Automated Reasoning for Multi-step Feature Model Configuration Problems

J. White\textsuperscript{d,*}, D. Benavides\textsuperscript{e}, T. Saxena\textsuperscript{f}, B. Dougherty\textsuperscript{d}, D.C. Schmidt\textsuperscript{f}

\textsuperscript{d}Virginia Tech, Blacksburg, Virginia, USA
\textsuperscript{e}University of Seville, Seville, Spain
\textsuperscript{f}Vanderbilt University, Nashville, Tennessee, USA

Abstract

A key challenge of using a software product-line (SPL) to reuse software over long periods of time is reasoning about configuration changes over multiple steps. For example, an SPL for an automobile may have discrete model years that are released sequentially over a series of years and build on one another’s features. This paper presents an approach for modeling the configuration of an SPL over multiple steps, where each step yields a complete and valid configuration, as a constraint satisfaction problem. The paper provides the following contributions to the study of automated SPL configuration: 1) it shows how multi-step feature model configuration problems can be modeled as constraint satisfaction problems (CSPs); 2) the paper explains how changes in the feature model over time can be incorporated and reasoned about in the CSPs; 3) the paper describes how objective functions can be used to derive configuration paths over multiple steps to minimize overall product cost or other metrics; and 4) it presents empirical results demonstrating the scalability of the approach.

1. Introduction

The development and sustainment of software constitutes a large—and growing—expense in modern information and embedded systems, such as automobiles, mobile devices, cloud computing environments, and medical equipment [1]. The ability to reuse software across multiple development projects is one means to amortize the cost of software development and sustainment. Reusable software artifacts include design models, source code, test plans, and component architectures.

To reuse software systematically, organizations must employ techniques that facilitate not only the reuse of original software artifacts but also mass customization [2], which involves customization of software on a large-scale to handle a wide range of disparate tasks. Capturing customization opportunities, known as points of variability, is an important activity that enables developers to catalog the valid ways in which software artifacts can be reused. In addition to describing how software artifacts can be reused, it is essential to document the assumptions an artifact makes about its environment, as well as any constraints that preclude its reuse.

Corresponding author

Email addresses: julesw@vt.edu (J. White), benavides@us.es (D. Benavides), tsaxena@isis.vanderbilt.edu (T. Saxena), brianpd@vt.edu (B. Dougherty), schmdit@dre.vanderbilt.edu (D.C. Schmidt)
Software product-lines [3] (SPLs) are a paradigm for managing the complexity of tracking reusable software artifacts, describing their points of variability, and ensuring they are reused appropriately. A key part of an SPL is scope, commonality, and variability (SCV) analysis. The scope defines the collection of software artifacts that constitute the SPL. The commonality defines the attributes that are common across different sets of artifacts. The variability describes the differences that exist across the artifacts, such as various implementations and algorithms for different environments and/or requirements.

SPL’s use models to codify the results of SCV analysis [4]. A feature model [5] is a common type of models used to capture commonality and variability information in an SPL. A feature model describes points of commonality and variability in terms of features. Each feature represents a unit or increment in SPL functionality, ranging from high-level end-user capabilities (such as the presence of an anti-lock braking system in a car) to implementation details [6] (such as the usage of a specific software library).

A common format for a feature model is a tree that describes successive refinement of the variability in a product-line. For example, Figure 1 depicts the feature model of a car that contains configuration options for its sensors and advanced driving control capabilities. The car can contain different types of automated driving controllers, such as automated parallel parking or collision avoidance braking.

Each individual driving controller that the car can be customized with requires a different set of sensors and sensor software, e.g., the ParallelParking controller requires a LateralRangeFinder. These types of configuration rules are encoded into the hierarchical relationships in the tree. For example, the filled circle above CollisionAvoidance denotes that it is a required child feature of the AutomatedDrivingController feature.

To reuse software in a new context, developers use the feature model to determine how the SPL can be customized into a new variant. A variant is a complete and unique configuration of the SPL’s software artifacts. In a feature model, a variant is manifested as a selection of features that adheres to the configuration constraints captured in the feature relationships.

