Automating Deployment Planning with an Aspect Weaver

Jules White and Douglas C. Schmidt

Vanderbilt University

{jules,schmidt}@dre.vanderbilt.edu

Abstract

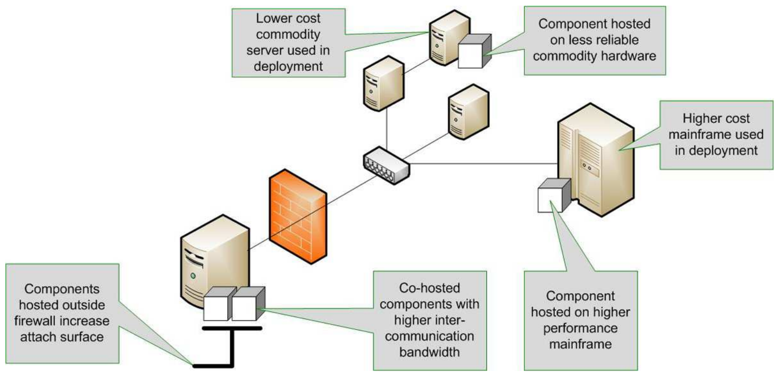
Deployment has emerged as a major challenge in distributed real-time and embedded (DRE) systems. Application deployment planners must integrate numerous functional and non-functional constraints, such as security and performance, to produce correct deployment plans. The numerous deployment constraints and their complex interactions make manually deducing correct/efficient deployments hard.

This paper presents four contributions to the study of automated deployment processes. First, it shows that a deployment planner and an aspect weaver accomplish the same abstract problem—*i.e.*, mapping items from a source set (advice or components) to items in a target set (joinpoints or nodes) according to a set of rules—and uses this abstract definition of deployment planning to automate it with an aspect weaver. Second, this paper describes how the ScatterML domain-specific aspect language incorporates complex global constraints for specifying deployment pointcuts. Third, we show how aspect weaving can be reduced to a constraint satisfaction problem and a constraint solver used to deduce a correct weaving. Fourth, we show that phrasing weaving as a constraint satisfaction problem and automating deployment through a constraint solver based weaver yields several key benefits, ranging from guaranteed deployment plan correctness to bounds on worst case solution quality.

1 Introduction

Deployment planning is the process of building an architecture to associate software components with computational nodes. This process has emerged as a key challenge in both modern enterprise and distributed real-time and embedded (DRE) systems [11]. Enterprise and DRE systems are increasing developed using component-based software that allows the assembly and deployment of reusable software artifacts into distributed applications. The intricate details of communicating between components is encapsulated by the middleware layers underlying the component-based application. By automatically handling the low-level communication details of wiring components together, middleware allows developers to deploy their components onto multiple physical nodes and in many different configurations without changing application functionality or rewriting code.

A challenging dimension of deployment, however, is that certain non-functional concerns, such as performance and security, pull the deployment solution (deployment plan) in different directions, as shown in Figure 1. For example, given



Lower cost commodity server used in deployment
Component hosted on less reliable commodity hardware
Higher cost mainframe used in deployment
Components hosted outside firewall increase attach surface
Co-hosted components with higher inter-communication bandwidth
Component hosted on higher performance mainframe

Figure 1: Complex Opposing Concerns in Deployment Planning

two components to deploy, if one component must be placed in a de-militarized zone outside the protection of a firewall and the other component is collocated with it, the application's throughput may improve at the cost of increasing its attack surface [24]. In this example, deploying to meet performance concerns can compromise security. As shown in Section 3, it is hard to manually deploy a group of components that meets a complex set of non-functional requirements.

A key goal of an aspect weaver is to map items from a source set (the set of crosscutting advice) to elements in a target set (the set of joinpoints in an application) [2, 21]. The aspect weaver produces this mapping via a series of rules (*e.g.*, patterns matching the names of invoked methods) that determine which source item (advice) should be mapped to which target item (joinpoint). This paper shows how the problem of matching items from a source set to a target set appears in the domain of deployment planning.

In most aspect weaving domains, the aspect weaver must map the source items to an extremely large set of target items (the numerous method and constructor calls in an application). Moreover, applications often use late-binding and the aspect weaver does not know the exact items in the target set until runtime. Finally, the aspect weaver may never know the entire set of target items, only the items that have been encountered (*e.g.*, the set of methods, etc. that have been invoked). Since the aspect weaver does not know the contents (and may never know them) before runtime, it is extremely hard to honor global constraints and nearly impossible to perform optimization. Most weavers use patterns, such as regular expressions, to match advice to joinpoints on the fly.

This paper shows how an aspect weaver can be used as a deployment planner by building a *domain-specific aspect language* (DSAL) [27, 13, 28] for specifying deployment rules as pointcuts. Existing pointcut languages, however, are not well-suited for specifying the complex global constraints, such as resource constraints, that are typical in deployment since they are designed for mapping advice to an indefinite set of joinpoints. In the domain of deployment planning, there is a known and finite set of target joinpoints.

This paper shows how aspect weaving can be mapped to a *constraint satisfaction problem* (CSP) [8, 30, 25] and how a constraint solver can be used to derive a correct—and in some cases optimal—association of advice to joinpoints. We have used this CSP reduction of weaving to create a constraint solver-based aspect weaver, called Scatter, that can incorporate complex global constraints. Scatter takes an instance of a DSAL for specifying crosscutting deployment concerns as input and outputs a deployment plan that is guaranteed to be correct with respect to both global and local deployment rules (and may be optimal).

Building an aspect weaver based on a constraint solver is hard since it requires performing a complex transformation of the pointcuts into a format, such as a system of linear equations, used by a constraint solver. A key attribute of our solution that simplifies its use by domain-experts is our DSAL, called ScatterML, that can specify pointcuts and overall global weaving constraints. We have also developed a series of model transformations that reformulate ScatterML pointcuts as a CSP. ScatterML helps bridge the semantic gap between the existing pointcut terminology familiar to domain experts and the CSP notation of a constraint solver. In particular, ScatterML provides a general-purpose set of language primitives that allow it to capture a wide variety of pointcut types.

The solution presented in this paper significantly extends our previous work developing constraint solver weaving and configuration solutions, which focused on techniques and tools for deploying software components to electronic control units (ECUs) in automobiles [26, 32], Enterprise Java Beans to application servers [31], and Java 2 MicroEdition (J2ME) components to mobile devices [33]. The experience we gained from developing model-driven and aspect-oriented deployment tools for these domains enabled us to identify the similarities between aspect weaving and deployment planning. We have leveraged these similarities to model deployment constraints as pointcuts and automate deployment planning with a constraint solver-based aspect weaver. This paper generalizes our prior experience and shows

- The challenges of developing a DSAL for specifying deployment concerns,
- How the ScatterML DSAL addresses these language design challenges,
- The challenges of weaving using an instance of ScatterML,
- How aspect weaving can be mapped tod to a CSP, and
- How our weaving engine uses a constraint solver to tackle the combinatorial problems of aspect weaving when global constraints are present.

Throughout the paper, we refer to our constraint-based weaver as Scatter and our DSAL as ScatterML.

The remainder of this paper is organized as follows: Section 2 shows how aspect weaving and deployment planning can be mapped to the same abstract problem; Section 3 describes the challenges of developing a DSAL to specify crosscutting deployment concerns and how ScatterML addresses them; Section 4 analyzes empirical results that compare a manual deployment planning process to a Scatter-based planning process; Section 5 compares our work on ScatterML with related research; and Section 6 presents concluding remarks.

2 Aspect Weaving for Deployment

This section evaluates the process of aspect weaving [2] and shows how the process of associating advice with joinpoints is similar to deriving a deployment of components to nodes (a deployment plan). By describing the similarities between the two processes we show how deployment planning can be automated with an aspect weaver, which enables developers to leverage aspect-oriented programming [22] techniques to simplify deployment planning. The section also motivates why conventional pointcut languages are not sufficient to specify all the rules needed to guide a deployment planning process.

