Domain-Specific Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains[*]

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

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

ABSTRACT

Domain-Specific Modeling Languages (DSMLs) are a means of simplifying the development of a large class of systems. There are many domains, however, where the domain constraints are so restrictive and the solution spaces so large that it is extremely difficult for a modeler to manually produce a correct solution using a DSML. For example, modeling the deployment of software components to nodes and observing configuration and resource constraints, even when only a few tens of model entities are present, can easily generate solution spaces with millions or more possibile deployments and few correct ones. This paper address the challenge of creating a Domain-Specific Intelligence Framework (DSIF) to help a modeler solve combinatorically challenging modeling problems. The paper provides five key contributions to the integration of DSMLs and model intelligence. 1) We present experiments showing how a domain-specific knowledge base and domain-specific solution algorithms can greatly reduce the complexity of integrating a solver. 2) We provide methods for templatizing a Knowledge Base (KB) and KB-based constraint solvers so that they can be paramertized by a metamodel. 3) We present experiments validating that a metamodel-parameterized KB and solvers require less code and provide superior performance to generic KBs and solvers. 4) We illustrate methods for using an Observer, templatized by a metamodel, to translate events from the object graph of a DSML tool into DSKB events and DSKB events back to object graph events. 5) Finally, we present a case-study based on a tool for sovling AUTOSAR, an automotive modeling standard, deployment problems.

1. INTRODUCTION

Domain-Specific Modeling Languages (DSMLs) are a means of combining high-level visual abstractions, specific to a domain, with constraint checking and code-generation to simplify the development of a large class of systems[10]. There are many domains, however, where the domain constraints are so restrictive and the solution spaces so large that it is extremely difficult for a modeler to manually produce a correct solution using a DSML. In these domains, modeling tools that merely provide solution-correctness checking via constraints, provide few real benefits over a DSML-less approach. The true complexity in these domains is their combinatorial nature and not code construction. For example, specifying the deployment of software components to hardware units in a car, while observing configuration and resource constraints, even when only a few tens of model entities are present, can easily generate solution spaces with millions or more possibile deployments and few correct ones. For these combinatorially complex modeling problems, it is impractical, if not impossible, to create a complete and valid model manually.

One approach to creating solutions for problems in these domains is to transform partial solutions, such as a listing of components and the nodes they may be deployed to, into a format that can be solved using a general purpose solver, such as a constraint logic programming solver. There are a large number of optimization, constraint solver, and inference engines available that can be utilized for this purpose. As noted in [7], however, *modeling is emerging as a major challenge: automating the formulation of real problems in a suitable form for efficient algorithmic processing* is hard. Transforming an arbitrary graphical DSML model into a format suitable for a general purpose solver is extremely tedious and error-prone. Integrating the results of the solver back into a DSML tool and providing interactive capabilities is also difficult.

To address the challenges of modeling combinatorically complex domains, methods are needed to reduce the cost of integrating Model Intelligence, or mechanisms that can guide the user from a partially specified model to a complete and correct one. Furthermore, these methods should respect the domain-specificity of the modeling tool and provide a flexible mechanism for specifying solvers using domain notations.

[*]Work is funded by and conducted in cooperation with Siemens Coroprate Technology (CT SE 2)

This paper address the problem of creating and maintaining a Domain-Specific Intelligence Framework (DSIF) to help a modeler solve combinatorically challenging modeling problems. The paper provides five key contributions to the integration of DSMLs and model intelligence. 1) We present experiments showing how a domain-specific knowledge base and domain-specific intelligence framework can greatly reduce the complexity of integrating a solver. 2) We provide methods for templatizing a Knowledge Base (KB) and KB-based constraint solvers so that they can be parametized by a metamodel. 3) We present experiments validating that a metamodel-parameterized KB and solvers require less code and provide superior performance to generic KBs and solvers. 4) We illustrate methods for using an Observer [5], templatized by a metamodel, to translate events from the object graph of a DSML tool into DSKB events and DSKB events back to object graph events. 5) Finally, we present a case-study based on a tool for sovling AUTOSAR, an automotive modeling standard, deployment problems.

