Constraint-based Model Weaving

Jules White[1], Jeff Gray[2], and Douglas C. Schmidt[1]

[1] Vanderbilt University
Nashville, TN, USA
{jules, schmidt}@dre.vanderbilt.edu
[2] University of Alabama at Birmingham
Birmingham, AL, USA
gray@cis.uab.edu

Abstract. Aspect-Oriented Modeling (AOM) is a promising technique for untangling the concerns of complex enterprise software systems. AOM decomposes the cross-cutting concerns of a model into separate models that can be woven together to form a composite solution model. In many domains, such as multi-tiered e-commerce web applications, separating concerns is much easier than deducing the proper way to weave the concerns back together into a solution model. For example, modeling the types and sizes of caches that can be leveraged by a web application is much easier than deducing the optimal way to weave the caches back into the solution architecture to achieve high system throughput.

This paper presents a technique called constraint-based weaving that maps model weaving to a constraint satisfaction problem (CSP) and uses a constraint-solver to deduce the appropriate weaving strategy. By mapping model weaving to a CSP and leveraging a constraint solver, our technique (1) generates solutions that are correct with respect to the weaving constraints, (2) can incorporate complex global weaving constraints, (3) can provide weaving solutions that are optimal with respect to a weaving cost function, and (4) can eliminate manual effort that would normally be required to specify pointcuts and maintain them as target models change. The paper also presents the results of a case study that applies our CSP weaving technique to a representative enterprise Java application. Our evaluation of this case study showed a reduction in manual effort that our technique provides.

Keywords: Model Weaving, Aspect-Oriented Programming, Constraint Satisfaction, Global Constraints

1 Introduction

Developers of complex enterprise applications are faced with the daunting task of managing not only numerous functional concerns, such as ensuring that the application properly executes key business logic, but also meeting challenging non-functional requirements, such as end-to-end response time and security. Enterprise domain solutions have traditionally been developed using large monolithic models that either provide a single view of the system or a limited set of views [20]. The result of using a limited

set of views to build the system is that certain concerns are not cleanly separated by the dominant lines of decomposition and are scattered throughout the system's models.

Aspect-Oriented Modeling (AOM) [7, 17, 38] has emerged as a powerful method of untangling and managing scattered concerns in large enterprise application models [19, 21]. With AOM, any scattered concern can be extracted into its own view. For example, caching considerations of an application can be extracted into an aspect. Once caching is separated into its own aspect, the cache sizes and types can be adjusted independently of the application components where the caches are applied. When a final composite solution model for the application is produced, the various aspects are woven back into the solution model and the numerous affected modeling elements are updated to reflect the independently modeled concerns.

Although concerns can often be separated easily into their own aspects or views, it is hard to correctly or optimally merge these concerns back into the solution model. Merging the models is hard because there are typically numerous competing non-functional and functional constraints, such as balancing encryption levels for security against end-to-end performance, that must be balanced against each other without violating domain constraints (such as maximum available bandwidth). Manual approaches for deriving solutions to these types of constraints do not scale well.

Most current model weavers [9, 16, 21, 38, 45] rely on techniques, such as specifying queries or patterns to match against model elements, that are ideal for matching advice against methods and constructors in application code, but are not necessarily ideal for static weaving problems. Many enterprise applications require developers to incorporate global constraints into the weaving process that can only be solved in a static weaving problem. As discussed in Section 3.2, the techniques used to match against dynamic joinpoints, such as pattern matching, cannot capture global constraints, such as resource constraints (*e.g.*, total RAM consumed < available RAM), that are common in enterprise applications. Because global constraints are not honored by the model weaver, developers are forced to expend significant effort manually deriving weaving solutions that honor them.

When weavers cannot handle global constraints, optimization, or dependency-based constraints, traditional model weaving becomes a manual four-stage process, as shown in Figure 1. The left column shows the steps involved in model weaving problems with global constraints in general. The right column shows how these steps manifest themselves in the cache weaving example. First, the advice and joinpoint elements (*e.g.*, caches and components) available in the solution model are identified in step 1. Second, as shown in steps 2 and 3, because a weaver cannot handle global constraints or optimization, developers manually determine which advice elements should be matched to which model elements (*e.g.*, the cache types, cache sizes, and the components to apply the caches to). This second step requires substantial effort because it involves deriving a solution to a complex set of global constraints.

In terms of deriving cache placements in an enterprise application, the second step involves determining cache architectures that fit within the required memory budget and respect the numerous dependency and exclusion constraints between caches. After viable cache architectures are identified, a developer must use the expected request distribution patterns and queueing theory to predict the optimal cache architecture. As

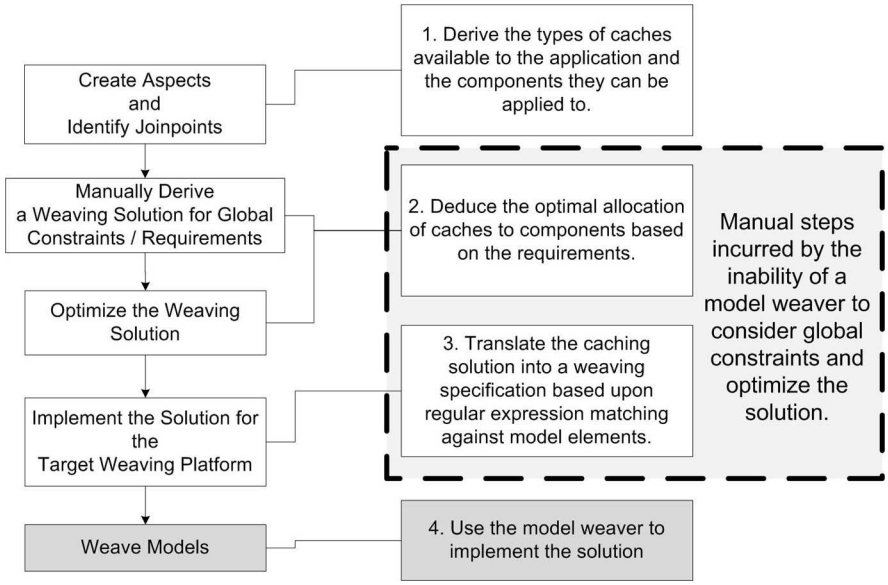

Fig. 1: The Model Weaving Process Applied to Cache Allocation

the examples in Section 3 show, even for a small set of caches and potential cache locations, the cache placement process requires significant work.

In the third step, developers take this manually-derived solution and translate it into pointcut definitions that match against model elements using regular expressions or queries (*e.g.*, a specification of how to insert the caching model elements into the models to implement the caching architecture). In some cases, the manually derived solution needs to be translated into the pointcut specification languages of multiple model weavers so that the architecture can be implemented in a set of heterogeneous models spanning multiple modeling tools. The model weavers then take these final specifications and merge the models. Each time the underlying solution models change (*e.g.*, the available memory for caching changes), the global constraints can cause the entire solution to change (*e.g.*, the previously used caches no longer fit in the budgeted memory) and the entire three steps must be repeated from scratch.

This paper shows that the manual steps of deriving a weaving solution that meets the global application requirements (steps 2 and 3) can be automated in many cases by creating a weaver capable of handling global constraints and optimization. Creating a weaver that can honor these constraints and optimize weaving allows developers to translate the high-level application requirements into pointcut specifications and optimization goals that can be used by the weaver when producing a weaving solution. Finally, because the weaver is responsible for deducing a weaving solution that meets the overall application requirements, as the individual solution models change, the weaver can automatically update the global weaving solution and re-implement it on behalf of the developer for multiple model weaving platforms.

This paper shows how model weaving can be mapped to a constraint satisfaction problem (CSP) [13, 34, 44]. With a CSP formulation of a model weaving problem, a constraint solver can be used to derive a correct—and in some cases optimal—weaving solution. Using a constraint solver to derive a correct weaving solution provides the following key benefits to model weaving:

- It ensures that the solution is correct with respect to the various modeled functional and non-functional weaving constraints.
- A constraint solver can honor global constraints when producing a solution and not just local regular expression or query-based constraints.
- A constraint solver automates the deduction of the correct weaving and saves considerable manual solution derivation effort.
- The weaving solution can automatically be updated by the solver when the core solution models (and hence joinpoints) change.
- The solver can produce a platform-independent weaving solution (a symbolic weaving solution that is not coupled to any specific pointcut language) where model transformations [8, 15] are applied to create a weaving solution for each required weaving platform and
- The solver can derive an optimal weaving solution (with respect to a cost function) in many cases.

The remainder of this paper is organized as follows: Section 2 presents the multi-tiered web application used as a case study throughout the paper; Section 3 shows current challenges in applying existing model weaving techniques to our case study; Section 5 describes how constraint solving can be used to derive a correct weaving solution and how it addresses the gaps in existing solutions; Section 4 presents a mapping from model weaving to a constraint satisfaction problem; Section 7 summarizes empirical results obtained from applying constraint-based weaving to our case study; Section 8 compares constraint-based weaving with related work; and Section 9 presents concluding remarks and lessons learned.

2 Case Study: The Java Pet Store

This paper uses a case study based on Sun's Java Pet Store [5] multi-tiered e-commerce application. The Pet Store is a canonical e-commerce application for selling pets. Customers can create accounts, browse the Pet Store's product categories, products, and individual product items (*e.g.*, male adult Bulldog vs. female adult Bulldog).

The Pet Store application was implemented by Sun to showcase the capabilities of the various Java 2 Enterprise Edition frameworks [43]. The Pet Store has since been re-implemented or modified by multiple parties, including Microsoft (the .NET Pet Store) [4] and the Java Spring Framework [6]. The Spring Framework's version of the Pet Store includes support for aspects via AspectJ [2] and Spring Interceptors and is hence the implementation that we base our study on.

2.1 Middle-tier Caching in the Pet Store

Our case study focuses on implementing caching in the middle-tier (*i.e.*, the persistent data access layer) of the Pet Store through caching aspects. The business logic and views in the Pet Store are relatively simple and thus the retrieval and storage of persistent data is the major performance bottleneck. In performance tests that we ran on the Pet Store using Apache JMeter [1], the average response time across 3,000 requests for viewing the product categories was 3 times greater for a remotely hosted database versus a remotely hosted database with a local data cache (25% hit rate). The same tests also showed that caching reduced the worst case response time for viewing product categories by a factor of two.

Our experiments tested only a single middle-tier and back-end configuration of the Pet Store. Many different configurations are possible. The Spring Pet Store can use a single database for product and order data or separate databases. Data access objects (DAOs) are provided for four different database vendors. Choosing the correct way of weaving caches into the middle-tier of the Pet Store requires considering the following factors:

- The workload characteristics or distributions of request types, which determine what data is most beneficial to cache [32]. For example, keeping the product information in the cache that is most frequently requested will be most beneficial.
- The architecture of the back-end database servers providing product, account, and order data to the application determines the cost of a query [31]. For example, in a simple Pet Store deployment where the back-end database is co-located with the Pet Store's application server, queries will be less expensive than in an arrangement where queries must be sent across a network to the database server.
- The hardware hosting the cache and the applications co-located with it will determine the amount of memory available for caching product data. If the Pet Store is deployed on small commodity servers with limited memory, large caches may be undesirable.
- The number of possible cache keys and sizes of the data associated with each cache item will influence the expected cache hit rate and the penalty for having to transfer a data set across the network from the database to the application server [35]. For example, product categories with large numbers of products will be more expensive to serialize and transfer from the database than the information on a single product item.
- The frequency that the data associated with the various middle-tier DAOs is updated and the importance of up-to-date information will affect which items can be cached and any required cache coherence schemes [35]. For example, product item availability is likely to change frequently, making product items less suitable to cache than product categories that are unlikely to change.