A core aspect of reusing software artifacts from an SPL is determining a complete and correct variant of the SPL that satisfies the target requirement set. For simple feature models, such as the one shown in Figure 1, developers can manually derive a selection of features for a variant. For more complex feature models—or in situations where cost optimization or resource constraints are involved—automated mechanisms are needed.

Prior research has developed a variety of automated techniques for deriving SPL variants to fit a requirement set. For example, some techniques the model feature selection problem as a constraint
satisfaction problem (which is a set of variables and a set of constraints over the variables) and use a general-purpose constraint solver (which is an automated tool for finding solutions to these problems) to derive a suitable variant [7, 8]. Other research has modeled feature selection problems as boolean satisfiability (SAT) problems or grammars and used SAT solvers to derive variants [9, 10, 11, 12]. The common aspect of this prior research is that one configuration is derived that satisfies a set of requirements in a single step.

**Open problems.** Not all software reuse scenarios are well-suited to a single-step approach for choosing an SPL variant. In some cases, product features must be introduced gradually over a series of steps. For example, an automotive manufacturer may want to include the automated driving features shown in Figure 1 into the base model of the car over a series of years. Adding these advanced features in a single year to the base model would increase costs excessively and suppress consumer demand.

Ideally, an automotive manufacturer would like to derive a sequence of successive configurations that build upon one another so that more advanced features are included each year. A manufacturer, however, cannot arbitrarily choose features to add in a given model year. Instead, each set of features for a model year must constitute a complete and correct configuration of the SPL to avoid selling a defective and non-viable configuration.

Further complicating this scenario is that a manufacturer is constrained in its introduction of features. For example, a manufacturer must introduce features in a manner that ensures no two successive model years differ by more than the price increase a consumer is willing to pay from one year to the next. Not only must the individual model year configurations be correct, therefore, but the delta between any two successive configurations must be valid.

This process of producing a series of intermediate configurations between a starting configuration and a desired ending configuration—*i.e.*, a **configuration path**—is shown in Figure 2. This sequence of
Moreover, the costs of introducing features may vary over the steps (e.g., as suppliers lower costs from one year to the next), making it hard to identify exactly the right step to introduce a feature.

**Solution overview and contributions.** 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* (MUSCLES). MUSCLES transforms multi-step feature configuration problems into *Constraint Satisfaction Problems* (CSPs) [13]. Once a CSP has been produced for the problem, MUSCLES uses a constraint solver to generate a series of configurations that meet the multi-step constraints.

This paper extends our prior work on automated multi-step configuration of software product-lines [14]. The paper presents a new approach for handling *feature model drift*, which is change in the constraints of a feature model over time. We present a formal mapping of feature model drift to a CSP and so that multi-step configuration problems involving non-constant product-lines can be automated. We also show how ordering and branching constraints can be applied to models of feature model drift.

The 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 present methods for modeling feature model drift as a feature model changes over time.
5. We describe mechanisms for optimally deriving a set of configurations that meet the requirements and minimize or maximize a property (such as total configuration cost) of the configurations or configuration process.
6. 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.
7. We present empirical results from experiments that demonstrate that MUSCLES can scale to feature models with hundreds of features and configured over multiple steps.

**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 multi-step configuration; Section 4 explains MUSCLES’s CSP-based automated multi-step configuration reasoning approach; Section 6 analyzes empirical results from experiments that evaluate the scalability of MUSCLES; Section 7 compares MUSCLES with related work; and Section 8 presents concluding remarks.

**2. Multi-step SPL Configuration 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).

Each successive series of points represents potential configurations of the feature model at a given step. For example, the configurations $B_0 \ldots B_i$ represent the intermediate configurations that can be reached in one step from the starting configuration. This section uses the graph formulation of the problem’s solution space to showcase the challenges of finding valid solutions.
2.1. Challenge 1: Graph Complexity

Developers attempting to derive solutions to multi-step configuration problems manually or via a graph algorithm face 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) and thus $2^n$ 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 $2^n - 1$ points at step $T + 1$ in the worst case 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 MUSCLES uses CSP-based automation to eliminate the need for developers to find solutions to these multi-step configuration problems manually, thereby minimizing configuration time and effort.

