Applying Model Intelligence Frameworks for Deployment Problem in Real-Time and Embedded Systems

Andrey Nechypurenko,
Egon Wuchner
Siemens AG,
Corporate Technology (SE 2)
Otto-Hahn-Ring 6
81739 Munich, Germany
{andrey.nechypurenko,
egon.wuchner}@siemens.com

Jules White,
Douglas C. Schmidt
Vanderbilt University,
Department of Electrical Engineering and
Computer Science
Box 1679 Station B
Nashville, TN, 37235, USA
{jules, schmidt}@dre.vanderbilt.edu

ABSTRACT

There are many application domains, such as distributed real-time and embedded (DRE) systems, where the domain constraints are so restrictive and the solution spaces so large that it is infeasible for modelers to produce correct solution manually using a conventional graphical model-based approach. In DRE systems the available resources, such as memory, CPU, and bandwidth, must be managed carefully to ensure a certain level of quality of service. This paper provides three contributions to simplify modeling of complex application domains: (1) we present our approach of combining model intelligence and domain-specific solvers with model-driven engineering (MDE) environments, (2) we show techniques for automatically guiding modelers to correct solutions and how to support the specification of large and complex systems using intelligent mechanisms to complete partially specified models, and (3) we present the results of applying an MDE tool that maps software components to Electronic Control Units (ECUs) using the AUTOSAR automotive modeling and middleware standard.

1. INTRODUCTION

Graphical modeling languages, such as UML, can help to visualise certain aspects of the system and automate particular development steps via code-generation. Model-driven engineering (MDE) tools and domain-specific modeling languages (DSMLs) [7] are graphical modeling technologies that combine high-level visual abstractions that are specific to a domain with constraint checking and code-generation to simplify the development of certain types of systems. In many application domains, however, the domain constraints are so restrictive and the solution spaces so large that it is infeasible for modelers to produce correct solutions manually. In these domains, MDE tools that simply provide solution correctness checking via constraints provide few benefits over conventional approaches that use third-generation languages.

Regardless of the modeling language and notation used, the inherent complexity in many application domains is the combinatorial nature of the constraints, and not the code construction per se. For example, specifying the deployment of software components to hardware units in a car in the face of configuration and resource constraints can easily generate solution spaces with millions or more possible deployments and few correct ones, even when only scores of model entities are present. For these combinatorially complex modeling problems, it is impractical, if not impossible, to create a complete and valid model manually. Even connecting hundreds of components to scores of nodes by pointing and clicking via a GUI is tedious and error-prone. As the number of modeling elements increases into the thousands, manual approaches become infeasible.

To address the challenges of modeling combinatorially complex domains, therefore, we need techniques to reduce the cost of integrating a graphical modeling environment with *Model Intelligence Guides (MIGs)*, which are automated MDE tools that help guide users from partially specified models, such as a model that specifies components and the nodes they need to be deployed to but not how they are deployed, to complete and correct ones, such as a model that not only specifies the components to be deployed but what node hosts each one. This paper describes techniques for creating and maintaining a *Domain Intelligence Generator* (DIG), which is an MDE that helps modelers solve combinatorially challenging modeling problems, such as resource assignment, configuration matching, and path finding.

The rest of the paper is organised as follows: Section 2 discusses challenges of creating deployment models in the context of the AUTOSAR[3] middleware and modeling standard, which we use as a motivating example; Section 3 describes key concepts used to create and customize MIGs; Section 4 shows the results of applying MIGs to AUTOSAR component deployments; and Section 5 presents concluding remarks and outlines future work.

2. MOTIVATING EXAMPLE

MODELS06WS6 '06 Genova, Italy

AUTOSAR is a new standard for automotive middleware and software development modeling [3]. The goal of AUTOSAR is to standardize solutions to many problems that arise when developing large-scale, distributed real-time and embedded (DRE) systems for the automotive domain. For instance, concert efforts is required to relocate components between Electronic Control Units (ECUs), *i.e.*, computers and micro-controllers running software components within a car. Key complexities of relocation include: (1) components often have a many constraints that need to be met by the target ECU and (2) there are many possible deployments of components to ECUs in a car and it is hard to find the optimal one.

For example, it is hard to manually find a set of interconnected nodes able to run a group of components that communicate via a bus. Modelers must determine whether the available communication channels between the target ECUs meet the bandwidth, latency, and framing constraints of the components that communicate through them. In the automotive domain—as with other embedded systems domains— it is also important to reduce the overall cost of the solution, which necessitates optimizations, such as finding deployments that use as few ECUs as possible or minimize bandwidth to allow cheaper buses. It is infeasible to find these solutions manually for a production systems.

