Application of Aspect-based Modeling and Weaving for Complexity Reduction in Development of Automotive Distributed Real-time Embedded System

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

To meet the stringent resource and costs constraints in developing modern automotive embedded electronic systems requires careful consideration of various aspects, such as the target hardware structure, component collaboration model, and timing models. An emerging trend in automotive systems is to apply Model-Driven Development (MDD) to understand and formalize these aspects. The growing size and complexity of automotive systems, however, can yield models that are hard to develop and evolve manually without violating domain constraints, such as resource allocation limits.

This paper presents our experiences applying aspect-oriented design and modeling to develop a component-based distributed real-time embedded (DRE) automotive system. We summarize our findings and show the key technological shortcomings with conventional weaving approaches that make it hard to leverage the full power of AOSD to design and model large-scale DRE systems. We also evaluate the effectiveness of various aspect merging techniques to help overcome these shortcomings.

1. INTRODUCTION

It is hard to develop traditional real-time and embedded systems due to extensive constraints, such as (*e.g.*, limits on the available memory, CPU time, and APIs) and tight coupling between the hardware (*e.g.*, sensors and actuators) and the corresponding software artifacts (*e.g.*, such as components and libraries) [17, 9]. It is even harder to develop larger-scale distributed real-time embedded (DRE) systems, such as vehtronics and infotainment systems in modern automobiles [23]. These systems incur many of the same constraints as traditional real-time and embedded systems, but also must address new scalability and networking challenges. For example, the current generation of high-end cars (such as the BMW 7 Series, Mercedes S-class, and Audi D3) can have ∼80 interconnected electronic control units (ECUs) (which are the automotive equivalent of a CPU) and ∼300 software components to deploy to the ECUs.

Due to physical restrictions, such as form-factor, power consumption, and heat emission, automotive DRE system designs often use hardware that is highly specialized for the functions it performs. To satisfy the extensive constraints on software artifacts that this type of specialized hardware creates, developers have historically written highly proprietary, hardware-specific code. This code made assumptions about many system configuration aspects (*e.g.*, the deployment structure, available communication paths, and component collaboration patterns) in early phases of system development. As a result, it was extremely tedious, error-prone, and costly to maintain existing automotive designs and evolve these designs to meet the feature requirements of newer car models.

To help alleviate the high cost and effort associated with developing custom automative DRE system hardware and software, designers are moving towards commercial-off-the-shelf (COTS) components [20, 15], which can provide a significantly less expensive and higher quality product, while reducing time-to-market. As a result of advances in hardware and software development, customers now expect automotive DRE systems to have significantly more functionality, such as parking assistance, collision avoidance, and seamless integration of internet-enabled devices. Although COTS components have decreased the software development costs of automotive applications, the integration complexity has risen significantly and is now the leading cause of automotive system failures [4, 8].

To deal with this broad spectrum of complexity, developers of automotive DRE system are increasingly using *model-driven development* (MDD) technologies [12, 14], such as Matlab [24], to capture many aspects of the system being constructed, including the target hardware structure,

AOSD '07 Vancouver, Canada

component collaboration model, and timing models. These models can then be used to automatically check, validate, and generate many implementation artifacts, such as state machine implementations, configuration files, and highly optimized middleware layers. MDD tools for automotive DRE systems allow modelers to manipulate models using various aspects, *e.g.*, users can build models and switch between aspects for hardware configuration, software collaboration, and software to hardware mapping. The software-to-hardware mapping (*i.e.* the deployment model) then weaves the hardware and software components together to create a complete application.

Despite the benefits of applying MDD approaches to automotive DRE system development [16, 5, 22], certain limitations have become more pronounced as the size of models increase, which is significant since the next-generation of automobiles will have thousands of components and computing nodes. A particularly vexing problem is that conventional MDD tools require users to manually specify relationships between model entities to specify how weaving is done.

For example, to show where to deploy each component, a line can be drawn from the component to the ECU that hosts it. In models with thousands of entities, it is tedious and error-prone to manually draw lines from components to ECUs to specify deployment relationships. Moreover, each component can have multiple constraints [23] (such as the required bandwidth to exchange messages, and RAM and CPU requirements) that restrict which ECUs can host it. For production models, therefore, the number of modeling entities and the complexity of the constraints makes it infeasible to manualy specify the mapping to weave aspects.

In prior work [26, 27], we demonstrated a tool based on the Generic Eclipse Modeling System (GEMS), called AUTODeploy, that integrates a declarative constraint language with a domain-specific modeling language for automotive software development. We found that adding a declarative language to specify system constraints can simplify the modeling of complex automotive DRE systems. This paper explores a previously unexplored dimension of our prior work by (1) describing our experience applying AUTODeploy to merge the component collaboration aspect with the hardware aspect in an automotive DRE system and (2) providing the following contributions on tackling the complexity of applying AOSD to large DRE systems:

- We describe the initial challenges faced when developing an automotive modeling application and how we addressed these challenges by using aspects to model hardware configuration concerns (such as the available computation nodes and buses) separately from the logical component collaboration concerns (such as component types, interfaces, and interactions).
- We describe the significant challenges faced when manually weaving these automotive aspects together to produce a deployment model that met the stringent component configuration and resource constraints required in automotive systems. We motivate why weaving must take domain-constraints into consideration and must be automated to handle production automotive applications.
- We summarize our experience leveraging constraint solvers to produce the mapping for automotive aspects and provide constraint- and semantic-aware weaving.