The rest of the paper is organized as follows: Section 2 discusses challenges of AUTOSAR deployment, which we will use as a motivating example; Section 3 presents KB, solver, and Observer templatization techniques for creating DSIFs; Section 4 presents our results from applying Model Intelligence to modeling AUTOSAR deployments; and Section 5 presents concluding remarks.

2. MOTIVATING EXAMPLE

AUTOSAR is a new standard for automotive software development modeling. The goal of the standardisation is to solve the set of typical problems inherent when developing large scale, distributed real-time systems for the automotive domain.

In particular, large efforts are required to relocate functions between Electronic Control Units (ECUs), i.e. computers and micro-controllers running software components within a car. The main reason for the difficulties in this case are: a) each component typically has a large set of constraints that need to be met by the target ECU; b) there are large numbers of possible deployments of components to ECUs in an automobile.

For example, finding a set of interconnected nodes able to run a group of components that communicate via a bus is difficult 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 very important in automotive domain to reduce the overall price of the solution, which necessitates optimizations, such as finding deployments that use as few ECUs as possible or that minimize bandwidth requirements. It is hard, if not impossible, to answer these questions manually for a real system model.

In order to illustrate the practical benefits of integrating a DSIF with a DSML, we present a case-study based on a tool we have developed to solve AUTOSAR-like constraints for the valid deployment of software components to ECUs.

There are two views on the AUTOSAR systems:

- **Logical collaboration structure** specifies components or functions that should communicate with each other and through what interfaces.

- **Physical deployment structure** captures the capabilities of each ECU, their interconnecting buses, and their available resources.

The mapping from components in the logical to physical structure (the deployment model) is generally specified with a graphical tool. Figure 1 illustrates the idea of mapping from the logical collaboration structure to the physical deployment structure using a graphical tool.

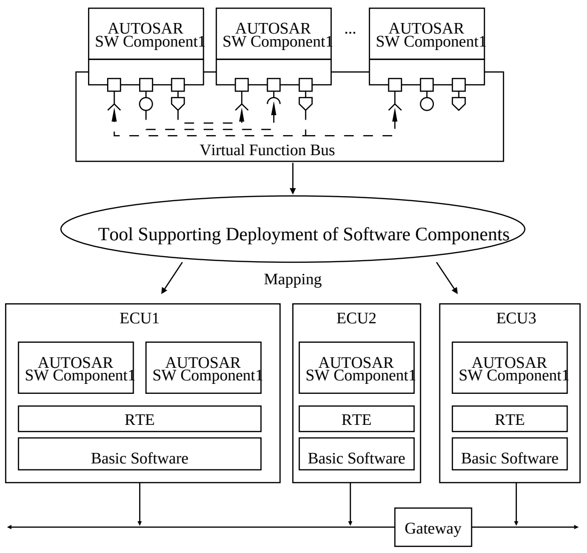

Figure 1: Mapping from the logical collaboration to the physical deployment structure

Modern cars 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, are requirements governing the valid ECUs that can host a component, such as the amount of memory required to run, CPU power, operating system type and version (if any), etc. These constraints have to be carefuly considered while deciding where to deploy a particular component. The problem is further exacerbated when the physical communication pathes and aspects, such as available bandwidth in conjunction with periodical real-time messaging, are considered.

In the research work presented in this paper, we use a tool to specify the mapping of components to nodes in AUTOSAR models as our case study. The goal of our case-study tool was to create *(semi)automated mechanisms to map software components to ECUs without violating the known constraints.* The following sections describe our approach and show how a DSIF can provide significant reductions in complexity for our AUTOSAR deployment examples.

3. CREATING DOMAIN-SPECIFIC INTELLIGENCE FRAMEWORKS

Based on the motivation in the previous section, our main research goals were to: a) define the infrastructure for providing a DSIF for a DSML; b) provide a mechanism to automate the creation and integration of a DSIF and c) provide mechansims to automatically complete a partially specified model using information extracted from the domain constraints.

