Evolving feature model configurations in software product lines

Jules White [a,*], José A. Galindo [b], Tripti Saxena [a], Brian Dougherty [a], David Benavides [b], Douglas C. Schmidt [a]

[a] *Vanderbilt University, Nashville, TN, USA*
[b] *University of Seville, Seville, Spain*

Keywords:
Software product line
Feature model
Multi-step configuration

A B S T R A C T

The increasing complexity and cost of software-intensive systems has led developers to seek ways of reusing software components across development projects. One approach to increasing software reusability is to develop a software product-line (SPL), which is a software architecture that can be reconfigured and reused across projects. Rather than developing software from scratch for a new project, a new configuration of the SPL is produced. It is hard, however, to find a configuration of an SPL that meets an arbitrary requirement set and does not violate any configuration constraints in the SPL.

Existing research has focused on techniques that produce a configuration of an SPL in a single step. Budgetary constraints or other restrictions, however, may require multi-step configuration processes. For example, an aircraft manufacturer may want to produce a series of configurations of a plane over a span of years without exceeding a yearly budget to add features.

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 SPL configuration problems can be automatically derived with a constraint solver by mapping them to CSPs. Moreover, we show how feature model changes can be mapped to our approach in a multi-step scenario by using feature model drift. Third, we present empirical results demonstrating that our CSP-based reasoning technique can scale to SPL models with hundreds of features and multiple configuration steps.

1. Introduction

The development and sustainment of software constitutes a large – and growing – expense in modern information and embedded systems, such as avionics, mobile devices, cloud computing environments, and medical equipment (Boehm, 1975). 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.

Reuse of software components is a common task in the mass customization of software products (Pohl et al., 2005). Mass customization refers to the adaptation of a software product to a variety of different users by varying the features in the software in order to satisfy their needs. Capturing customization opportunities, known as *variability points*, 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.

Software product-lines (SPLs) (Clements et al., 2001) are a paradigm for managing the complexity of tracking and creating reusable software artifacts, as well as 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.

SPLs use models to codify the results of SCV analysis (Coplien et al., 1998). A *feature model* (Kang et al., 1998) is used to capture commonality and variability information existing 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

* Corresponding author. Tel.: +34667368095.
E-mail addresses: jules@dre.vanderbilt.edu (J. White), jagalindo@us.es (J.A. Galindo), tsaxena@isis.vanderbilt.edu (T. Saxena), briand@dre.vanderbilt.edu (B. Dougherty), benavides@us.es (D. Benavides), schmdit@dre.vanderbilt.edu (D.C. Schmidt).

(such as the presence of an entertainment system on an aircraft) to implementation details (Metzger et al., 2007) (such as the usage of a specific software library).

A common format for a feature model is a tree that describes successive refinements of the variability in a product-line. For example, Fig. 1 depicts the feature model of a flight avionics system that contains configuration options for its sensors and flight avionics navigation capabilities.

The plane can contain different types of advanced navigation systems, such as `Inertial Navigation` or `GPS`.

Each individual advanced navigation avionics system that the aircraft can be customized with requires a different set of sensors and software, *e.g.*, the `Laser Gyro` software requires `Laser Gyro Hardware`. These types of configuration rules are encoded into the hierarchical relationships in the tree. For example, the filled circle above `Inertial Nav.` denotes that it is a required child feature of the `Adv. Nav. Avionics` feature.

An example of where software reuse can play a role is in software evolution, which is a key task in software engineering. Software evolution is the process of determining how existing software can be adapted to support new customer requirements. Lenhman et al. (Lehman, 1996), for example, have explored how software reuse can be employed in software evolution. In particular, their work showed that there is often a set of evolution rules that must be adhered to during the evolution process. In software product-lines, software functionality is usually encapsulated in components, enabling the reuse of code between different versions of the software through these components. Because these software components cannot be arbitrarily composed, rules are needed to specify how features can be composed across multiple steps.

