

# Automated Reasoning for Multi-step Feature Model Configuration Problems

Jules White, Brian Dougherty, and Douglas C. Schmidt  
Vanderbilt University, Nashville, TN USA  
Email: {jules, briand, schmidt}@dre.vanderbilt.edu

David Benavides  
University of Seville, Seville, Spain  
Email: benavides@us.es

## Abstract

*The increasing complexity and cost of software-intensive systems has led developers to seek ways of increasing software reusability. One software reuse approach is to develop a Software Product-line (SPL), which is a reconfigurable software architecture that can be reused across projects. Creating configurations of the SPL that meets arbitrary requirements is hard.*

*Existing research has focused on techniques that produce a configuration of the SPL in a single step. This paper provides three contributions to the study of multi-step configuration for SPLs. First, we present a formal model of multi-step SPL configuration and map this model to constraint satisfaction problems (CSPs). Second, we show how solutions to these CSP configuration problem CSPs can be derived automatically with a constraint solver. Third, we present empirical results demonstrating that our CSP-based technique can solve multi-step configuration problems involving hundreds of features in seconds.*

## 1 Introduction

The high-cost of developing distributed real-time and embedded (DRE) systems has pushed developers to find novel solutions to increase the reusability of software. One promising reuse approach is Software Product-lines (SPLs) [6], which are software architectures that are designed with built-in points of variability that can be altered so that the software can be more readily reused across projects. For example, an SPL for a car can be built with the ability to use multiple engine control software components so it can be adapted to cars with different engine types.

Ensuring that a correct software product is produced from an SPL involves building models of the rules for configuring the points of variability. For example, an SPL configuration for a car cannot simultaneously employ two different engine control software components or the wrong component for the given engine type. A common technique for specifying SPL configuration rules is a *feature model* [11], which abstracts the components and points of

variability in a software product as *features*.

Feature models are typically implemented as tree-like structures that specify how the components and points of variability affect one another. For example, the feature model of a car in Figure 1 can optionally include an Automated Driving Controller. If the car includes this feature it must also include the Collision Avoidance Breaking feature. Any arbitrary configuration can be checked against the feature model to determine if it is a complete and correct configuration of a software product.

When an SPL is configured for a new set of requirements, developers must find a selection of the features from the feature model that (1) satisfy the requirements and (2) adhere to the rules in the feature model. This configuration process involves reasoning over a complex set of constraints to meet an end goal. Various tools [2, 13, 1, 4, 5, 15] have been developed to help reduce the complexity of this process by automating parts of the feature selection process.

**Open problems.** Some configuration problems require starting at an arbitrary state and deriving a new configuration that meets the target requirements. For instance, an automotive software designer using an SPL may start with no features selected and derive a selection of features for the automobile software to meet the needs of a new model year car. Often, however, constraints limit developers from directly transitioning from the starting state to the desired end configuration.

For example, assume that a group of automotive SPL developers want to modify the configuration of an existing SPL car model to include automated driving capabilities, as shown in Figure 1. The developers have determined that the cost of adding all the new features will be 88 million dollars. The developers only have a annual development budget of 35 million dollars to reconfigure the SPL variant, however, which means that the developers cannot simply add all features in a single year. Management has also asked the developers to make continual progress on developing the car by adding new features to it every year.

To manage these constraints, the developers must incrementally add the desired features over a series of steps, *i.e.*, over several years the developers will produce a series of in-

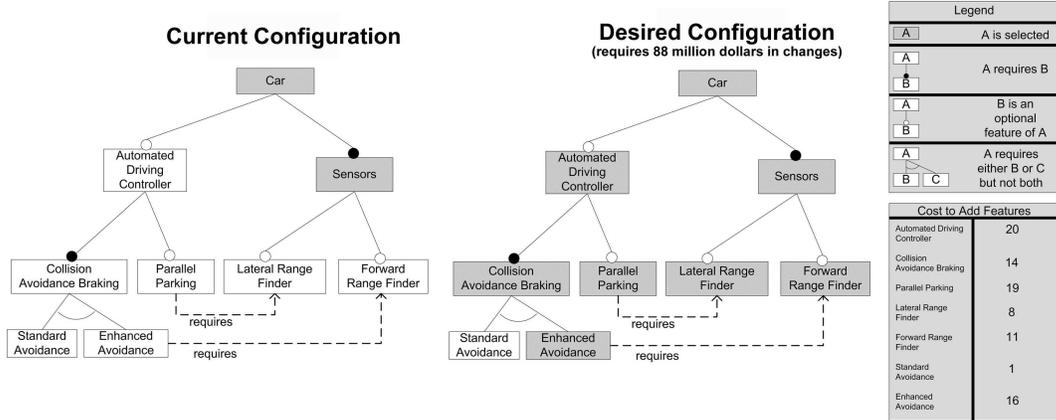


Figure 1: A Configuration Problem Requiring Multiple Steps

intermediate configurations that leverage each other to reach the desired configuration. For example, they can develop a subset of the new features in the first year’s car configuration, add more of the remaining desired features in the second year, and add the rest of the desired features and reach the new configuration in the third year. This process of producing a series of intermediate configurations—*i.e.*, a *configuration path*—is shown in Figure 2. We call this sequence of activities a *multi-step configuration problem*.

A key challenge is that the developers cannot arbitrarily pick and choose features to add in a given year to meet the budget constraint. For example, developers will violate the feature model rules if they choose to add Parallel Parking in year 3 without Lateral Range Finder, which is required via a cross-tree constraint, as shown in Figure 2. Developers must therefore not only adhere to their constraints on the changes that can be produced in a given year (such as the maximum allowed annual development budget) but also ensure that the changes they choose create a valid configuration at the end of each year. The developers also cannot choose an intermediate configuration to transition through that will not function and hence cannot be sold.