Throughout the paper we summarize our experiences decomposing and composing automotive systems based on concerns and present the lessons learned while working on the project.

The rest of the paper elaborates on these topics as follows: Section 2 presents an overview of the structure, functionality, and scope of the automotive project that provides the context for our work; Section 3 describes the problems we faced when applying a multi-aspect development approach to our automotive project; Section 4 examines the techniques we used to address these challenges, which involved building a constraint-aware aspect weaver based on semantic pointcuts; Section 5 compares our approach with related work; and Section 6 presents concluding remarks and summarizes lessons learned.

2. AUTOMOTIVE SYSTEM OVERVIEW

This section presents an overview of the structure, functionality, and scope of the automotive project that provides the context for our experience reported in this paper. The goal of this project was to provide a modeling environment to support the specification of complex automotive electronic systems. Users of the modeling environment need to specify the component types in their application, the interaction of the components, the hardware resources available for running the components, and the mapping from component instances to hardware units. From these various system aspects, the modeling environment generates software artifacts, such as XML component deployment descriptors, XML configuration descriptors, component interfaces, and component skeletons. Figure 1 represents a view of an example system.

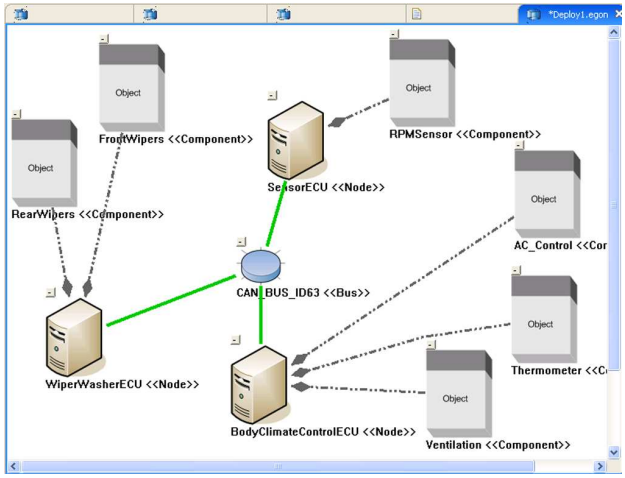

Figure 1: Example of Interconnected ECUs and Components

Figure 1 shows the hardware configuration for a system with

three ECUs interconnected through a Controller Area Network (CAN) bus. The *BodyClimateControl* ECU hosts components responsible for measuring internal temperature and controlling ventilation and the air conditioner. The *Wiper-Washer* ECU runs different software components required to control the forward and rear wipers and washers. The third ECU hosts the set of components that receive raw sensor information (such as speed) and convert it for viewing on driver displays (such as the dash speedometer and tachometer).

Figure 1 also shows the set of components hosted on each ECU. Since this diagram depicts the *Deployment* aspect, there are no explicit connections shown between collaborating components, though most components communicate with each other. The interaction between the components is defined via views from other aspects. For example, the rain sensor provides measurements that are processed by the wiper/washer controllers to decide whether to turn the corresponding actuators on or off. Likewise, the body temperature sensor provides information to the instrument pannel. Each aspect allows modelers to focus on a specific system concern, such as deployment, software collaboration, or hardware infrastructure.

Each software component within the system has a set of requirements that must be met when deciding which hardware component can host it. In addition to simple CPU and RAM requirements, components may require certain actuators and sensors be connected to a hosting ECU, or that the communication channel between the hosting ECU and the hosting ECU of another component provided a minimum guaranteed bandwidth. For example, the Anti-lock Braking System (ABS) will mix multiple constraint types in the requirements specification for its hosting ECU. The ABS component will have fault-tolerance constraints that restrict its host's distance from the perimiter of the car, connectivity requirements to ensure that it can activate the brake actuators, resource requirements to ensure it has sufficient processing power to calcuate how the brakes should be applied, and infrastructure requirements specifying the types of APIs or middleware on its host.

We designed AUTODeploy to focus on a specific set of these requirement types that are particularly challenging to address during the development of an automotive application. The main automotive concerns AUTODeploy focused on were:

- **Physical hardware configuration concerns**, including the available ECUs, communication busses and connection topology, connected devices, such as actuators and controllers.
- **Logical component collaboration concerns**, including specifying the interfaces exposed by software components and the dependencies between components.
- **Component requirement concerns**, including specifying the constraints that must be met by a valid deployment of components to ECUs.
- **Hardware resource concerns**, which capture the capabilities of each ECU, their interconnecting buses, and their available resources.

The initial AUTODeploy prototype encompassed 28,292 lines of Java code and 2,751 lines of Prolog code for solver implementations. Three developers produced the prototype over the course of three months. The work was part of a larger effort to build tools to support the modeling of large-scale automotive systems.