2.2 Modeling and Integrating Caches into the Pet Store

Aspect modeling can be used effectively to weave caches into the Pet Store to adapt it for changing request distribution patterns and back-end database configurations. We

used this scenario for our case study to show that although caches can be woven into code and models to adapt the Pet Store for a new environment, creating and maintaining a cache weaving solution that satisfies the Pet Store's global application requirements takes significant manual effort due to the inability of model weavers to encode and automate weaving with the global application constraints. Each time the global application requirements change, the manually deduced global cache weaving solution must be updated. Updating the global cache weaving solution involves a number of models and tools. Figure 2 shows the various models, code artifacts, and tools involved in implementing caching in the Pet Store.

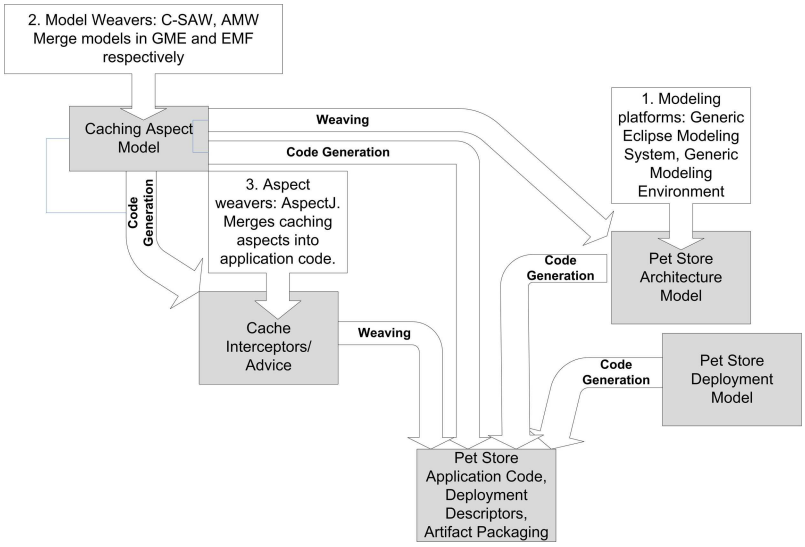

Fig. 2: Models and Tools Involved in the Pet Store

1. Modeling platforms. We have implemented models of different parts of the Pet Store in two different modeling tools: the Generic Eclipse Modeling System (GEMS) [48] and the Generic Modeling Environment (GME) [30]. GME was chosen due to its extensive support for different views, while GEMS was selected for its strengths in *model intelligence*, which was used for automating parts of the deployment modeling process. Using different tools simplifies the derivation of the deployment plan and the understanding of the system architecture, but also requires some level of integration between the tools.

GEMS is a graphical modeling tool built on top of Eclipse [41] and the Eclipse Modeling Framework (EMF) [12]. GEMS allows developers to use a Visio-like graphical interface to specify metamodels and generate domain-specific modeling language (DSML) tools for Eclipse. In GEMS, a deployment modeling tool has been implemented to capture the various deployment artifacts, such as required Java Archive Resources (JAR) files, and their placement on application servers. Another Neat Tool

(ANT) [24] build, configuration, and deployment scripts can be generated from the GEMS deployment model.

GME [30] is another graphical modeling tool similar to GEMS that allows developers to graphically specify a metamodel and generate a DSML editor. A modeling tool for specifying the overall component architecture of the Pet Store has been implemented in GME. The GME architecture model is used to capture the component types, the various client types, back-end database architecture, and expected distribution of client requests to the Pet Store. The GME architecture model is shown in Figure 3.

2. Model weaving tools. The caching aspect of the Pet Store is modeled separately from the GEMS deployment model and GME architecture model. Each time the caching model is updated, model weaving tools must be used to apply the new caching architecture to the GEMS and GME models. For the GME models, the C-SAW [42] model weaver is used to merge the caching architecture into the architecture model. C-SAW relies on a series of weaving definition files to perform the merger. Each manually derived global cache weaving solution is implemented in C-SAW's weaving definition files to apply to the GME architecture models. Again, because we need two separate modeling tools to produce the best possible deployment and architecture models, we must also utilize and integrate two separate model weavers into the development process.

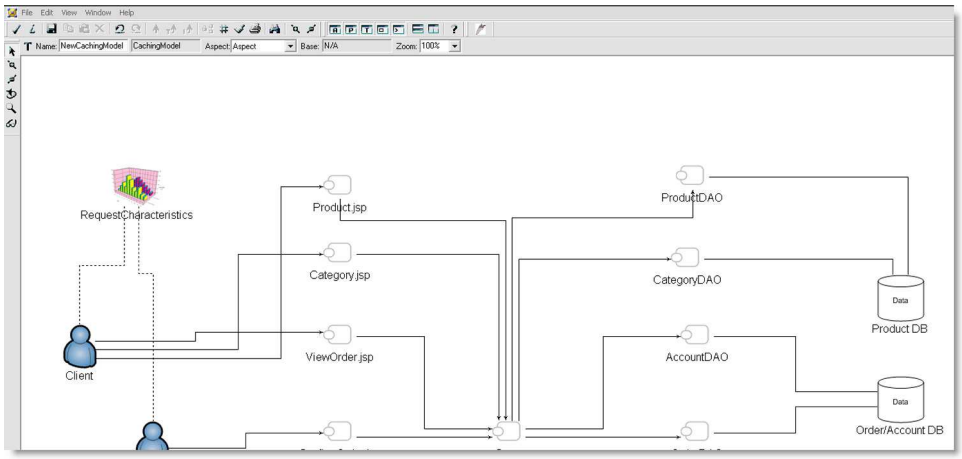

Fig. 3: GME Pet Store Architecture Model

The deployment models in GEMS need to be updated via a model weaver, such as the Atlas Model Weaver (AMW) [16], which can interoperate with models based on EMF. With AMW, developers specify two EMF models and a series of merger directives (*i.e.*, a weaving specification). AMW produces a third merged EMF model from the two source models. Each global cache weaving solution must also be implemented as a weaving specification for AMW. Once the AMW specification is implemented, the

cache weaving solution can be merged into the GEMS EMF-based deployment model to include any required JAR files and cache configuration steps.

3. Code weaving tools. Finally, to apply the cache weaving solution to the legacy Pet Store code, the Java cache advice implementations must be woven into the Pet Store's middle-tier objects using AspectJ [2], which is a framework for weaving advice into Java applications. Although the Spring framework allows the application of AspectJ advice definitions to the Pet Store, it requires that the Spring bean definition files for the Pet Store be updated to include the new AspectJ pointcuts and advice specifications. A final third implementation of the global cache weaving solution must be created and specified in terms of Spring bean definitions and AspectJ pointcuts.

Overall, there are three separate tool chains that the Pet Store cache weaving solution must be implemented in. First, C-SAW weaving specifications must be created to update the GME architectural models. Second, AMW weaving specifications must be produced to update the GEMS deployment models. Finally, the weaving solution must be turned into AspectJ advice/pointcut definitions for weaving the caches into the Pet Store at runtime.

3 Model Weaving Challenges

One of the primary limitations of applying existing model weavers to the Pet Store case study described in Section 2 is that existing model weaver pointcut specifications cannot encode global application constraints, such as memory consumption constraints, and also cannot leverage global constraints or dependency-based weaving rules to produce an overall global weaving solution. Developers must instead document and derive a solution for the overall global application constraints and implement the solution for each of the numerous modeling and weaving platforms for the Pet Store. Moreover, each time the underlying global application constraints change (*e.g.*, the memory available for caches is adjusted) the overall global weaving solution must be recalculated and implemented in the numerous modeling tools and platforms.

3.1 Differences Between Aspect Weavers and Model Weavers

To understand why model weavers do not currently support global constraints and how this can be rectified, we first must evaluate aspect weavers at the coding level, which have influenced model weavers. Aspect weavers, such as AspectJ and HyperJ [3], face an indeterminate number of potential joinpoints (also referred to as *joinpoint shadows* [23]) that will be passed through during application execution. For example, late-binding can be used in a Java application to dynamically load and link in multiple libraries for different parts of the application.

Each library may have hundreds or thousands of classes and numerous methods per class (each a potential joinpoint). An aspect weaver cannot know which classes and methods the execution path of the application will pass through before the process exits. The weaver can therefore never ascertain the exact set of potential joinpoints that will be used ahead of time. Although the weaver may have knowledge of every

joinpoint shadow, it will not have knowledge of which are actually used at runtime. Model weaving, however, faces a different situation than a runtime aspect weaver. The key differences are:

- Model weaving merges two models of finite and known size.
- Because models have no thread of execution, the weaver can ascertain exactly what joinpoints are used by each model.
- Model weaving speed is less critical than aspect weaving speed at runtime and adding additional seconds to the total weaving time is not unreasonable.

Because a model weaver has knowledge of the entire set of joinpoints used by the models at its disposal it can perform a number of activities that are not possible with runtime weaving where the entire used set of target joinpoints is not known. For example, a model weaver can incorporate global constraints into the weaving process. A runtime weaver cannot honor global constraints because it cannot see the entire used joinpoint set at once. To honor a global constraint, the weaver must be able to see the entire target joinpoint set to avoid violating a global constraint.

Runtime aspect weaving involves a large number of potential joinpoints or joinpoint shadows and is not well-suited for capturing and solving global application constraints as part of the weaving process. When weaving must be performed on an extremely large set of target joinpoints, the weaver must use a high-efficiency technique for matching advice to joinpoints (every millisecond counts). The most common technique is to use a query or regular expression that can be used to determine if a pointcut matches a joinpoint. The queries and regular expressions are independent of each other, which allows the weaver to quickly compare each pointcut to the potential joinpoints and determine matches.

If dependencies were introduced between the queries or expressions (*e.g.*, only match pointcut A if pointcut B or C do not match), the weaver would be forced to perform far less efficient matching algorithms. Moreover, since the weaver could not know the entire joinpoint set passed through by the application's execution thread ahead of time, it could not honor a dependency, such as match pointcut A only if pointcuts B and C are *never* matched, because it cannot predict whether or not B and C will match in the future. Finally, when dependencies are introduced, there is no longer necessarily a single correct solution. Situations can arise where the weaver must either choose to apply A or to apply B and C.

3.2 Challenge 1: Existing Model Weaving Poinctut Specifications Cannot Encode Global Application Constraints

Most model weavers, such as C-SAW, AMW, and the Motorola WEAVR [14], have adopted the approach of runtime weavers and do not allow dependencies between pointcuts or global constraints. Because the model weaver does not incorporate these types of constraints, developers cannot encode the global application constraints into the weaving specification. Figure 4 presents the manual refactoring steps (the first six steps) that must be performed when the modeled distribution of request types to the Pet Store changes.

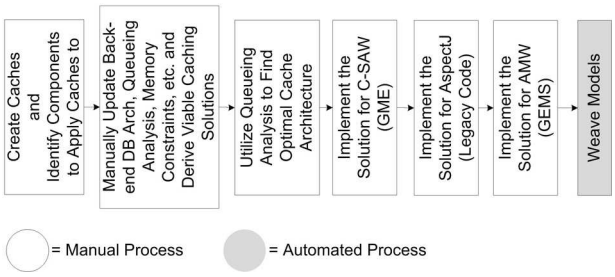

Fig. 4: Solution Model Changes Cause Weaving Solution Updates

In the Pet Store case study, there are a number of dependencies and global constraints that must be honored to find a correct weaving. We created caching advice implementations that capture all product queries and implementations that are biased towards specific data items, such as the `FishCache`. The biased cache is used when the majority of requests are for a particular product type. The `FishCache` and the generic product cache should be mutually exclusive. The use of the `FishCache` is excluded if the percentage of requests for fish drops below 50%. Moreover, the generic product cache will then become applicable and must be applied.

A small change in the solution model can cause numerous significant ripple effects in the global application constraints and hence weaving solution. This problem of changes to the solution models of an applicaiton causing substantial refactoring of the weaving solution is well-known [22]. The problem becomes even more complex, however, with the global weaving solution where significant refactoring causes multiple implementations of the weaving specification to change.