Further complicating the multi-step configuration problem is that developers may need to foresee tradeoffs that must be made along the way. For example, it is not possible to simultaneously add Collision Avoidance Braking and Enhanced Avoidance in the same year since their total development cost is 36 million dollars, as shown in Figure 2. Developers must therefore add Collision Avoidance Braking and Standard Avoidance one year (to simultaneously meet the budget constraint and the feature model constraints) and then remove the Standard Avoidance feature at a later step to add the desired Enhanced Avoidance feature.

Prior work on configuration analysis and automation [13, 1, 4, 5] focused on creating one configuration that meets a specific set of requirements, *i.e.*, they find a configuration in one step and assume that it is possible to directly transition to it. These techniques do not, however, support the need

to split the configuration over multiple steps to adhere to a change constraint, such as the maximum development budget per year. A gap therefore exists in current techniques when developers need to reason about and automate configuration over multiple steps.

**Solution overview and contributions.** To fill the gap in existing research, we have developed an automated method for deriving a set of configurations that meet a series of requirements over a span of configuration steps. We call our technique the *MULTI-step Software Configuration problem solver* (MUSCLE). MUSCLE transforms multi-step feature configuration problems into constraint satisfaction problems (CSPs) [9]. Once a CSP has been produced for the problem, MUSCLE uses a constraint solver (which is an automated tool for finding solutions to CSPs) to generate a series of configurations that meet the multi-step constraints.

This paper provides the following contributions to the study of feature model configuration over a span of multiple steps:

1. We provide a formal model of multi-step configuration,
2. We show how the formal model of multi-step configuration can be mapped to a CSP,
3. We show how multi-step requirements, such as limits on the cost of feature changes between two successive configurations, can be specified using our CSP formulation of multi-step configuration,
4. We describe mechanisms for optimally deriving a set of configurations that meet the requirements and minimize or maximize a property of the configurations or configuration process, such as total configuration cost,
5. We show how multi-step optimizations can be performed, such as deriving the series of configurations that meet a set of end-goals in the fewest time steps, and

**Paper organization.** The remainder of the paper is organized as follows: Section 2 summarizes the challenges of performing automated configuration reasoning over a sequence of steps; Section 3 describes a formal model of

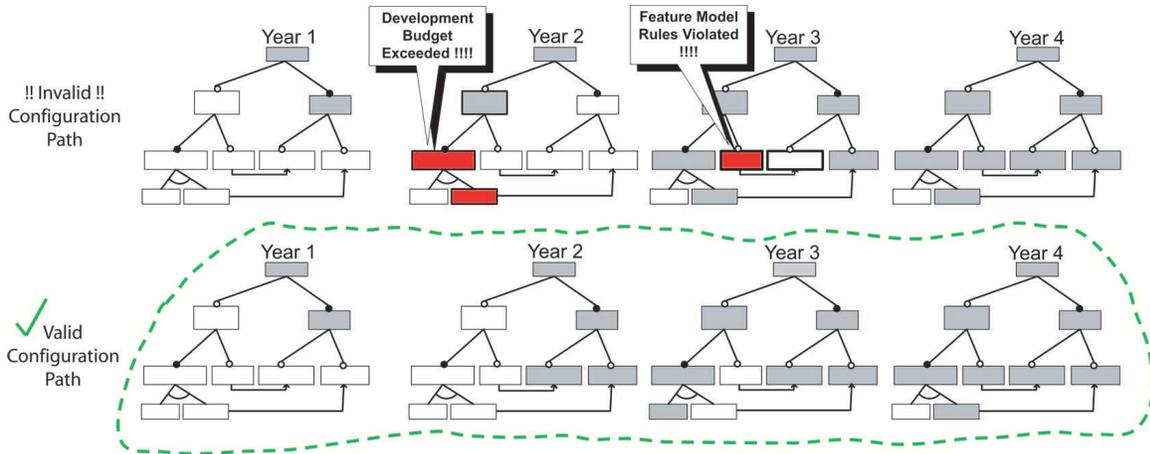
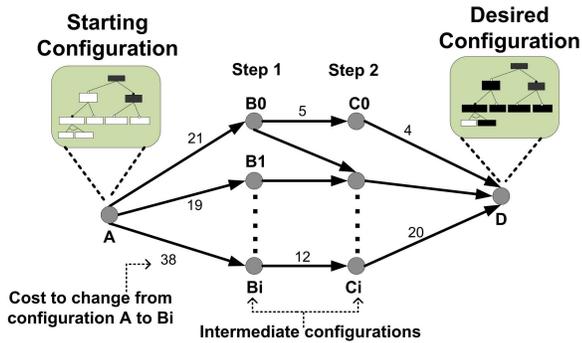


Figure 2: Potential Configuration Paths

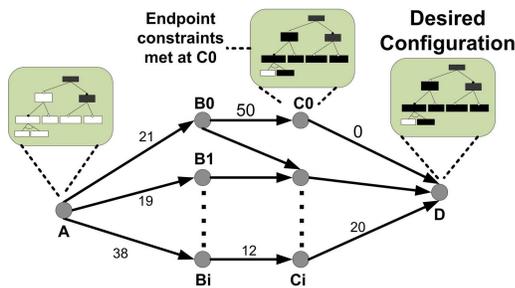
multi-step configuration; Section 4 explains MUSCLE’s CSP-based automated multi-step configuration reasoning approach; Section 5 analyzes empirical results from experiments demonstrating the scalability of MUSCLE; Section 6 compares MUSCLE with related work; and Section 7 presents concluding remarks.

## 2 Challenges

A multi-step configuration problem for an SPL involves transitioning from a starting configuration through a series of intermediate configurations to a configuration that meets a desired set of end state requirements. The solution space for producing a series of successive intermediate configurations to reach the desired end state can be represented as a directed graph, as shown in Figure 3(a).