In previous work [11, 13, 12], we have 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. 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 convenient infrastructure (such as built-in support for Visitor pattern implementation) to simplify model traversal. We used GEMS as the basis for our AUTOSAR deployment modeling tool and our work on DSIFs.

3.1 Describing Domain Constraints

One of the key research challenges was determining how to specify the set of model constraints in such a way that they could be used *not only to check the model for correctness but also to guide the user through a series of model modifications to bring it to a valid state*. We considered using Java, the Object Constraint Language (OCL) and Prolog for our constraint specification language. Initially, we implemented our AUTOSAR deployment constraints in each of the three languages to evaluate their pros and cons. The following list summarize our observations:

- **Java** can be used to define constraints by writing code fragments which access the in-memory object graph of the model and ensure that constraint conditions are met. GEMS, can dynamically compile arbitrary Java constraints that are invoked by modeling events, such as the user creating a deployment connection from a component to a node. Java constraint specification is convenient for exposing simple constraints, but its utility as a constraint language decreases as models become more complicated and consequently the Java constraints require more object graph traversal code. Java also has the disadvantage that the constraint writer must have intimate knowledge of the internal data structures used by GEMS to maintain the model. In many domains, modelers should not be expected to have any Java experience. Our key criterion for not choosing Java-based constraints, however, was that there is no method for deriving the intput to an arbitrary Java constraint that will cause it to evaluate to true. Java constraints can be used to check for correctness but cannot be used to infer possible valid additions to the model and hence provide modeling suggestions.
- **OCL** is very compact, in terms of the amount of code that must be written by a user, and does not require the constraint writer to have detailed knowledge of the internal object graph. OCL, however, has the same core disadvantage of Java. Given an arbitrary OCL expression, there is no easy way to deduce a sequence of model changes that will satisfy the constraint. It is worth noting, however, that the OCL language itself does not prevent finding the solutions to an expression but the currently available tools do not provide this capability. Theoretically, OCL could be implemented on top of a Prolog engine and provide this capability.
- **Prolog** is a declarative programming language where the programmer can 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 satisifed 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 what the valid fact combinations are that will satisfy it. If the user completely specifies the input variables, Prolog merely checks whether the rule holds for the provided input. If, however, some of the input variables are not specified, *Prolog has the unique ability to return the set of possible facts from the KB that lead the rule to evaluate to "true"*.

As a result of the evaluation we conducted, we came to conclusion that Prolog is the most appropriate language for providing both constraint checking and model suggestions. The declarative nature of the language reduces the number of lines of code needed to be written to both transform an instance of a DSML into a knowledge base and create constraints (its roughly comparable to OCL for writing constraints). Most importantly, Prolog provides the ability for the modeling tool to derive sequences of modeling actions that will take the model from an incomplete or invalid state to a valid one. This capability, as we will discuss later, is crucial for domains, such as deployment, where completely manual model specification is infeasible or extremely time consuming.

The following section details our approach to creating DSIFs using Prolog and GEMS.

3.2 Prolog-based approach

To provide Model Intelligence, the modeling tool must be able to 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 AUTOSAR deployment example, the DSIF must provide a mechanism for, given a set of components, their requirements, nodes, and their resources, suggest a valid assignment of components to nodes. Our work provides this reasoning capability to GEMS by automatically generating a Prolog representation of each model, allowing the user 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 entites and the role-based relationships between them. For each model, our DSIF parameterizes 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, a component in our AUTOSAR 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 with 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 KB provides a domain-specific set of predicate statements.

The domain-specific interface to the KB provides several advantages over a generic format, such as the format that would be used by a general purpose constraint solver. 1. The KB maintains the domain-specific notations from the DSML making the format more intuitive and readable to domain experts. 2. Maintaining the domain-specific notations allows constraints to be specified using domain notations, improving a developer's ability to understand how requirements map to constraints. 3. In experiments that we conducted, writing constraints using the domain-specific predicates produced rules that had fewer levels of indirection and greatly outperformed rules written using a generic format. As would be expected, the size of the performance advantage was dependent on the generality of the KB format. For accessing properties of the model entities, the predicate syntax presents the most specific KB 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.