The problem with managing this ripple effect with existing model weavers is that both the `FishCache` and the generic product cache have a pointcut that matches the same model element, the `ProductDAO`. With existing pointcut languages based on regular expressions or queries, there is no way to specify that only one of the two pointcut definitions should be matched to the `ProductDAO`. The pointcut definitions only allow the developer to specify matching conditions based on joinpoint properties and not on the matching success of other pointcuts.

Developers often need to restrict the overall cache selection to use less than a specified amount of memory. For example, rather than having the `FishCache` and `GenericCache` be mutually exclusive, the two caches could be allowed to be applied if there is sufficient memory available to support both. Requiring that the woven caches fit within a memory budget is a resource constraint on the total memory consumed by the weaving solution and relies on specifying a property over the entire weaving solution. Existing regular expression and query-based pointcut languages usually do not capture these types of rules.

Another challenge of producing this weaving constraint on the memory consumed by the caches is that it relies on properties of both the advice objects (*e.g.*, the memory consumed by the cache) and the joinpoint objects (*e.g.*, the memory available to the hosting object's application server). Most model weaving pointcut languages allow

specifying conditions only against the properties of the target joinpoints and not over the advice elements associated with the pointcut. To circumvent this limitation, developers must manually add up the memory consumed by the advice associated with the pointcut and encode it into the pointcut specification's query (*e.g.,* find all elements hosted by an application server with at least 30 MB of memory).

3.3 Challenge 2: Changes to the Solution Model Can Require Significant Refactoring of the Weaving Solution

As the solution models of the application that determine the set of joinpoints change, each manual step in Figure 4 may need to be repeated. The caching solution relies on multiple solution models, such as the server request distribution model, the cache hit ratio and service times model, and the PetStore software architecture model. A change in any of these models can trigger a recalculation of the global weaving solution. Each recalculation of the global weaving solution involves multiple complex caculations to determine the new targets for caches. After the new cache targets are identified, the implementation of the solution for each weaving platform, such as the C-SAW weaving definition files, must be updated to reflect the new caching architecture.

For example, the correct weaving of caches into the Pet Store requires considering the back-end organization of the product database. If the database is hosted on a separate server from the Pet Store's application server, caching product information can significantly improve performance, as described in Section 2. The cache weaving solution is no longer correct, however, if biased caches are applied to various product types that are being retrieved from a remote database and the database is co-hosted with the Pet Store's application server. A Developer must then update the weaving solution to produce a new and correct solution for the updated solution model.

As seen in Figure 5, not only are numerous manual steps required to update the weaving solution when solution model changes occur, but each manual step can be complex. For example, re-caculating the optimal placement of caches using a queueing model is non-trivial. Moreover, each manual step in the process is a potential source of errors that can produce incorrect solutions and require repeating the process. The large numbers of solution model changes that occur in enterprise development and the complexity of updating the weaving solution to respect global constraints, make manually updating a global weaving solution hard.

3.4 Challenge 3: Existing Model Weavers Cannot Leverage a Weaving Goal to Find an Optimal Concern Merging Solution

Another challenge of encoding global application constraints into a weaving specification is that global constraints create situations where there are multiple correct solutions. Existing model weavers do not allow situations where there are multiple possible weaving solutions. Because the weaver cannot choose between weaving solutions, developers must manually deduce the correct and optimal solution to use.

Optimizing a solution bound by a set of global constraints is a computationally intensive search process. Searching for an optimal solution involves exploring the solution space (the set of solutions that adhere to the global constraints) to determine the

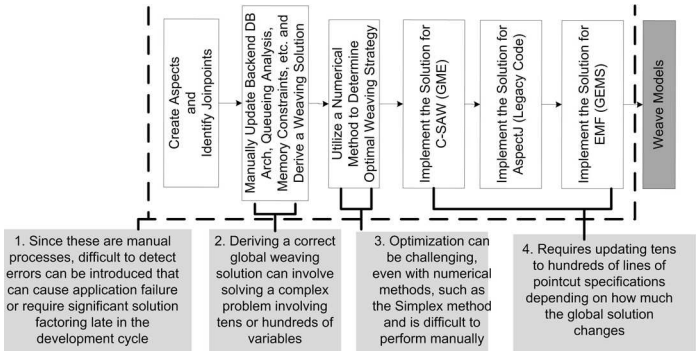

Fig. 5: Challenges of Updating a Weaving Solution

optimal solution. This type of optimization search can sometimes be performed manually with numerical methods, such as the Simplex [37] method, but is typically hard. In particular, each time the solution models change, developers must manually derive a new optimal solution from scratch.

For example, to optimize the allocation of caches to DAOs in the Pet Store, developers must:

- Evaluate the back-end database configuration to determine if product, account, or other data must be cached to reduce query latency.
- Derive from the cache deployment constraints what caches can be applied to the system and in what combinations.
- Determine how much memory is available to the caches and how memory constraints restrict potential cache configurations.
- Exhaustively compare feasible caching architectures using queuing analysis to derive the optimal allocation of caches to DAOs based on DAO service rates with and without caching and with various cache hit rates.

It is hard to manually perform these complex calculations each time the solution models change or caching constraints are modified.

4 CSP-based Model Weaving

To address the challenges described in Section 3, we have developed *AspectScatter*, which is a static model weaver that can:

1. Transform a model weaving problem into a CSP and incorporate global constraints and dependencies between pointcuts to address Challenge 1 from Section 3.2.
2. Using a constraint solver, automatically derive a weaving solution that is correct with respect to a set of global constraints, eliminating the need to manually update the weaving solution when solution models change, as described in Challenge 2 from Section 3.3

3. Select an optimal weaving solution (when multiple solutions exist) with regard to a function over the properties of the advice and joinpoints, allowing the weaver rather than the developer to perform optimization, thereby addressing Challenge 3 from Section 3.4.
4. Produce a platform-independent weaving model and transform it into multiple platform-specific weaving solutions for AspectJ, C-SAW, and AMW through model transformations, thus addressing the problems associated with maintaining the weaving specification in multiple weaving platforms.

Figure 6 shows an overview of AspectScatter's weaving approach. First, develop-

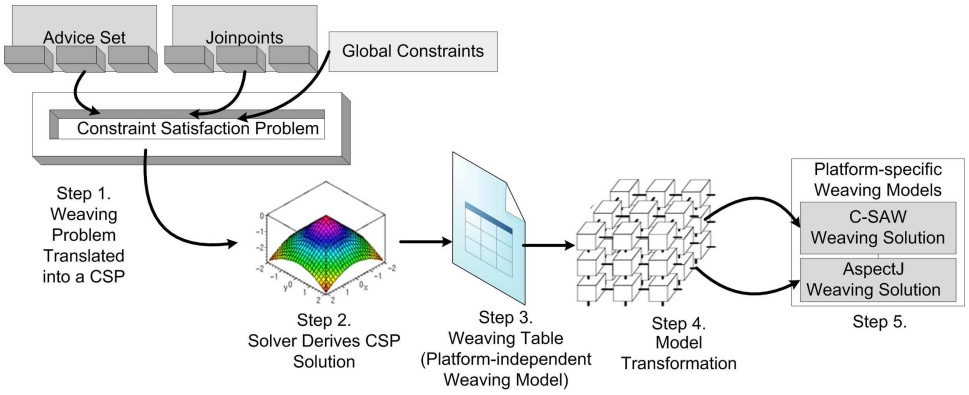

Fig. 6: Constraint-based Weaving Overview

ers describe the advice, joinpoints, and weaving constraints to AspectScatter using its domain-specific language (DSL) for specifying aspect weaving problems with global constraints. In Step 1, AspectScatter transforms the DSL instance into a CSP. In Step 2, AspectScatter uses a constraint solver to derive a guaranteed correct and, if needed, optimal weaving solution. In Step 3, AspectScatter transforms the solution into a platform-independent weaving model. In Step 4, model transformations are used to transform the platform-independent weaving model into specific implementations, such as C-SAW weaving definition files, for each target weaving platform. Finally, in Step 5, the platform-specific weaving models are applied to their target models or code.

The remainder of this section presents a mapping from model weaving to a CSP. By producing a CSP for model weaving, a constraint solver can be used to deduce a correct and in many cases optimal solution to a weaving problem.

4.1 CSP Background

A CSP is a set of variables and a set of constraints over those variables. For example, $A < B < 100$ is a CSP over the integer variables A and B. A solution to a CSP is a set of values for the variables (called a labeling) that adheres to the set of constraints. For example, $A = 10, B = 50$ is a valid labeling (solution) of the example CSP.

Solutions to CSPs are obtained by using *constraint solvers*, which are automated tools for finding CSP solutions. Constraint solvers build a graph of the variables and constraints and apply techniques, such as arc-consistency, to find the ranges that variable values can be set to. Search algorithms then traverse the constraint network to hone in on a valid or optimal solution.

A constraint solver can also be used to derive a labeling of a CSP that maximizes or minimizes a specific goal function (*i.e.*, a function over the variables). For example, the solver could be asked to maximize the goal function $A+B$ in our example CSP. A maximal labeling of the variables with respect to this goal function would be $A = 98, B = 99$.

4.2 Mapping Cache Weaving to a CSP

Cache weaving can be used as a simple example of how a CSP can be used to solve a weaving problem. In the following example, we make several assumptions, such as the hit ratio for the caches being the same for both joinpoints, to simplify the problem for clarity. Real weaving examples involving optimal caching or other types of global constraints are substantially more difficult to solve manually and hence motivate our constraint solver weaving solution.

Assume that there are two caches that can be woven into an application, denoted $C1$ and $C2$. Furthermore, assume that there are two joinpoints that the caches can be applied to, denoted $J1$ and $J2$. Let there be a total of 200K of memory available to the caches. Furthermore, the two caches are mutually exclusive and cannot be applied to the same joinpoint. Let the time required to service a request at $J1$ be 10ms and the time at $J2$ be 12ms.

Each cache hit on $C1$ requires 2ms to service and each cache hit on $C2$ requires 3ms. All requests pass through both $J1$ and $J2$ and the goal is to optimally match the caches to joinpoints and set their sizes to minimize the total service time per request. The size of each cache, $C1_{size}$ and $C2_{size}$, determines the cache's hit ratio. For $C1$ the hit ratio is $C1_{size}/500$ and for $C2$ the hit ratio is $C2_{size}/700$. Let's assume that cache $C1$ is woven into joinpoint $J1$ and $C2$ is woven into joinpoint $J2$, the service time per request can be calculated as

$$SvcTime = 2(C1_{size}/500) + 10(1 - C1_{size}/500) + 3(C1_{size}/700) + 12(1 - C1_{size}/700)$$

With this formulation, we can derive the optimal sizes for the caches subject to the global weaving constraint:

$$C1_{size} + C2_{size} < 200$$

The problem, however, is that we want to know not only the optimal cache size but also where to weave the caches. The above formulation assumes that cache $C1$ is assigned to $J1$ and $C2$ to $J2$. Thus, instead we need to introduce variables into the service time calculation to represent the joinpoint that each cache is actually applied to so that we do not assume an architecture of how caches are applied to joinpoints. That is, we want to deduce not only the cache sizes but also the best allocation of caches to

joinpoints (the caching architecture). Let the variable M_{jk} have value 1 if the j_{th} cache C_j is matched to joinpoint J_k and 0 otherwise. We can update our service time formula so that it does not include a fixed assignment of caches to joinpoints:

$$SvcTime = 2(M_{11} * C1_{size}/500) + 3(M_{21} * C2_{size}/700) +$$
$$10(1 - ((M_{11} * C1_{size}/500) + (M_{21} * C2_{size}/700))) +$$
$$2(M_{12} * C1_{size}/500) + 3(M_{22} * C2_{size}/700) +$$
$$12(1 - ((M_{12} * C1_{size}/500) + (M_{22} * C2_{size}/700)))$$

The new formulation of the response time takes into account the different caches that could be deployed at each joinpoint. For example, the service time at joinpoint $J1$ is defined as:

$$J1SvcTime = 2(M_{11} * C1_{size}/500)$$
$$+3(M_{21} * C2_{size}/700) +$$
$$+10(1 - ((M_{11} * C1_{size}/500) + (M_{21} * C2_{size}/500)))$$

In this formulation the variables M_{11} and M_{21} are influencing the service time calcuation by determining if a specific cache's servicing information is included in the calculation. If the cache $C1$ is applied to $J1$, then $M_{11} = 1$ and the cache's service time is included in the calculation. If the cache is not woven into $J1$, then $M_{11} = 0$, which zeros out the effect of the cache at $J1$ since:

$$J1SvcTime = 2(0) \ldots 10(1 - (0 + (M_{21} * C2_{size}/500)))$$

Thus, by calculating the optimal values of the M_{ij} variables, we are also calculating the optimal way of assigning the caches (advice) to the joinpoints.