(a) A Graph of a Multi-step Configuration Problem



(b) Optimization of Total Steps

Figure 3: Multi-step Configuration Graphs

Each successive series of points represents potential configurations of the feature model at a given step. For example, the configurations  $B_0 \dots B_i$  represent the intermediate configurations that can be reached in one step from the starting configuration. In this section we use this graph formulation of the problem’s solution space to showcase the challenges of finding valid solutions.

### 2.1 Challenge 1: Graph Complexity

A critical challenge to developers attempting to derive solutions to multi-step configuration problems manually or to use a graph algorithm is that there are an exponential number of potential intermediate configurations and paths that could be used to reach the desired end state. In the worst case, at any given intermediate step, there can be  $O(2^n)$  points (where  $n$  is the number of features in the feature model). In the worst case, therefore, there are  $2^n$  potential subsets of the features in the feature model that could form a configuration. Moreover, for a multi-step configuration problem over  $K$  time steps, there are  $O(K2^n)$  possible intermediate points.

Further compounding this problem is that for any intermediate configuration at step  $T$ , there are in the worst case  $2^n - 1$  points at step  $T + 1$  that could be reached from it by adding or removing features to its feature selection. The intermediate configurations that do not precede the end point will therefore have  $2^n - 1$  outgoing edges. Section 4 discusses how MUSCLE uses CSP-based automation to eliminate the need for developers to manually find solutions to these multi-step configuration problems, which reduces configuration time and cost.

### 2.2 Challenge 2: Point Configuration Constraints

Although there are a substantial number of potential intermediate configurations, many of these configurations will not meet developer requirements. For example, many of the  $K2^n$  arbitrary subsets of feature selections will represent configurations that do not adhere to the feature model

constraints. Moreover, other external constraints, such as safety constraints requiring a specific feature to be selected at all times, may not be met. We term these constraints on the allowed configurations at a given step *point configuration constraints*.

Point configuration constraints eliminate many potential configuration paths. These constraints may create small additional restrictions, such as that a particular feature must always be selected. Complex step-based constraints may also be present, such as a particular automotive feature becomes unavailable after a specific time step (year) because the supplier discontinues it. Finally, a multi-step configuration problem may not dictate an exact starting and ending configuration, but merely a series of point configuration constraints that must hold for the start and end points of the configuration path. The myriad of possible point configuration constraints significantly increases the challenge of finding a valid configuration path for a multi-step configuration problem. Section 4.3 describes how MUSCLE models these constraints using a CSP, which enables a CSP solver to automatically derive solutions that adhere to these constraints and thus reduce tedious and error-prone manual configuration.

### 2.3 Challenge 3: Configuration Change/Edge Constraints

The automotive example in Figure 1 requires that developers adding new features spend no more than 35 million dollars in one year. The cost of adding/removing features can be captured as the length or weight of the edges connecting two transitions. For example, to transition directly from the starting configuration to the desired end configuration requires 88 million dollars and has an edge weight of 88.

Developers must not only find a path that reaches the desired end state without violating the point configuration constraints in Section 2.2, but also ensure that any constraints on the edges connecting successive configurations are met. Transitioning directly from the start configuration to end configuration would violate the edge constraint of the 35 million dollar yearly development budget. Edge constraints further reduce the number of valid paths and add complexity to the problem. Section 4.4 shows how these edge restrictions can be encoded as constraints on MUSCLE’s CSP variables to plan configuration paths that adhere to development budgets, which is hard to determine manually.

### 2.4 Challenge 4: Configuration Path Optimization

There may often be multiple correct configuration paths that reach the desired end point. In these cases, developers would like to optimize the path chosen, for example to

minimize total cost (the sum of the edge weights). In other cases, it may be more imperative to meet the desired end point constraints in as few time steps as possible.

For example, in Figure 3(b), developers have an initial development budget of 35 million dollars and then a subsequent yearly budget of 50 million dollars.

Although the cost of the path through intermediate configurations  $B_i$  and  $C_i$  is cheaper (70 million), developers may prefer to pass through  $B_0$  and  $C_0$  since they will already have a configuration that meets the end goals at  $C_0$ . Developers must therefore not only contend with numerous multi-step constraints, but must also perform complex optimizations on the properties of the configuration path. Section 4.5 shows how optimization can be performed on MUSCLE’s CSP formulation of multi-step configuration to allow developers to find the fastest and most cost-effective means of achieving a configuration goal.

## 3 A Formal Definition of Multi-step Configuration

This section presents a formal model of multi-step configuration. In its most general form, multi-step configuration involves finding a sequence of at most  $K$  configurations that satisfy a series of point configuration constraints and edge constraints. This definition requires the start and end configurations meet a set of point constraints, but does not dictate that there be a *single* valid starting and ending configuration.

**General formal model.** We define a multi-step configuration problem using the 6-tuple  $Msc = \langle E, PC, \Delta(F_T, F_U), K, F_{Start}, F_{end} \rangle$ , where:

- $E$  is the set of edge constraints, such as the maximum development cost per year for features,
- $PC$  is the set of point configuration constraints that must be met at each step, such as the feature model rules that developers may require to be adhered to across all steps (feature model rules do not have to be enforced at each time step),
- $\Delta(F_T, F_U)$  is a function that calculates the change cost or edge weight of moving from a configuration  $F_T$  at step  $T$  to a configuration  $F_U$  at step  $U$ ,
- $K$  is the maximum number of steps in the configuration problem,
- $F_{Start}$  is a set of configuration constraints on the starting configuration, such as a list of features that must initially be selected,
- $F_{end}$  is a set of configuration constraints on the final configuration, such as a list of features that must be selected or maximum cost of the final configuration.