The remainder of this paper concentrates on the challenges we faced while weaving these concerns of the automotive DRE system to produce the correct and consistent software-to-hardware mapping. In particular, a key task for automotive system developers using AUTODeploy is to specify on which ECU each component should run. This mapping from components in the logical to physical structure (the deployment model) can be specified graphically in AUTODeploy. Figure 2 shows a mapping from the logical collaboration structure to the physical deployment structure using AUTODeploy.

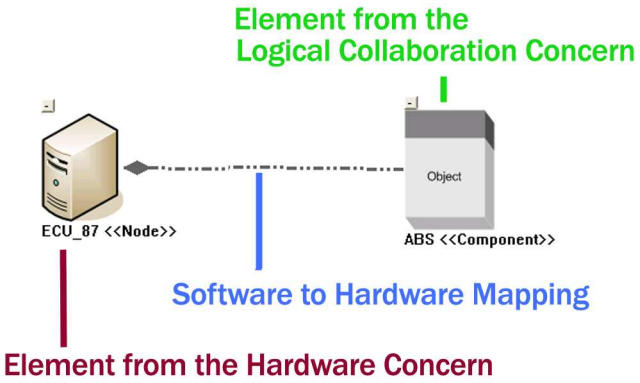

Figure 2: Mapping Logical Component Collaboration to Physical Deployment Structure

A key goal for our automotive project was to use the multiple aspects supported by AUTODeploy to reduce the complexity of developing software compoents, particularly for determining how software components should be deployed to hardware units in an automotive application. Another goal was to ensure that our multi-aspect modeling approach could support next-generation production automotive system models, which will be significantly larger and more complex.

3. CHALLENGES OF APPLYING AOSD TO AUTOMOTIVE MODELING

This section provides a detailed description of the problems we faced when applying a multi-aspect development approach to our DRE automotive system. To meet the requirements and functionality outlined in Section 2, we combined an MDD approach with the concept of *concern separation*. In particular, we applied the concept of HyperSlices [21, 19], which are a geometric analogue to hyperplanes in a multi-dimensional space. Each concern of an application is dealt with as a separate slice cutting across all designated dimensions of the system (*e.g.*, different dimensions of class

entities, whereby each hyperslice might only contain semantically cohesive methods of the representing concern).

We used AUTODeploy to model the hardware configuration (such as available ECUs) and available communication channels (such as CAN-buses) as one concern. We also used AUTODeploy to model the set of software components that implemented the application logic—and the logical collaborations between them—as another concern. After modeling these two aspects separately, weaving was performed to merge the concerns. The weaving process resulted in a new model—composed of the two concerns—that showed how to deploy the software components on the available hardware.

A key complexity with our AOSD approach was that the weaving must respect the various requirements of the components. For example, each component was annotated with information about its memory usage and required CPU power measured during the testing process. There were many domain-specific semantic constraints, such as any ECU that runs a safety critical component (*e.g.*, an Anti-lock Braking System (ABS) or airbag control) must be located at maximum distance from the car perimeter to reduce the probability of failure in an accident. The weaving process needed to produce a model where components were allocated to nodes without violating their resource, configuration, and fault-tolerance constraints. We quickly discovered, however, that manually specifying how to merge the models scaled poorly.

To illustrate the scalability problems, consider a group of components that must be deployed to ECUs within a car. Part of the complexity of this domain is how quickly the solution spaces grow as the number of model elements increases. Figure 3 visualizes the speed at which the solution spaces grows for our automotive example.

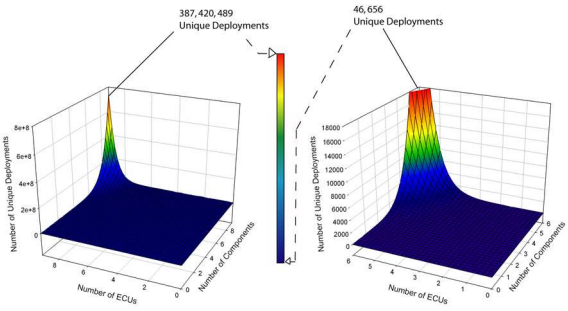

Figure 3: Measuring Modeling Complexity

With 9 components and 9 ECUs we have a total of 387,420,489 unique deployments. It appears from the left-hand graph in Figure 3 that the solution space size is relatively flat when there are less than 6 components and 6 nodes. The right-hand graph in Figure 3, however, shows that the solution space is actually not flat at all from 0-6 components/nodes, but only appears flat when scaled in comparison to 9 components/nodes. Clearly, any approach to finding a deployment that observes the deployment constraints must be efficient and employ a form of pruning to reduce the time taken to search the solution space. Often, pruning strategies can leverage domain-speicific information to create a set of heuristics for searching the solution space.

A manual approach may work for a model with 5 or so elements. As shown in Figure 3, however, the solution space can increase rapidly as the number of elements grows. Typically, each component in an automobile will have multiple constraints governing its placement. Not only must the ABS be hosted by a controller at least a certain distance from the perimeter of the car, it will also have requirements governing the CPU, memory, and bus bandwidth available on its host. When these constraints are considered for all the components, it becomes infeasible for modelers to handcraft a model that merges these concerns.