To illustrate the practical benefits of generating and integrating MIGs with a DSML, we describe an MDE tool we developed to solve AUTOSAR constraints for validly deploying software components to ECUs. There are two primary architectural views in AUTOSAR systems:

- The *logical collaboration structure* that specifies which components that should communicate with each other via which interfaces, and
- The *physical deployment structure* that captures the capabilities of each ECU, their interconnecting buses, and their available resources.

Historically, AUTOSAR developers have manually specified the mapping from components in the logical view to ECUs in the physical view via MDE deployment tools, as shown in Figure 1. This approach worked relatively when when there were a small number of components and ECU. Modern cars, however, can be equipped with 80 or more ECUs and several hundred or more software components. Simply drawing arrows from 160 components to 80 ECUs is tedious. Moreover, many requirements constrain which ECUs that can host certain components, including the amount of memory required to run, CPU power, programming language, operating system type and version, etc. These constraints must be considered carefully when deciding where to deploy a particular component. The problem is further exacerbated when developers consider the physical communication paths and aspects, such as available bandwidth in conjunction with periodical real-time messaging.

The remainder of this paper how the AUTOSAR MDE tool we developed helps automate the mapping of software components to ECUs in AUTOSAR models without violating

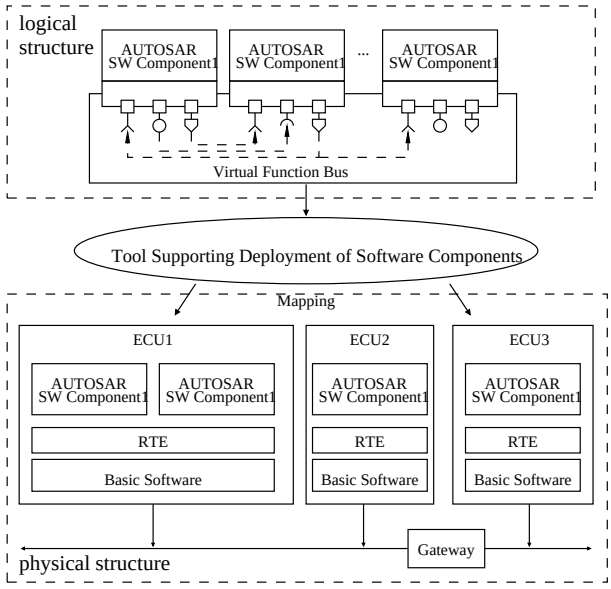

Figure 1: Mapping from the logical collaboration to the physical deployment structure

the known constraints. The following sections describe our approach and show how MIGs can significantly reduce the complexity of creating AUTOSAR deployment models.

3. DOMAIN-SPECIFIC MODEL INTELLIGENCE

Based on the challenges related to the AUTOSAR example presented in Section 2, the goals of our work on MIGs are to (1) specify an approach for guiding modelers from partially specified models to complete and coorrect ones and (2) automate the completion of partially specified models using information extracted from domain constraints.

In previous work [9, 8], we showed how MDE tools and DSMLs can improve the modeling experience and bridge the gap between the problem and solution domain by introducing domain-specific abstractions. At the heart of these efforts is the *Generic Eclipse Modeling System* (GEMS), which provides a convenient way to define the metamodel, *i.e.*, the visual syntax of the DSML. Given a metamodel, GEMS automatically generates a graphical editor that enforces the grammar specified in the DSML. GEMS provides convenient infrastructure (such as built-in support for the Visitor pattern[5]) to simplify model traversal and code generation. We used GEMS as the basis for our MIGs AUTOSAR deployment modeling tool and our work on domain-specific model intelligence.

3.1 Domain Constraints as the Basis for Automatic Suggestions

A key research challenge was determining how to specify the set of model constraints so they could be used by MIGs *not only to check the correctness of the model, but also to guide users through a series of model modifications to bring it to a state that satisfies the domain constraints.* We considered various approaches for constraint specification language, in-

cluding Java, the Object Constraint Language (OCL), and Prolog. To evaluate the pros and cons of each approach, we implemented our AUTOSAR deployment constraints in each of the three languages.

As a result of this evaluation, we selected Prolog since it provided both constraint checking and model suggestions. In particular, Prolog can return the set of possible facts from a knowledge base that indicate why a rule evaluated to "true." The declarative nature of Prolog significantly reduced the number of lines of code written to transform an instance of a DSML into a knowledge base and to create constraints (its roughly comparable to OCL for writing constraints). Moreover, Prolog enables MIGs to derive sequences of modeling actions that converts the model from an incomplete or invalid state to a valid one. As shown in Section 1, this capability is crucial for domains, such as deployment in complex DRE systems, where manual model specification is infeasible or extremely tedious and error-prone.