2.2. Challenge 2: Point Configuration Constraints

To reason about configuration over multiple steps, developers must ensure that at each step the configuration is in a valid state, i.e., the feature selection of the configuration should not violate the rules in the feature model. To plan the long-term configuration strategy, therefore, developers must devise a series of valid configurations that incrementally build upon one another while moving towards a desired end goal.

Figure 1 shows an example configuration problem with time for a low-end model car that has no automated driving capabilities. In three years, the manufacturer would like to migrate the automated driving features from a high-end model as standard features on the base model. The manufacturer’s costs (in millions) to add each feature to the base model is shown in the Cost to Add Features table in Figure 1. The manufacturer has budgeted at most 35 million dollars per year to add features to the car. The manufacturer would like to know what features to add each year to reach the three year goal without exceeding the budget or creating an invalid configuration in any year.

Although there are many potential intermediate configurations that could be used to reach the desired automobile configuration, most configurations will not meet developer requirements. For example, many of the $K2^n$ arbitrary subsets of feature selections 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. These point configuration constraints limit the
allowed configurations at a given step. The example in Figure 1 has 243 \(3^5\) different configuration paths that could be used to reach the end goal, although few of them are correct.

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 must be selected by a specific step so that manufacturer will be the first to market with that capability.

In addition, a multi-step configuration problem should 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 MUSCLES models these constraints using a CSP, which enables a CSP solver to derive solutions automatically that adhere to these constraints, thereby avoiding 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 MUSCLES’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, e.g., 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, e.g., 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 MUSCLES’s CSP formulation of multi-step configuration so developers can find the fastest and most cost-effective means of achieving a configuration goal.

2.5. Challenge 5: Feature Model Drift

Over time, a feature model will invariably need readjusting to account for changing external conditions (such as the newly released software features from vendors, deprecated APIs, or newly discovered bugs).

which we call *feature model drift*. In the simplest case, new features are added to the feature model. In more challenging scenarios, it may be necessary to remove features from the feature model or add new constraints between features to the model.

For example, the vendor that provides the software for the Collision Avoidance Braking controller, shown in Figure 1, may be bought by a competitor that intends to discontinue selling the existing software controller in two years. In place of the existing controller, a newer controller will be offered that is much more expensive and uses a different forward range finding sensor. In two years when the existing software controller is discontinued, developers must update the feature model to include the new range finder type and move the existing requires constraint to point to the new sensor type. As shown in this example, feature model drift substantially complicates the process of finding a sequence of configurations that will both meet the requirements of each configuration checkpoint and the end configuration goal. Section 5.1 shows how MUSCLES’s CSP representation of multi-step configuration can be modified to account for feature model drift.

3. A Formal Definition of Multi-step Configuration

This section presents a formal model of multi-step configuration used by MUSCLES to derive valid configuration paths of SPLs. 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 $Msc = < E, PC, \Delta(F_T,F_U), K, F_{Start}, F_{end} >$, 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

$$P = < F_T, F_{T+1}, \ldots, F_{T+K-1} >$$

, 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:

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

Edge constraints may also be attached to specific time steps:

$$\forall T \in (0..4, 6..K-1), \Delta(F_T, F_{T+1}) \leq \frac{35}{1-(.01 \cdot T)} \quad \Delta(F_5, F_6) \leq 40$$

**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 $F_{\text{start}}$ and reach another configuration $F_{\text{end}}$:

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

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 $F_{\text{start}}$, such as safety critical braking features, be selected at all times:

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

**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:

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

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

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 MUSCLES uses CSPs to derive solutions to multi-step configuration problems automatically. To address the challenges outlined in Section 2 we show how deriving a configuration path for a multi-step configuration problem can be modeled as a CSP [13] using the formal framework from Section 3. After a CSP formulation of a multi-step configuration problem is created, MUSCLES can use a CSP solver to derive a valid configuration path automatically, which addresses Challenge 1 in Section 2.1. Moreover, the CSP solver can be used to perform optimizations that would be hard to achieve manually.