To reuse software in a new context, developers use the feature model to determine how the SPL can be customized into a new *configuration*. A configuration is a complete and unique set of the SPL's software artifacts. In a feature model, a configuration 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 configuration of the SPL that satisfies the target requirement set. For simple feature models, such as the one shown in Fig. 1, developers can manually derive a selection of features for a configuration. 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 configurations to fit a requirement set. For example, some techniques model the 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 configuration (Benavides et al., 2013; White et al., 2007). Other research has modeled feature selection problems as boolean satisfiability (SAT) problems or grammars and used SAT solvers to derive configurations (Mannion, 2002; Batory, 2005; Beuche, 2004; Buhrdorf et al., 2004) or Binary Decision Diagrams (BDDs) (Czarnecki and Wasowski, 2007). This prior research have been focused on problems dealing with one configuration with a set of constraints while not considering multiple-steps derivation problems.

1.1. Open problems

When software evolves, its evolution may need to be broken into multiple steps to satisfy evolution constraints (Abran et al., 2001). In some cases, product features must be introduced gradually over a series of steps. For example, the Boeing 737 aircraft, introduced in 1966, has been continually upgraded and adapted over time and is still currently in service. Each successive configuration of the 737, which is called a *Variant* has been developed over multiple years and incorporated new features into the base aircraft configuration (Shaw, 1999). For example, development of the 737-300 configuration of the aircraft started in 1979 and first flew in 1984. The configuration added a variety of features, such as an Electronic Flight Instrumentation System. The 737 has had numerous successive configurations, such as the 737-400, 737-500, 737-600, 737-700, 737-800, and 737-900, all planned and developed over significant spans of time.

There are a number of scenarios where the evolution of a set of products may be performed over several predefined steps. For example, when a new Linux distribution such as an Ubuntu release is planned, developers have to decide the set of software artifacts that are going to be added and removed in the next release of the distribution (*e.g.* add and remove packages and change the dependencies between them). Moreover, in other domains, such as aircraft construction, nuclear power plants, etc., configurations and upgrades to product configurations are planned years in advance (*e.g.*, the configurations of the 737 have spanned 46 years) and must be analyzed years in advance of their actual production. Ideally, an aircraft manufacturer would like to derive a sequence of successive configurations that build upon one another, as the 737 variants do, so that more advanced features are included each

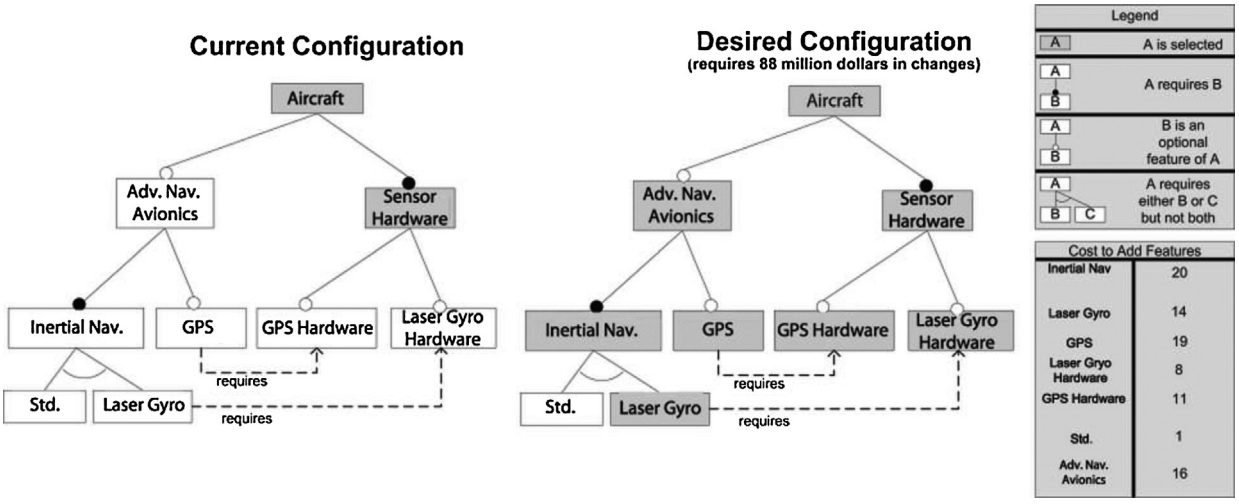


Fig. 1. A configuration problem requiring multiple steps.