2.1 Comparing Deployment to Aspect Weaving

Aspect-oriented programming (AOP) associates reusable crosscutting software logic (advice) with joinpoints (execution points in a program), as shown on the right-hand side of Figure 2). Advice is associated with a joinpoint through

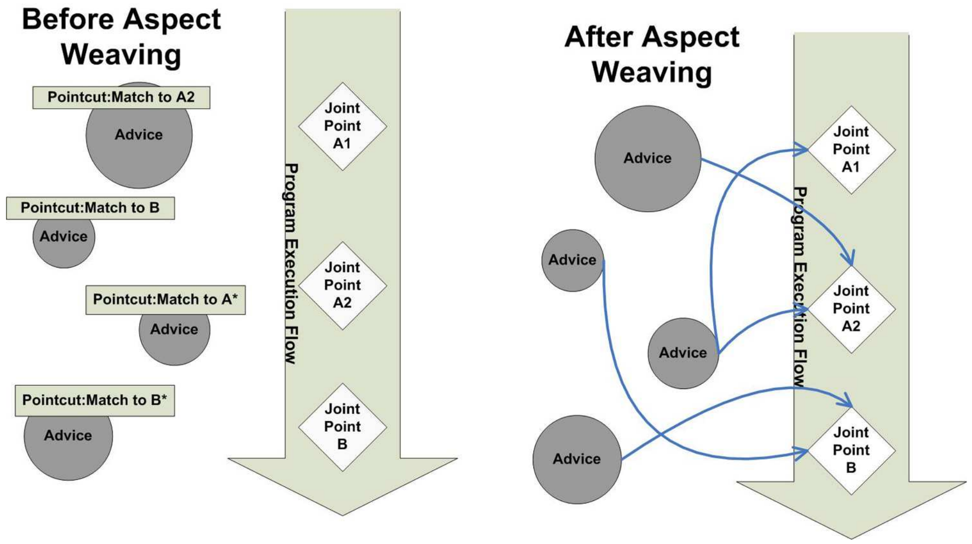


Figure 2: Overview of Aspect Weaving

a pointcut specification (left-hand side of Figure 2). The pointcut specifies rules or patterns to match a set of advice to joinpoints. The goal of an aspect weaver is to merge the pointcut specification, advice, and application code into an executable program that invokes the correct advice (based on the pointcut specifications) when it reaches a given joinpoint.

At an abstract level, the goal of an aspect weaver is to take two sets (the set of advice and the set of joinpoints) and produce a mapping of elements from the source set (the advice) to elements in the target set (the joinpoints). Developers guide the aspect weaver by providing it with a set of rules to determine if an element in the source set can be associated with an element in the target set. The association rules are specified as pointcuts or assertions that must hold true for the target joinpoint.

Pointcuts use assertions (typically patterns, such as regular expressions) that are checked against the properties of a joinpoint. For example, a typical pointcut will match a pattern against the name of a Java method. For example, the AspectJ [21] pointcut specification:

```
pointcut os() : call(void Node.setOS(String))


    before() : os() {
         //do something
    }
```

would match against any method call to a *Node* class' *setOS* method. This rule specifies a type to match against *Node* and a method to match *setOS*. The basic format is to specify a pattern that can be matched against the properties or attributes of a joinpoint to determine if a specific set of advice should be executed. In this case, the pointcut specification is being matched against the class name and method signature of the joinpoint.

The goal of deployment planning is to derive a valid mapping of software components to nodes, as shown in Figure 3. A deployment plan must meet *functional constraints*, such as having the appropriate OS for each component on its hosting node, and *non-functional constraints*, such as being able to process a minimum number of requests per second. Whereas aspect weaving is an automated process, deployment planning has historically been a manual pro-

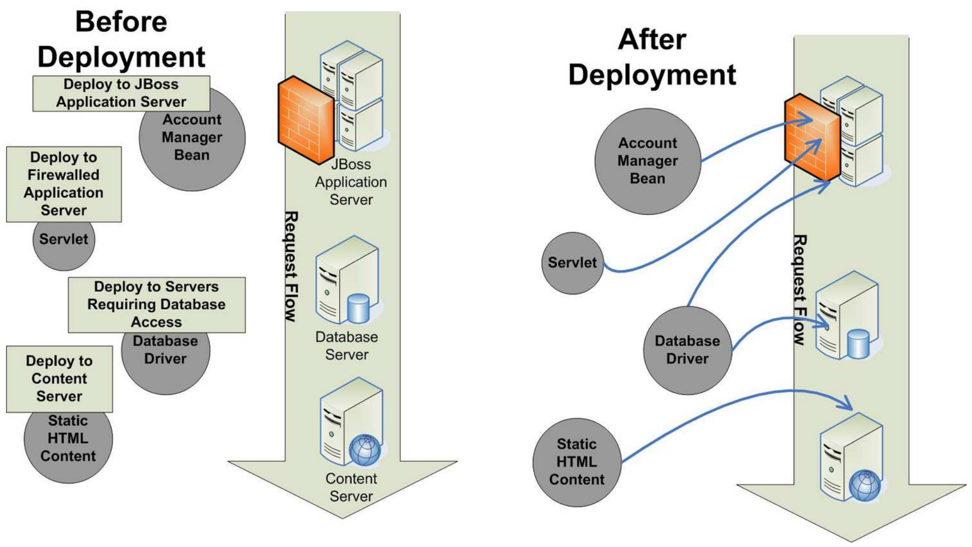


Figure 3: Overview of Deployment

cess [10].

Deployment planning also fits into the abstract problem definition of finding a valid association of items in a source set with items in a target set based on a set of rules. With deployment, the source set contains the software components, the target set contains the available nodes, and the rules are specified as assertions on the functional and non-functional properties of the nodes. For example, in Enterprise Java Beans (EJBs) [29], a deployment planning process assigns beans to nodes and specifies which EJBs, servlets, and other related resources should be placed on which servers (left-hand side of Figure 3).

In AOP, an aspect weaver automates the process of associating advice with joinpoints based on a set of pointcut pattern expressions. With deployment planning, however, an application deployer *manually* derives the correct placement of components on nodes to satisfy the various crosscutting concerns of the application, such as security and performance. Planning the deployment of components *manually* is tedious and error-prone and has been shown to be a major source of mistakes leading to system failure and down-time [10]. In particular, manual deployment planning creates a number of significant problems:

- The correctness of manually planned deployments is not guaranteed with respect to deployment constraints, *e.g.*, a developer may assign one or more components to nodes that have the wrong supporting libraries or insufficient memory.
- Manual planned deployment provides no guarantee of the deployment optimality, *e.g.*, a deployment planner may create a solution that uses far more nodes than is acutally necessary.
- Manual deployment is time-consuming and expensive due to the high complexity of deducing a deployment plan that adheres to the numerous cross-cutting concerns of an application.

By viewing deployment planning as a process for associating items from a source set with items in a target set, an aspect weaver can be used to automate the process of assigning components to nodes. Automating deployment planning with an aspect weaver helps address the three challenges outlined above. First, as we show in Section 3.3, a constraint solver can be used by a weaver to guarantee that a weaving solution respects the deployment constraints (addressing Challenge 1). Second, as discussed in Section 3.3, a constraint-solver based weaver can use different optimization and approximation algorithms to achieve different lower bounds on the solution optimality and upper bounds on the time spent weaving (addressing Challenge 2). Finally, automation eliminates considerable manual planning effort [31] (addressing Challenge 3).