Prior work on automated feature model configuration [15, 8, 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 to (1) incorporate multiple steps; (2) allow a constraint solver to derive a configuration path for evolving a feature selection over multiple intermediate steps to meet an end goal; (3) permit the specification of intermediate configuration constraints; (4) allow for change/edge constraints, which govern the selection/deselection of feature over time; and (5) 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, \((X - Y > 0) \land (X < 10)\) 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 (which is a set of values) for the variables that simultaneously satisfies all 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 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 information on creating a CSP from a feature model are described in [8, 15].

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 MUSCLES’s CSP model of configuration.

**CSP model of configuration steps.** To introduce configuration steps into MUSCLES’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, \( f_T \in F_T \). A solution to the CSP is configuration path defined by a labeling of all of the variables in the K-tuple: \(< F_T, F_{T+1} \ldots F_{T+K-1} >\).

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

\[
\begin{align*}
  f_{aT} &= 0 \\
  f_{aT+1} &= 1
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Feature Model</th>
<th>CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_0 )</td>
<td>( f_0 = 1 )</td>
</tr>
<tr>
<td>( f_{iT} )</td>
<td>( f_{iT} = 0 ) ( f_{iT+1} = 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature Model</th>
<th>CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1 )</td>
<td>( f_1 = 1 ) ( f_{1T+1} = 0 )</td>
</tr>
</tbody>
</table>

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

### 4.3. CSP Point Configuration Constraints

To address Challenge 2 from Section 2.2, the point configuration constraints (which are the constraints that define 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\(^{nd}\) and 3\(^{rd}\) 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 $F_T \ldots F_{T+K}$ 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:

$$\forall T \in (0 \ldots K), \ (f_{iT} = 1) \iff (F_{jT} = 1)$$

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. Moreover, 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).

<table>
<thead>
<tr>
<th>Feature Model Over K Time Steps</th>
<th>CSP</th>
</tr>
</thead>
</table>
| A                              | B is always a mandatory child of A | forall T in (0..K):
|                                | $(f_{iT} = 1) \Rightarrow (f_{jT} = 1)$ |
|                                | $(f_{jT} = 1) \Rightarrow (f_{iT} = 1)$ |
| B                              | B is always an optional feature of A | forall T in (0..K):
|                                | $(f_{iT} = 1) \Rightarrow (f_{jT} = 1)$ |
| A                              | A always requires either B or C but not both | forall T in (0..K):
|                                | $(f_{iT} = 1) \Rightarrow (f_{jT} + f_{cT} = 1)$ |
|                                | $(f_{jT} = 1) \Rightarrow (f_{iT} + f_{cT} = 1)$ |
|                                | $(f_{cT} = 1) \Rightarrow (f_{iT} = 1)$ |
|                                | $(f_{iT} = 1) \Rightarrow (f_{jT} = 1)$ |

(a) Point Configuration Constraints for Feature Model Structure

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 change/edge 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 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:

$$\forall s_{ITU} \in S_{TU}, \ (s_{ITU} = 1) \iff (f_{iT} = 0) \land (f_{IU} = 1)$$

$$\forall d_{ITU} \in D_{TU}, \ (d_{ITU} = 1) \iff (f_{iT} = 1) \land (f_{IU} = 0)$$

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:

$$edge(T,U) =< D_{TU}, S_{TU} >$$
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 $\text{weight}(\text{edge}(T, U))$ 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

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

**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 MUSCLES’s CSP formulation and constrain the configuration path so that it adds features at times when they are sufficiently cheap. We thus define an edge constraint that accounts for changing feature modification costs and limits the change in cost between two successive configurations to $35 \text{ million dollars}$.

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:

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

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

$$
\text{weight}(\text{edge}(T, T+1)) = \sum_{i=1}^{n} (s_{iTT+1} \cdot \text{Cost}(i, T+1))
$$

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

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

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 $T < K - 1$). We have found a configuration process that requires fewer configuration steps if we meet the final state constraints sooner than the final configuration.