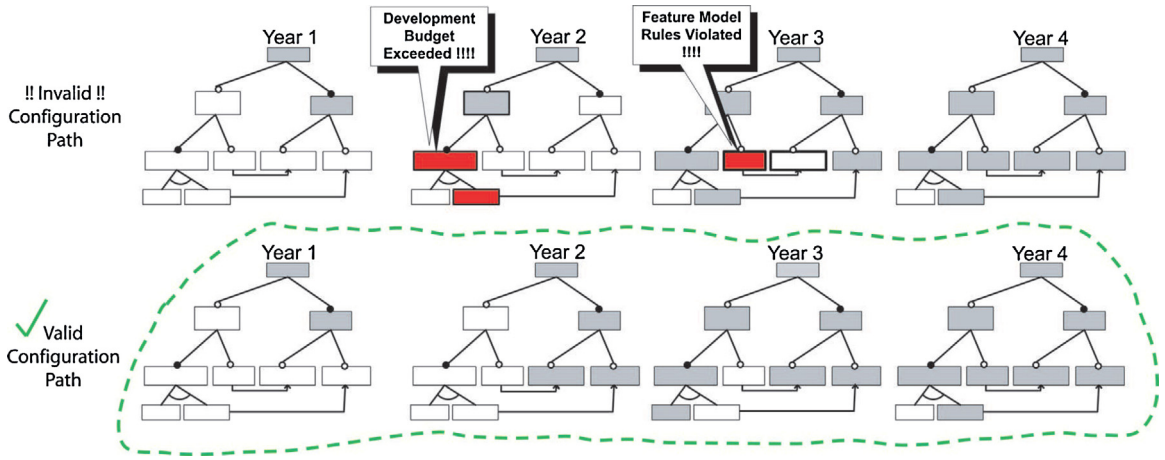


Fig. 2. Potential configuration paths.

year. A manufacturer, however, cannot arbitrarily choose features to add in a given year. Instead, each set of features for a 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 an order that ensures no two successive configurations will not differ by more than the price that a customer is willing to pay from one year to the next (*e.g.*, airline development or acquisition budget). Therefore, not only should every configuration step should satisfy a set of constraints, but the delta between any two successive configurations must be also acceptable.

Finally, when the product life spans years, such as the case of the 46 year history of the 737, the availability and capabilities of the processors, software, sensors, and other constituent components of the product inevitably change. Not only must manufacturers be able to plan and reason about configuration over multiple steps but have plans that account for the end-of-life of components and the significant increases in capabilities of newer components, which produce changes in the underlying feature model. For example, the processing power and availability of the processors used in the 737 have changed dramatically from 1966 to 2012. In some cases, the feature model may be specialized (*e.g.*, adapted so that its valid configurations at later steps are subsets of the starting set of valid configurations). In other cases, new features may be added to the feature model so that it is evolved to allow configurations that were not initially possible or valid. Thus, when the configuration problem should be reasoned in multiple steps, manufacturers must deal with two distinct forms of change: (1) changes to configuration and (2) changes to the underlying feature model that dictate what configurations are 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 Fig. 2. In Fig. 2, the selected features are colored in grey and the red features represent feature selections that violate a software evolution constraint.

This sequence of activities is called a *multi-step configuration problem*. Prior work on automated configuration (Mannion, 2002; Batory, 2005; Beuche, 2004; Buhrdorf et al., 2004) focuses on selecting a single configuration in a single step and not determining a configuration path. As a result, developers must manually derive a configuration path through feature models with hundreds or thousands of features and complex constraints on how successive configurations can differ.

Manually deriving configuration paths for a product-line is hard because developers must analyze a myriad of tradeoffs related to the order that the features are selected. For example, developers may temporarily add a feature that is not in the desired ending configuration to yield a valid variant at a particular step. Moreover, the costs of introducing features may vary over different steps making it hard to identify exactly the right step to introduce a feature. For example, the cost of adding an entertainment system in an aircraft may vary from one year to the next one because of variations in display prices, fluctuations in currency value, or changes in tax rates.

1.2. 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) (Van Hentenryck, 1989). 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. MUSCLES can return either all valid paths or a single optimized path from the initial configuration to the final one and the SPL engineer can decide which evolution path best fits the project's goals.