2.2 Using Pointcuts to Specify Deployment Rules

Pointcut expressions can be used to specify some types of deployment rules. For a simple local constraint on a component's deployment, a pattern can be used to match components to nodes based on attribute patterns:

```
pointcut os() : at(Node OS==Windows)


    at() : os() {
      //specify some Windows components to deploy
    }
```

In this example, we created a pattern that matches when a node is encountered ($at(Node...)$) and the node has a Windows OS $at(...OS == Windows)$. Later, we define a group of components that should be deployed to the node if the pattern matches $at() : os()....$ Using this simple pattern matching strategy, an aspect weaver can automatically choose components for nodes to satisfy numerous crosscutting deployment constraints. For example:

```
pointcut os() : at(Node OS==Windows && Firewalled=true && CPUModel=Xeon)


    at() : os() {
      //specify some sensitive computationally intensive
      //components to deploy
    }
```

In this case, the rule is associating some security-sensitive components with nodes that are firewalled, have a Windows OS, and have a powerful Intel Xeon processor. As this pointcut pseudo-code shows, pointcuts can capture deployment rules.

3 Challenge - Developing a DSAL for Specifying Crosscutting Deployment Concerns

Although it may appear that a slight modification of an existing pointcut language will suffice to allow deployment planners to express deployment rules and automate deployment planning with a aspect weaver, there are the following challenges to building a custom pointcut language for deployment:

Challenge 1 - Existing pointcut languages are designed to express mapping rules specifically for Java, C++, or other third generation programming languages. One goal of building a DSAL to specify deployment planning concerns is to provide a series of language abstractions that can be reused across application domains. Each domain, however, typically is concerned with different types of elements.

Most existing AOP pointcut languages, such as the Josh pointcut language [6], operate on a single type of source and input set (an advice set and a set of execution points in an application). AspectJ associates advice with points in the control flow of a Java application(*e.g.*, method invocations, constructor invocations, etc.). With deployment, the application domain varies and what constitutes the source set and the target set varies.

Automotive systems are focused on associating micro-controller code to ECUs, while enterprise Java systems are concerned with mapping EJBs to application servers. Designing a DSAL on a per-application domain basis to handle the variations in set types is both expensive and time consuming. At the same time, however, DSAL designers do not want to expose automotive developers to concepts, such as application servers, that are irrelevant to their domain. Developing a flexible set of language elements is tricky.

Challenge 2 - Existing pointcut languages allow patterns and other matching constraints based on the properties of the elements of a single type of target set. The constraints or concerns that guide the deployment of components vary greatly across domains. Again, in most pointcut languages, such as AspectJ's, the attributes of the joinpoints are fixed to the domain of Java execution flows and are limited to concepts such as the containing class type, signature of the method invoked, etc. Regardless of the type of Java application being developed, the AspectJ attributes are relevant. In the domain of deployment, the attributes that are important to each domain vary and thus a fixed set of attributes (like AspectJ uses) will not suffice.

For example, when deploying Java Midlets to mobile devices, screen size, JVM stack size, device memory, and installed JRE libraries are important to consider. For enterprise Java applications, firewalls, external library versions, and application server configuration are examined to derive a deployment. In the automotive domain, domain experts analyze the ECU memory, safety properties (such as distance from the perimeter of the car), and connected busses. Producing a language that allows domain experts to capture a wide variety of requirement types in a meaningful format across application domains is hard.

Challenge 3 - Existing pointcut languages do not provide facilities for specifying global constraints governing the matching of advice to joinpoints. Deployment involves many concerns, such as performance, that typically involve global constraints that are not addressed by current pointcut languages. For example, if two components are collocated on the same node, the latency on calls between the components is reduced. At the same time, however, placing the components on the same node increases the amount of memory that is consumed on the node. A typical constraint on a deployment would be that for all nodes, the memory requirements of the components deployed to a given node do not exceed its available RAM. The various global constraints pull the solution in different directions and make it hard to find a valid solution.

Most pointcut languages are not designed to specify these types of global constraints because they must quickly match items against an indefinite target set at runtime. For example, even though some poincut languages may offer generic pointcut specification mechanisms such as queries [12] across the target set, these still have the potential for reasonable execution time at runtime against an arbitrary sized target set. Some pointcut languages, such as FOAL [23], have explored adding time-based constraints to pointcuts (*e.g.*, pointcut A is applied after pointcut B), but these languages do allow for global constraints on the of mapping source and target items – only on when the mapping occurs.

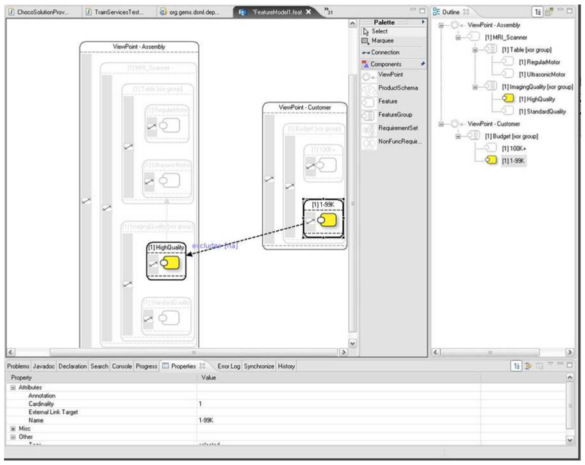
In deployment, if a component is matched to a node, it may subtract from the memory available on the node and change the future matches that can be made to the node. Matching a pointcut may also affect component placement due to safety concerns, such as "the anti-lock braking system and the steering control cannot be deployed to the same ECU." Although components may be *functionally* oblivious to where they and other components are deployed, they are often *non-functionally* affected by deployment locations.

3.1 ScatterML

Traditional pointcut languages are geared towards specifying methods for matching advice to joinpoints in a program's execution flow. We have developed a DSAL for capturing crosscutting deployment constraints as pointcuts. The pointcut specification is then used by our deployment weaver, called Scatter, to determine how to associate items from a source set (advice, components, etc.) with items in a target set (joinpoints, nodes, etc). We call our DSAL *ScatterML*.

ScatterML leverages the existing pattern matching pointcut approach for specifying the typical local deployment constraints on a component, such as the OS it requires, application server vendor, and available database connections. For global constraints that do not fit into the pattern matching paradigm, ScatterML provides an extended pointcut specification mechanism that is used by a constraint solver. This extended mechanism is discussed in Section 3.3 and addresses Challenge 3 from Section 3.

The ScatterML language allows developers to associate mapping rules (pointcuts) with components, advice, or other source set items. The pointcuts are represented as constraints that must be matched by the node, joinpoint, or other target element type that a source item is mapped to. ScatterML has been implemented both as a textual DSAL (which we use for the examples provided) and as a graphical modeling language built on top of the Generic Eclipse Modeling System (GEMS) [34], a part of the Eclipse GMT project. The graphical modeling language uses feature modeling [19, 9] notations and model transformation [5] to compile the graphical specifications into ScatterML textual specifications that can be leveraged by the Scatter deployment engine. A screenshot from the graphical binding of ScatterML is shown in Figure 4.



ViewPoint - Assembly
[1] MRI_Scanner
[1] Table [xor group]
[1] RegularMotor
[1] UltrasonicMotor
[1] ImagingQuality [xor group]
[1] HighQuality
[1] StandardQuality
ViewPoint - Customer
[1] Budget [xor group]
[1] 100K+
[1] 1-99K

Property	Value
Attributes	
Annotation	
Cardinality	1
External Link Target	
Name	1-99K
Misc	
Other	

Figure 4: The Graphical Implementation of ScatterML in GEMS

ScatterML allows each target item to have an arbitrary set of attributes. For example, a node may have *OS*, *RAM_Amount*, and *CPU_Type* attributes. Pointcut rules are specified as constraints on the attributes of a node. For example, the following ScatterML pointcut specification creates a pointcut that matches target items with the attributes *OS* and *InternetAccessible* and with attribute values of "*SELinux*" and "*false*", respectively:

```
SecureLinuxComponents {
  OS=Linux;
  InternetAccesible=false;
}
```

When the SecureLinuxComponents pointcut is matched against a node, that node is added to the list of acceptable deployment locations for the components associated with SecureLinuxComponents.