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:

\[
(w_T = 1) \Rightarrow (w_{T+1} = 1)
\]

\[
(w_T = 1) \Rightarrow (\sum_{i=0}^{n} s_{iT+1} + \sum_{i=0}^{n} d_{iT+1} = 0)
\]

**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 minimizes 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:

\[
C_K = \sum_{i=0}^{n} f_i \ast c_i
\]

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:

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

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

![Figure 7: Point Configuration Constraints for the Automobile Example](image)

We now provide a complete mapping of the automotive configuration problem in Section 1 to MUS-CLES’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 year:

\[
F_0 = (f_{00}, f_{10}, \ldots f_{80})
\]

\[
F_1 = (f_{00}, f_{11}, \ldots f_{81})
\]

\[
F_2 = (f_{00}, f_{12}, \ldots f_{82})
\]

\[
F_3 = (f_{00}, f_{13}, \ldots f_{83})
\]

\[
F_4 = (f_{00}, f_{14}, \ldots f_{84})
\]
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:

\[ P = \langle F_0, F_1, \ldots, F_4 \rangle \]

The point constraint, \( p_{c0} \in PC \), 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:

\[
F_{\text{start}} = (f_{00} = 1) \land (f_{10} = 1) \land (f_{20} = 0) \land (f_{30} = 0) \\
\land (f_{40} = 0) \land (f_{50} = 0) \land (f_{60} = 0) \land (f_{70} = 0) \\
\land (f_{80} = 0)
\]

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

\[
F_{\text{end}} = (f_{74} = 0) \land (f_{04} = 1) \land (f_{14} = 1) \land (f_{24} = 1) \land (f_{34} = 1) \land (f_{44} = 1) \land (f_{54} = 1) \land (f_{64} = 1) \land (f_{84} = 1)
\]

Our complete set of point configuration constraints is \( PC = (p_{c0}) \).

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.

\[
\Delta(F_T, F_{T+1}) = 20s_{2TT+1} + 14s_{3TT+1} + 19s_{4TT+1} + 8s_{5TT+1} \\
+ 11s_{5TT+1} + 1s_{7TT+1} + 16s_{8TT+1}
\]

\[
E = (\forall T \in (0..3), \Delta(F_T, F_{T+1}) \leq 35)
\]

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. Feature Model Drift

When configuration occurs over multiple steps, the configuration process may span a substantial period of time. For example, the automotive development example from Section 1, where automated driving is being added to a car, spans several years. In most multi-step configuration problems, developers reason about configuration over a span of days, months, or years.

Configuration time frames that span months or years introduce the possibility for feature model drift. Feature model drift is the evolution of a feature model, through the addition or removal of features and constraints, after the initial configuration step. Automotive manufacturers may rely on suppliers that plan to introduce new features in a component at a specific time. Moreover, suppliers may plan to discontinue support for older features in the future.

In many cases, developers know ahead of time which features will be introduced or discontinued. Moreover, developers often have an estimate of when the availability of the feature will change based on information provided by a supplier or other mechanism. This data on feature addition and removal times allows developers to incorporate this knowledge into the construction of a multi-step configuration problem. This section describes how feature model drift can be accounted for in a multi-step configuration CSP.
5.1. Modifying the CSP Model of Multiple Steps

In the original formulation of the CSP, the set of features that are present does not change over time. To account for feature model drift, we show how we can relax our requirement from Section 4.3 that feature model constraints remain static. Once feature model constraint changes over multiple steps are modeled in the CSP, the solver can derive a configuration path that respects the feature model constraints as they drift. This eliminates the burden on developers to derive configuration paths that must meet complex drifting feature model requirements.