This paper extends our prior work on automated multi-step configuration of software product-lines (White et al., 2009). The paper presents a new approach for handling *feature model drift*, which represent the problem of introducing one or more changes in a feature model's constraints that occur over time. As pointed out earlier, when configuration is performed in multiple steps, there are two types of changes that must be considered: (1) configuration changes and (2) feature model changes, which we term feature model drift. This paper adds new techniques for handling the second form of change, feature model drift, which was not addressed in our prior work. We present a formal mapping of feature model drift to a CSP 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 feature model changes over time.
5. We describe mechanisms for optimally deriving a set of configurations that meet 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.

1.3. 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 5 describes how feature model drift can be modeled as a CSP;

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 Fig. 3.

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.

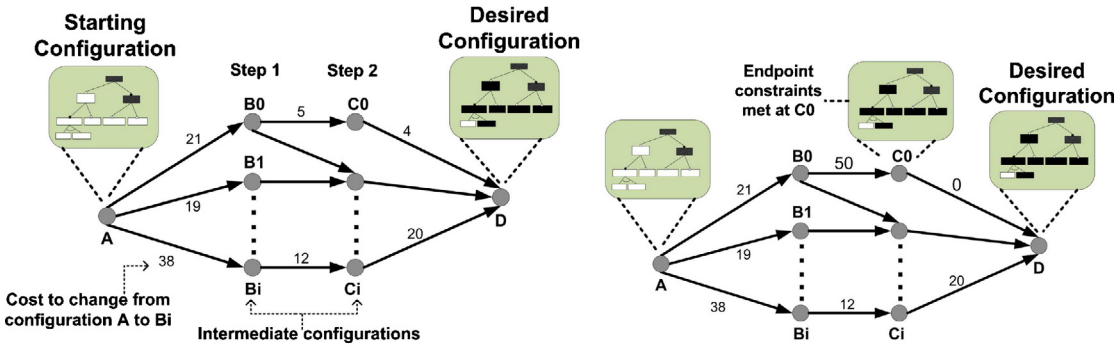
2.2. Challenge 2: point configuration constraints

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.

Fig. 1 shows an example configuration problem with time for an aircraft with no advanced navigation capabilities. In three years, the manufacturer would like to add the advanced navigation capabilities to the standard aircraft. The manufacturer's cost (in millions) to add each feature to the aircraft configuration is shown in the *Cost to Add Features* table in Fig. 1. The manufacturer has budgeted at most 35 million dollars per year to add features to the aircraft. 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 aircraft 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 Fig. 1 has multiple 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



(a) A Graph of a Multi-step Configuration Problem (b) Optimization of Total Steps

Fig. 3. Multi-step configuration graphs.

selected. Complex step-based constraints may also be present, such as a particular aircraft 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 4.4 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 aircraft example in Fig. 1 requires that developers adding new features spend no more than 35 million dollars in one year. The cost of selecting/deselecting 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. We term these constraints on the selection/deselection of features from one step to the next, *edge constraints*.

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 4.7 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 Fig. 3 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 4.10 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 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 `Laser Gyro` feature, shown in Fig. 1, may be bought by a competitor that intends to discontinue selling the existing software component in two years. In place of the existing component, a newer component will be offered that is much more expensive and uses a different and more precise algorithm. In two years when the existing software controller is discontinued, developers must update the feature model to include the new laser gyro type and add a requires constraint from the new laser gyro to the laser gyro hardware. 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 the multi-step configuration approach used by MUSCLES to derive valid configuration paths of SPLs. This paper also presents the techniques for modeling multi-step configuration problems as CSPs. These techniques give modeling tool developers the theoretical underpinnings to develop tools that can reason about configuration over multiple steps. We have developed domain-specific graphical modeling tools for our industry partners, using the Generic Eclipse Modeling System (http://eclipse.org/gmt/gems), for describing these problems and each of the various constraint types outlined in this paper and automating the transformation to CSP. For example, in past work with Boeing, Siemens, and others, we have developed modeling tools on top of these types of algorithmic approaches for the aeronautics, automotive, and mobile computing domains (Gill et al., 2004; Madl et al., 2006; Deng et al., 2008; Dougherty et al., 2011; White et al., 2008b,b, 2007; Nechypurenko et al., 2007). However, the process of building domain-specific languages and tooling on top of MUSCLE is beyond the scope of this paper and this paper focuses on the CSP modeling and solving process for these multi-step problems. SPL modeling experts can build modeling tools that use model to model transformation to create MUSCLES CSPs, solve for optimized configuration paths, and present the results to the user, very similar to our work in White et al. (2008b).