3.2 Addressing the Challenges of Developing a DSAL for Deployment Pointcuts

The ScatterML DSAL example for specifying crosscutting deployment concerns addresses the challenges from Section 3 in the following ways:

Challenge 1 → Solution: Allow arbitrary sets of source/target element types through flexible naming and a pluggable association mechanism.

The SecureLinuxComponents group can be a single component or set of components that are associated with the crosscutting constraints. The specifics of what constitutes the SecureLinuxComponents group is application domain dependent. In the domain of deploying components to mobile devices, the SecureLinuxComponents could resolve to a set of Java Midlets. If the domain is Enterprise Java, the components could resolve to EJBs and their associated Enterprise Archive Resource files. ScatterML does not specify what the names of different deployment groups map to. The association of names to artifacts (*e.g.*, AspectJ advice, Java Midlets, or EJBs) is built for each application domain by creating a small adapter component for the Scatter mapper that associates the appropriate artifacts to deploy with the named group.

The names of the groups can vary to match whatever application domain the mapping concerns are being modeled for. The mapping engine only needs to have a mechanism for retrieving the correct deployment artifacts for a given name. Moreover, what the components are being associated with is not specified in the mapping. The mapping specifies the properties of the target item but not whether it is a mobile device, EJB application server, or automotive ECU. Again, an adapter is used to perform the actual mapping function (*e.g.*, inserting a piece of advice into a Java application, adding an EJB to a deployment descriptor). The terminology of the ScatterML DSAL therefore supports application domain-specific concepts through the use of arbitrary attribute and component group names. The implementation of the mapping is provided through the user of domain-specific adapters.

The following four examples show the application domain-independence of ScatterML:

```
//Example 1. Java Advice to Joinpoints
TransactionPointcut {
 ContainingClass=com.my.bank.Account;
 InvokedMethod=transfer;
 Amount > 100;
}

// Exampe 2. EJB Applications
AccountEJB {
 ApplicationServerVendor = JBoss;
 JVM_Version > 1.5;
 HasAccountDatasource = true;
}

// Example 3. Mobile Devices
FoodServicesMidlet {
 JVM_Configuration = CDC;
 JVM_Profile = MIDP;
 MIDP_Version >= 1.1;
 Device_Display_Width >= 128;
 Device_Display_Height >= 256;
}

// Example 4. Automotive Software to ECUs
AntiLockBrakingControl {
 DistanceFromCarPerimeter >= 0.75m;
 ConnectedToBrakeActuator = true;
}
```

Example 1 creates a pointcut for Java advice that matches the invocation of an *Account* class' *transfer* method when it is called with an *Amount* parameter greater than 100 dollars. Example 2 associates an EJB with crosscutting concerns related to the version of the Java Virtual Machine (JVM) and vendor of the application server on the target node, as well as whether or not the target has a connection to the Account datasource. Example 3 specifies concerns relating to the configuration and profile of a JVM on a mobile device, as well as its display size. Example 4 shows a simple safety concern for the distance of a target ECU from the perimeter of the car. Moreover, the ECU must be connected to the brake actuators.

Although domain-specific terminology is used in each example, the application domains are shielded from each other. Ordinarily, these pointcuts would be in separate files for each application domain and are shown together only for comparison. Domain experts would only work with the files and notations for their associated domain. Instead, existing approaches would need to develop a DSAL for each domain to shield developers from the concerns of other domains.

Challenge 2 $\rightarrow$ Solution: Allow for constraints on arbitrary attribute/-value pairs but require application domain-specific mechanisms for obtaining attribute values. Pointcuts are built by specifying constraints on the attributes of the items they are matched against. Constraints are specified as an attribute name, a comparison (*e.g.* <,=), and a comparison value. The attribute name can be an arbitrary name that has meaning in the target domain. Similarly, the comparison value can be any value for which there is a known way of applying the comparison operator to the value. This flexible attribute/value architecture allows the creation of domain-specific constraints through the use of domain-specific attribute names and values.

If there are a small number of target items, the values of attributes can be assigned manually. If there are a large number of target items, automated probes can be used to evaluate the target nodes and produce attribute/value pairs for each node [31]. The only restriction on the names and types of attributes is that there exist a method for applying the comparison operator to the type.

For example, the < operator should not be applied to strings since the meaning is not well defined. Since ScatterML can handle arbitrary attributes and attribute types, it allows domain experts to leverage the key domain-specific attributes from their application domain. New operation types, such as matching against regular expressions, can be plugged into Scatter and associated with a new operator.

Challenge 3 → Solution: Create global constraint operators. As shown in Section 2.2, existing pointcut models can be used to handle types of concerns, such as security. Deployment, however, involves many concerns, such as performance, that typically involve global constraints that are not addressed by current pointcut languages.

For example, if two components are collocated on the same node, the latency on calls between the components is reduced. At the same time, however, placing the components on the same node increases the amount of memory that is consumed on the node. A typical constraint on a deployment would be that for all nodes, the memory requirements of the components deployed to a given node do not exceed its available RAM. The various global constraints pull the solution in different directions and make it hard to find a valid solution.

To address the limitations of applying existing pointcut languages to deployment planning, ScatterML includes four new global constraint operators commonly needed in deployment (and may useful in weaving for other domains too) that are summarized in Table 1. The operators are: *Requires*, *Excludes*, *Select*, and "−". The *Requires*, *Excludes*, and *Select* operators are used to govern the mapping of two source items to the same target item. For example, the following pointcut uses the *Excludes* operator to implement a safety constraint and ensure that the AntiLockBrakingSystem components are not deployed on the same node (ECU) as the SteeringControlSystem components:

```
AntiLockBrakingSystem {
  DistanceFromPerimiterOfCar > 0.75;
  Excludes: SteeringControlSystem;
}
```

The *Requires* operator can be used to guarantee the opposite, *i.e.*, that two items are always placed together. The *Select* operator is used for constraints

Operator	Applied To	Description
Requires	one or more other source items	Ensures that all of the specified source items are mappedto the same target item
Excludes	one or more other source items	Ensures that none of the specified source items are mappedto the same target item
Select$[MIN..MAX]$	a cardinality expression and one or more other source items	Ensures that at least MIN and at most MAX of the specified source items are mapped to the same target item
"$-$"	an integer value and an attribute name	Ensures that the sum of the "$-$" values on a given attribute by a matched set of pointcuts does not exceed the attribute value on the target item

Table 1: ScatterML Global Pointcut Constraints

involving the cardinality of a set of items placed with each other. For example, this pointcut allows either the SteeringControlSystem or the TractionControlSystem to be deployed with the AntiLockBrakingSystem:

```
AntiLockBrakingSystem {
  DistanceFromPerimiterOfCar > 0.75;
  Select: [0..1],SteeringControlSystem,TractionControlSystem;
}
```

The *Select* operator takes a cardinality expression ($MIN...MAX$) and tells the weaver that in order to match the source item to a target item, that at least MIN and at most MAX of the source items from the select statement must also be mapped to the target item.