As we showed in Section 2.3, we constrain the feature selection variables \( F_T \) to respect the feature model constraints. Since each variable represents the selection state of a feature at a specific step, we do not have to apply the same constraints to every step. For example, assume that a software vendor for the automotive manufacturer announces that in two years, its software package must be purchased with a currently optional feature. If the \( j_{th} \) feature is an optional child of the \( i_{th} \) feature (the software package) at step \( T \) and at step \( K \), the \( j_{th} \) feature becomes mandatory, we can model this as:

\[
(f_{jT} = 1) \implies (f_{iT} = 1)
\]

At Step \( K \), the \( j_{th} \) feature becomes mandatory, changing the constraints on selection of the feature:

\[
(f_{iK} = 1) \implies (f_{jK} = 1) \quad (f_{jK} = 1) \implies (f_{iK} = 1)
\]

That is, at step \( T \), if \( f_i \) is selected \( (f_{iT} = 1) \) there is no constraint requiring \( f_j \) to be selected. At step \( K \), however, there is the constraint that \( (f_{iK} = 1) \implies (f_{jK} = 1) \), which makes \( f_j \) mandatory.

Examples of other feature model drifts as CSP constraints are shown in Figure 8.

![Figure 8: A CSP Model of Feature Model Drift](image)

5.2. Feature Drift Epochs

Because feature model drift may take place far in the future, it may not always be possible to precisely predict the time step at which a particular feature becomes available. For example, a supplier may
indicate that in the next 3-5 years, they plan to phase out the usage of a particular component. In these scenarios, SPL engineers need a way to be able to reason about configuration and place bounds, rather than exact times, on feature model drift.

The formal model of feature model drift that we have presented can be extended to account for these types of inexact timeframes on the drift of a feature model. Feature model drift is a change to a feature model at a future point in time. We introduce a new concept, which we call the change epoch, which is the period of time during which a change due to feature model drift is in effect.

Each change epoch includes both a start time and a duration. For example, a supplier may phase out a component in 3-5 years, causing the feature model to have several modifications. Let, \( E_i \) be the change epoch of the \( i \)th set of changes that need to be applied to the feature model as a result of feature model drift. When the \( E_i \) change epoch is in effect, it means that its starting point is \( E_i^{\text{start}} \) and \( 3 \leq E_i^{\text{start}} \leq 5 \). The duration of the epoch, \( E_i^{\text{dur}} \), is \( E_i^{\text{dur}} = \infty \).

The key change that needs to be made to the formal model of feature model drift is to introduce constraints to bound, rather than exactly specify, the values for \( E_i^{\text{start}} \) and \( E_i^{\text{dur}} \). We introduce the function,

\[ S(E_i^{\text{start}}, E_i^{\text{dur}}, F_0, F_1, \ldots, F_{\text{end}}) \]

to determine the beginning of a change epoch as a value of time and the configurations of the feature model at each step. For example, if a supplier was expected to phase out a part 3-5 years in the future, then:

\[ 3 \geq W(E_i^{\text{start}}, E_i^{\text{dur}}, F_0, F_1, \ldots, F_{\text{end}}) \geq 5 \]

Similarly, a separate function,

\[ S(E_i^{\text{dur}}, E_i^{\text{dur}}, F_0, F_1, \ldots, F_{\text{end}}) \]

calculates the duration of the change epoch. In the case of a part phased out of existence, the duration of the change epoch would be indefinite, or:

\[ S(E_i^{\text{dur}}, E_i^{\text{dur}}, F_0, F_1, \ldots, F_{\text{end}}) = \infty \]

5.3. Epoch-based Feature Model Constraints

The concept of feature model drift epochs introduces the situation that the exact point in time that specific feature model constraints will be in effect is not known. Instead, constraints are placed upon when the feature model drift epochs will occur and their duration. In order to account for epochs in the multi-step configuration CSP, additional constraints must be added. In the previous examples, if the \( j_{ih} \) feature is an optional child of the \( i_{ih} \) feature (the software package) at step \( T \) and at step \( K \), where \( 3 \geq K \geq 5 \), the \( j_{ih} \) feature becomes mandatory, we can model this as:

\[ (f_{jT} = 1) \Rightarrow (f_{iT} = 1) \]

At Step \( K \), the \( j_{ih} \) feature becomes mandatory, changing the constraints on selection of the feature:

\[ (f_{iK} = 1) \Rightarrow (f_{jK} = 1) \]
\[ (f_{jK} = 1) \Rightarrow (f_{iK} = 1) \]