The remainder of this section describes how Domain Intelligence Generation (DIG) uses Prolog and GEMS to support the creation of customizable and extensible domain-specific constraint solver and optimization frameworks for MIGs. Our research focuses on providing modeling guidance and automatic model completion, as described below.

3.2 Modeling Guidance on-the-fly

To provide domain-specific model intelligence, an MDE tool must capture the current state of a model and reason about how to assist and guide modelers. To support this functionality, MIGs use a Prolog knowledge base format that can be parameterized by a metamodel to create a domain-specific knowledge base. GEMS metamodels represent a set of model entities and the role-based relationships between them. For each model, DIG populates a Prolog knowledge base using these metamodel-specified entities and roles. For each entity, DIG generates a unique id and a predicate statement specifying the type associated with it.

In the context of our AUTOSAR example, a 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 entity it is relating and the value it is relating it to. For example, if a component with id 23 has a *TargetHost* relationship to a node with id 25 the predicate statement *targethost(23,25)* is generated. This predicate statement specifies that the entity with id 25 is a *TargetHost* of the entity with id 23. Each knowledge base generated by DIG provides a domain-specific set of predicate statements.

The domain-specific interface to the knowledge base provides several advantages over a generic format, such as the format used by a general-purpose constraint solver like for exampl CLIPS. First, the knowledge base maintains the domain-specific notations from the DSML, making the format more intuitive and readable to domain experts. Second, maintaining the domain-specific notations allows the specification of constraints using domain notations, thereby enabling developers to understand how requirements map to constraints. Third, in experiments that we conducted, writing constraints using the domain-specific predicates produced rules that had fewer levels of indirection and thus outperformed rules written using a generic format. In general, the size of the performance advantage depended on the generality of the knowledge base format. To access properties of the model entities, the predicate syntax presents the most specific knowledge base format. Given an entity id and role name, the value can be accessed with the statement *role(id, Value)*, which has exactly zero or one facts that match it.

Based on this domain-specific knowledge base, modelers can specify user-defined constraints in form of Prolog rules for each type of metamodel relationship. These constraints semantically enrich the model to indicate the requirements of a correct model. They are also used to automatically deduce the sets of valid model changes to create a correct model.

For example, consider the following constraint to check if a node (ECU) is a valid host of a component: *is_a_valid_component_targethost(Comp, Nodes).* It can be used to both check a Component/Node combination (*e.g.*, *is_a_valid_component_targethost(23,[25]).*) and to find valid *Nodes* that can play the *TargetHost* role for a particular component (*e.g.*, *is_a_valid_component_targethost(23, Nodes).*). The latter example uses Prolog's ability to deduce the correct solution, *i.e.*, the *Nodes* variable will be assigned with the list of all constraint-valid nodes for the *TargetHost* role of the specified component. This example illustrates how constraints can be used to check *and* to generate the solution, if one exists.

Figure 2 shows how dynamic suggestions from Prolog are presented to modelers. The upper part of the figure shows

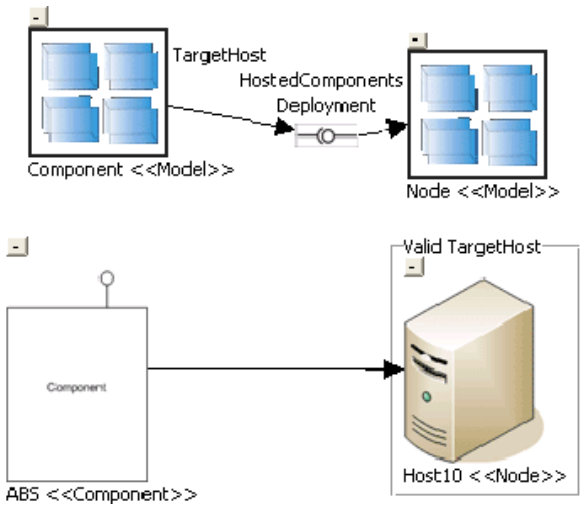

Figure 2: Highlighting valid target host

the fragment of the metamodel that describes the *Deployment* relationship between *Component* and *Node* model entities. The lower part of the picture shows how the generated editor displays the corresponding entity instances. This screenshot was made at the moment a modeler had begun dragging a connection begining from the "ABS" compo-

nent. The rectangle around "Host10" labelled "Valid TargetHost" is drawn automatically as a result of triggering the corresponding solver rule and receiving a valid solution as feeback. GEMS also can also trigger arbitrary Prolog rules from the modeling tool and incorporate their results back into a model. This mechanism can be used to solve for complete component to ECU deployments and automatically add deployment relationships based on a (partially) complete model.