The "$-$" operator is used to specify resource consumption values for items and enforce resource constraints. Given a set of pointcuts matched to a node, the sum of the "$-$" attribute constraints of the matched pointcuts cannot exceed the value of that attribute on the target item. The "$-$" operator can be used to specify constraints, such as the sum of the memory consumed by the components deployed to a node cannot exceed 10 MB. For example, the following ScatterML rule ensures that if both the *AntiLockBrakingSystem* and *BrakeActuators* are deployed to the same node, the node will have at least $Memory \geq 10$:

```
AntiLockBrakingSystem {
  DistanceFromPerimiterOfCar > 0.75;
  Memory - 5;
}
BrakeAcuators {
  Memory - 2;
}
```

3.3 Weaving with ScatterML Pointcuts

Adding new operators to specify global constraints is not in itself sufficient to address the problems of applying existing weaving techniques to deployment. Most existing weavers do not honor global constraints since they use a pattern matching based approach to pair advice with joinpoints at runtime. Solving complex global constraints is time-consuming and usually not feasible (nor desired) at runtime. Using a runtime pattern matching approach makes sense in most applications where there is either not an enumeration of all joinpoints ahead of time (you don't know what components will be loaded) or there are too many joinpoints to consider (an application usually has a significant number of method and constructor invocations).

With deployment planning, there is a known and limited set of joinpoints that are being matched against. This allows global constraints to be incorporated and dealt with at design-time. To make it possible to use an aspect weaver as a deployer, we implemented a new weaving technique that adheres to not only local pattern-based pointcuts but global constraints as well. The weaving technique uses a constraint solver to derive a solution to the abstract mapping problem (outlined in Section 2.1) that meets both the local and global mapping constraints. The solution produced is a pre-calculated table that can be used at runtime to lookup the target items that a source item should be mapped to. The weaver can be used to map components to nodes, match advice to joinpoints, and other set types.

Our prior work [26] on using an aspect weaver for deployment focused on the use of Prolog as an aspect weaver. This paper examines a previously unexplored aspect of weaving that uses a Java-based constraint solver called Choco [1] to perform weaving while honoring global constraints. Below, we present our formulation of constraint-aware aspect weaving as a Constraint Satisfaction Problem (CSP). The mapping also incorporates an adaptation of our techniques for handling resource constraints on feature models that we outlined in [31].

3.4 Mapping Aspect Weaving to a CSP

A CSP is a set of variables and a set of constraints over these variables. A constraint solver derives a *labeling* (set of values) for the variables that simultaneously satisfies the set of constraints. For example, "$a+b < c$" is a CSP involving the variables a, b, and c. A correct labeling of the CSP is $a = 1, b = 1, c = 3$.

To map aspect weaving to a CSP, we create a matrix of variables where the columns are the items in the target set (*e.g.*, joinpoints) and the rows are the source items (*e.g.*,advice). We call this matrix the weaving matrix. The value at a position $< i, j >$ is 1 if the i_{th} item should be mapped to the j_{th} joinpoint and zero otherwise. For deployment weaving, the rows represent the component groups and the columns represent the nodes. Each value in the matrix is represented by the variable W_{ij}. The weaving matrix is shown in the bottom of Figure 5.

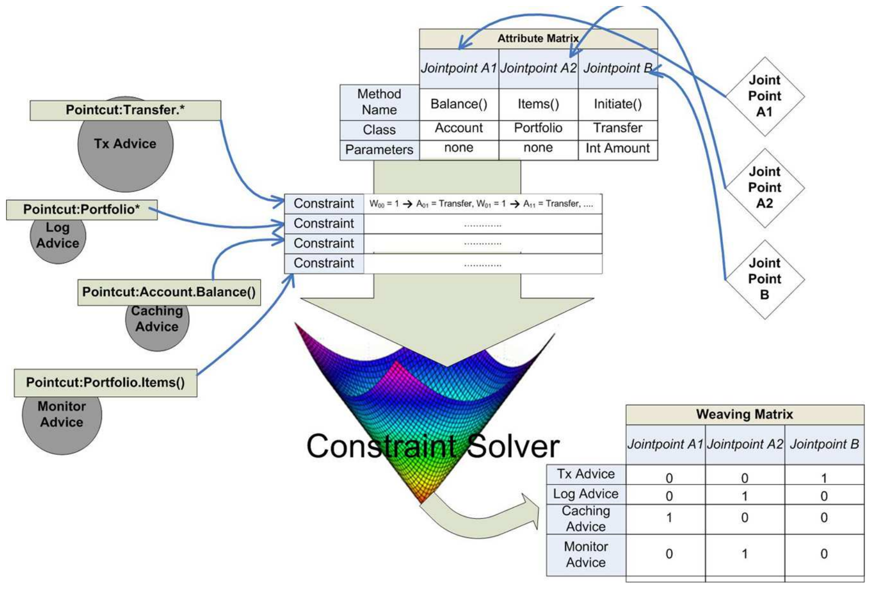


Figure 5: Reducing Aspect Weaving to a CSP and Generating a Weaving with a Constraint Solver

The weaving matrix is supplemented with an attribute matrix. The rows of the attribute matrix represent the different attributes that can be present on a target item. The columns represent the target items. The value at a position $< i, j >$ is the attribute value of the i_{th} attribute at the j_{th} joinpoint. For example, if the "RAM" attribute is the 2nd attribute in the matrix and "WindowsNode1" is the 3rd target item, the value at $< 2, 3 >$ is the value of WindowsNode1's RAM attribute. Each cell of the attribute matrix is represented by a variable A_{ij}. The attribute matrix is shown in the middle of Figure 5.

3.5 Mapping ScatterML to a CSP

The ScatterML pointcut specification is transformed into a set of constraints over the W and A variables for the matrix cells. For example, the following ScatterML fragment can be translated into a CSP for deriving a deployment over two different ECUs (weaving matrix columns 0 and 1):

```
AntiLockBrakingSystem {
  DistanceFromPerimiterOfCar > 0.75;
  Excludes: SteeringControlSystem;
}
```

Assume that the DistanceFromPerimeterofCar attribute is the 0th attribute. Moreover, assume that the AntiLockBrakingSystem is row 0 and the Steering-ControlSystem is row 1. The ScatterML fragment is translated as follows:

$(W_{00} = 1 \rightarrow A_{00} > 0.75) \wedge$ $(W_{00} = 1 \rightarrow W_{10} = 0) \wedge$ $(W_{01} = 1 \rightarrow A_{01} > 0.75) \wedge$ $(W_{01} = 1 \rightarrow W_{11} = 0) \wedge$ $(W_{00} + W_{01} = 1)$

The $(W_{00} = 1 \rightarrow A_{00} > 0.75)$ specifies that if the AntiLockBrakingSystem is deployed to ECU_0, the DistanceFromThePerimeterOfCar attribute for ECU_0 must be greater than 0.75. The constraint $(W_{00} = 1 \rightarrow W_{10} = 0)$ specifies that if the AntiLockBrakingSystem is deployed to ECU_0, then the SteeringControlSystem is not deployed there as well. The next two constraints can be mapped back to the original ScatterML in a similar fashion. The final constraint ensures that the AntiLockBrakingSystem is never deployed in two locations. This constraint can easily be removed or turned into a range to specify the minimum/maximum number of joinpoints the component group can be matched against.

3.6 Solving the Weaving CSP

To solve for a valid labeling of the CSP, values must be filled in for the attributes A_{00} and A_{01}. These values can either be specified through and auxiliary input file to Scatter that provides values for each ECU's attributes, through static analysis of the classes and methods of an application, or through probes that evaluate the ECUs and extract their attribute values. This initial step of filling in values for the attributes of the joinpoints is shown in the top of Figure 5.

Once the CSP has been produced and the values of the A_{ij} variables set, a constraint solver can be used to derive a labeling for the W_{ij} variables. The result of labeling the CSP will be a matrix that can be used to determine if the i_{th} source item should be mapped into the j_{th} target item. The weaving matrix can then be used to statically associate advice with joinpoints at compilation, dynamically associate advice with joinpoints at runtime, or to assign components to nodes during deployment.

3.7 Benefits of a CSP Weaving Model

Using a constraint solver to calculate a table for matching advice to joinpoints provides several benefits. For example, the solution produced by the solver (if one is found) is guaranteed to be correct with respect to both local and global weaving constraints. Moreover, a solver can handle many complex global constraints and large CSPs that would be infeasible to manage manually or with a strictly pattern matching type approach. Solvers can also be used to produce CSP labelings that maximize or minimize a function.