To optimally weave the caches into the application, we need to derive a set of values for the variables in the service time equation that minimizes its value. Furthermore, we must derive a solution that not only minimizes the above equation's value, but respects the constraints:

$$C1_{size} + C2_{size} < 200$$
$$(M_{11} = 1) \Rightarrow (M_{21} = 0)$$
$$(M_{12} = 1) \Rightarrow (M_{22} = 0)$$

because the cache sizes must add up to less than the alloted memory (200K) and both caches cannot be applied to the same joinpoint.

When the constraint solver is invoked on the CSP, the output will be the values for the M_{ij} variables. That is, for each Advice, i, and Joinpoint, j, the solver will output the value of the variable M_{ij}, which specifies if Advice, A_i, should be mapped to Joinpoint, B_j. The M_{ij} variables can be viewed as a table where the rows represent the advice elements, the columns represent the joinpoints, and the values (0 or 1) at each cell are the solver's solution as to whether or not a particular advice should be applied to a specific joinpoint. Furthermore, any variables that do not have values set, such as the cache sizes ($C1_{size}$ and $C2_{size}$), will have optimal values set by the constraint solver.

Even for this seemingly simple weaving problem, deriving what joinpoints the caches should be applied to and how big each cache should be is not easy to determine manually. However, by creating this formulation of the weaving problems as a CSP, we can use a constraint solver to automatically derive the optimal solution on our behalf. The solution that the solver creates will include not only the optimal cache sizes, but also which joinpoints each cache should be applied to.

A General Mapping of Weaving to a CSP: The previous subsection showed how a CSP could be used to solve a weaving problem involving optimization and global constraints. This section presents a generalized mapping from a weaving problem to a CSP so that the technique can be applied to arbitrary model weaving problems with global constraints.

We define a solution to a model weaving problem as a mapping of elements from an advice set α to a joinpoint set β that adheres to a set of constraints γ. To represent this mapping as a CSP, we create a table—called the *weaving table*—where for each advice A_i in α and joinpoint B_j in β, we define a cell (*i.e.*, a variable in the CSP) M_{ij}. If the advice A_i should be applied to the joinpoint B_j, then $M_{ij} = 1$ (meaning the table cell <i,j> has value 1). If A_i should not be applied to B_j, then $M_{ij} = 0$. The rules for building a weaving solution are described to the constraint solver as constraints over the M_{ij} variables. An example weaving table where the `ProductsCache` is applied to the `ProductDAO` is shown in Table 1.

	ProductDAO	ItemDAO
ProductsCache	$M_{00} = 1$	$M_{01} = 0$
FishCache	$M_{10} = 0$	$M_{11} = 0$

Table 1: An Example Weaving Table

Some weaving constraints are described purely in terms of the weaving table. For example, Challenge 1 from Section 3.2 introduced the constraint that the `FishCache` should only be used if the `ProductsCache` is not applied to any component. This constraint can be defined in terms of a constraint over the weaving table. If the `FishCache` is A_0 and the `ProductsCache` is A_1, then we can encode this constraint as for all joinpoints, j:

$$(\sum_{j=0}^{n} M_{0j} > 0) \rightarrow (\sum_{j=0}^{n} M_{1j} = 0)$$

Some examples of dependency constraints between advice elements that can be implemented as CSP constraints on the weaving table are:

$Advice_0$ requires $Advice_1$ to always be applied to the same joinpoint:

$$\forall B_j \subset \beta, (M_{0j} = 1) \rightarrow (M_{1j} = 1)$$

$Advice_0$ excludes $Advice_1$ from being applied to the same joinpoint:

$$\forall B_j \subset \beta, (M_{0j} = 1) \rightarrow (M_{1j} = 0)$$

$Advice_0$ requires between $MIN \ldots MAX$ of $Advice_1 \ldots Advice_k$ at the same joinpoint:

$$\forall B_j \subset \beta, (M_{0j} = 1) \rightarrow (\sum_{i=1}^{k} M_{ij} \geq MIN) \wedge (\sum_{i=1}^{k} M_{ij} \leq MAX)$$

Advice and Joinpoint Properties Tables: Other weaving constraints must take into account the properties of the advice and joinpoint elements and cannot be defined purely in terms of the weaving table. To incorporate constraints involving the properties of the advice and joinpoints, we create two additional tables: the *advice properties table* and *joinpoint properties table*. Each row P_i in the advice properties table represents the properties of the advice element A_i. The columns of the advice table represent the different property types. Thus, the cell <i,j>, represented by the variable PA_{ij}, contains A_i's value for the property associated with the j_{th} column. The joinpoint properties table is constructed in the same fashion with the rows being the joinpoints (*i.e.*, each cell is denoted by the variable PT_{ij}). An example joinpoint properties table is shown in Table 2.

	%Fish Requests	%Bird Requests
ProductDAO	65% ($PT_{00} = 0.65$)	20% ($PT_{01} = 0.2$)
ItemDAO	17% ($PT_{10} = 0.17$)	47% ($PT_{11} = 0.47$)

Table 2: An Example Joinpoint Properties Table

Challenge 1 from Section 3.2 introduced the constraint that the `FishCache` should only be applied to the `ProductDAO` if more than 50% (the majority) of the requests to the `ProductDAO` are for fish. We can use the advice and joinpoint properties tables to encode this request distribution constraint. Let the joinpoint properties table column at index 0 be associated with the property for the percentage of requests that are for Fish, as shown in the the joinpoint properties table shown in Table 2. Moreover, let A_1 be the `FishCache` and B_0 be the `ProductDAO`. The example request distribution constraint can be encoded as $M_{10} \rightarrow (PT_{00} > 50)$.

4.3 Global Constraints

In enterprise systems, global constraints are often needed to limit the amount of memory, bandwidth, or CPU consumed by a weaving solution. Global constraints can naturally be incorporated into the CSP model as constraints involving the entire set of variables in the weaving table. For example, the memory constraint on the total amount of RAM consumed by the caches, described in Challenge 1 from Section 3.2, can be specified as a constraint on the weaving and properties tables.

Let the joinpoint property table column at index 5, as shown in Table 3, represent the amount of free memory available on the hosting application server of each joinpoint. Moreover, let the advice property table column at index 4, as shown in Table 4, contain

	...	RAM on Application Server
ProductDAO	...	1024K ($PT_{05} = 1024$)
...	...	...

Table 3: An Example Joinpoint Properties Table with Available Memory

	...	RAM Consumed
ProductCache	...	400K ($PA_{04} = 400$)
FishCache	...	700K ($PA_{14} = 700$)

Table 4: An Example Advice Properties Table with RAM Consumption

the amount of memory consumed by each cache. The memory consumption constraint can be specified as:

$$\forall B_j \subset \beta, (\sum_{i=0}^{n} PA_{i4} * M_{ij}) < PT_{j5}$$

If an advice element is matched against a joinpoint, the corresponding M_{ij} variable is set to 1 and the advice element's memory consumption value, PA_{i4}, is added to the total consumed memory on the target application server. The constraint that the consumed memory be less than the available memory is captured by the stipulation that this sum be $< PT_{j5}$, which is the total amount of free memory available on the joinpoint's application server.

4.4 Joinpoint Feasibility Filtering with Regular Expressions and Queries

Some types of constraints, such as constraints that require matching strings against regular expressions, are more naturally represented using existing query and regular expression techniques. The CSP approach to model weaving can also incorporate these types of constraint expressions. Regular expressions, queries, and other pointcut expressions that do not have dependenices can be used as an initial filtering step to explicitly set zero values for some M_{ij} variables. The filtering step reduces the set of feasible joinpoints that the solver must consider when producing a weaving solution.

For example, the `FishCache` should only be applied to DAOs with the naming convention "Product*". This rule can be captured with an existing pointcut language and then checked against all possible joinpoints, as shown in Figure 7. For each joinpoint, j, that the pointcut does not match, the CSP variable, M_{ij}, for each advice element, i, associated with the pointcut is set to 0. Layering existing dependency-free pointcut languages as filters on top of the CSP based weaver can help to increase the number of labeled variables provided to the solver and thus reduce solving time.

4.5 CSP-Weaving Benefits

Challenge 3 from Section 3.4 showed the need for the ability to incorporate a weaving goal to produce an optimal weaving. Using a CSP model of a weaving problem, a weaving goal can be specified as a function over the M_{ij}, PA_{ij}, and PT_{ij} variables. Once

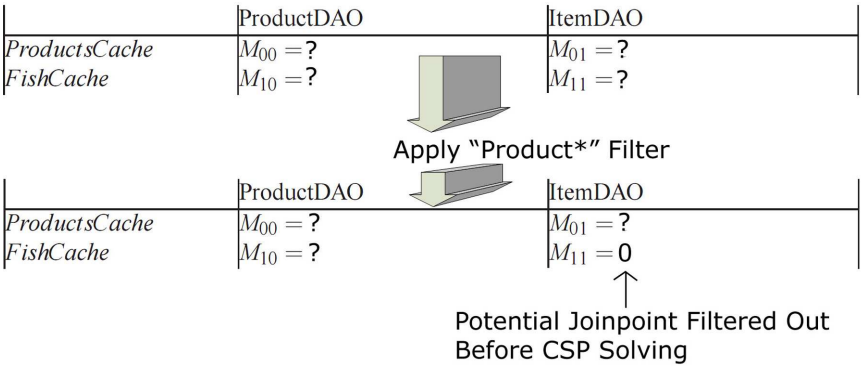

Fig. 7: Joinpoint Feasibility Filtering

the goal is defined in terms of these variables, the solver can be used to derive a weaving solution that maximizes the weaving goal. Moreover, the solver can set optimal values for attributes of the advice elements, such as cache size.

Allowing developers to specify optimization goals for the weaver enables different weaving solutions to be obtained that prioritize application concerns differently. For example, the same Pet Store solution models can be used to derive caching solutions that minimize response time at the expense of memory, balance response time and memory consumption, or minimize the response time of particular user actions, such as adding items to the shopping cart. To explore these various solution possibilities, developers update the optimization function provided to AspectScatter and not the entire weaving solution calculation process. With the manual optimization approaches required by existing model weavers, it is typically too time-consuming to evaluate multiple solution alternatives.

Mapping aspect weaving to a CSP and using a constraint solver to derive a weaving solution addresses Challenge 1 from Section 3.2. CSPs can naturally accomodate both dependency constraints and complex global constraints, such as resource or scheduling constraints. With existing model weaving approaches developers manually identify and document solutions to the global weaving constraints. With a CSP formulation of weaving, conversely, a constraint solver can perform this task automatically as part of the weaving process.

Manual approaches to create a weaving solution for a set of constraints have numerous points where errors can be introduced. When AspectScatter is used to derive a weaving solution, the correctness of the resulting solution is assured with respect to the weaving constraints. Moreover, in cases where there is no viable solution, AspectScatter will indicate that weaving is not possible.