There are two types of Prolog rules needed to enable Model Intelligence:

- **User-defined constraints** - they are a semantical enrichment of the model that specify the requirements of a correct model. These constraints are also used to automatically deduce the sets of valid model changes to create a correct model. As an example, consider the following constraint to check if a node is a valid host of a component:

  ```
  is_a_valid_component_targethost(Component,
                                  Node).
  ```

  It can be used to both check a Component-Node combination:

  ```
  is_a_valid_component_targethost(23,25).
  ```

  and to find valid Nodes that can play the TargetHost role for a particular components:

  ```
  is_a_valid_component_targethost(23, Nodes).
  ```

  In this example, the Nodes variable will be assigned the list of all valid nodes for the TargetHost role of the specified component. This example illustrates how the constraint can be used to check as well as to generate the solution. It is also worth noting that there are a lot of existing algorithms and libraries developed by the artificial intelligence community and other Prolog enthusiasts which can be used while performing complex model analysis. In other work, we have used the new DSIF capabilities of GEMS to integrate an existing Prolog Qualitative Differential Equation simulator into one of our DSMLs. The integration was very straightforward and required less than 100 lines of Prolog code.

- **Generated reusable rules** - during the development of our DSIF infrastructure and while writing different constraints we found that certain types of problems occur frequently. For example, a common problem is given a model entity and a role that can be assigned on the entity, to find the set of valid values for that role. This directly corresponds to our AUTOSAR deployment example where we need to find the valid nodes for each component's TargetHost role. It is possible to formulate the solutions to these common problems in a template form so that they can be parameterized by the metamodel and included in the generated KB. Many of the common constraint solutions are based on inference and thus our DSIF is modularized according to the architecture proposed in [6]. The DSIF separates the inference based solvers and the domain-specific solvers, such as a bin-packing solver. As a result, we have developed a set of reusable inference-based rules which can be used by a constraint writer to reduce manual coding. As discussed in [7], constraint frameworks that integrate multiple solution methods provide the best approach. The metamodel-parameterized rules generated into the DSIF can also be used to automatically provide visual feedback to a modeler. We have integrated a tool infrastructure into GEMS that listens for user-initiated connections, finds all valid connection 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-generated DSMLs. Modelers only need to specify the constraints and the generated DSIF framework can automatically find valid solutions for some types of problems. The feedback mechanism and solver are also dynamic, users can add constraints at modeling time and immediately begin receiving guidance from the DSIF.

The DSIFs generated by GEMS leverage SWI Prolog and use the Java Prolog Library (JPL) 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. One complexity resulted by the use of a domain-specific knowledge base was that translating events from the object graph of the model into KB events was much more difficult. The complexity is that the knowledge base format does not correspond directly to the object graph. To overcome this problem, a Observer is used that is parameterized by the metamodel at runtime so that object graph events can dynamically be translated into type and relationship predicate statements. For each object graph event, the Observer looks up the meta type of the event source, deduces the modified role, translates the event into a predicate, and asserts the predicate into the DSKB.

The DSIF also provides mechansims to trigger arbitrary Prolog rules from the modeling tool and incorporate their results into the model object graph. This mechanism is useful for providing global rules for solving complex problems involving multiple model entities and roles. For example, for the AUTOSAR deployment tool, a rule was developed to check not only configuration constraints but also resource constraints for all components and derive a valid TargetHost for every component. This feedback interface allows Prolog rules to return further assertions or retractions to the knowledge base which are then translated into object graph events. The assertions and retractions use the native DSKB format. Again, the metamodel-paramaterized Observer is used to decompose the predicate statements into changes involving the role of an object in the object graph. The decomposition is performed by using the predicate as the role name, the id to lookup the correct model object, and the metamodel to find the correct code entry point to change the role value.