To enable modeling assistance, different subsystems must collaborate within the modeling environment. It is the responsibility of the modeler (or MDE tool creator) to provide the set of constraints and supply solvers for new constraint types. The GEMS metamodel editor updates the knowledge base and incorporates the new rules into the generated MIG. User-defined solver(s) can be based on existing Prolog algorithms, the reusable rules generated by GEMS, or a hybrid of both. Solvers form the core of the basic MIG generated by GEMS. Below we describe the solver we developed for completing partially specified models in our AUTOSAR deployment tool.

3.3 Model Completion Solvers

Using a global deployment (completion) solver, it is possible to ask for the completion of partially specified models constrained by user-defined rules. For example, in the AUTOSAR modeling tool, the user can specify the components, their requirements, the nodes (ECUs), and their resources and ask the tool to find a valid deployment of components to nodes. After deploying the most critical components to some nodes by using MIGs step-wise guidance, modelers can trigger a MIG global deployment solver to complete the deployment. This solver attempts to calculate an allocation of components to nodes that observes the deployment constraints and update the connections between components and nodes accordingly. This global solver can aim for an optimal deployment structure by using constraint-based Prolog programs or it could integrate some domain-specific heuristics, such as attempting to find a placement for the components that use the most resources first.

In some cases, however, the modeled constraints cannot be satisfied by the available resources. For example, in a large AUTOSAR model, a valid bin-packing of the CPU requirements for the components into EPUs may not exist. In these cases the complexity of the rules and entity relationships could make it extremely hard to deduce *why* there is no solution and *how* to change the model to overcome the problem. For such situations, we developed a solver that can identify failing constraints and provide suggestions on how to change the model to make the deployment possible.

4. CASE STUDY: SOLVING AUTOSAR DEPLOYMENT PROBLEM

To validate our DIG MDE tool, we created a DSML for modeling AUTOSAR deployment problems. This DSML enables developers to specify partial solutions as sets of components, requirements, nodes (ECUs), and resources. A further requirement was that the MIGs should produce both valid assignments for a single component's *TargetHost* role and global assignments for the *TargetHost* role of all components. In the automotive domain certain software components often cannot be moved between ECUs from one model car to the next due to manufacturing costs, quality assurance, or other safety concerns. In these situations, developers must fix the *TargetHost* role of certain components and allow MIGs to solve for valid assignments of the remaining unassigned component *TargetHost* roles.

For the first step, we created a deployment DSML metamodel that allows users to model components with arbitrary configuration and resource requirements and nodes (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 *TargetHost*. For example, *OSVersion* $>$ *3.2* would be a valid configuration constraint. Resource constraints were created by specifying a resource name and the amount of that resource consumed by the component. Each Node could only have as many components deployed to it as its resources could support. Typical resource requirements were the RAM usage and CPU usage.

Each host can provide an arbitrary number of resources. Constraints comparisons on resources were specified using the $<$, $>$, -, and $=$ relational operators to denote that the value of the resource with the same name and type (*e.g.*, OS version) must be less, greater, or equal to the value specified in requirement. The "-" relationship indicates a summation constraint, *i.e.*, the total value of the demands on a resource by the components deployed to the providing node must not exceed the amount present on the node. After defining the metamodel and generating the graphical editor for the deployment DSML using GEMS, we added a set of Prolog constraints to enforce the configuration and resource constraint semantics of our models.

4.1 Defining Constraints and Solvers

Our constraint rules specified that for each child requirement element of a component, a corresponding resource child of the TargetHost must satisfy the requirement. Our complete configuration constraint rules are as following.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% specifying validness of a requirement-resource
% pair
requirement_resource_valid_pair(Req, Res) :-
    (requirement_spec(Req, Name, '>'),!
     ;
     requirement_spec(Req, Name, '<'),!
     ;
     requirement_spec(Req, Name, '=')
    ),
    resource_spec(Res, Name, '=').

requirement_to_resource(Req, Host, Res) :-
    requirement(Req),
    resource_to_node(Res, Host),
    requirement_resource_valid_pair(Req, Res).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% configuration requirement resource solver
comparevalue(V1,V2,'>') :- V1 > V2.
comparevalue(V1,V2,'<') :- V1 < V2.
comparevalue(V1,V2,'=') :- V1 == V2.
```