We define a configuration path from step  $T$  over  $K$  steps as a  $K$ -tuple

$$P = \langle F_T, F_{T+1}, \dots, F_{T+K-1} \rangle$$

, where the configuration at step  $T$  is denoted by  $F_T$ . Each configuration,  $F_T$ , denotes the set of selected features at step  $T$ .

Section 4 shows how this formal model can be specified as a CSP. Although we use CSPs for reasoning on the formal model, we could also use SAT solvers, propositional logic, or other techniques to reason about this model. The formal model is thus applicable to a wide range of reasoning approaches.

### 3.1 Constraint and Function Examples

We now describe how the formal model presented above can be used to model typical SPL configuration constraints. We show how common configuration needs, such as the selection of specific features or budgetary constraints, can be mapped to portions of our multi-step configuration problem tuple.

**Edge constraint examples.** The set of edge constraints  $E$  can include numerous types of constraints on the transition from one configuration to another. For example, a constraint  $e_1 \in E$  may dictate that the maximum weight of any edge between successive configurations in  $F_T, F_{T+1} \in P$  have at most weight 35 (for the automotive problem from Figure 1):

$$\forall T \in (0..K-1), \Delta(F_T, F_{T+1}) \leq 35$$

Edge constraints may also vary depending on the step, for example a development budget may start at \$35 million and may expand as a function of the step:

$$\forall T \in (0..K-1), \Delta(F_T, F_{T+1}) \leq \frac{35}{1 - (.01 * T)}$$

Edge constraints may also be attached to specific time steps:

$$\forall T \in (0..4, 6..K-1), \Delta(F_T, F_{T+1}) \leq \frac{35}{1 - (.01 * T)}$$

$$\Delta(F_5, F_6) \leq 40$$

**Point configuration constraint examples.** The point configuration constraints specify properties that must hold for the feature selection at a given step. Both the starting and ending points for the multi-step configuration problem are defined as point configuration constraints on the first and last steps. For example, we want to start at a specific configuration  $F_{start}$  and reach another configuration  $F_{end}$ :

$$(F_0 = F_{start}) \wedge (F_K = F_{end})$$

Another general constraint  $pc_1 \in PC$  could require that for any step  $T$ , the feature selection  $F_T$  satisfies the feature model constraints  $Fc$ :

$$\forall T \in (0..K-1), F_T \Rightarrow Fc$$

Developers could also require that a specific set of features  $F_{start}$ , such as safety critical braking features, be selected at all times:

$$\forall T \in (0..K-1), F_{start} \subset F_T$$

**Change calculation function examples.** The function  $\Delta(F_T, F_U)$  calculates the cost of changing from one configuration to another configuration at a different step. For example, the following change calculation function computes the cost of changing from one configuration to another:

$$F_{added} = F_U - F_T$$

$$\Delta(F_T, F_U) = \sum f_i * c_i, f_i \in F_{added}$$

where  $f_i$  is the  $i_{th}$  added feature and  $c_i$  is the price of adding that feature.

## 4 A CSP Model of Multi-step Configuration

This section describes how MUSCLE uses CSPs to automatically derive solutions to multi-step configuration problems. To address the challenges outlined in Section 2 we show that deriving a configuration path for a multi-step configuration problem can be modeled as a CSP [9] using the formal framework from Section 3. After a CSP formulation of a multi-step configuration problem is built, MUSCLE can use a CSP solver to automatically derive a valid configuration path on behalf of the developer. Automating the configuration path derivation helps reduce the complexity from Challenge 1 in Section 2.1. Moreover, the CSP solver can be used to perform optimizations that would be extremely hard to achieve manually.

Prior work on automated feature model configuration [3, 15, 16] has yielded a framework for representing feature models and configuration problems as CSPs. This section shows how a new formulation of feature models and configuration problems can be developed that (1) incorporates multiple steps, (2) allows a constraint solver to derive a configuration path for evolving a feature selection over multiple intermediate steps to meet an end goal, (3) permits the specification of intermediate configuration constraints, (4) allows for change/edge constraints on the transition between feature selections, and (5) can be leveraged to optimize configuration path properties, such as path length or cost.

### 4.1 CSP Automated Configuration Background

A CSP is a set of variables and a set of constraints over the variables. For example,  $(X - Y > 0) \wedge (X < 10)$  is a simple CSP involving the integer variables  $X$  and  $Y$ . A constraint solver is an automated tool that takes a CSP as input and produces a *labeling* or set of values for the variables that simultaneously satisfies all of the constraints. The solver can also be used to find a labeling of the variables that maximizes or minimizes a function of the variables *e.g.* maximize  $X + Y$  yields  $X = 9, Y = 8$ .

A feature model can be modeled as a CSP through a series of integer variables  $F$ , where the variable  $f_i \in F$  corresponds to the  $i_{th}$  feature in the feature model. A configuration is defined as a series of values for these variables

Feature Model	CSP
	$(f_a = 1) \rightarrow (f_b = 1)$ $(f_b = 1) \rightarrow (f_a = 1)$
	$(f_b = 1) \rightarrow (f_a = 1)$
	$(f_a = 1) \rightarrow (f_b + f_c = 1)$ $(f_b = 1) \rightarrow (f_a = 1)$ $(f_c = 1) \rightarrow (f_a = 1)$

Figure 4: Mapping Feature Model Rules to a CSP

such that  $f_i = 1$  implies that the  $i_{th}$  feature is selected in the configuration. If the  $i_{th}$  feature is not selected,  $f_i = 0$ . Configuration rules from the feature model are represented as constraints over the variables in  $F$ , as shown in Figure 4.

More details on building a CSP from a feature model are described in [15, 3].