Now, assume that the \( j_{ih} \) feature is an optional child of the \( i_{ih} \) feature (the software package) at the start and at some step, \( K \), where \( 3 \geq K \geq 5 \), the \( j_{ih} \) feature becomes mandatory, we can no longer directly
model this as before. Instead, we must define the enforcement of the new feature model constraint in terms of its feature drift epoch. In this situation, we model this as:

\[(f_j^T = 1) \Rightarrow (f_i^T = 1)\]

If Step \(K\) is within the time period of the feature drift epoch, the \(j\)th feature becomes mandatory, changing the constraints on selection of the feature:

\[((f_i^K = 1) \Rightarrow (f_j^K = 1)) \iff (E_i^{start} + E_i^{dur} \geq K)\]
\[((f_j^K = 1) \Rightarrow (f_i^K = 1)) \iff (E_j^{start} + E_j^{dur} \geq K)\]

where:

\[3 \leq E_i^{start} \leq 5\]

Using the concept of a feature model epoch, developers can encode ambiguity into the feature model drift. Developers can model periods of time during which changes are expected and reason about how variations in when those epochs occur will impact configuration. Most importantly, feature model epochs allow developers to create configuration scenarios that more closely mirror the uncertainty in real-world development at when a particular feature will be completed and become part of a feature model.

### 5.4. Ordered Epochs

Another issue that developers face is that the development or deprecation of a feature from a feature model is dependent upon the development or deprecation of several other features. For example, developers may know that the next generation of a mobile phone platform is going to support connectors that can communicate with an automobile’s CAN bus. Within 1 year from the time that this new mobile phone platform is developed, they will be able to develop a diagnostic interface for the car on the same mobile platform.

In this scenario, the development of the mobile phone diagnostic interface feature is dependent upon the occurrence of the mobile platform’s CAN bus feature. The exact point in time at which the diagnostic interface feature will be developed is only known relative to the occurrence of another epoch. We term these types of epoch constraints, ordered epochs.

Using the modified model of multi-step configuration, we can defined an ordered by constraining an epoch’s start, \(E_i^{start}\), and duration, \(E_i^{dur}\), in terms of another epoch, \(E_j\). For example, if we wish to define the epoch, \(E_j\), as occurring at least two steps after the epoch, \(E_i\), we can say:

\[E_j^{start} \geq E_i^{start} + E_i^{dur}\]

### 5.5. Feature Drift Branches

Using these CSP constraints, developers can encode ordering into the occurrence of epochs. Another key attribute of epoch ordering is the ability to encode branching into the occurrence of epochs. For example, developers may know that they will develop one of two different sets of features, but not both. For example, developers might develop a mobile automobile diagnostic interface or a in-car LCD diagnostic panel, but not both.
To encode branching constraints into feature model drift, developers can use the $E_{start}^{i}$ variable to encode branching constraints. For example, if the changes described by the $i_{th}$ feature model drift are mutually exclusive with the changes in $j_{th}$ feature model drift, this constraint can be encoded as:

$$E_{start}^{i} \geq 0 \iff E_{start}^{j} = -1$$

$$E_{start}^{j} \geq 0 \iff E_{start}^{i} = -1$$

where, $E_{start}^{j} = -1$ indicates that the $j_{th}$ feature model drift never is in effect. Using this same strategy, arbitrary constraints on the branching of feature model drift can be encoded into the CSP.

6. Evaluating the Scalability of MUSCLES

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 MUSCLES.

Our experiments were performed with an implementation of the MUSCLES provided by the open-source Ascent Design Studio (available from code.google.com/p/ascent-design-studio). The Ascent Design Studio’s implementation of MUSCLES is built using the Java Choco open-source CSP solver (available from 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 MUSCLES we needed thousands 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 also 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.1

We measured the solving time of MUSCLES 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. MUSCLES 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 $\sim$3 seconds. The worst case solving time for feature models configured over 10 steps was 16 seconds. These initial results indicate that the technique should be sufficiently fast for feature models with hundreds of features.