While developing these feedback mechanisms, we also found that it could be useful to base the model on the state of an application running outside of GEMS for monitoring purposes. To enable this type of monitoring-based DSML, we implemented a CORBA interface within gems which can receive updates to the DSKB using a predicate/arguments format. As a result, with a small amount of effort, GEMS can be turned into a very powerful monitoring and control system. For example, it is straightforward to convert GEMS into an Eclipse Rich Client Platform (RCP) application and run it as a standalone, only showing the model and corresponding set of rules. This can be used, for example, to show failed Nodes in our AUTOSAR deployment example, recalculate a new deployment strategy, and trigger the redeployment of components. In this case, the information about failed nodes is received from a remote observer using the provided CORBA interface. The following CORBA IDL fragment shows the core part of the CORBA interface we are providing (exceptions, and some other definitions are omitted).

```
enum Operation {Insert, Update, Delete};
struct EntityRecord {
  Operation op;
  string predicate;
  StringSeq params;
};
typedef sequence<EntityRecord> EntityRecordSeq;
interface Model {
  void applyChanges(in EntityRecordSeq entities);
  void getEntities(in long offset, in long count,
                   out EntityRecordSeq entities);
};
```

4. CASE STUDY: DEPLOYMENT MODELING FOR AUTOSAR

To validate our DSIF approach to Model Intelligence, we created a DSML for modeling AUTOSAR deployment problems. Our goal was to create a modeling tool that enabled the developer to specify partial solutions, as sets of components, requirements, nodes, and resources. A further requirement was that the tool could produce both valid assignments for a single component's TargetHost role and global assignments for the TargetHost 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 TargetHost role of certain components and allow the tool to solve for valid assignments of the remaining unassigned component TargetHost roles. Thus, we also required that our tool be able to complete a partially specified deployment of components to nodes, if a valid deployment exists.

For the first step, we created a deployment metamodel which defines a DSML that allows the user to model components with arbitrary configuration and resource requirements and nodes 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 was not allowed to have more components deployed to it than its resources could support. Typical resource requirements were the RAM usage and CPU usage. Each host in turn could provide an arbitrary number of resources. Constraints comparisons on resources were specified using $<$, $>$, - and $=$ signs to denote that the value of the resource with the same name and type (for example OS version) must 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 node, does not exceed the amount present on the node. Figure 2 shows a screen-shot of the our deployment DSML.

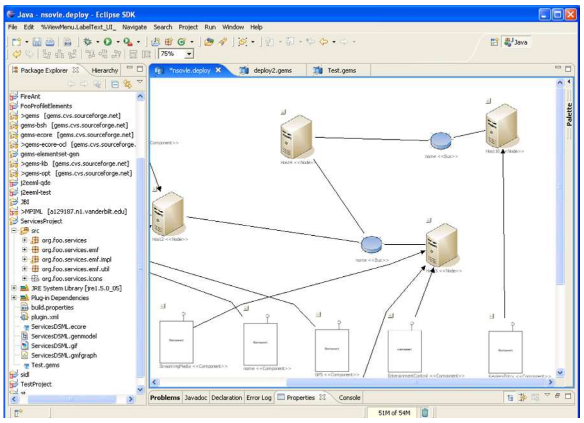

Figure 2: Mapping from the logical collaboration to the physical deployment structure

After defining the metamodel and generating the deployment DSML using GEMS, we added the a set of Prolog constraints 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 TargetHost must satisfy the requirement. Our complete configuration constraint rules are listed below:

```
comparevalue(V1,V2,'>') :- V1 > V2.
comparevalue(V1,V2,'<') :- V1 < V2.
comparevalue(V1,V2,'=') :- V1 == V2.

%Resource constraints were checked in further
%rules. This rule helped to reduce the possible
%nodes that could be check for a deployment
%configuration, since a node cannot meet a
%resource constraint if it does not have at least
%the sepecified amount of the resource.
comparevalue(V1,V2,'-') :- V1 >= V2.


matchesResource(Req,Resources) :-
    member(Res,Resources),
    self_name(Req,RName),
    self_name(Res,RName),
    self_type(Req,Type),
    self_value(Req,Rqv),
    self_value(Res,Rsv),
    comparevalue(Rsv,Rqv,Type).

is_a_valid_component_targethost(Comp,Host) :-
    self_requires(Comp,Requirements),
    self_provides(Host,Resources),
     forall( member(Req,Requirements),
             matchesResource(Req,Resources)
     ....
```

It is worth noting that these 16 lines of code are the *ENTIRE* solution for providing not only configuration constraint checking for an arbitrary set of requirements and resources but also to enable the DSIF to provide valid suggestions for deploying a component. In our experiments, Prolog could solve for a valid global deployment of 900 components to 300 nodes, that observed configuration constraints, in .08 seconds.