## 4.2 Introducing Multiple Steps into the CSP

The goal of automated configuration over multiple-steps is to find a configuration path that permutes a given starting configuration through a sequence of intermediate configurations to reach a desired end state. For example, the configuration paths in Figure 2 capture sequential modifications to the car configuration, shown in Figure 1, that will incorporate high-end features into the base automobile model. To reason about a configuration path over a span of steps, we first introduce a notion of a configuration step into MUSCLE’s CSP model of configuration.

**CSP model of configuration steps.** To introduce configuration steps into MUSCLE’s configuration CSP, we modify the configuration CSP formulation outlined in Section 4.1. We no longer use a variable  $f_i$  to refer to whether or not the  $i_{th}$  feature is selected or deselected. Instead, we refer to the selection state of each feature at a specific step  $T$  with the variable  $f_{iT}$ , *i.e.*, if the  $i_{th}$  feature is selected at step  $T$ ,  $f_{iT} = 1$ . We refer to an entire configuration at a specific step as a set of values for these variables,  $f_{iT} \in F_T$ . A solution to the CSP is configuration path defined by a labeling of all of the variables in the K-tuple:  $\langle F_T, F_{T+1} \dots F_{T+K-1} \rangle$ .

For example, if the ABS feature (denoted  $f_a$ ) is not selected at step  $T$  and is selected at step  $T + 1$ , then:

$$f_{aT} = 0$$

$$f_{aT+1} = 1$$

Figure 5 shows a visualization of how the  $f_{iT} \in F_T$  variables map to feature selections.

## 4.3 CSP Point Configuration Constraints

To address Challenge 2 from Section 2.2, the point configuration constraints (which are the constraints that define

Feature Model	CSP
	$f_0 = 1$
	$f_{1T} = 0$ $f_{1T+1} = 1$
	$f_{1T} = 1$ $f_{1T+1} = 0$

Figure 5: An Example of Variables Representing Feature Selection State at Specific Steps

what constitutes a valid intermediate configuration) can be modeled as constraints on the variables  $f_{iT} \in F_T$ . Each point configuration constraint has a specific set of steps,  $T_{pc}$ , during which it must be met, *i.e.*, the constraint must only evaluate to true on the precise steps for which it is in effect. For example, a simple constraint would be that the 2<sup>nd</sup> and 3<sup>rd</sup> configurations must have the feature  $f_1$  selected. The set of steps for which this constraint must hold would be  $T_{pc} = \{2, 3\}$ .

**CSP model of point configuration constraints.** A CSP point configuration constraint,  $pc_i \in PC$ , requires that:

$$\forall T \in T_{pc}, F_T \Rightarrow pc_i$$

Arbitrary point configuration constraints can be built using this model to restrict the valid configurations that are passed through by the configuration path. This flexible point configuration constraint mechanism allows developers to specify and automatically find solutions to problems involving the constraints from Challenge 2 in Section 2.2.

**CSP point configuration constraint example.** Assume that we want to find values for  $F_T \dots F_{T+K}$  such that we never violate any of the feature model constraints at any step. Further assume that the constraints in the feature model remain static over the  $K$  steps (feature model changes over multiple steps can also be modeled). If the  $j_{th}$  feature is a mandatory child of the  $i_{th}$  feature, we add the constraint:

$$\forall T \in (0 \dots K), (f_{iT} = 1) \Leftrightarrow (f_{jT} = 1)$$

That is, we require that at any step  $T$ , if the  $i_{th}$  feature ( $F_{iT}$ ) is selected, the  $j_{th}$  feature ( $f_{jT}$ ) is also selected. Furthermore, at any step  $T$ , if the  $j_{th}$  feature ( $F_{jT}$ ) is selected, the  $i_{th}$  feature ( $f_{iT}$ ) is also selected. Other example point configuration constraints can be mapped to the CSP as shown in Figure 6(a) and Figure 6(b).

Feature Model Over K Time Steps	CSP
	B is <b>always a</b> mandatory child of A forall T in (0..K): $(f_{aT} = 1) \rightarrow (f_{bT} = 1)$ $(f_{bT} = 1) \rightarrow (f_{aT} = 1)$
	B is <b>always an optional</b> feature of A forall T in (0..K): $(f_{bT} = 1) \rightarrow (f_{aT} = 1)$
	A <b>always requires</b> either B or C but not both forall T in (0..K): $(f_{aT} = 1) \rightarrow (f_{bT} + f_{cT} = 1)$ $(f_{bT} = 1) \rightarrow (f_{aT} = 1)$ $(f_{cT} = 1) \rightarrow (f_{aT} = 1)$

(a) Point Configuration Constraints for Feature Model Structure

Feature Model	CSP
	$\sum_{t=0}^K f_{1t} = K + 1$ $\sum_{t=0}^{K-1} d_{1t} = 0$
	$F_{1(T=0)} = 1$ and $\sum_{t=0}^{\infty} d_{1t} = 0$

(b) Point Configuration Constraint for Feature Selection

Figure 6: Point Configuration Constraint Examples

#### 4.4 CSP Edge/Change Constraints

Challenge 3 from Section 2.3 described how developers must be able to specify and adhere to constraints on the difference between two configurations at different steps. These edge/change constraints can be modeled in the CSP as constraints over the variables in two configurations  $F_T$  and  $F_U$ . By extending the CSP techniques we have developed in past work [16], we can specifically capture which features are selected or deselected between any two steps and constrain these changes via budget or other restrictions.

**CSP model of edge/change constraints.** To capture differences between feature selections between steps  $T$  and  $U$ , we create two new sets of variables  $S_{TU}$  and  $D_{TU}$ . These variables have the following constraints applied to them:

$$\begin{aligned} \forall s_{iTU} \in S_{TU}, (s_{iTU} = 1) &\Leftrightarrow (f_{iT} = 0) \wedge (f_{iU} = 1) \\ \forall d_{iTU} \in D_{TU}, (d_{iTU} = 1) &\Leftrightarrow (f_{iT} = 1) \wedge (f_{iU} = 0) \end{aligned}$$

If a feature is selected at time step  $T$  and not at time step  $U$ , then  $d_{iTU}$  is equal to 1. Similarly, if a feature is not selected at step  $T$  and selected at step  $U$ ,  $s_{iTU}$  is equal to 1.