This approach assumes that developers have advance knowledge of the feature model changes that will occur. In some cases, unforeseen changes may arise that impact the configuration paths that were previously derived with MUSCLES. Unforeseen changes are always a challenge in software development. A key attribute of MUSCLES is that it can help developers to quickly analyze a number of different configuration paths to understand the impact of an unforeseen change that has arisen. MUSCLES does not guarantee that the configuration paths may not change due to unforeseen circumstances, but it does help engineers to reason about how those changes may impact future configuration decisions and aid them in understanding corrective remedies.

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 a *single* valid starting and ending configuration exist. All derived configurations at each step must be complete and valid feature model configurations – no partial configurations are allowed.

3.1. 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 (the point configuration constraints do not have to be identical across all steps. For example, if feature A is active in the step K is not required to explicitly set it to true in the step $K+1$),
- $\Delta(F_T, F_U)$ 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. Those constraints can be a list of features that must initially be selected. For example, the basic security required when building an aircraft.
- F_{end} is a set of configuration constraints on the final configuration to be reached at the end of the configuration steps. Those constraints can be composed by the features that must be selected or maximum cost of the final configuration. For example, the maximum cost for the entertainment system on an aircraft.

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

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

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 that support the required constraints. Not all techniques may have sufficient expressiveness, particularly to encode optimization-related goals, such as path length optimizations.

3.2. Constraint and optimization functions

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.

3.3. Edge constraints

We define an edge constraint as a bound on the feature state (selected or deselected) of features over time. An edge constraint, $e_i \in E$, is defined as:

$$\gamma(F_T, F_{T+k})$$

where γ is a constraint defined over a set of features at steps T and $T+k > T$. The set of edge constraints E can include numerous types of constraints on the transition from one configuration to another. 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 avionics problem from Fig. 1):

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

In this case, $\gamma = \Delta(F_T, F_{T+1}) \leq 35$. 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:

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

Edge constraints may also be attached to specific time steps:

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

3.4. Point configuration constraints

The point configuration constraints specify properties that must hold for the set of selected features at a given step. A point configuration constraint is defined as a set of feature selection states, F_r, for step T, $F_T = F_r$. 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:

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

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

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

3.5. Change calculation functions

A change function, defined as $\Delta(F_T, F_{T+K})$, where $K > 0$, 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:

$$F_{added} = F_{T+K} - F_T$$
$$\Delta(F_T, F_{T+K}) = \sum f_i \times c_i, f_i \in F_{added}$$

where f_i is the ith selected feature and c_i is the price of selecting 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 (Van Hentenryck, 1989) 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 (Benavides et al., 2005; White et al., 2007, 2008a) 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, $(X - Y > 0) \wedge (X < 10)$ 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* (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 ith 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 ith feature is selected in the configuration. If the ith feature is not selected, $f_i = 0$. Configuration rules from the feature model are represented as constraints over the variables in F. More information on creating a CSP from a feature model are described in White et al. (2007), Benavides et al. (2005).

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 Fig. 2 capture sequential modifications to the aircraft based configuration (shown in Fig. 1) that will incorporate high-end features into the base 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.

4.3. 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 ith 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 ith 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: $\langle F_T, F_{T+1}, \ldots, F_{T+K-1} \rangle$. All paths are of the same length, except that some paths may arrive at the desired configuration earlier than other paths.

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

$$f_{aT} = 0 f_{aT+1} = 1$$

Fig. 4 shows a visualization of how the $f_{iT} \in F_T$ variables map to feature selections.