A further benefit of mapping an aspect weaving problem to a CSP is that extensive prior research on CSPs can be applied to deriving aspect weaving solutions. Existing research includes different approaches to finding solutions [27], incorporating soft constraints [40], selecting optimal solutions or approximations in polynomial time [11, 18, 39], and handling conflicting constraints. Conflict resolution has been singled out in

model weaving research as a major challenge [49]. Numerous existing techniques for over-constrainted systems [10, 25, 46] (*i.e.*, CSPs with conflicting constraints), such as using higher-order constraints, can be applied by mapping model weaving to a CSP.

5 The AspectScatter DSL

Manually translating an aspect weaving problem into a CSP using the mapping presented in Section 4 is not ideal. A CSP model can accomodate global constraints and dependencies but requires a complex mapping that must be performed correctly to produce a valid solution. Working directly with the CSP variables to specify a weaving problem is akin to writing assembly code as opposed to Java or C++ code.

AspectScatter provides a textual DSL for specifying weaving problems and can automatically transform instances of the DSL into the equivalent CSP model for a constraint solver. AspectScatter's DSL allows developers to work at the advice/joinpoint level of abstraction and use leverage a CSP solver for deriving a weaving solution.

The CSP formulation of an aspect weaving problem is not specific to any one particular type of joinpoint or advice. The construction and solving of the CSP is a mathematical manipulation of symbols representing a set of joinpoints and advice. As such, the joinpoints could potentially be Java method invocations or model elements. In Section 6, we discuss how these symbols are translated into platform-specific joinpoints and advice. For this section, however, it is important to remember that we are only declaring and stating the symbols and constraints that are used to build the mathematical CSP weaving problem.

For example, in the context of the cache weaving example, there are two different types of platform-specific joinpoints. First, there are the joinpoints used by C-SAW, which are types of model elements in a GME model. Second, there are AspectJ type joinpoints, which are the invocation of various methods on the Java implementations of the `ProductDAO`, `OrderDAO`, etc. In the platform-independent model used by the CSP, the joinpoint definition `OrderDAO` is merely a symbolic definition of a joinpoint. When the platform-specific solution is translated into a platform-specific weaving solution, `OrderDAO` is mapped to a model element in the GME model used by C-SAW and an invocation of a query method on the Java implementation of the `OrderDAO`.

The basic format for an AspectScatter DSL instance is shown below:

```
ADVICE_1_ID
{
  (DIRECTIVE;)*
}
...
ADVICE_N_ID
{
  (DIRECTIVE;)*
}
JOINPOINT_1_ID
{
  (VARIABLENAME=EXPRESSION;)*
}
...
JOINPOINT_N_ID
{
  (VARIABLENAME=EXPRESSION;)*
}
```

The JOINPOINT declaration specifies a joinpoint, an element $B_j \subset \beta$, that ADVICE elements can be matched against. The JOINPOINT_ID is the identifier, such as "OrderDAO," that is given as a symbolic name for the joinpoint. Each JOINPOINT element contains one or more property declarations in the form of VARIABLENAME=EXPRESSION. The columns for the joinpoint properties table are created by traversing all of the JOINPOINT declarations and creating columns for the set of VARIABLENAMEs. The EXPRESSION that a JOINPOINT specifies for a VARIABLENAME becomes the entry for that JOINPOINT's row in the specified VARIABLENAME column, PT_{ij}.

Each ADVICE declaration specifies an advice element that can be matched against the set of JOINPOINT elements, an element $A_i \subset \alpha$. The DIRECTIVES within the advice element specify the constraints that must be upheld by the weaving solution produced by AspectScatter and the properties of the ADVICE element (values for the PA_{ij} variables). The directives available in AspectScatter are shown in Table 5.

As an example, the AspectScatter ADVICE definitions:

```
GenericCache
{
  Excludes:FishCache;
  DefineVar:CacheSize;
}
FishCache
{
}
```

defines two advice elements called GenericCache and FishCAche. The DIRECTIVEs within the GenericCache declaration (between "{..}") specify the constraints that must be upheld by the joinpoint it is associated with and the properties the advice element defines. The GenericCache excludes the advice element FishCache from being applied to the same joinpoint as the GenericCache. The GenericCache declaration also specifies a property variable, called CacheSize, that the weaver must determine a value for.

Assume that the GenericCache is A_2 and the FishCache is A_1. The AspectScatter specification would be transformed into: the mapping variables $M_{20} \ldots M_{2n}$, the advice property variables $PA_{20} \ldots PA_{2k}$, an advice property table column for CacheSize, and the CSP constraint $\forall B_j \subset \beta, (M_2 j = 1) \rightarrow (M_1 j = 0)$.

The final part of an AspectScatter DSL instance is an optional set of global variable definitions and an optimization goal. The global variable definitions are defined in an element named Globals. Within the Globals element, properties can be defined that are not specific to a single ADVICE or JOINPOINT. Furthermore, the Goal directive key word can be used within the Globals element to define the function that the constraint solver should attempt to maximize or minimize.

DIRECTIVE	Applied To	Description
Requires : *ADVICE*+	one or more other ADVICE elements	Ensures that all of the specified ADVICE elements are applied to a JOINPOINT if the enclosing ADVICE element is
Required : (*true*|*false*)	an ADVICE element	The enclosing ADVICE element must be applied to at least one JOINPOINT (if true).
Excludes : *ADVICE*+	one or more other ADVICE elements	Ensures that none of the specified ADVICE are applied to the same JOINPOINT as the enclosing ADVICE
Select : [*MIN*..*MAX*],*ADVICE*+	a cardinality expression and one or more other ADVICE	Ensures that at least MIN and at most MAX of the specified ADVICE are mapped to the same JOINPOINT as the enclosing ADVICE
Target : *CONSTRAINT*	an ADVICE element	Requires that CONSTRAINT hold true for the ADVICE and JOINPOINT's properties if the ADVICE is mapped to the JOINPOINT
Evaluate : (*ocl*|*groovy*), *FILTER_EXPRESSION*	an ADVICE element	Requires that FILTER_EXPRESSION defined in OCL or Groovy hold true for the ADVICE and JOINPOINT's properties if the ADVICE is mapped to the JOINPOINT
DefineVar : *VARIABLENAME* (= *EXPRESSION*)?	a weaving problem	Defines a variable. The final value for the variable is bound by the weaver and must cause the optional EXPRESSION to evaluate to true
Define : *VARIABLENAME* = *EXPRESSION*	a weaving problem	Defines a variable and sets a constant value for it
Goal : (*maximize*|*minimize*), *VARIABLE_EXPRESSION*	a weaving problem	Defines an expression over the properties of ADVICE and JOINPOINTS that should be maximized or minimized by the weaving

Table 5: AspectScatter DSL Directives

EXPRESSION	(*CONSTANT*|*VARIABLE_EXPRESSION*) (+|−|×) (*CONSTANT*|*VARIABLE_EXPRESSION*)	An expression
CONSTRAINT	(*VARIABLE_EXPRESSION*|*CONSTANT*) (<|>|=|!=|=<|>=) (*VARIABLE_EXPRESSION*|*CONSTANT*)	Defines a constraint that must hold true in the final weaving solution.
VARIABLE_EXPRESSION	(*VARIABLE_V_EXPRESSION*|*CONSTANT*) (+|−|×) (*VARIABLE_V_EXPRESSION*|*CONSTANT*)	An expression over a set of variables
VARIABLE_V_EXPRESSION	(Target|Source).VARIABLENAME	The value of the specified defined variable (VARIABLENAME) on a ADVICE or JOINPOINT element. *Target* specifies that the variable should be resolved against the JOINPOINT matched by the enclosing ADVICE. *Source* specifies that the variable should be resolved against the enclosing ADVICE element.

Table 6: AspectScatter DSL Expressions

The values for variables provided by the weaver are determined by labeling the CSP for the weaving problem. For example, the global constraints for the Pet Store weaving problem define the goal as the minimization of the response time of the `ItemDAO` and `ProductDAO`, as can be seen below:

```
Globals {
  Define:TotalFish = 100;
  Define:TotalBirds = 75;
  Define:TotalOtherAnimals = 19;
  Constraint:Sources.CacheSize.Sum < 1024;
  Goal:minimize, ProductDAO.RequestPercentage * ProductDAO.ResponseTime +
                          ItemDAO.RequestPercentage * ItemDAO.ResponseTime;
}
```

Each `Define` creates a variable in the CSP and sets its value. The variable created by the `Define` can then have a constraint bound to it. For example, a constraint could be created that leveraged the `TotalBirds` variable declared above. An example constraint might be $(\sum_{j=0}^{n} M_{0j} > 0) \rightarrow (TotalBirds < 80)$. This simple constraint states that the 0th advice element can only be applied to a joinpoint if there are less than 80 birds.

The `Constraint` directive adds a constraint to the CSP. In the example above, the specification adds a constraint that the sum of the cache sizes must be less than 1024. The statement "`Sources.CacheSize.Sum`" is a special AspectScatter language expression for obtaining a value from a properties table (the advice properties table), a column (`CacheSize`), and an operation (summation). Assuming `CacheSize` is the 0th column in the advice properties table, the statement adds the following constraint to the CSP:

$$\forall B_j \subset \beta, (\sum_{i=0}^{n} (M_{ij} * PA_{i0}) < 1024)$$

Since no explicit values for each advice element's `CacheSize` is set, these will be variables that the solver will need to find values for as part of the CSP solving process.

Because the response times of the DAOs are dependent on the size of each cache, the `CacheSize` variables will be set by the weaver to minimize response time. Developers can use the AspectScatter DSL to produce complex aspect weaving problems with both global constraints and goals.

AspectScatter's DSL also includes support for the filtering operations described in Section 4.4. Filters to restrict the potential joinpoints that an advice element can be mapped to can be defined using an Object Constraint Language (OCL) [47] or Groovy [26] language expression that must hold true for the advice/joinpoint mapping (*i.e.*, the choice of expression language is up to the user). Filters are defined via the `Evaluate` directive. For example, a Groovy constraint can be used to restrict the FishCache from being applied to any order related DAOs via a regular expression constraint:

```
FishCache {
 ...
 Evaluate:groovy,{!target.name.contains("Order")};
}
```

An OCL constraint could be used to further restrict the FishCache to only be applied to DAOs that receive requests from a category listing page:

```
FishCache {
 ...
 Evaluate:ocl,{target.requestsFrom->collect(x | x.name = 'ViewCategories.jsp')->size() > 0};
}
```

As described in Section 4.4, the filter expressions defined via `Evaluate` are used to preprocess the weaving CSP and eliminate unwanted advice/joinpoint combinations.

6 AspectScatter Model Transformation Language

The result of solving the CSP is a platform-independent weaving solution that symbolically defines which advice elements should be mapped to which joinpoints. This symbolic weaving solution still needs to be translated into a platform-specific weaving model, such as an AspectJ weaving specification. The platform-specific weaving specification can then be executed to perform the actual code or model weaving.

Each platform-independent weaving representation of the weaving solution can be transformed into multiple platform-specific weaving solutions, such as AspectJ, C-SAW, or AMW specific weaving specifications. Producing a platform-independent weaving model of the solution and transforming it into implementations for specific tools allows AspectScatter to eliminate much of the significant manual effort required to synchronize multiple weaving specifications across a diverse set of models, modeling languages, and modeling tools. For example, when the modeled request distribution changes for the Pet Store, the C-SAW, AspectJ, and GEMS weaving specifications can automatically be re-generated by AspectScatter, as shown in Step 4 of Figure 6.

AspectScatter's platform-independent weaving model can be transformed into a platform-specific model with a number of model transformation tools, such as ATL [28]. AspectScatter also includes a simple model transformation tool based on pointcut generation templates that can be used to create the platform-specific weaving model. In this

section, we show the use of the built-in transformation language in the context of the C-SAW weaving definition files needed for the GME model.