An edge  $edge(T, U)$  between the configurations at steps  $T$  and  $U$  is defined as a 2-tuple:

$$edge(T, U) = \langle D_{TU}, S_{TU} \rangle$$

An edge is thus defined by the features deselected and selected to reach configuration  $F_U$  from configuration  $F_T$ . The

weight of the edge  $weight(edge(T, U))$  can then be calculated as a function of the edge tuple. For example, if the  $i_{th}$  feature costs  $c_i$  to add or remove then

$$weight(edge(T, U)) = \sum_{i=0}^n s_{iTU} * c_i + \sum_{i=0}^n d_{iTU} * c_i$$

**CSP edge/change constraint example.** The cost of including a particular feature may change over time. For example, the cost of adding a GPS guidance system to a car does not remain fixed, but instead typically decreases from one year to the next as GPS technology is commoditized. We can model and account for these changes in MUSCLE's CSP formulation and constrain the configuration path so that it adds features at times when they are sufficiently cheap. We will define an edge constraint that takes into account changing feature modification costs and limits the change in cost between two successive configurations to \$35 million dollars.

We assume we can calculate that the price of including the  $i_{th}$  feature so that it is included in the feature selection at step  $T$  by the function:

$$Cost(i, T) = \frac{c_i}{T + 1}$$

We can then define the cost of adding features to a configuration as:

$$weight(edge(T, T + 1)) = \sum_{i=1}^n (s_{iT+1} * Cost(i, T + 1))$$

We can now limit the cost of any two successive configurations via the edge constraint:

$$\forall T \in (0..K - 1), weight(edge(T, T + 1)) \leq 35$$

#### 4.5 Multi-step Configuration Optimization

Challenge 4 from Section 2.4 showed that optimizing the configuration path is an important issue. CSP solvers can automatically perform optimization while finding values for the variables in a CSP (though it may be impractical time-wise for some problems). We can define goal functions over the CSP variables to leverage these optimization capabilities and address Challenge 4.

In some cases, developers may not want to just find any configuration path that ends in the desired state. Instead, they may want a path that produces a configuration that meets the end goals as early as possible. For example, in the automotive problem from Section 1 developers may want to find a configuration path that meets their constraints and includes the high-end features in the base model in fewer than five years.

**CSP model of path length.** To support path length optimization, we define a measure of the number of steps needed to reach a valid end state. We must therefore determine if the constraints on the final configuration  $F_{end}$  (which is the goal state) are met by some configuration prior to the

last configuration ( $F_T$  where  $T < K - 1$ ). If we meet the final state constraints sooner than the final configuration, then we have found a configuration process that requires fewer configuration steps.

To track whether or not a configuration has met the constraints on the ending configuration  $F_{end}$ , we create a series of variables  $w_T \in W$  to represent whether or not the configuration  $F_T \in P$  satisfies  $F_{end}$ . For each configuration,  $F_T \in P$ , if  $F_{end}$  is satisfied:

$$(F_T \Rightarrow F_{end}) \Rightarrow (w_T = 1)$$

*i.e.*, if at any step (up to and including the last step) we satisfy the end state requirements, set  $w_T$  equal to 1. We also require that after one step has reached a correct ending configuration, the remaining steps also keep the correct configuration and do not alter it:

$$\begin{aligned} (w_T = 1) &\Rightarrow (w_{T+1} = 1) \\ (w_T = 1) &\Rightarrow (\sum_{i=0}^n s_{iTT+1} + \sum_{i=0}^n d_{iTT+1} = 0) \end{aligned}$$

**Optimization examples.** We can optimize to find the shortest configuration path to reach the goals over  $K$  steps by asking the solver to maximize:

$$\sum_{T=0}^{K-1} w_T$$

The reason that maximizing this sum will minimize the number of steps taken to reach the desired end state is that the sooner the state is reached, the more steps  $w_T$  will equal 1.

The most straightforward optimization functions are defined as functions of the variables in the configuration path  $P$ . For example, we can instruct the solver to minimize the cost of the ending configuration. Assume that the cost of  $i_{th}$  feature at step  $K$  is denoted by the variable  $c_i \in C_K$ , minimize  $C_K$ , where:

$$C_K = \sum_{i=0}^n f_i * c_i$$

Other optimizations can be performed on the weights of the edges. For example, to find the configuration path with the lowest development cost, where the development cost is the edge weight the goal is to minimize:

$$\sum_{T=0}^{K-1} weight(edge(T, T+1))$$

#### 4.6 A Complete Multi-step CSP Example

We now provide a complete mapping of the automotive configuration problem in Section 1 to MUSCLE's multi-step CSP. For this problem, the automotive developers want to include the high-end features into the base model over the course of five years ( $K = 5$ ). We first create a series of configuration variables to represent the feature selection at the end of each of the five years:

Feature Variables $F_T$	Point Constraint $pc_0$
Car	$f_{0T}$
Sensors	$f_{1T}$
Automated Driving Controller	$f_{2T}$
Collision Avoidance Braking	$f_{3T}$
Parallel Parking	$f_{4T}$
Lateral Range Finder	$f_{5T}$
Forward Range Finder	$f_{6T}$
Standard Avoidance	$f_{7T}$
Enhanced Avoidance	$f_{8T}$