After developing some prototypes, we discovered that although we could apply AOSD techniques to separate concerns, the weaving of various concerns into a deployment model had to account for the semantic relations between model entities. These relationships are specified in form of domain-specific constraints. For our automotive project, we needed a weaver that could consider all these constraints, deduce the proper merging strategy based on the component requirements, and then update the internal model representation to establish new relationships between hardware and software elements.

This merging process introduced a conceptual and technical challenge because current approaches [1, 2] to specify concern composition rules rely primarily on syntactic matching and do not adequately capture semantic constraints and rules. For example, it is not possible with wildcard, method signature, or type-based pointcuts to specify that a weaving should ensure each ECU's memory capacity is not exceeded by the components allocated to it in the resulting model. To compose different concerns in an automotive system, we identified numerous semantic constraints that the weaver needed to leverage to deduce the proper composition strategy. These constraints underscore that manual weaving does not work effectively for mapping software components to ECUs.

The domain analysis briefly presented above led us to create a set of desirable properties and requirements for an effective and scalable automotive modeling solution. The following list represents the most important requirements and challenges we identified:

1. **Support for capturing complex semantically-enriched concern composition rules exposed as domain constraints**. For example, to deduce the proper composition, the concern weaver must combine and leverage certain statements, such as "the value of the *OSVersion* property should be greater then or equal to *N*" or "the cumulative RAM usage of all components assigned to an ECU should not exceed the available amount of RAM."

2. **Support a combination of automatic and manual weaving**. It is hard to formalize certain composition decisions, such as those guided by historical or political constraints. For example, a particular ECU may come pre-loaded with a software component for controlling the windshield wipers. Although it may

be possible to relocate the windshield wiper software component to another ECU, it requires an extra software configuration step during manufacturing and is thus undesirable. In these situations, it must be possible for modelers to partially specify the composition manually and then complete weaving automatically.

3. **Support model repairs**. A large set of semantic composition rules may yield situations where no valid composition can be found and where modelers may not understand how to fix the problem. In cases where rules conflict, or other forces prevent successful merging, reasonable feedback must be provided to modelers so they can repair the model and enable weaving to succeed. It is therefore critical to have the ability to automatically reason about the problem and provide suggestions on how to reconcile the conflicting forces in each concern. For example, if there is a component that requires a certain amount of RAM, but there is no ECU with a sufficient amount available, the tool could suggest increasing the amount of RAM on an ECU or reduce RAM consumption on the component side.

We built the AUTODeploy tool on top of a generic modeling framework (*i.e.*, not specific to deployment) that allows modelers to define model aspects, specify concern composition rules using a declarative composition and constraint definition language, and then automatically generate a constraint compliant weaver using a set of constraint solvers. The rule definitions in this constraint language are sufficiently general to specify conventional merging rules based on known techniques, such as pattern matching for model entity names. In addition, it is possible to specify advanced semantic rules, such as do not allow the sum of the memory demands of the components hosted by a node to exceed the RAM available on the node or do not allow this component to be deployed to a node that is less than a quarter meter from the perimeter of the car. Definitions are based on domain constraints defined by application domain experts, such as the minimum distance from the car perimeter, operating system of the hosting ECU, or required memory allocation.

4. CONSTRAINT-AWARE CONCERN WEAVING

This section describes techniques we used to address the challenges outlined in Section 3. We chose the open-source Generic Eclipse Modeling System (GEMS) [26, 27], which is a part of the Eclipse Generative Modeling Technologies (GMT) project, as the basis for our AUTODeploy tool. GEMS provides a convenient way to define the metamodel, *i.e.*, the visual syntax of the modeling language. Based on the metamodel, GEMS automatically generates a graphical editor that enforces the grammar specified in the metamodel.

GEMS enables the definition of aspects in its metamodel that can be used to specify concern boundaries and views in a modeling language. We used these aspects extensively when modeling the various automotive system concerns described in Section 3. To facilitate code generation, GEMS also provides an infrastructure for model traversal and event listening that can be used in conjunction with other MDD tools, such as ATL [10] or open Architecture Ware (oAW) [3].

Although GEMS supports the definition of various modeling aspects, it originally did not support the extensive constraint-aware weaving required for our automotive DRE system project. During the development of AUTODeploy, therefore, we extended GEMS with a generic mechanism to automatically weave metamodel-defined aspects together based on domain constraints. A key implementation and conceptual challenge we faced, however, was how to specify the domain constraints so that they could be used to not only check the correctness of the model, but also be leveraged by our weaver to *automatically merge concerns together without violating constraints*. We considered various approaches for constraint specification, including Java, the OMG Object Constraint Language (OCL), and Prolog. To evaluate the pros and cons of each approach, we implemented our composition constraints in each of these three languages.

After an extensive evaluation [18, 25], we selected Prolog since it provided both constraint checking on already woven models and the ability to deduce composition strategies (constraint solving), based on domain constraints, for unwoven models. In particular, Prolog can return the combination of known facts from a knowledge base that lead a conjunction of constraints to evaluate to "true," *i.e.*, a Prolog constraint can be invoked both as a constraint check and as a constaint solver.