4.4. 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

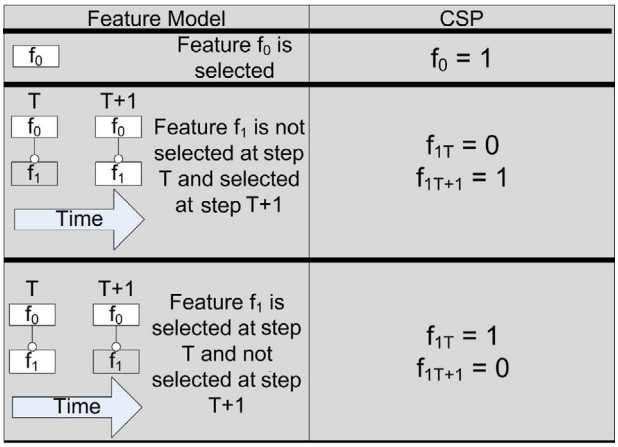


Fig. 4. Representing feature selection state at specific steps.

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. A simple constraint would be that the 2nd and 3rd configurations must have the feature f_1 selected. The set of steps for which this constraint must hold would be $T_{pc} = \{2, 3\}$.

4.5. 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.

4.6. CSP point configuration constraints

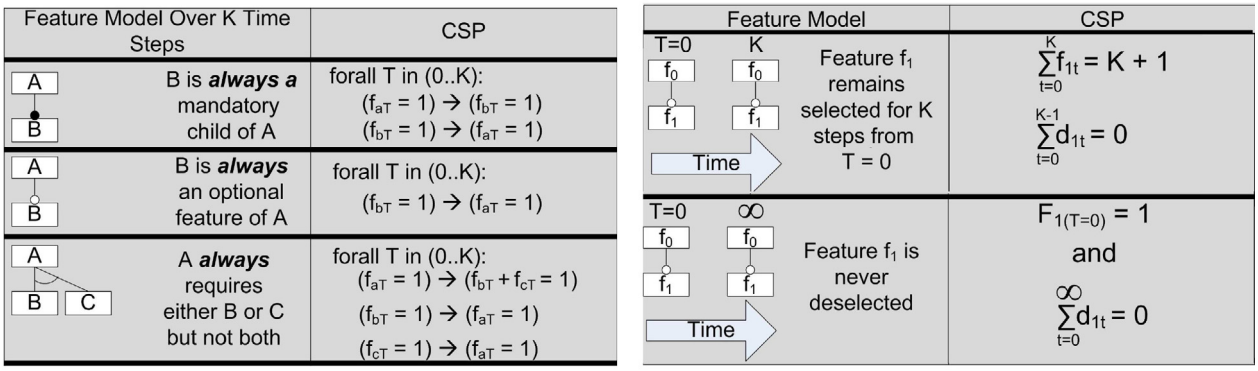
Assume that we want to find values for $F_T, \ldots, F_{T+K}$ such that we never violate any of the feature model constraints at any step. Further assume that the constraints in the feature model remain static over the K steps (feature model changes over multiple steps can also be modeled). If the jth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

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

That is, we require that at any step T, if the ith feature (F_{iT}) is selected, the jth feature (f_{jT}) is also selected. Moreover, at any step T, if the jth feature (F_{jT}) is selected, the ith feature (f_{iT}) is also selected. Other example point configuration constraints can be mapped to the CSP as shown in Fig. 5(a) and (b).

4.7. 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 (White et al., 2008a), we can specifically capture which features are selected or deselected between any two steps and constrain these changes via budget or other restrictions. Note that this slightly differs from the feature



(a) Point Configuration Constraints for Feature Model Structure

(b) Point Configuration Constraint for Feature Selection

Fig. 5. Point configuration constraints.

model drift because there is no modification of any relationship (*e.g.*, mandatory to optional).

4.8. 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:

$$\forall s_{iTU} \in S_{TU}, (s_{iTU} = 1) \Leftrightarrow (f_{iT} = 0) \wedge (f_{iU} = 1)$$

$$\forall d_{iTU} \in D_{TU}, (d_{iTU} = 1) \Leftrightarrow (f_{iT} = 1) \wedge (f_{iU} = 0)$$

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. If the ith feature costs c_i to select or deselect then

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

4.9. CSP edge/change constraints

The cost of including a particular feature may change over time. For example, the cost of selecting a GPS guidance system 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 selects 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 million dollars. In practice, the edge weights should be defined by the SPL managers and the product stakeholders.