Point Constraint $pc_0$	Constraints
$pc_0 =$ for all $T$ in $(0..4)$	$f_{0T} = 1$
	$(f_{0T} = 1) \leftrightarrow (f_{1T} = 1)$
	$(f_{2T} = 1) \rightarrow (f_{0T} = 1)$
	$(f_{2T} = 1) \leftrightarrow (f_{3T} = 1)$
	$(f_{3T} = 1) \rightarrow (f_{8T} + f_{7T} = 1)$
	$(f_{4T} = 1) \rightarrow (f_{2T} + f_{5T} = 2)$
	$(f_{5T} = 1) \rightarrow (f_{1T} = 1)$
	$(f_{6T} = 1) \rightarrow (f_{1T} = 1)$
	$(f_{7T} = 1) \rightarrow (f_{3T} = 1)$
	$(f_{8T} = 1) \rightarrow (f_{3T} + f_{6T} = 2)$

Figure 7: Point Configuration Constraints for the Automobile Example

$$\begin{aligned} F_0 &= (f_{00}, f_{10}, \dots, f_{80}) \\ F_1 &= (f_{00}, f_{11}, \dots, f_{81}) \\ &\dots \\ F_4 &= (f_{00}, f_{14}, \dots, f_{84}) \end{aligned}$$

The mappings of the automobile features from Figure 1 to CSP variables can be seen in Figure 7. A configuration path is defined by a set of feature selections for each of the five years:

$$P = \langle F_0, F_1, \dots, F_4 \rangle$$

The point constraint,  $pc_0 \in PC$ , ensures that the feature model constraints are met by each year's configuration, as shown in Figure 7. We must also specify the point configuration constraint for the starting configuration:

$$\begin{aligned} F_{start} &= (f_{00} = 1) \wedge (f_{10} = 1) \wedge (f_{20} = 0) \wedge (f_{30} = 0) \\ &\wedge (f_{40} = 0) \wedge (f_{50} = 0) \wedge (f_{60} = 0) \wedge (f_{70} = 0) \\ &\wedge (f_{80} = 0) \end{aligned}$$

Moreover, we must ensure that the high-end features are included in the last configuration:

$$\begin{aligned} F_{end} &= (f_{74} = 0) \wedge (f_{04} = 1) \wedge (f_{14} = 1) \wedge (f_{24} = 1) \wedge \\ &(f_{34} = 1) \wedge (f_{44} = 1) \wedge (f_{54} = 1) \wedge (f_{64} = 1) \\ &\wedge (f_{84} = 1) \end{aligned}$$

Our complete set of point configuration constraints is  $PC = (pc_0)$ .

Finally, we must specify how the change cost between two configurations is calculated and enforce the edge constraint that at most \$35 million dollars is spent per year.

$$\begin{aligned} \Delta(F_T, F_{T+1}) &= 20s_{2TT+1} + 14s_{3TT+1} + 19s_{4TT+1} + 8s_{5TT+1} \\ &\quad + 11s_{6TT+1} + 1s_{7TT+1} + 16s_{8TT+1} \\ E &= (\forall T \in (0..3), \Delta(F_T, F_{T+1}) \leq 35) \end{aligned}$$

Given this CSP formulation, we can use a constraint solver to automatically derive a solution to the multi-step automotive configuration problem described in Section 1.

## 5 Results

As described in Section 2.1, configuring an SPL over multiple steps is a highly combinatorial problem. An automated multi-step SPL configuration technique should be able to scale to hundreds of features and multiple steps. This section presents empirical results from experiments we performed to determine the scalability of MUSCLE.

Our experiments were performed with an implementation of the MUSCLE provided by the open-source Ascent Design Studio (available from [code.google.com/p/ascent-design-studio](http://code.google.com/p/ascent-design-studio)). The Ascent Design Studio’s implementation of MUSCLE is built using the Java Choco open-source CSP solver (available from [choco.sourceforge.net](http://choco.sourceforge.net)). The experiments were performed on a computer with an Intel Core DUO 2.4GHZ CPU, 2 gigabytes of memory, Windows XP, and a version 1.6 Java Virtual Machine (JVM). The JVM was run in server mode using a heap size of 40 megabytes (-Xms40m) and a maximum memory size of 256 megabytes (-Xmx256m).

To test the scalability of MUSCLE we needed 1,000s of feature models to test with, which posed a problem since there are not many large-scale feature models available to researchers. To solve this problem, we used a random feature model generator developed in prior work [16]. The feature model generator and code for these experiments is available in open-source form along with the Ascent Design Studio. We used a maximum branching factor of 5 children per feature and a maximum of 1/3 of the features were in an XOR group.<sup>1</sup>

We measured the solving time of MUSCLE by generating random multi-step configuration problems and solving for configuration paths that involved larger and larger numbers of steps. The problems were created by generating a semi-random feature model with 500 features as well as starting and ending configurations. MUSCLE was used to derive a configuration path between the two configurations.

Our experiments were performed with *large-scale configuration paths*, which were produced by forcing the solver to find a configuration path that involved switching between two children of the root feature that were involved in an XOR group. For a feature model with 500 features configured over 3 steps, the worst case solving time we observed was roughly 3 seconds. The worst case solving time for feature models configured over 10 steps was 16s. These initial results indicate that the technique should be sufficiently fast for feature models with hundreds of features.

## 6 Related Work

This section compares MUSCLE with related work.

---

<sup>1</sup>XOR feature groups are features that require the set of their selected children to satisfy a cardinality constraint (the constraint is 1..1 for XOR).

**Automated Single-step Configuration:** Several single-step feature model configuration and validation techniques have been proposed [2, 13, 1, 4, 5, 15]. These techniques use CSPs and propositional logic to derive feature model configurations in a single stage as well as assure their validity. These techniques help address the high complexity of finding a valid feature selection for a feature model that meets a set of intricate constraints.