C-SAW weaves the caching specification into the GME architecture according to a set of weaving directives specified in a weaving definition file. The implementation of the C-SAW weaving definition file that is used to merge caches into the architecture model is produced from the platform-independent weaving solution model. To transform the platform-independent solution into a C-SAW weaving definition file, an AspectScatter model transformation is applied to the solution to create C-SAW *strategies* to update model elements with caches and C-SAW *aspects* to deduce the elements to which the strategies should be applied. For each cache inserted into the GME architecture model, two components must be added to the C-SAW weaving definition file. First, the *Strategy* for updating the GME model to include the cache and connect it to the correct component must be created, as shown below:

```
strategy ProductDAOAddGenericCache( ) {
    declare parentModel : model;
    declare component, cache : atom;
    parentModel := parent();
    component := self;
    cache := parentModel.addAtom("Cache", "GenericCacheForProductDAO");
    parentModel.addConnection("CacheInstallation",cache,component);
}
```

A root `Aspect` and `Strategy` must also be created that matches the root element of the GME model and invokes the weaving of the individual DAO caches. The root definitions are shown below:

```
aspect RootAspect()
{
  rootFolder().models()->AddCaches();
}
strategy AddCaches()
{
    declare parentModel : model;
    parentModel := self;
    parentModel.atoms("Component")->select(m|m.name() == "ProductDAO")->ProductDAOAddGenericCache ( );
    ....
}
```

For each advice/joinpoint combination, the `Strategy` to weave in the cache must be created. Moreover, for each advice/joinpoint combination, a weaving instruction must be added to the root `AddCaches` strategy to invoke the advice/joinpoint specific weaving strategy.

To create the advice/joinpoint specific cache weaving strategy, an AspectScatter template can be created, as follows:

```
#advice[*](for-each[list=targets]){#
strategy ${value}Add${advice}Cache( ) {
    declare parentModel : model;
    declare component, cache : atom;
    parentModel := parent();
    component := self;
    cache := parentModel.addAtom("Cache", "${advice}CacheFor${value}");
    parentModel.addConnection("CacheInstallation",cache,component);
}
#}#
```

The template defines that for all advice elements matched against joinpoints "*advice*[∗]", iterate over the joinpoints that each advice element is applied to "`for-each[list=targets]`", and create a copy of the template code between "{#" and "#}" for each target joinpoint. Moreover, each copy of the template has the name of the advice element and target element inserted into the placeholders "${advice}" and "${value}", respectively. The "${advice}" placeholder is filled with the symbolic name of the advice element from its `ADVICE` declaration in the AspectScatter DSL instance.

The "${value}" placeholder is the symbolic name of the joinpoint, also obtained from its definition in the AspectScatter DSL instance, that the advice element has been mapped to. The properties of an advice element can also be referred to using the placeholder "`${PROPERTYNAME}`." For example, the property `CacheSize` of the advice element could be referred to and inserted into the template by using the placeholder "${CacheSize}".

After deriving a weaving solution, AspectScatter uses the templates defined for C-SAW to produce the final weaving solution for the GME model. Invoking the generated C-SAW file inserts the caches into the appropriate points in the architecture diagram. A final woven Pet Store architecture diagram in GME can be seen in Figure 8.

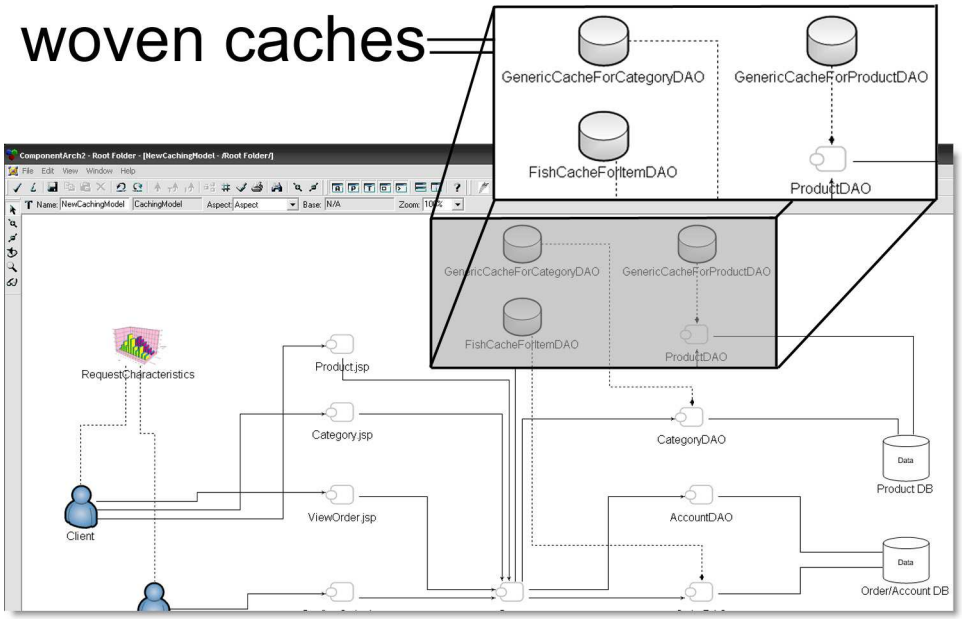

Fig. 8: The GME Architecture Model with Caches Woven in by C-SAW

With existing weaving approaches, each time the global properties, such as request distributions change, developers must manually derive a new weaving solution. When the properties of the solution models change, however, AspectScatter can automatically

solve for new weaving solutions, and then use model transformation to generate the platform-specific weaving implementations, thereby addressing Challenge 2 from Section 3.3. The CSP formulation of a weaving problem is based on the weaving constraints and not specific solution model properties. As long as the constraint relationships do not change, AspectScatter can automatically re-calculate the weaving solution and regenerate the weaving implementations. For example, if new request distributions are obtained, AspectScatter can re-calculate the weaving solution to accomodate the new information. Automatically updating the weaving solution as the solution model properties change can save substantial development effort across multiple solution model refactorings.

7 Applying Constraint-based Weaving to the Java Pet Store

This section demonstrates the reduction in manual effort and complexity achieved by applying AspectScatter to the Spring Java Pet Store to handle global constraints and generate platform-specific weaving implementations. For comparison, we also applied the existing weaving platforms C-SAW and AspectJ to the same code base using a manual weaving solution derivation process. The results document the manual effort required to derive and implement a caching solution for the Pet Store's `ItemDAO` and `ProductDAO`.

7.1 Manual Complexity Overview

It is difficult to directly compare the manual effort required to execute two different aspect weaving processes. The problem is that there is no way of correlating the relative difficulty of the individual tasks of each process. Furthermore, the relative difficulty of tasks may change depending on the developer.

Although it is difficult to quantify the relative difficulty of the individual steps, we can define functions $M(WP)$ and $M'(WP)$ to calculate the total number of manual steps required for each process as a function of the size of the weaving problem (WP) input. That is, as more advice elements, joinpoints, and constraints are added to the weaving problem, how is the number of manual steps of each process impacted? What we can show is that one process exhibits a better algorithmic O bound for the number of manual steps as a function of the input size.

Let's assume that each step in one process is E times harder than the steps in the second process. This gives the formula:

$$E * M_{step} = M'_{step}$$

Even if there is some unknown coefficient E, representing the extra effort of each step in the process yielding $M'(WP)$, if $M'(WP)$ posseses a better O bound, then there must exist an input, $wp_i \subset WP$ (WP is sorted in ascending order based on size), for which:

$$E * M'(wp_i) \leq M(wp_i)$$

and for all $wp_x \subset (wp_{i+1} \ldots wp_n)$:

$$E * M'(wp_x) < M(wp_x)$$

Once the size of the weaving problem reaches size wp_{i+1}, even though the steps in M' are E times more complicated than the steps in $M(WP)$, the faster rate of growth of the function $M(WP)$ makes it less efficient. If we can calculate O bounds for the number of manual steps required by each process as a function of the size of the weaving problem, then we can definitively show that for large enough problems, the process with the better O bound will be better.

In order to compare the AspectScatter based approach to our original C-SAW and AspectJ approach, we provide an example weaving problem involving global constraints and optimization. We apply each process to the problem to show the manual steps involved in the two processes. Next, we calculate functions $M(WP)$ and $M'(WP)$, for the traditional and AspectScatter processes respectively, and show that $M'(WP)$ exhibits a superior O bound.

7.2 Experimental Setup

We evaluated both the manual effort required to use the existing weaving solutions to implement a potentially non-optimal caching solution and the effort required to derive and implement a guaranteed optimal caching solution. By comparing the two different processes using existing weavers, we determined how much of the manual effort results from supporting multiple weaving platforms and how much results from the solution derivation process. Both processes with existing tools were then compared to a process using AspectScatter to evaluate the reduction in solution derivation complexity and solution implementation effort provided by AspectScatter.

7.3 Deriving and Implementing a Non-Optimal Caching Solution with Existing Weaving Techniques

The results for applying existing weavers to derive and implement a non-optimal caching solution are shown in Figure 9. Each individual manual set of steps is associated with an activity that corresponds to the process diagram shown in Figure 4. The results tables contain minimum and maximum values for the number of steps and lines of code. The implementation of each step is dependent on the solution chosen. The minimum value assumes that only a single cache is woven into the Pet Store, whereas the maximum value assumes every possible cache is used.

The top table in Figure 9 shows the effort required to produce the initial caching solution and implementation for the Pet Store. In the first two steps, developers identify and catalog the advice and joinpoint elements. Developers then pick a caching architecture (which may or may not be good or optimal) that will be used to produce a weaving solution. In the next three steps, developers must implement the weaving solution as a C-SAW weaving definition file. Finally, developers must update the Spring bean definition file with various directives to use AspectJ to weave the caches into the legacy Pet Store code base.

Existing Model Weaving Approach w/o Optimization					
Initial Implementation					
Activity	**Step**	**Min Lines of Code**	**Max Lines of Code**	**Min Steps**	**Max Steps**
Create Aspects				1	1
Identify/Define Joinpoints				1	1
Derive Caching Strategy				1	1
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	100	8	24
Refactoring for Request Distribution Change					
Derive New Caching Strategy				1	1
Implement Weaving Specification for C-SAW	Remove Unused AddCache Strategies	0	48	1	6
Implement Weaving Specification for C-SAW	Remove Unused AddCaches Strategy	0	6	1	1
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Remove Previous Proxies	0	22	1	2
Implement Weaving Specification for AspectJ	Remove Previous Cache Beans	0	18	1	6
Implement Weaving Specification for AspectJ	Remove Unused Cache Beans from ProductDAO/ItemDAO Methods	0	6	1	6
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	200	11	43

Fig. 9: Manual Effort Required for Using Existing Model Weaving Techniques Without Caching Optimization

The bottom table in Figure 9 documents the steps required to update the caching architecture and weaving implementation to incorporate a change in the distribution of request types to the Pet Store. In the first step, the developer derives a new caching architecture. In the next 12 steps, developers remove any caches from the original C-SAW and AspectJ implementations that are no longer used by the new solution and implement the new caching solution using C-SAW and AspectJ.

7.4 Deriving and Implementing an Optimal Caching Solution with Existing Weaving Techniques

Figure 10 presents the manual effort to derive and implement an optimal caching solution for the Pet Store using existing weavers. The change in this experiment is that it measures the manual effort required to derive an optimal solution for the Pet Store by calculating the Pet Store's response time using each potential caching architecture and choosing the optimal one. The steps for implementing the weaving solution are identical to those from the results presented in Figure 9.