In contrast to Prolog, Java and OCL have no equivalent to forward and backward chaining (constraint checking and solving). OCL is easier to write rules with than Java, but we could not find any constraint solvers that allowed us to specify constraints as OCL, so we could only check to see if a woven model was valid. With Java, we could leverage a constraint solver and specify constraints, but the semantic gap between the input format of the constraint solver and the modeling tool was large and required significant work to overcome.

For example, to solve a configuration constraint using a Java constraint solver, the constraint must be transformed into a system of linear equations. We found this transformation prohibitively expensive and inflexible for our system. Moreover, the specialized knowledge required to perform this transformation prevented domain experts, *i.e.*, automotive engineers, from specifying constraints.

Our model-driven approach to constraint-aware weaving was the key that allowed us to raise the level of abstraction and allow domain experts to apply Prolog constraint solvers successfully. Without the domain-specific predicates and the ability to display constraint solving results graphically in the modeling tool, we found that Prolog was too hard for domain experts to use effectively since they were forced to learn predicate logic without a concrete sense of how to build predicates and map them to their application domain.

With the appropriate domain-specific predicates and graphical feedback provided by GEMS, however, we found that domain experts could quickly grasp how the predicates applied

to their model and easily write complex domain constraints. Moreover, when users manually specified predicates with bare Prolog, they were also forced to manually specify how to map model instances to these predicates. Manually specifying this mapping required significant work and required developers to update the mapping each time the metamodel changed.

We often found the need to add constraints to specific model instances. For example, a component implementation by a vendor could have superior performance characteristics when paired with hardware by the same vendor. Encoding constraints for a specific model instance in a format for a Java constraint solver was prohibitively expensive. We found that Prolog combined the declarative expressivity of OCL (which allowed domain experts to specify constraints) with the power of a Java constraint solver (which allowed the modeling tool to derive solutions to the domain constraints). In fact, the standard Prolog distributions we tried had implementations of constraint solvers with features comparable or superior to the Java constraint solver we evaluated.

Our experiments found no significant performance difference between the Java solvers and the Prolog solvers that justified the increased development cost of writing Java constraints. Moreover, we found that having the right type of solver was more important than the implementation language. Since many constraints, particularly resource constraints, were highly combinatorial, constant differences in performance coefficients between the languages was not nearly as significant as large algorithmic improvements that could be obtained by using the right solver for the problem. Prolog provided significantly more solver implementations than Java.

Our AUTODeploy solution allowed domain experts to specify the rules governing concern composition as a series of declarative rules using Prolog. When two aspects were woven together, the constraint solver was invoked to find an assignment of the constraints' variables that would satisfy the domain constraints. For example, when mapping components to ECUs, we would invoke the deployment constraint and have Prolog return an assignment of components to ECUs that would satisfy the resource and configuration constraints specified in the model. The weaver would then take this assignment of components to nodes to merge the component collaboration and physical hardware concerns into a deployment model. Figure 4 shows the inovcation of the constraint-aware weaver and the presentation of a merger solution to the user.

The remainder of this section summarizes the lessons learned while implementing the AUTODeploy-based solution presented above. This solution addresses the list of requirements presented in Section 3.

4.1 Capturing Semantically-enriched Concern Composition Rules

As outlined in Section 4, we found domain experts had a hard time providing domain constraints for specifying concern composition rules by programming conventional constraint solvers written in C or Java. We alleviated this problem with AUTODeploy by generating a *Domain Specific Knowledge Base* (DSKB) in Prolog, which respected the domain-specific concepts from the concerns and provided a flexible mechanism for specifying both solvers and constraints using domain notations. AUTODeploy's domain-specific format is created automatically from the AUTODeploy metamodel we specified in GEMS and requires no explicit user specification or generation. As we continue to evolve AUTODeploy, and the metamodel changes, GEMS automatically updates the underlying Prolog DSKB format.