While these techniques are useful for the derivation and validation of configurations in a single step, they do not consider feature configuration over the course of multiple steps. In scenarios, such as the automotive example from Section 1, the ability to reason about configuration over multiple steps is critical. MUSCLE provides this automated reasoning across multiple steps. Moreover, MUSCLE can also be used for single-step configurations since it is a special case of multi-step configuration where there is only one step  $K = 1$ .

**Staged Configuration:** Czarnecki et al. [7] describe a method for using staged feature selection to achieve a final target configuration. This multi-stage selection considers cases in which the selection of features in a previous stage impacts the validity of later stage feature selections. Our technique also examines the production of a feature model configuration over multiple configuration steps. MUSCLE is complementary to Czarnecki et al.’s work and provides a general formal framework that can be used to perform automated reasoning on staged configuration processes. Moreover, MUSCLE can also be used to reason about other multi-step configuration processes that do not fit into the staged configuration model, such as the the example from Section 1 where each step must reach a valid configuration.

Staged configuration can be modeled as a special instance of multi-step configuration. Specifically, staged configuration is an instance of a multi-step configuration problem where:  $E = \emptyset$ ,  $F_{start} = \emptyset$ ,  $F_{end} = (F_{K-1} \Rightarrow Fc)$ ,  $K$  is set to the number of stages,  $\Delta(F_T, F_U)$  is not defined, and  $Fc$  is the set of feature model constraints. That is, there are no limitations on the changes that can be made between successive configurations, the starting configuration has no features selected, and the ending configuration yields a valid feature model configuration. The staged configuration definition can be refined to guarantee that successive stages only add features:  $\forall T \in (0..K - 1), F_T \subset F_{T+1}$ .

Classen et al. [12] have investigated creating a formal semantics for staged configuration. Furthermore, they have provided a definition of a configuration path through a series of stages for a feature model. Whereas Classen et al. have focused on configuration paths that continually reduce variability, MUSCLE is a formal model that allows for both the reduction and introduction of variability in the configuration process. Moreover, MUSCLE allows for complete

configurations to be met at multiple points in the configuration process.

**Quality Attribute Evaluation:** Several techniques have been proposed for using quality attribute evaluation [8, 10, 14] to guide a configuration process. These techniques provide a framework for assessing the impact of each feature selection on the overall capabilities of the configured system. As a result, quality characteristics, such as reliability, can be taken into account when selecting features. These techniques are also designed for single step configuration processes. These techniques could be used in a complementary fashion to MUSCLE to produce the point configuration, edge, and other constraints in the multi-step configuration model.

## 7 Concluding Remarks

Many production SPL configuration problems require developers to evolve a configuration over multiple steps, rather than in a single iteration. Multi-step configuration, however, must take into account constraints on the change between successive configurations, such as the increase in cost of an automobile's configuration from one year to the next. Moreover, even though configuration is performed over multiple steps, a valid configuration must still be produced at the end of each year, further adding complexity while maintaining a functional system configuration.

It is hard to determine a sequence of feature model configurations and feature selections such that an initial configuration can be transformed into a desired target configuration. This paper introduces a technique, called the *MULTI-step Software Configuration problem solver* (MUSCLE), for modeling and solving multi-step configuration problems. MUSCLE represents the problem as a CSP, which enables CSP solvers to determine a path from a starting configuration to a target configuration. The output from MUSCLE is a valid sequence of feature selections that will lead from a starting configuration to the desired target configuration while also taking into account resource constraints.

Open-source implementations of MUSCLE are available in the Ascent Design Studio ([ascent-design-studio.googlecode.com](http://ascent-design-studio.googlecode.com)) and FAMA ([www.isa.us.es/fama](http://www.isa.us.es/fama))<sup>2</sup>.

## References

- [1] D. Batory. Feature Models, Grammars, and Propositional Formulas. *Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005: Proceedings*, 2005.
- [2] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.
- [3] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering*, Porto, Portugal, 2005. ACM/IFIP/USENIX.
- [4] D. Beuche. Variant Management with Pure::variants. Technical report, Pure-Systems GmbH, <http://www.pure-systems.com>, 2003.
- [5] R. Buhrdorf, D. Churchett, and C. Krueger. Salion's Experience with a Reactive Software Product Line Approach. In *Proceedings of the 5th International Workshop on Product Family Engineering*, Siena, Italy, November 2003.
- [6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, USA, 2002.
- [7] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004: Proceedings*, 2004.
- [8] L. Etxeberria and G. Sagardui. Variability Driven Quality Evaluation in Software Product Lines. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 243–252, 2008.
- [9] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, USA, 1989.
- [10] A. Immonen. A method for predicting reliability and availability at the architectural level. *Research Issues in Software Product-Lines-Engineering and Management*, T. Käkölä and JC Dueñas, Editors, 2005.
- [11] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering*, 5(0):143–168, January 1998.
- [12] B. Littlewood and L. Strigini. A Formal Semantics for Multi-level Staged Configuration. In *Proceedings of the Third Workshop on Variability Modelling of Software-intensive Systems*, pages 51–60, January 2009.
- [13] M. Mannion. Using first-order logic for product line model validation. *Proceedings of the Second International Conference on Software Product Lines*, 2379:176–187, 2002.
- [14] F. Olumofin and V. Misic. Extending the ATAM Architecture Evaluation to Product Line Architectures. In *IEEE/IFIP Working Conference on Software Architecture, WICSA*, 2005.
- [15] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt. Automating Product-Line Variant Selection for Mobile Devices. In *Proceedings of the 11th Annual Software Product Line Conference (SPLC)*, Kyoto, Japan, Sept. 2007.
- [16] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortez. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In *Proceedings of the Software Product Lines Conference (SPLC)*, Limerick, Ireland, Sept. 2008.

<sup>2</sup>This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533)