---

1XOR feature groups are features that require the set of their selected children to satisfy a cardinality constraint (the constraint is 1..1 for XOR).
7. Related Work

This section compares MUSCLES with related work on automated single-step configuration, staged configuration, legacy configuration evolution, quality attribute evaluation, and step-wise refinement.

**Automated single-step configuration.** Several single-step feature model configuration and validation techniques have been proposed [7, 9, 10, 11, 12, 8]. 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 many production scenarios (such as the automotive example from Section 1) the ability to reason about configuration over multiple steps is critical. MUSCLES provides this automated reasoning across multiple steps. Moreover, MUSCLES can be used for single-step configurations since it is a special case of multi-step configuration with only one step $K = 1$.

**Staged configuration.** Czarnecki et al. [17] describe a method for using staged feature selection to achieve a final target configuration. Their multi-stage selection considers cases in which the selection of features in a previous stage impacts the validity of later stage feature selections.

MUSCLES is complementary to Czarnecki et al.’s work since it (1) examines the production of a feature model configuration over multiple configuration steps and (2) provides a general formal framework that can be used to perform automated reasoning on staged configuration processes. Moreover, MUSCLES 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, i.e., 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. [18] have investigated creating a formal semantics for staged configuration. Moreover, they provide a definition of a configuration path through a series of stages for a feature model. Whereas Classen et al. focus on configuration paths that continually reduce variability, MUSCLES is a formal model that allows for both the reduction and introduction of variability in the configuration process. Moreover, MUSCLES can produce a complete configuration at multiple points in the configuration process.

**Legacy configuration evolution.** A legacy configuration problem involves a previously valid feature selection that no longer satisfies the current feature model constraints. The goal is to find a way of modifying the legacy configuration to satisfy the new feature model constraints. For example, an automotive manufacturer may have been informed that in two years a supplier is discontinuing the part needed to produce a specific feature, so the manufacturer must find a way of modifying the current configurations to eliminate the unavailable feature. The problem is a refinement of a multi-step configuration problem where:

$$F_{start} = \emptyset, F_{end} = (F_{K-1} \Rightarrow Fc)$$
\[ K = \text{maximum time to adhere to new feature model requirements} \]

\[ E \text{and } \Delta(F_T, F_U) \text{ may or may not be defined} \]

These methods are similar to MUSCLES in that they guarantee that should a valid solution exists, it will be determined. These methods, however, require an exhaustive search of the configuration space, resulting in an exponential runtime. While MUSCLES shares this worst case runtime, it can add multiple features in a single step, thus reducing the amount of constraint validation required. Moreover, our technique uses heuristics to determine which variable(s) to instantiate, resulting in less backtracking in most cases.

**Quality attribute evaluation.** Several techniques have been proposed for evaluating quality attributes [19, 20, 21] 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 MUSCLES to produce the point configuration, edge, and other constraints in the multi-step configuration model.

8. **Concluding Remarks**

Many production SPL configuration problems require developers to evolve a configuration over multiple steps, rather than in a single step. Multi-step SPL 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 step (e.g., prior to shipping the new year’s model car), which further complicates 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 (MUSCLES)**, for modeling and solving multi-step configuration problems. MUSCLES 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 MUSCLES is a valid sequence of feature selections that will lead from a starting configuration to the desired target configuration, while accounting for resource constraints.

The following are lessons learned from our efforts examining multi-step configuration using MUSCLES:

- **SPL multi-step optimization.** Multi-step optimizations can be performed to minimize both the number of time steps and the resource consumption required to reach a target SPL configuration.
- **Multi-step SPL configuration complexity.** There exists a worst case exponential number of intermediate configurations for each time step. It is therefore paramount to employ an algorithm that does not rely on exhaustive exploration. Our future work plans to improve our solving methods to increase performance.
- **Multi-step SPL CSP linear scaling.** Empirical data has shown that our technique scales linearly for varying numbers of time steps. Our future work will therefore test the scalability of MUSCLES in response to other factors, including problem size and tightness of constraints.
Open-source implementations of MUSCLES are available in the Ascent Design Studio (ascent-design-studio.googlecode.com) and FAMA (www.isa.us.es/fama).²


²This 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).