The steps labeled *Derive Optimal Caching Strategy* in Figure 10 presents the manual optimal solution derivation effort incorporated into this result set. First, developers must enumerate and check the correctness according to the domain constraints, or each potential caching architecture for both the `ProductDAO` and `ItemDAO`. Developers must then enumerate and check the correctness of the overall caching architectures produced from each unique combination of `ProductDAO` and `ItemDAO` caching architectures. After determining the set of valid caching architectures, developers must use

Existing Model Weaving Approach w/ Optimization					
Initial Implementation					
Activity	**Step**	**Min Lines of Code**	**Max Lines of Code**	**Min Steps**	**Max Steps**
Create Aspects				1	1
Identify/Define Joinpoints				1	1
Derive Optimal Caching Strategy	Arch			19	115
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	100	26	138

Fig. 10: Manual Effort Required for Using Existing Model Weaving Techniques With Caching Optimization

the Pet Store's modeled request distribution, memory constraints, and response time goals to derive the optimal cache sizes and best possible response time of each caching architecture. Finally, developers select the optimal overall architecture and implement it using C-SAW and AspectJ.

As shown in Figure 11, refactoring the weaving solution to accomodate the solution model change in request type distributions forces developers to repeat the entire process. First, they must go back and perform the optimal solution derivation process again. After a new result is obtained, the existing solution implementations in C-SAW and AspectJ must be refactored to mirror the new caching structure.

Existing Model Weaving Approach w/ Optimization					
Refactoring for Request Distribution Change					
Activity	**Step**	**Min Lines of Code**	**Max Lines of Code**	**Min Steps**	**Max Steps**
Derive Optimal Caching Strategy				19	115
Implement Weaving Specification for C-SAW	Remove Unused AddCache Strategies	0	48	1	6
Implement Weaving Specification for C-SAW	Remove Unused AddCaches Strategy	0	6	1	1
Implement Weaving Specification for C-SAW	Create AddCache Strategies	8	48	1	6
Implement Weaving Specification for C-SAW	Create Root AddCaches Strategy	1	6	1	1
Implement Weaving Specification for AspectJ	Remove Previous Proxies	0	22	1	2
Implement Weaving Specification for AspectJ	Remove Previous Cache Beans	0	18	1	6
Implement Weaving Specification for AspectJ	Remove Unused Cache Beans from ProductDAO/ItemDAO Methods	0	6	1	6
Implement Weaving Specification for AspectJ	Add ProductDAO / ItemDAO Proxy	11	22	1	2
Implement Weaving Specification for AspectJ	Add Cache Beans	3	18	1	6
Implement Weaving Specification for AspectJ	Apply Cache Beans to ProductDAO/ItemDAO Methods	1	6	1	6
Totals		24	200	29	157

Fig. 11: Manual Effort Required for Using Existing Model Weaving Techniques to Refactor Optimal Caching Architecture

7.5 Deriving and Implementing an Optimal Caching Solution using AspectScatter

Figure 12 contains the steps required to accomplish both the initial implementation of the Pet Store caching solution and the refactoring cost when the request distribution

Aspect Scatter					
Initial Implementation					
Activity	**Step**	**Min Lines of Code**	**Max Lines of Code**	**Min Steps**	**Max Steps**
Create Aspects		12	12	6	6
Identify/Define Joinpoints		12	12	2	2
Derive Optimal Caching Strategy	Define Weaving Goal	1	1	1	1
Implement Weaving Specification for C-SAW	Create AddCache Model Transformation	8	8	1	1
Implement Weaving Specification for C-SAW	Create Root AddCaches Model Transformation	6	6	1	1
Implement Weaving Specification for AspectJ	Create ProductDAO / ItemDAO Proxy Model Transformation	22	22	2	2
Implement Weaving Specification for AspectJ	Create Cache Beans Model Transformation	18	18	6	6
Implement Weaving Specification for AspectJ	Create Cache Beans to ProductDAO/ItemDAO Methods Model Transformation	1	1	1	1
Implement Weaving Specification	Invoke AspectScatter	1	1	1	1
Totals		81	81	21	21
Refactoring for Request Distribution Change					
Identify/Define Joinpoints	Update Request Distribution Properties	1	2	1	2
Implement Weaving Specification	Invoke AspectScatter	1	1	1	1
Totals		2	3	2	3

Fig. 12: Manual Effort Required for Using AspectScatter With Caching Optimization

changes. In steps 1 and 2, developers use AspectScatter's DSL to specify the caches, joinpoints, and constraints for the weaving problem. Developers then define the weaving goal, the response time of the application in terms of the properties of the joinpoints and advice elements woven into a solution. The goal is later used by AspectScatter to ensure that the derived weaving solution is optimal.

The next two steps (3 and 4) require the developer to create a model transformation, using AspectScatter's transformation templates, as described in Section 6, to specify how to transform the platform-independent weaving solution into a C-SAW implementation. The approach thus represents a higher-order transformation where C-SAW transformations are generated from more abstract transformation rules. The subsequent three steps define a model transformation to produce the AspectJ implementation. Finally, AspectScatter is invoked to deduce the optimal solution and generate the C-SAW and AspectJ implementations.

The bottom of Figure 12 presents the steps required to refactor the solution to accomodate the change in request distributions. Once the aspect weaving problem is defined using AspectScatter's DSL, the change in request distributions requires updating one or both of the request distribution properties of the two joinpoints (*i.e.*, the `ProductDAO` and `ItemDAO`) in the AspectScatter DSL instance. After the properties are updated, AspectScatter is invoked to recalculate the optimal caching architecture and regenerate the C-SAW and AspectJ implementations using the previously defined model transformations.

7.6 Results Analysis and Comparison of Techniques

By comparing the initial number of lines of code (shown in Figures 9-12) required to implement the caching solution using each of the three techniques, the initial cost of

defining an AspectScatter problem and solution model transformations can be derived. AspectScatter initially requires 81 lines of code versus between 24 and 100 for the approach based on existing techniques. The number of lines of code required to implement the initial weaving specification grows at a rate of $O(n)$, where n is the number of advice and joinpoint specifications, for both AspectScatter and existing approaches. The more advice and joinpoint specifications, the larger each weaving specification.

The benefit of AspectScatter's use of model transformations becomes most apparent by comparing the refactoring results. AspectScatter only requires the developer to change between 1-2 lines of code before invoking AspectScatter to regenerate the C-SAW and AspectJ implementations. Using the existing weaving approaches, the developer must change between 24-200 lines of code. Moreover, this manual effort required by the existing approaches is incurred *per solution model change*. Thus, AspectScatter requires a constant or $O(1)$ number of changes per refactoring while existing approaches require $O(n)$ changes per refactoring.

For a single aspect weaving problem without optimization that is implemented and solved exactly once, both AspectScatter and the manual weaving approach exhibit roughly $O(n)$ growth in lines of code with respect to the size of the weaving problem. The more caches that need to be woven, the larger the weaving specifications have to be for both processes. For a *single weaving* in this scenario, we cannot directly show that AspectScatter provides an improvement since it has an equivalent big O bound.

If we calculate the weaving cost over K refactorings, however, we see that AspectScatter exhibits a bound of $O(2K+n) = O(K+n)$ lines of code. ApsectScatter requires an initial setup cost of $O(n)$ lines of code and then each of the K refactorings requires manually changing 1-2 lines of code. The manual approach requires $O(n)$ lines of code changes for each of the K refactorings because the developer may have to completely rewrite all of the joinpoint specifications. Over K refactorings, the manual process requires $O(Kn+n) = O(Kn)$ lines of code changes. Thus, AspectScatter provides a better bound, $O(K+n) < O(Kn)$ on the rate of growth of the lines of code changed over multiple refactorings.

When optimization is added to the scenarios, AspectScatter's reduction in manual complexity becomes much more pronounced. With existing approaches, each time the weaving solution is implemented, the developer must calculate the optimal cache weaving architecture. Let γ be the number of manual steps required to calculate the optimal cache weaving architecture, then the cost of implementing the initial weaving solution with an existing approach is $O(n+\gamma)$. The developer must implement the $O(n)$ lines of code for the weaving specification and derive the optimal architecture.

Since we are doing a big O analysis, we will ignore any coefficients or differences in difficulty between a step to implement a line of code and a step in the derivation of the optimal caching architecture. We will say that n lines of code require n manual steps to implement. The next question is how the number of steps γ grow as a function of the size of the weaving problem. The caching optimization problem with constraints is an instance of a mixed integer optimization problem, which is in NP, and thus has roughly exponential complexity. Thus, $\gamma = \theta^n$, where θ is a constant

The overall complexity of the existing approach for the optimization scenario is $O(n+\theta^n)$. Note, this complexity bound is for solving a single instance of the weaving

problem. Over K refactorings, the complexity bound is even worse at $O(n+K(n+\theta^n))$. With AspectScatter, the solver performs the optimization step on the developer's behalf and the θ^n manual steps are eliminated. When optimization is included and K refactorings are performed, AspectScatter shows a significantly better bound on manual complexity than existing approaches:

$$O(n+K) < O(n+K(n+\theta^n))$$

One might argue that a developer would not manually derive the optimal caching architecture by hand but would instead use some automated tool. We note, however, that this is essentially arguing for our approach, since we are using an external tool to derive the caching architecture and then using code generation to automatically implement the solution. Thus, even using an external tool would still require a developer to rewrite the weaving specification after each refactoring and would also add setup cost for specifying the weaving problem for the external tool and translating the results back into a weaving solution. Our approach automates all of these steps on behalf of the developer.

A final analysis to consider is the effect of the number of weaving platforms on the complexity of the weaving process. For both processes, the overhead of the initial setup of the weaving solution is linearly dependent on the number of weaving platforms used. In the experiments, AspectJ and C-SAW are used as the weaving platforms. Given P weaving platforms, both processes exhibit an initial setup complexity of $O(Pn)$.

With existing processes, when K refactorings are performed, the number of weaving platforms impacts the complexity of each refactoring. Rather than simply incurring $O(n)$ complexity for each refactoring, developers incur $O(Pn)$ per refactoring. This leads to an overall complexity bound of $O(Pn+KPn)$ for existing processes versus a bound of $O(Pn+K)$ for AspectScatter. As we showed in the previous analyses, even for a single weaving platform, such as AspectJ, AspectScatter reduces complexity. However, when numerous weaving platforms are used AspectScatter shows an even further reduction in complexity.

7.7 Weaving Performance

There is no definitive rule to predict the time required to solve an arbitrary CSP. The solution time is dependent on the types of constraints, the number of variables, the degree of optimality required, and the initial variable values provided to the solver. Furthermore, internally, the algorithms used by the solver and solver's implementation language can also significantly affect performance.

Our experience with AspectScatter indicated that the weaving process usually takes 10ms to a few seconds. For example, to solve a weaving problem involving the optimal weaving of 6 caches that can be woven into any of 10 different components with fairly tight memory constraints requires approximately 120ms on an Intel Core 2 Duo processor with 2 gigabytes of memory. If a correct—but not necessarily optimal solution is needed—the solving time is roughly 22ms. Doubling the available cache memory budget essentially halves the optimal solution derivation time to 64ms. The same problem expanded to 12 caches and 10 components requires a range from 94ms to 2,302ms depending on the tightness (i.e., amount of slack memory) of the resource constraints.

In practice, we found that AspectScatter quickly solves most weaving problems. It is easy to produce synthetic modeling problems with poor performance, but realistic model weaving examples usually have relatively limited variability in the weaving process. For example, although a caching aspect could theoretically be applied to any component in an application, this behavior is rarely desired. Instead, developers normally have numerous functional and other constraints that bound the solution space significantly. In the Pet Store, for example, we restrict caching to the four key DAOs that form the core of the middle-tier.

In cases where developers do encounter a poorly performing problem instance, there are a number of potential courses of action to remedy the situation. One approach is to relax the constraints, *e.g.*, allow the caches to use more memory. Developers can also improve solving speed by accepting less optimal solutions, *e.g.*, solving for a cache architecture that produces an average response time below a certain threshold rather than an optimal response time. Finally, developers can try algorithmic changes, such as using different solution space search algorithms, *e.g.*, simulated annealing [39], greedy randomized adaptive search [39], and genetic algorithms [39].