```
requirement_resource_constraint(Req, Res) :-
    requirement(Req),
    self_type(Req, Type),
    (Type = '<' ; Type = '>' ; Type = '='),
    !,
    resource(Res),
    self_value(Res, ResValue),
    self_value(Req, ReqValue),
    comparevalue(ResValue, ReqValue, Type).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% local role base component targethost
% relationship solver
is_a_valid_component_targethost(Owner, Value) :-
( self_targethost(Owner, [Value]), ! %deployed
  ;
  (is_a(Value,node),
   self_requires(Owner, Requirements),
   forall( member(Req,Requirements)
     ,
     (requirement_to_resource(Req, Value, Res),
     requirement_resource_constraint(Req, Res))
) ) ).
```

These lines of code are the *entire* solution, providing not only configuration constraint checking for an arbitrary set of requirements and resources but also enabling domain-specific GEMS editors to provide valid suggestions for deploying a component. Moreover, this solution was intended as a proof-of-concept to validate the approach and thus could be implemented with even fewer lines of code. The rest of the required predicates to implement the solver were generated by GEMS.

In our experiments with global solvers, Prolog solved a valid global deployment of 900 components to 300 nodes in approximately 0.08 seconds. This solution met all configuration constraints.

The rules required for solving for valid assignments using resource constraints were significantly more complicated since resource constraints are a form of bin-packing (an NP-Hard problem). We were able to devise heuristic rules in Prolog, however, that could solve a 160 component and 80 ECU model deployment in approximately 1.5 seconds and an entire 300 component and 80 ECU deployment, a typical AUTOSAR sized problem, in about 3.5 seconds. These solution times are directly tied to the difficulty of the problem instance. For certain instances, times could be much higher, which would make the suggestive solver from Section 3 discussed in the previous section applicable. In cases where the solver ran too long, the suggestive solver could be used to suggest ways of expanding the underlying resources and making the problem more tractable.

5. RELATED WORK

Many complex modeling tools are available for describing and solving combinatorial constraint problems, such as those presented in [2, 6, 4]. 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 generated domain-specific solvers based on a metamodel. These tools also do not support the generation of a DSML graphical environment and integrated graphical suggestions. In contrast, our domain-specific model intelligence, based on GEMS, is automatically integrated with any DSML tool generated from a GEMS metamodel.

Decision support systems are similar to the domain-specific model intelligence approach proposed in this paper. In [1], Achour and all propose a modeling tool based on the Unified Medical Language System (UMLS), to create knowledge bases for diagnosing and treating diseases. Both their UMLS approach and our approach attempt to glean domain knowledge and constraints from an expert and simplify users abilities to find the correct solution to a partially specified problem. Their approach, however, differs significantly from our approach in several ways. First, our approach is designed to facilitate the creation of decision support systems for any domain-specific modeling language. In particular, MIGs are not limited solely to decision tree type guidance but also complex analysis and optimizations specified by users. Second, MIGs are automatically generated from a metamodel and integrated with a graphical modeling tool via GEMS, which supports the creation of graphical modeling tools with integrated modeling decision support for arbitrary domains.

6. CONCLUDING REMARKS

The work presented in this paper addresses scalability problems of conventional 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 often infeasible to create correct models manually, so constraint solvers are therefore needed.

Turning a DSML instance into a format that can be used by a constraint solver is a time-consuming task. Our DIG MDE tool generates a domain-specific constraint solver that leverages a semantically rich knowledge base in Prolog format. It also allows users to specify constraints in declarative format that can be used to derive modeling suggestions.

In future work, we plan to continue our development of templatized solver-frameworks for modeling tools and incorporate new types of constraint solvers into the framework. We plan to investigate the use of both automatic control and monitoring of running systems using domain-specific model intelligence and human-assisted monitoring and control. Finally, we intend to extend our infrastructure to allow other types of constraint solving platforms, such as bin-packing solvers written in C, to be integrated into a GEMS-based modeling environment.

GEMS and the MIGs generation framework is an open-source project available from:
http://www.sf.net/projects/gems.

7. REFERENCES

[1] 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.

[2] J. Cohen. Constraint logic programming languages.

Commun. ACM, 33(7):52–68, 1990.

[3] H. H. et al. Autosar current results and preparations for exploitation. In *7th EUROFORUM conference*, may 2006.

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

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[6] G. Smolka. The oz programming model. In *JELIA '96: Proceedings of the European Workshop on Logics in Artificial Intelligence*, page 251, London, UK, 1996. Springer-Verlag.

[7] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[8] J. White and D. C. Schmidt. 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.

[9] J. White and D. C. Schmidt. Reducing enterprise product line architecture deployment costs via model-driven deployment and configuration testing. In *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2006.