In the AntiLockBrakingSystem example from Section 3.5, the solver could be asked to maximize the sum $W_{00}A_{00} + W_{01}A_{01}$, which would have the effect of placing the AntiLockBrakingSystem as far from the perimeter of the car as possible. In cases where only a certain percentage of optimality is required, the solver can use different approximation techniques to trade solution quality for increased solving speed. The tradeoff, however, can be performed in a manner to still guarantee a worst case bound on solution quality.

A key motivation for creating ScatterML was to provide a high-level domain-specific language to specify constraints. Directly using the CSP formulation shown in this section is tedious and error-prone. ScatterML raises the level of abstraction used to specify mapping constraints and allows domain experts to leverage the automated CSP approach to weaving. The Scatter deployment engine automatically performs the transformation from ScatterML models to CSPs.

An interesting effect of viewing aspect weaving as a specialized instance of our abstract set mapping problem is that it can be used to show that product [7, 17] configuration from feature models can be viewed as a weaving activity. The reduction of product configuration to CSP labeling provided by Benavides [4] and the CSP model of deployment we have produced in other work is identical to the CSP model for weaving. Advances in aspect weaving techniques using this view of matching components or advice to joinpoints can therefore be applied to both the domains of deployment and product configuration.

4 Empirical Results

Experimental design. To evaluate the benefits of using an aspect weaver as a deployment planner, we developed a deployment scenario in the automotive domain to apply both a manual and ScatterML-based deployment planning process to. The automotive scenario encompasses the deployment of five components to a set of three ECUs. The components and their associated constraints are shown in the ScatterML pointcuts below:

```
ABS{
 Memory - 1;
 Requires:BrakeActuators;
}
BrakeActututators{
 Memory  10;
 FirmwareVendor = Z;
 Firmware Version > 1.2;
}
```

```
SteeringControl{
 Excludes: ABS;
 Firmware Version > 1.6;
}
Infotainment{
 Memory - 3;
}
GPSNavSystem{
 Memory - 5
}
```

The ScatterML specification includes a set of pointcut rules. For example, the *ABS* component must be deployed with the *BrakeActuators* component. The *SteeringControl* component cannot be placed on the same ECU as the *ABS* component. The components consume 1, 10, 0, 3, and 5 memory units respectively. The components can be deployed to three ECUs with the following capabilities:

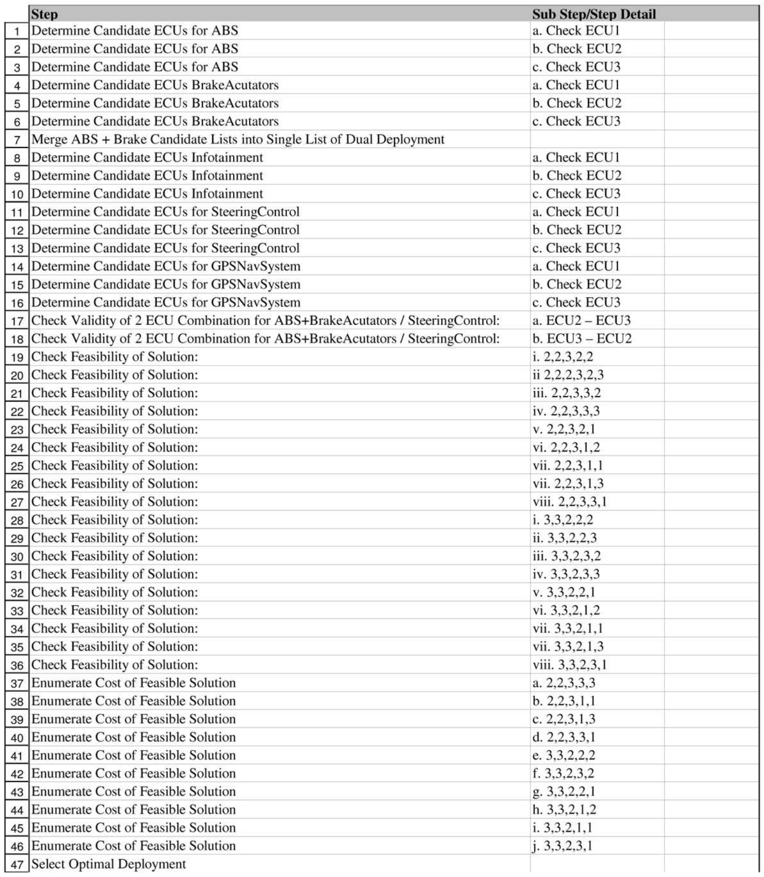
```
ECU1
Memory = 9
Cost = 10
FirmwareVendor = Z
Firmware Version = 1.5

ECU2
Memory = 12
Cost = 16
FirmwareVendor = Z
Firmware Version = 1.9

ECU3
```

```
Memory = 15

Cost = 19

FirmwareVendor = Z

Firmware Version = 1.9
```

As can be seen from the listing of ECUs, each ECU has a cost associated with it. The goal of the deployment planning process is to produce the minimum cost deployment plan.

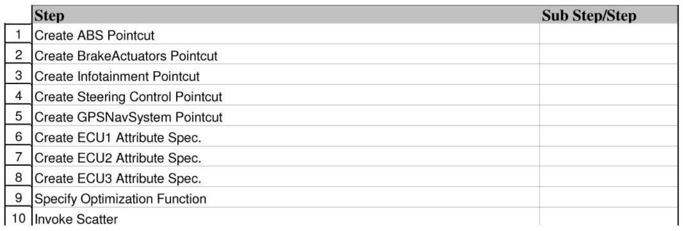
Analysis of manual deployment planning. First, we applied a manual process to derive an optimal deployment plan. The steps required to perform the deployment planning process are shown in Figure 6. In the first 16 steps, each component is analyzed and a list of feasible hosting ECUs is produced. In step 7, the feasible ECU lists for the *ABS* and *BrakeAcutators* are merged, since they must be deployed together. In steps 19-36, the valid hosting ECUs for the *ABS+BrakeAcutators* and the *SteeringControl* are used to enumerate and check all remaining possibly valid deployment combinations for the components (the deployment plans are specified as strings X,Y,Z,A,B that indicate the ABS is deployed to ECUX, the BrakeActuators are deployed to ECUY, etc.). In steps 37-46, the cost is calculated for each feasible deployment plan. Finally, in step 47, the optimal cost deployment plan is chosen for each of the remaining valid deployment scenarios.

Analysis of scatter deployment planning. After conducting the analysis of manually deriving a deployment plan for the example application, we performed the same analysis using Scatter. The set of steps to find an optimal deployment plan with Scatter is shown in Figure 7. In steps 1-5, the pointcuts for capturing the deployment rules for the five components are created. In steps 6-8, the attribute values of the target ECUs are specified. In step 9, the cost