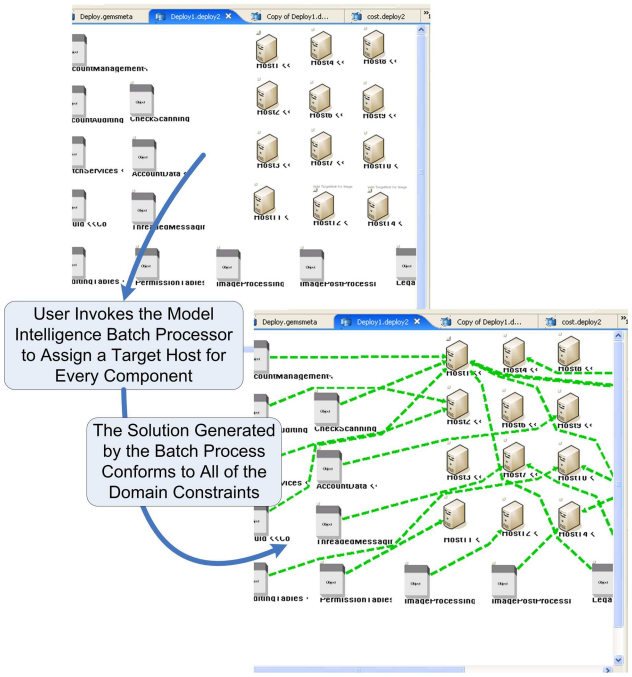

Figure 4: Constraint Solver Guided Weaving

We found that AUTODeploy users can easily leverage the automotive-specific notation to define composition rules and constraints. In particular, for most types of weaving constraints, existing constraint solvers can be used. Users therefore did not require constraint solver experts to specify solution strategies, but instead specified how AUTODeploy and GEMS should map their constraints to the appropriate existing probem types, such as mapping the assignment of components to nodes, while respecting resource constraints, to bin-packing. AUTODeploy and GEMS would then take care of choosing the appropriate solver for the problem.

AUTODeploy views the concern models as a set of model entites and the role-based relationships between them. For example, Node (ECU), Component, and DeploymentPlan are entities. Each entity may participate in multiple role-based relationships. Relationships that cross concern boundaries, such as Deployment (the mapping from an element in the logical collaboration concern to an element in the physical hardware concern), indicate mergers that must take place.

For example, a deployment relationship between a Component instance and an ECU specifies that in the merger of the software and hardware concerns of the system into a deployment model, the ECU should become the target host

of the component. Each relationship that crosses concern boundaries may have Prolog constraints bound to it. Before weaving occurs, the Prolog constraint solvers are used to find the valid endpoints for each of these cross-concern relationships. The valid endpoints are then provided to the weaver to perform the actual concern merging.

To generate a DSKB format for each concern model, GEMS 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 is transformed into the predicate statement

```
self_type(id,component)
```

, 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

```
self_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.

Users create rules, based on the DSKB format, to bind constraints to the various cross-concern relationships. For example, in AUTODeploy, the rule