The rules required for solving for valid assignments using resource constraints were, as expected, significantly more complicated since resource constraints are a form of bin-packing (an NP-Hard problem). Still, however, we were able to devise heuristic-based rules in Prolog that could solve a 160 component / 80 node model deployment in 1.5 seconds. Our resource rules were not meant to provide the most optimal solution algorithm but mere to show the feasability of using Prolog for the domain.

5. RELATED WORK

Decision support systems are similar to the 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 KBs for diagnosing and treating diseases. Both their approach and the Model Intelligence 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 proposed, however, differs significantly from the research proposed in this paper. First, the research proposed here is designed to facilitate the creation of decision support systems for any domain-specific modeling language. Furthermore, the DSIF is not limited solely to decision tree type guidance but also complex analysis and optimizations specified by a user. Finally, the DSIF proposed in this paper is automatically generated from a metamodel and integrated with a graphical modeling tool. GEMS and its DSIF 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 [8, 3, 9, 2, 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. Finally, as discussed previously, these tools do not provide automation of the problem specification as our GEMS-based DSIFs do.

6. CONCLUSIONS

The work presented in this paper addresses the scalability problems of completely manual modeling approaches. These scalability issures are particularly problematic for domains that have large solutions spaces and few correct solutions. In such domains, it is impossible or extremely time consuming to create correct models manually and therefore constraint solvers are needed. Constructing an phrasing a problem instace from a DSML instance is a time-consuming task.

Our solution uses the idea of generating a DSIF 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 advantage of this approach is that the same set of constraints can not only be used check whether a manually defined model is correct but also to ask for valid solutions by keeping some parameters, like TargetHost, open. The approach also allows the modeler to both specify constraints using the domain-specific notation of the knowledge base and to specify solution algorithms, for constraint types not covered by the reusable rules, in a domain specific manner. Another key capability of the DSIF approach is that it automates the problem specification for the underlying constraint solver. Finally, our results have also shown that it is possible to templatize many common solvers so that they can be parameterized by a metamodel and made domain-specific.

From our work, we have learned several important lessons:

- Many typical types of constraints that use $<$, $>$, and $=$ comparisons between two values can easily be solved by the rules generated in the DSIF. For these types of comparisons, the user merely needs to specify the constraint and the GEMS-generated DSIF can automatically solve them and provide valid modeling feedback.

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

- Many combinatorically complex problems have been previously solved using Prolog and expert system approaches. The DSIFs generated by GEMS can often easily incorporate these existing algorithms with a minimal amount of code.

- Optimization of a model is a much more challenging task than finding a valid solution. For our deployment example, we were able to create algorithms that minimze the number of nodes used by a deployment. These algorithms, however, do not scale well if there are a large number of valid solutions. Much more work will be needed to provide templatizable optimization rules.

In future work, we plan to apply DSIF to multiple combinatorically challenging domains, such as service configuration, failure analysis, and workflow composition. We are also collaborating with optimization researchers to develop more reusable solution algorithms that can be generated into the DSIF. GEMS and its DSIF 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] 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.

[3] J. Cohen. Constraint logic programming languages. *Commun. ACM*, 33(7):52–68, 1990.

[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] J.-K. Hao and J.-J. Chabrier. A modular architecture for constraint logic programming. In *CSC '91: Proceedings of the 19th annual conference on Computer Science*, pages 203–210, New York, NY, USA, 1991. ACM Press.

[7] P. V. Hentenryck. *The OPL optimization programming language.* MIT Press, Cambridge, MA, USA, 1999.

[8] 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.

[9] 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.

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

[11] J. White, D. Schmidt, and A. Gokhale. The j3 process for building autonomic enterprise java bean systems. *icac*, 00:363–364, 2005.

[12] 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.

[13] 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.