	Step	Sub Step/Step Detail
1	Determine Candidate ECUs for ABS	a. Check ECU1
2	Determine Candidate ECUs for ABS	b. Check ECU2
3	Determine Candidate ECUs for ABS	c. Check ECU3
4	Determine Candidate ECUs BrakeAcutators	a. Check ECU1
5	Determine Candidate ECUs BrakeAcutators	b. Check ECU2
6	Determine Candidate ECUs BrakeAcutators	c. Check ECU3
7	Merge ABS + Brake Candidate Lists into Single List of Dual Deployment	
8	Determine Candidate ECUs Infotainment	a. Check ECU1
9	Determine Candidate ECUs Infotainment	b. Check ECU2
10	Determine Candidate ECUs Infotainment	c. Check ECU3
11	Determine Candidate ECUs for SteeringControl	a. Check ECU1
12	Determine Candidate ECUs for SteeringControl	b. Check ECU2
13	Determine Candidate ECUs for SteeringControl	c. Check ECU3
14	Determine Candidate ECUs for GPSNavSystem	a. Check ECU1
15	Determine Candidate ECUs for GPSNavSystem	b. Check ECU2
16	Determine Candidate ECUs for GPSNavSystem	c. Check ECU3
17	Check Validity of 2 ECU Combination for ABS+BrakeAcutators / SteeringControl:	a. ECU2 – ECU3
18	Check Validity of 2 ECU Combination for ABS+BrakeAcutators / SteeringControl:	b. ECU3 – ECU2
19	Check Feasibility of Solution:	i. 2,2,3,2,2
20	Check Feasibility of Solution:	ii 2,2,2,3,2,3
21	Check Feasibility of Solution:	iii. 2,2,3,3,2
22	Check Feasibility of Solution:	iv. 2,2,3,3,3
23	Check Feasibility of Solution:	v. 2,2,3,2,1
24	Check Feasibility of Solution:	vi. 2,2,3,1,2
25	Check Feasibility of Solution:	vii. 2,2,3,1,1
26	Check Feasibility of Solution:	vii. 2,2,3,1,3
27	Check Feasibility of Solution:	viii. 2,2,3,3,1
28	Check Feasibility of Solution:	i. 3,3,2,2,2
29	Check Feasibility of Solution:	ii. 3,3,2,2,3
30	Check Feasibility of Solution:	iii. 3,3,2,3,2
31	Check Feasibility of Solution:	iv. 3,3,2,3,3
32	Check Feasibility of Solution:	v. 3,3,2,2,1
33	Check Feasibility of Solution:	vi. 3,3,2,1,2
34	Check Feasibility of Solution:	vii. 3,3,2,1,1
35	Check Feasibility of Solution:	vii. 3,3,2,1,3
36	Check Feasibility of Solution:	viii. 3,3,2,3,1
37	Enumerate Cost of Feasible Solution	a. 2,2,3,3,3
38	Enumerate Cost of Feasible Solution	b. 2,2,3,1,1
39	Enumerate Cost of Feasible Solution	c. 2,2,3,1,3
40	Enumerate Cost of Feasible Solution	d. 2,2,3,3,1
41	Enumerate Cost of Feasible Solution	e. 3,3,2,2,2
42	Enumerate Cost of Feasible Solution	f. 3,3,2,3,2
43	Enumerate Cost of Feasible Solution	g. 3,3,2,2,1
44	Enumerate Cost of Feasible Solution	h. 3,3,2,1,2
45	Enumerate Cost of Feasible Solution	i. 3,3,2,1,1
46	Enumerate Cost of Feasible Solution	j. 3,3,2,3,1
47	Select Optimal Deployment	

Figure 6: Steps Required for Manual Deployment Planning



	Step	Sub Step/Step
1	Create ABS Pointcut	
2	Create BrakeActuators Pointcut	
3	Create Infotainment Pointcut	
4	Create Steering Control Pointcut	
5	Create GPSNavSystem Pointcut	
6	Create ECU1 Attribute Spec.	
7	Create ECU2 Attribute Spec.	
8	Create ECU3 Attribute Spec.	
9	Specify Optimization Function	
10	Invoke Scatter	

Figure 7: Steps Required for Scatter Deployment Planning

function to optimize is specified. Finally, in step 10, Scatter is invoked to derive an optimal mapping of components to nodes based on the specified cost function.

Analysis of results. Comparing the steps for the manual deployment planning process in Figure 6 and the Scatter process in Figure 7 shows that the Scatter-based process requires 78.7% fewer steps. Scatter has a total of 9 steps in which errors that are not guaranteed to be detected can be made (invoking Scatter incorrectly will be flagged as an error). All 47 steps of the manual process, in contrast, can potentially create errors that will not be detected.

To test the running time of Scatter on the example problem, we invoked the Scatter weaver 1,000 times and recorded the average, worst, and best case execution times. The average execution time of Scatter was 16ms. The worst case execution time of Scatter was 34ms. The best case execution time of Scatter was 1ms. To out perform Scatter, a human would need to perform the 47 steps, outlined in Figure 6, in at most 34ms.

In summary, not only does Scatter save a large amount of deployment planning steps, but it also reduces the number of points at which errors can be made. As the number of components and nodes increases, the savings produced by Scatter will become even more significant. There will be $components^{nodes}$ possible deployments to check with a manual process. Scatter sorts through these possible deployments on the deployment planner's behalf.

5 Related Work

This section compares our work on Scatter with related work on aspect-oriented weaving and deployment planning.

5.1 AOP-based Related Work

The idea of using modeling and aspect weaving to weave deployment aspects into Domain-specific Modeling Languages (DSMLS) is presented by Balasubramanian [3]. That work focuses on a different problem from Scatter, namely how to improve the comprehensibility and help automate modeling tasks related to the specification of a component-based system's functional structure and and the artifacts associated with the structures. Balasubramanian's work helps to automate the determination of what comprises the source and target sets that must be woven in a deployment. In contrast, Scatter focuses on the complex task of finding a way of mapping the source and target sets to each other. Moreover, their approach relies on a weaving engine, called C-Saw that does not have Scatter's ability to capture or solve global weaving constraints. Scatter and Balasubramanian's work are complementary in that Balasubramanian's work can be used to specify and manage the source and target sets through models and C-Saw aspect weaving, while Scatter can be used to specify mapping rules and deduce complex mappings for the source and target sets.

Gray et al. [16] present research showing how aspect-orientation can be used to reduce tangling and improve model quality in model-driven development. The techniques of capturing constraints as crosscutting concerns and associating them with model components is closely related to the ideas of capturing deployment concerns and associating them with component groups. A key differentiator between the two approaches is that Scatter is designed to perform constraint-aware weaving with the constraints whereas the goal of Gray's approach is to improve model quality and reduce tangling. Gray's approach does not provide deployment weaving based on global constraints as Scatter does.

In [14], France et al. layout a framework for composing aspects and determining if the composition has errors. The technique proposed by France et al. is not focused on deployment but has similar goals of allowing developers to create automated mechanisms for composing and ensuring the correctness of aspects. Scatter and Scatter allow aspects to be composed along with a model of components and nodes to create a deployment plan. Whereas France et al. propose using testing to check for correctness, the Scatter approach uses a constraint solver to guarantee that the derived composition is correct. Moreover, the Scatter approach is specifically designed to handle concerns related to deployment, such as resource constraints and dependencies.

Klose et al. [23] present an aspect language called FOAL that transforms program traces into Prolog knowledge bases and specifies pointcuts as conditions over the traces. FOAL is focused on providing temporal constraints between pointcuts (*e.g.*, advice A is applied before advice B). Although FOAL allows for a form of context-aware matching of pointcuts, the constraints are based on time. Scatter focuses on allowing constraints based on the structure and resources of joinpoints, which is a requirement for using an aspect weaver for deployment. Moreover, as Klose points out in their paper, they have no known application for their technique. In contrast, Scatter has clear application domains and also provides the ability to perform optimization, which FOAL does not.

5.2 Deployment Planning Related Work

Ivan et al. [18] present an AI-based deployment planning engine. Ivan's work allows the specification of similar constraints to Scatter. Ivan's planner is focused on runtime migration of services closer to a client and thus does not perform optimization. In contrast, Scatter is a design-time tool and can perform optimization of deployments based on a complex cost function.