```
is_a_valid_component_targethost(Component,ECU) :-
    self_requiredOS(Component,RequiredOS),
    self_providedOS(ECU,RequiredOS).
```

could be used to specify that a correct Component to ECU mapping requires that the target ECU provide the correct OS for the Component. This rule can be used by the constraint solver to collect all valid ECUs for each component, solve the various global constraints, such as resource consumption, and produce a valid mapping from Components to ECUs. The mapping from Components to ECUs is then input into the weaver to perform the concern model merging and produce a deployment plan. The deployment plan can then be leveraged by code generators to produce XML deployment descriptors to drive a deployment and configuration framework, such as DAnCE [11].

4.2 Semi-Automatic Concern Weaving

As described in Section 4.1, modelers can use AUTODeploy to specify user-defined constraints, based on the domain-specific knowledge base, in the form of Prolog rules for each kind of cross-concern relationship. As an example, consider the following constraint to check whether a node is a valid host of a component:

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

This constraint can be used to check a Component-Node or concern merging combination, *i.e.*:

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

It can also be used to find valid Nodes that can play the TargetHost role (a join point) for a particular component solution, *i.e.*:

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

In this example, the Nodes variable will be assigned the list of all nodes for the TargetHost role of the specified component that satisfy its deployment constraints. We initially used a single-layered approach to deducing a concern merging strategy. The single-layer would invoke the local rules to find valid endpoints for each element in the concerns being merged that was involved in a cross-concern relationship. When resource and other global constraints began being added, we had to adopt a multi-layered approach to solve for merging strategies.

As can be seen in Figure 5, local constraint solving is the initial step of our automatic constraint-aware weaving. In

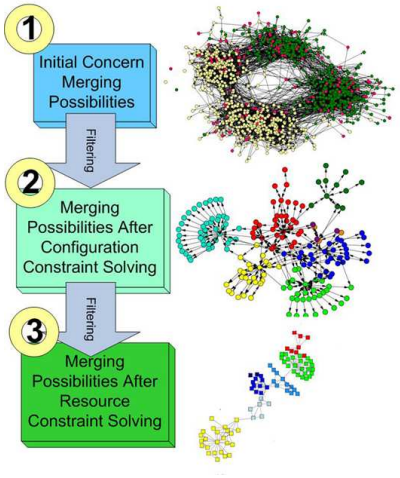

Figure 5: Iteratively Reducing the Concern Merging Solution Space

AUTODeploy, the local constraints correspond to the configuration constraints, such as required OS, that impact only the valid hosting sites for a single component. The solution space initially contains many millions or more possible concern merging combinations, as seen in step 1 of Figure 5. Any global constraint-aware weaving has to start by iterating over each unassigned component and considering only valid ECUs respecting the configuration constraints. After pruning the solution space, global constraints, such as resource requirements, are considered, as shown in step 2 of Figure 5. After solving the global constriants, AUTODeploy is left with a drastically reduced number of concern merging 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 critera, such as the number of nodes used.

Even after implementing this multi-layered solving approach, finding a merger that met the global constraints could still take a long time. What makes the situation complicated is the fact that a blind iteration approach through possible solutions may be initiated very far from a valid solution. The key to quickly finding a solution is to start as near as possible to it.

For example, 20% of the components may be computationally intensive and require substantial CPU, but less RAM than the other 80% of the components that store a large amount of state in memory. The second type of components have high RAM requirements and only a medium demand of CPU usage. We found that AUTODeploy could find a merging strategy in significantly less time by developing domain-specific heuristics for distinguishing CPU- and RAM-intensive components and assigning them to nodes having comparable proportions of each resource. Moreover, as described in [7], by choosing the right search strategy, different guarantees can be made on the optimality of the solution found. AUTODeploy's solver uses a domain-specific heuristic to prioritize the assignment of components and their respective ECU candidates by selecting the largest weighted average of each component's required resources and each ECU's remaining resources.

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 parts of the mappings between concerns. We introduced a mechanism into AUTODeploy so that developers could first fix the deployment locations of certain components with unspecifiable constraints an then use the constraint solver to complete the partially specified deployment.

A liability of allowing developers to manually assign components to nodes is that it can easily lead to situations where the fixed component-node mappings makes finding a valid concern merger impossible. For these cases, strategies for repairing the model automatically were needed. These AUTODeploy's merger repair mechanisms are discussed next.

4.3 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 weaving impossible. With numerous complex composition rules guiding the weaving process, it was extremely hard for modelers to figure out *why* there was no valid way of merging concerns and *how* to repair the model to overcome the problem. Simply failing to weave the model 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 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 host all of the components, it is not necessarily a single component that is causing the problem. Marking the first component that could not be placed would not make 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, the modeler would still need to manually figure out 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 of allowing modelers to express a set of legal model modifications that could be performed, which we call "repair operations," and using AUTODeploy's constraint solver to apply these repair operations to the model to make weaving possible. For example, a repair operator *IncreaseCPUPower* could be used to allow the constraint solver to place a component on a node or if no suitable node was found to increase the CPU power (a suggestion to improve the hardware) of an upgradable ECU. By specifying a series of repair operations, such as *IncreaseCPUPower* 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 instance, modelers could try to deploy a component manually and find that the automatic weaving guidance does not provide any valid EPUs. This failure might occur if no EPU matching the configuration requirements of the component (*e.g.*, the required operating system or hosting server type). Another reason for the failure could be that all resources of valid EPUs have been exhausted by previous component assignments. Corresponding suggestions could therefore be to create a compliant EPU or to increase respective resources of a single EPU. On the other hand, a global solver may use the repair operations to apply a batch of corrections to the model to make weaving possible.

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

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

It can be used to pass the following operator to a call of *is_a_valid_component_targethost*:

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

which uses the same pair of Component/Node variables. In addition, there are some input arguments and output arguments. 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 node must first check whether the currently considered invalid component/node pair is caused by a lack of resources on the node and wether 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 node assignments it has made.

The AUTODeploy 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 weaving failure. In contrast, AUTODeploy model repair raises the level of abstraction by specifying possible domain-specific corrections within the underlying domain structure and domain entities.

5. RELATED WORK

This section compares our AUTODeploy approach with related work. The techniques we have developed to solve the concern merging challenges for AUTODeploy build on the techniques presented by Clarke in [6]. Clarke presents a method for decomposing and composing models based on subject oriented design concerns. One of the key ideas presented in their paper are mechanisms for model merger and model conflict resolution.