8 Related Work

This section compares our research on AspectScatter to related work. Section 8.1 compares and constrasts AspectScatter to other model weavers. Section 8.2 compares the CSP-based model weaving approach to other aspect-oriented modeling techniques. Finally, Section 8.3 compares AspectScatter to other approaches for incorporating application requirements into aspect-oriented modeling.

8.1 Model Weaving

Reddy et al. [38] propose a technique that uses model element signatures and composition directives to perform model weaving. Reddy's approach focuses on different challenges of model weaving and is complementary to the constraint-based weaving approach used by AspectScatter. AspectScatter focuses on incorporating and automating the solution and optimization of global weaving constraints. Reddy's approach, however, is targeted towards the difficulties of identifying joinpoints and correctly modifying the structure of a model to perform a merger. First, model element signatures can be incorporated as a CSP filtering step, as described in Section 4.4. Second, the composition directives developed by Reddy can be used to implement the platform-specific weaving model produced by AspectScatter. In contrast, AspectScatter can derive and optimize the global weaving solution, which Reddy's techniques are not designed to do.

Cottenier et al. [14] have developed a model weaver called the Motorola WEAVR. The WEAVR provides complex weaving and aspect visualization capabilities for models. Although WEAVR has numerous capabilities, it is designed for a different part of the model weaving process than AspectScatter. AspectScatter sits above multiple weaving platforms to manage the overall global weaving solution. Motorola WEAVR, in contrast, is a specific weaving platform used to merge models and visualize model

weaving results. The two tools are synergistic. Motoroal WEAVR is a weaving platform that provides numerous analytic and modeling capabilities. AspectScatter is a high-level weaver that can be used to produce weaving specifications for WEAVR. Furthermore, WEAVR is not designed to model and solve the complex global constraints that represent the strength of AspectScatter.

8.2 Aspect-Oriented Modeling

Lahire et al. [29] motivate the need for and describe a potential solution for incorporating variability into AOM. Their work motivates some of the challenges addressed in this paper, namely the challenge of managing variability in how advice can be applied to joinpoints. AspectScatter offers an implementation of a solver designed to: (1) handle the solution variability presented by Lahire et al., (2) incorporate global constraints to ensure that individual variable solution weaving decisions produce an overall correct solution, and (3) optimally choose values for points of variability when multiple solutions are possible. Lahire et al. initially explore and describe a potential solution for capturing and handling AOM variability. AspectScatter provides a concrete approach to handling numerous facets described by Lahire et al.

Morin et al. [36] have also developed a generic model of aspect-oriented modeling. Their technique generalizes joinpoints to model snippets and pointcuts to model templates. AspectScatter also adopts a generalized view of pointcuts and joinpoints. AspectScatter provides global weaving constraints and optimization, whereas the techniques developed by Morin et al. are for situations where there is no ambiguity in which potential joinpoints a template should be matched against. AspectScatter automates part of the weaving design process, the derivation of the global weaving solution, whereas Morin et al. propose techniques to generically model how a weaving solution is applied. Each technique is geared towards a different phase of the weaving process. AspectScatter solves the weaving solution derivation challenges and Morin et al.'s techniques address the platform-specific weaving solution implementation.

8.3 Models and Constraints

Lengyel et al. [33] present a technique for validating the correctness of model transformations by tying constraints to transformation rules. Lengyel's technique provides a method for identifying cross-cutting constraints and refactoring them into aspects. These techniques for capturing transformation constraints as aspects is complementary to AspectScatter. Whereas Lengyel's techniques are designed to help maintain the correctness of model transformations, AspectScatter is designed to automatically maintain the correctness of model weaving. Moreover, AspectScatter is designed to derive solutions to constraints but Lengyel's techniques are for checking constraints and identifying aspects. Lengyel's techniques could be used to help guarantee the correctness of the transformations that AspectScatter uses to produce the platform-specific weaving implementations.

Baniassad et al. [7] have developed an approach to help identify aspects in designs and trace the relationship between aspects and requirements. Their approach is related

to AspectScatter's incorporation of global system requirements and goals into the aspect weaving specification. Baniassad et al.'s techniques help to identify and trace the aspects and their relationship with requirements whereas AspectScatter is designed to capture and *solve* requirements guiding the placement of aspects into a system. Thus, although the approaches are both related to understanding and managing how requirements affect aspects, the challenges that Baniassad et al. address (*i.e.*, identification and tracing of aspects) are different than AspectScatter's (*i.e.*, capture and solving of weaving requirements and goals).

9 Concluding Remarks

A significant amount of manual effort is incurred by the inability to encode the global application requirements into the model weaving specification and honor them during the weaving process. This gap in existing model weavers encourages developers to manually derive and maintain solutions to the global weaving constraints as the underlying solution models evolve. Moreover, developers may need to implement the global weaving solution in the pointcut languages of multiple model weavers.

This paper describes how providing a model weaver with knowledge of the entire set of joinpoints used during the weaving process ahead of time makes it possible to map model weaving to a CSP and use a constraint solver to derive a weaving that can incorporate global, dependency, and expression-based constraints. From our experience using AspectScatter's approach of mapping model weaving to a CSP, we have learned that CSP-based model weaving reduces manual effort by:

1. Capturing and allowing the weaver to solve the global application constraints required to produce a weaving solution
2. Informing the weaver of the overall solution goals so that the weaver can derive the best overall weaving solution with respect to a cost function and
3. Encoding using model transformations to automatically generate implementations of the global weaving solution for each required weaving platform.

By capturing and leveraging this critical set of domain knowledge, AspectScatter can automate the complex process of deriving weaving solutions and maintaining them as solution models change. By applying Aspect Scatter to the Java Pet Store case study, we showed that the CSP-based weaving approach scaled significantly better than existing approaches in terms of the number of manual weaving steps. Although this paper has focused on cache weaving, the same techniques could be applied to other domains, such as optimally configuring applications for mobile devices.

AspectScatter is an open-source tool available from http://www.eclipse.org/gmt/gems.

10 Acknowledgements

This work was supported in part by the National Science Foundation under NSF CAREER CCF-0643725.

References

1. Apache Foundation's JMeter, http://jmeter.apache.org.
2. AspectJ, http://www.eclipse.org/aspectj/.
3. HyperJ, http://www.alphaworks.ibm.com/tech/hyperj.
4. .NET Pet Store, http://msdn2.microsoft.com/en-us/library/ms978487.aspx.
5. Sun Microsystem's Java Pet Store Sample Application, http://java.sun.com/developer/releases/petstore/.
6. The Spring Framework, http://www.springframework.org/about.
7. E. Baniassad and S. Clarke. Theme: an Approach to Aspect-oriented Analysis and Design. In *Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Scotland, UK, May 2004.
8. J. Bézivin. From Object Composition to Model Transformation with the MDA. In *Proceedings of TOOLS*, pages 350–354, Santa Barbara, CA, USA, August 2001.
9. J. Bézivin, F. Jouault, and P. Valduriez. First Experiments with a ModelWeaver. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development Workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, March 2004.
10. S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison. *Over-Constrained Systems*, 1106:111–150, 1996.
11. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-Based Constraint Satisfaction and Optimization. *Journal of the ACM*, 44(2):201–236, 1997.
12. F. Budinsky. *Eclipse Modeling Framework*. Addison-Wesley Professional, New York, NY, USA, 2003.
13. J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, 1990.
14. T. Cottenier, A. van den Berg, and T. Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *Proceedings of the International Conference on Aspect-Oriented Software Development, Industry Track*, Vancouver, Canada, March 2006.
15. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
16. M. Del Fabro, J. Bézivin, and P. Valduriez. Weaving Models with the Eclipse AMW plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe*, Esslingen, Germany, October 2006.
17. T. Elrad, O. Aldawud, and A. Bader. Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In *Generative Programming and Component Engineering (GPCE)*, pages 189–201, Pittsburgh, PA, USA, October 2005.
18. R. Fletcher. *Practical Methods of Optimization*. Wiley-Interscience, New York, NY, USA, 1987.
19. R. France, I. Ray, G. Georg, and S. Ghosh. An Aspect-Oriented Approach to Early Design Modeling. *IEE Proceedings-Software*, 151(4):173–185, 2004.
20. H. Gomaa. *Designing Concurrent, Distributed, and Real-time Applications with UML*. Addison-Wesley, Reading, MA, USA, 2000.
21. J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-specific Modeling. *Communications of the ACM*, 44(10):87–93, 2001.
22. J. Hannemann, G. Murphy, and G. Kiczales. Role-based Refactoring of Crosscutting Concerns. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 135–146, Chicago, Illinois, USA, March 2005.

23. E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 26–35, Lancaster, UK, March 2004.
24. S. Holzner. *Ant: The Definitive Guide*. O'Reilly, Sebastopol, CA, USA, 2005.
25. M. Jampel, E. Freuder, and M. Maher. *Over-Constrained Systems*. Springer-Verlag London, UK, 1996.
26. D. König, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning Publications, 2007.
27. V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.
28. I. Kurtev, K. van den Berg, and F. Jouault. Rule-based Modularization in Model Transformation Languages Illustrated with ATL. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1202–1209, Dijon, France, April 2006.
29. P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J.-M. Jézéquel. Introducing variability into Aspect-Oriented Modeling Approaches. In *In Procedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, Nashville, TN, USA, October 2007.
30. A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-specific Design Environments. *Computer*, 34(11):44–51, 2001.
31. W. Li, W. Hsiung, D. Kalshnikov, R. Sion, O. Po, D. Agrawal, and K. Candan. Issues and Evaluations of Caching Solutions for Web Application Acceleration. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 1019–1030, Hong Kong, China, August 2002.
32. Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay, and J. Naughton. Middle-tier Database Caching for E-business. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 600–611, Madison, Wisconsin, June 2002.
33. H. C. László Lengyel, Tihamér Levendovszky. Identification of Crosscutting Concerns in Constraint-Driven Validated Model Transformations. In *Proceedings of the Third Workshop on Models and Aspects at ECOOP 2007*, Berlin, Germany, July 2007.
34. 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, San Diego, CA, USA, 2003.
35. C. Mohan. Caching Technologies for Web Applications. In *Proceedings of the 27th International Conference on Very Large Data Bases*, page 726, Rome, Italy, September 2001.
36. B. Morin, O. Barais, J.-M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *Models and Aspects Workshop, at ECOOP 2007*, Berlin, Germany, July 2007.
37. J. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7(4):308–313, 1965.
38. Y. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. *Transactions on Aspect-Oriented Software Development*, 3880:75–105, 2006.
39. C. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Inc., New York, NY, USA, 1993.
40. T. Schiex. Possibilistic Constraint Satisfaction Problems or How to Handle Soft Constraints. In *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 268–275, San Mateo, CA, USA, 1992.
41. S. Shavor, J. D'Anjou, P. McCarthy, J. Kellerman, and S. Fairbrother. *The Java Developer's Guide to Eclipse*. Pearson Education, Upper Saddle River, NJ, USA, 2003.

42. Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). www.cis.uab.edu/gray/research/C-SAW, University of Alabama at Birmingham, Birmingham, AL.
43. T. Valesky. *Enterprise JavaBeans*. Addison-Wesley, Reading, MA, USA, 1999.
44. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press Cambridge, MA, USA, 1989.
45. M. Voelter, I. Groher, and G. Heidenheim. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th International Software Product Line Conference*, pages 233–242, Kyoto, Japan, September 2007.
46. R. Wallace and E. Freuder. Heuristic Methods for Over-constrained Constraint Satisfaction Problems. *Over-Constrained Systems*, 1106:207–216, 1996.
47. J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, Reading, MA, USA, 2003.
48. J. White, D. C. Schmidt, and S. Mulligan. The Generic Eclipse Modeling System. In *Proceedings of the Model-Driven Development Tool Implementors Forum at TOOLS 2007*, Zurich, Switzerland, June 2007.
49. J. Zhang, T. Cottenier, A. van den Berg, and J. Gray. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 6(7).