discussed previously, these tools do not provide automation of the problem specification as our GEMS-based role-based constraint solvers do.

The Eclipse Graphical Modeling Framework (GMF) [1] provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. Similar to GEMS, GMF supports meta-modeling and can generate graphical editors for a meta-model. Both GEMS and GMF use EMF as the in-memory model representation. The core advantage of GEMS is the role-based constraint solving capabilities for guiding complex modeling tasks. As we have shown, for realistic size models, manual modeling approaches do not work. GEMS, unlike GMF, provides extensive support for handling large and complex models with constraint solver assistance.

6 Concluding Remarks

The work presented in this paper addresses the scalability problems of manual modeling approaches. These scalability issues are particularly problematic for domains that have large solutions spaces and few correct solutions. In such domains, it is extremely time consuming to handcraft correct models, so some type of automatic constraint solver is needed. Moreover, transforming a DSML instance into a native constraint solver input format is a time-consuming task.

Our solution generates a DSML editor that encompasses a semantically rich knowledge base in Prolog format and allows users to specify constraints in declarative format that can be used to derive modeling suggestions. The key advantages of this approach are that

- The same set of constraints can be used to check whether a manually defined model is correct and to generate valid solutions by keeping open some parameters, such as TargetHost.
- Modelers can specify constraints using the domain-specific notation of the knowledge base and specify constraint solver rules in a domain-specific manner.
- The role-based constraint solving model repair capabilities allow modelers to not only construct valid models but to repair broken models, which is a key capability for complex modeling domains.

From our work thus far, we learned the following lessons about how to write effective role-based constraint solvers, as well as which problems are still hard to solve:

- Certain types of constraint comparisons, such as summation comparisons, require significantly more work to solve. For these constraint types, it is critical that the generated solvers include templatizations of known solution algorithms since they are too costly to reinvent and rediscover.

- Many complex problems we faced have been solved using Prolog in the past and validated with expert system approaches. The solvers generated by GEMS can often easily incorporate these existing algorithms with a minimal amount of code.
- Automating the transformation from the modeling domain to native constraint solver input formats is a crucial element of reducing the cost of integrating a constraint solver with a modeling environment. The level of abstraction used to specify constraints must also be raised to make constraint solvers usable by domain experts.
- Optimization of a model is a much harder than finding a valid solution. In our EAST-EEA deployments, for example, we created algorithms that minimized the number of nodes used by a deployment. These algorithms, however, do not scale well if there are a large number of valid solutions. More work is needed to provide templatizable optimization rules.
- Constraint solvers can easily be converted to repair mechanisms by adding a simple check to see if an assignment decision meets the domain constraint *OR* meets a relaxed constraint specified by a repair operator. We have found that adding these checks is very straightforward.

In future work, we plan to apply role-based constraint solving to multiple combinatorially challenging domains, such as service configuration, failure analysis, and workflow composition. We are also collaborating with optimization researchers to develop reusable solution algorithms that can be generated automatically.

References

[1] GMF, Eclipse Graphical Modeling Framework. www.eclipse.org/gmf.

[2] JPL, A Java interface to Prolog. www.swi-prolog.org/packages/jpl/java_api/index.html.

[3] EAST-EEA Embedded Electronic Architecture Web Site - www.east-eea.net. 2004.

[4] Automotive Open System Architecture - www.autosar.org/fi nd02_ns6.php. 2006.

[5] S. L. Achour, M. Dojat, C. Rieux, P. Bierling, and E. Lepage. A UMLS-based Knowledge Acquisition Tool for Rule-based Clinical Decision Support System Development. *Journal of the American Medical Information Association*, 8(4):351–360, July 2001.

[6] M. Calejo. Interprolog: Towards a Declarative Embedding of Logic Programming in Java. *Theory and Practice of Logic Programming, Lecture Notes in Computer Science*, 3229:714–717, 2004.

[7] Y. Caseau, F.-X. Josset, and F. Laburthe. CLAIRE: Combining Sets, Search And Rules To Better Express Algorithms. *Theory and Practice of Logic Programming*, 2:2002, 2004.

[8] J. Cohen. Constraint Logic Programming Languages. *Commun. ACM*, 33(7):52–68, 1990.

[9] O. for blind review. Application of Aspect-based Modeling and Weaving for Complexity Reduction in the Development of Automotive Distributed Real-time Embedded Systems: Technical Report ISIS-06-709.

[10] O. for blind review. Simplifying the Development of Product-Line Customization Tools via MDD. In *Workshop: MDD for Software Product Lines, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, October 2005.

[11] O. for blind review. The J3 Process for Building Autonomic Enterprise Java Bean Systems. *icac*, 00:363–364, 2005.

[12] O. for blind review. Reducing Enterprise Product Line Architecture Deployment Costs via Model-Driven Deployment and Confi guration Testing. In *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2006.

[13] R. for Blind Submission. Introduction to the Generic Eclipse Modeling System. *Eclipse Magazine, Software and Support Media*, 4, 2007 (to appear).

[14] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, November 2002.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[16] J. Jaffar and J. Lassez. *Constraint Logic Programming*. ACM Press New York, NY, USA, 1987.

[17] L. Michel and P. V. Hentenryck. Comet in Context. In *PCK50: Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge*, pages 95–107, New York, NY, USA, 2003. ACM Press.

[18] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Effi cient Deductive Database Engine. pages 442–453, 1994.

[19] G. Smolka. The Oz Programming Model. In *JELIA '96: Proceedings of the European Workshop on Logics in Artifi cial Intelligence*, page 251, London, UK, 1996. Springer-Verlag.

[20] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *Computer*, 30(4):110–111, 1997.

[21] J. Wielemaker. SWI-Prolog Reference Manual. University of Amsterdam, The Netherlands, gollem.science.uva.nl/SWI-Prolog/Manual/, 1997, 1997.