Kichkaylo et al. [20] present an extension of Sekitei AI planning to find optimal resource-based deployment plans for components. Kichkaylo's work labels available resources, such as network bandwidth, with discrete levels and indications if they can be upgraded or downgraded. Kichkaylo's planning techniques are designed to improve the placement of components to nodes as resource availability changes. Whereas Kichkaylo is primarily concerned with resource-based constraints, Scatter is designed for deriving deployments when there are a high level of dependency, platform, security, and other hetergeneous constraint types. Moreover, Scatter is geared towards design-time deployment planning and optimization whereas Kichkaylo's work is focused on online planning. Kichkaylo's techniques attempt to optimize bandwidth and CPU resources whereas Scatter also permits the inclusion of other arbitrary domain-specific optimization criteria, such as hardware costs and distance from car perimeter.

Benavides et al. [4] have presented a method for reducing a feature selection problem to a CSP. This paper builds on the work of Benavides et al. by showing how deployment and aspect weaving can be reduced to a related CSP model. Moreover, Scatter extends Benavides' solution with resource constraints. Finally, Benavides does not provide a domain-specific language for capturing crosscutting deployment concerns like Scatter.

SmartFrog [15] is a tool for specifying and automating complex deployments. SmartFrog uses a similar view of deployment to Scatter, namely components and nodes. SmartFrog and Scatter although seemingly similar are designed to solve different problems of deployment. Scatter is focused on addressing the challenges of deducing the correct allocation of components to nodes or deployment plan whereas SmartFrog is designed to execute a deployment plan correctly. SmartFrog does not provide any facilities for deriving a correct deployment plan from a model of components and their crosscutting concerns as Scatter does.

6 Concluding Remarks

Deployment is a complex process that involves merging the non-functional and functional concerns of numerous components into a correct deployment plan. Manually producing a deployment plan, however, is a tedious and error-prone process. By viewing deployment as an process for associating items from a source set with items from a target set, many complex and time-consuming parts of deployment can be automated with an aspect weaver.

This paper described how deployment can be automated with an aspect weaver. In particular, we used pointcuts to specify rules for how components are matched to nodes and then used an aspect weaver to merge components automatically into their correct deployment locations. Deployment often relies on global constraints, however, which are not well-supported with existing pointcut languages. This paper therefore describes how the ScatterML DSAL specifies deployment rules by extending their pattern matching-based syntax with elements for capturing global deployment concerns, such as dependencies between components and resource constraints. We also showed how a constraint solver can be used to weave crosscutting deployment concerns, specified as pointcuts, into a deployment plan.

From our experience developing and applying ScatterML to deploy Java Midlets to mobile devices, EJBs to application servers, and CCM applications to servers, we learned the following lessons:

- Aspect weaving and deployment can both be reduced to the abstract problem of mapping items from a source set (advice, components) to items in a target set (joinpoints, nodes) based on a set of rules (pointcuts, deployment constraints). Using this abstract definition of weaving allows the use of an aspect weaver to automate deployment planning. There are other domains, such as the configuration of product-lines, that can be viewed as specializations of this same abstract problem [31].

- The abstract source → target mapping problem can be phrased as a CSP and a constraint solver used to derive a correct mapping. It is tedious and error-prone, however, to formulate the CSP to solve this mapping problem. The CSP formulation of the problem can be made amenable to developers through the use of a DSAL and automated transformation from DSAL instances to CSP instances.

- Mistakes in global pointcut constraints are hard to debug. In future work, we plan to investigate different techniques for debugging CSP labeling failures.

- Although a constraint solver can produce an optimal labeling, optimization can require large amounts of time for complex mappings. Developers must carefully test each mapping problem to ensure that it is not too time-consuming to optimize.

- Although pointcuts are typically associated with AOP they are a natural way of capturing deployment constraints. For example, the feasiblity of matching an EJB to an application server can be expressed as a pattern that the properties of the application server must possess.

In future work, we plan to extend our constraint solver weaving approach to weaving advice into Java applications. ScatterML is an open-source project available from CVS at `www.sf.net/projects/gems`.

References

[1] Choco constraint programming system. http://choco.sourceforge.net/.

[2] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 86–95, 2002.

[3] K. Balasubramanian, A. Gokhale, Y. Lin, J. Zhang, and J. Gray. Weaving Deployment Aspects into Domain-specific Models. *International Journal on Software Engineering and Knowledge Engineering (IJSEKE)*, 16(3), 2006.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. *17th Conference on Advanced Information Systems Engineering (CAiSE05, Proceedings), LNCS*, 3520:491–503, 2005.

[5] J. Bézivin. From Object Composition to Model Transformation with the MDA. *Proceedings of TOOLS*, pages 350–354, 2001.

[6] S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111, 2004.

[7] P. C. Clements and L. Northrop. *Software Product Lines Practices, and Patterns.* Addison-Wesley, 2001.

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

[9] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 2005.

[10] D. P. D. Oppenheimer, A. Ganapathi. Why do Internet Services Fail, and What can be Done about It? *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2003.

[11] G. Deng, J. Balasubramanian, W. Otte, D. Schmidt, and A. Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. *Proceedings of the 3rd Working Conference on Component Deployment*, 2005.

[12] M. EICHBERG, M. MEZINI, and K. OSTERMANN. Pointcuts as functional queries. *Lecture notes in computer science*, 3302(1):366–381.

[13] P. Fradet and M. Sudholt. An aspect language for robust programming. *Proceedings of the ECOOP*, 99.

[14] R. France, G. Georg, and I. Ray. Supporting multi-dimensional separation of design concerns. *Proceedings of the Third International Workshop on Aspect-Oriented Modeling, March*, 2003.

[15] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and Automatic Ignition of Distributed Applications. *HP Openview University Association conference*, 2003.

[16] 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.

[17] J. Greenfield and K. Short. *Software factories: assembling applications with patterns, models, frameworks and tools.* ACM Press New York, NY, USA, 2003.

[18] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable services: A framework for seamlessly adapting distributed applications to heterogeneous environments. In *Proceedings of the 11th IEEE International Symposium on High performance Distributed Computing*, July 2002.

[19] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented project line engineering. *IEEE Softw.*, 19(4):58–65, 2002.

[20] T. Kichkaylo and V. Karamcheti. Optimal Resource-aware Deployment Planning for Component-based Distributed Applications. In *Proceedings of the 13th IEEE International Symposium on High performance Distributed Computing*, June 2004.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. in proc. *11th European Conference on Object-Oriented Programming*, pages 220–242.

[23] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. *Foundations of Aspect-Oriented Languages Workshop*, March 2005.

[24] P. Manadhata and J. Wing. *Measuring a System's Attack Surface*. School of Computer Science, Carnegie Mellon University, 2004.

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

[26] A. Nechypurenko, E. Wuchner, J. White, and D. C. Schmidt. Application of Aspect-based Modeling and Weaving for Complexity Reduction in the Development of Automotive Distributed Realtime Embedded Systems. In *Proceedings of the Sixth International Conference on AspectOriented Software Development*, March 2007.

[27] M. Shonle, K. Lieberherr, and A. Shah.

[28] E. Tanter and J. Noye. A versatile kernel for multi-language AOP. *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, pages 173–188, 2005.

[29] T. Valesky. *Enterprise JavaBeans*. Addison-Wesley Reading, MA, 1999.

[30] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press Cambridge, MA, USA, 1989.

[31] J. White, K. Czarnecki, D. C. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege. Automated Model-based Configuration of Enterprise Java Applications. In *EDOC 2007*, 2007 (to appear).

[32] J. White, A. Nechypurenko, E. Wuchner, and D. Schmidt. *Designing Sofware-Intensive Systems, Methods and Principles*, chapter Reducing the Complexity of Designing and Optimizing Large-scale Systems by Integrating Constraint Solvers with Graphical Modeling Tools. Idea Group, Inc., New York, NY, 2008 (to appear).

[33] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt. Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference*, September 2007.

[34] J. White, D. C. Schmidt, and S. Mulligan. The generic eclipse modeling system. In *Model-Driven Development Tool Implementors Forum at TOOLS 2007*, June 2007.