Our work is a complementary enhancement of Clarke's work. We provide a framework for handling the complexity of domains, such as the automotive domain, where many types of constraints are too combinatorial in nature to handle without a constraint solver. We also enable a constraint solver to resolve model conflicts, which allows far more complex model repair strategies. Another key difference between our work and Clarke's is that [6] focuses on UML, whereas we have created domain-specific modeling languages for each concern we are merging.

In [13], Georg et al. present a method for composing aspects that can handle composition conflicts by obeying user specified merger directives. Our AUTODeploy tool acknowledges and handles the complexities and frequent conflicts in aspect merger identified by Georg et al. Our approach is similiar in that it allows users to specify a series of repair operators, to apply in the case that conflicts prevent a merger. Moreover, our approach provides the flexibility of allowing users to override and force particular merging decisions. The key difference between AUTODeploy and the work described in [13] is that AUTODeploy can use multiple constraint solvers to merge complex concerns that would be unmanagable using manual methods or traditional algorithms.

6. CONCLUDING REMARKS

This paper presents the results of applying the AUTODeploy aspect-oriented design and modeling tool to develop a component-based automotive DRE system. The following is a summary of our lessons learned from this project:

- The complexity of DRE systems can be managed effectively by applying the principle of separation of concerns at the model level. Our initial model-driven approach separated concerns with respect to specifying and merging the component collaboration aspect and the hardware aspect. Merging these concern models to map software to hardware components is complicated, however, and cannot be done manually in the automotive domain due to (1) the size of each concern model, which can comprise hundreds of entities, and (2) the complexity of domain constraints that must be met when merging different concern models. We discovered that augmenting our weaver with guidance from a constraint solver helped to overcome problems.
- Java and C constraints use notations and concepts that are not intuitive to experts in the automotive domain, who are a key source of knowledge on domain constraints that guide concern composition. For instance, the engineers developing a specific automobile model are intimately familiar with values about a safe distance for an ECU from the car perimeter and tolerable resource allocation values. Although we initially hoped to use a Java- or C-based constraint solver, we found that they were not well-suited for domain experts since they require assistance from constraint solver experts when domain experts needed to introduce new composition constraints.
- When augmented with the right model-driven abstractions and predicates, Prolog's declarative nature makes an effective choice as a constraint solver framework for domain experts. We found that using Prolog's declarative expressivity to implement constraint solvers allowed domain experts to write constraints. Prolog's

role-based domain constraints are a type of semantic pointcut specification that form the basis of our approach to weaving. A modeling-based approach and domain-specific abstractions are the key to making Prolog amenable by domain-experts.

- Standard Prolog tracing is not sufficient to debug weaving problems for this domain. In order to handle weaving failures, the repair operator framework we implemented in Prolog is needed to make deducing weaving strategies, as well as debugging weaving failures, possible.

- Even though semantic weaving mechanisms are powerful, domain experts tend to devise new and more difficult constraint types, such as guaranteeing paths between components with various characteristics, that are not well supported by current solvers.

- We have found that it is extremely important to have a flexible and powerful concern composition language that provides the ability to capture complex semantic composition rules, *i.e.*, semantic pointcuts. Most conventional AOSD tools provide only syntactic means to describe composition rules. For large-scale DRE systems with complex constraints, however, semantic knowledge must be leveraged to deduce proper weaving strategy. The approach taken for our project is one possible direction to overcome this problem. Our experience shows that combining a Prolog knowledge base generated from models with a Prolog-based set of constraints and weaving rules can scaleup to support the larger-scale models in next-generation automotive DRE systems.

GEMS and the AUTODeploy prototype are opensource projects available from: http://www.sf.net/projects/gems.

7. REFERENCES

[1] Aspectj, http://www.eclipse.org/aspectj/.

[2] Hyperj, http://www.alphaworks.ibm.com/tech/hyperj.

[3] The openarchitectureware. http://www.eclipse.org/gmt/oaw.

[4] B. Boehm and C. Abts. Cots integration: Plug and pray? *IEEE Computer*, 32(1):135–138, 1999.

[5] D. Chen, J. Elkhoury, and M. Trngren. A modelling framework for automotive embedded control systems. In *SAE World Congress*, 2003.

[6] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 44(1):71–100, 2002.

[7] E. Coffman Jr, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: combinatorial analysis. *Handbook of Combinatorial Optimization. Kluwer Academic Publishers*, 1998.

[8] I. Crnkovic, J. Axelsson, S. Graf, M. Larsson, R. van Ommering, and K. Wallnau. Cots component-based embedded systems–a dream or reality?

[9] G. De Michell and R. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[10] M. Del Fabro, J. Bzivin, and P. Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*, 2006.

[11] G. Deng, J. Balasubramanian, W. Otte, D. Schmidt, and A. Gokhale. Dance: A qos-enabled component deployment and configuration engine. *Proceedings of the 3rd Working Conference on Component Deployment*, 2005.

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

[13] G. Georg, R. France, and I. Ray. Composing aspect models. *The 4th AOSD Modeling With UML Workshop*, 2003.

[14] S. Gérard, F. Terrier, and Y. Tanguy. Using the model paradigm for real-time systems development: Accord/uml. *Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 260–269, 2002.

[15] B. Hall, B. Sellner, and R. Maier. Automated safety critical software development for distributed control systems: A cots approach. *SAE transactions*, 110(7):293–302, 2001.

[16] P. Hastono and S. Huss. Automatic generation of executable models from structured approach real-time specifications. In *To Appear at The 25th IEEE International Real-Time Systems Symposium (RTSS)*, 2004.

[17] Kopetz and Hermann. *Real-Time Systems : Design Principles for Distributed Embedded Applications (The International Series in Engineering and Computer Science)*. Springenr, April 1997.

[18] A. Nechypurenko, J. White, E. Wuchner, and D. C. Schmidt. Applying model intelligence frameworks to deployment problems in real-time and embedded systems. In *Proceedings of MARTES: Modeling and Analysis of Real-Time and Embedded Systems at the 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2006*, 2006.

[19] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. *IBM Research Report 21452*, 1999.

[20] J. Srinivasan and K. Lundqvist. Real-time architecture analysis: a cots perspective. *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, 1, 2002.

[21] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, 1999.

[22] M. Trngren, P. Johanessen, and N. Adamsson. Lessons learned from model based development of a distributed embedded automotive control system. In *SAE World Congress*, 2003.

[23] M. Weber and J. Weisbrod. Requirements engineering in automotive development-experiences and challenges. pages 331–340, 2002.

[24] R. Weeks and J. Moskwa. Automotive engine modeling for real-time control using matlab/simulink. *SAE paper*, 950417:123–137, 1995.

[25] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt. Intelligence frameworks for assisting modelers in combinatorically challenging domains. In *Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems at the Fifth International Conference on Generative Programming and Component Engineering (GPCE 2006)*, 2006.

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

[27] 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.