Assume that the cost of selecting the ith feature at step T can be calculated by the function:

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

We can then define the cost of selecting new features for the configuration as:

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

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

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

In practice, the cost to add or remove features from a configuration will be domain-specific. For example, add or removing features may be as straightforward as consulting a supplier's pricing sheet to determine the cost of an add-on that will provide the necessary feature. In more complex cases, the feature selection may involve development of a new software component, in which software cost estimation models may be needed.

4.10. 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 avionics problem from Section 1 developers may want to find a configuration path that meets their constraints and includes the laser gyro as soon as possible.

4.11. 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 satisifed:

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

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:

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

$$(w_T = 1) \Rightarrow \left(\sum_{i=0}^{n} s_{iTT+1} + \sum_{i=0}^{n} d_{iTT+1} = 0\right)$$

When the final configuration is reached, the number of feature selections, $\sum_{i=0}^{n} s_{iTT+1}$, and deselections, $\sum_{i=0}^{n} d_{iTT+1}$, is forced to be zero by constraining the sum of the selections and deselections to zero. Thus, MUSCLES does not allow any configurations to have changes after the goal state is reached because that would cause $\sum_{i=0}^{n} s_{iTT+1} + \sum_{i=0}^{n} d_{iTT+1}$ to be nonzero.

4.12. Path length optimization

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.

4.13. Cost optimization

We can instruct the solver to minimize the cost of the ending configuration by defining an optimization goal over the variables in P. Assume that the cost of ith feature at step K is denoted by the variable $c_i \in C_K$, minimize C_K, where:

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

4.14. Path cost optimization

An optimization to minimize the costs of changes can be defined based on the weights of the edges. To find the configuration path with the lowest development cost, where the development cost is the edge weight the goal is to minimize:

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

4.15. Optimization flexibility

A subset of the possible objective functions have been defined above. Other arbitrary objective functions can be defined over the variables in *Msc*.

4.16. Catalog of feature model constraints over multiple steps

In this section, we show that any of the feature model constraints described in the previously discussed semantics by Benavides et al. (Schobbens et al., 2007; Benavides et al., 2010) can be converted into a multi-step constraint using MUSCLES. Feature model constraint semantics are described by Benavides et al. (Benavides et al., 2010) both in terms of propositional logic and CSP semantics. Below is a table that includes each of the constraints described by Benavides et al. and maps SPL constraints to multi-step constraints.

Comprehensive list of feature model constraints in MUSCLES

	CSP (single step)	CSP with multiple steps ($T_1, T_2, \ldots, T_n$)
Mandatory	$F_i = F_j$	$F_{iT_1} = F_{jT_2}$
Optional	if $F_j = 0$ then $F_i = 0$	if $F_{jT_2} = 0$ then $F_{iT_1} = 0$
Or	if $F_i = 1$ then $\sum(F_j, F_k, \ldots, F_n) in \{1, \ldots, n\}$ else $\sum(F_j, F_k, \ldots, F_n) = 0$	if $F_{iT_1} = 1$ then $\sum(F_{jT_2}, F_{kT_3}, \ldots, F_{nT_n}) in \{1, \ldots, n\}$ else $\sum(F_{jT_2}, F_{kT_3}, \ldots, F_{nT_n}) = 0$
Alternative	if $F_i = 1$ then $\sum(F_j, F_k, \ldots, F_n) = 1$ else $\sum(F_j, F_k, \ldots, F_n) = 0$	if $F_{iT_1} = 1$ then $\sum(F_{jT_2}, F_{kT_3}, \ldots, F_{nT_n}) = 1$ else $\sum(F_{jT_2}, F_{kT_3}, \ldots, F_{nT_n}) = 0$
Excludes	if $F_i > 0$ then $F_j = 0$	if $F_{iT_1} > 0$ then $F_{jT_2} = 0$
Implies	if $F_i > 0$ then $F_j = 1$	if $F_{iT_1} > 0$ then $F_{jT_2} = 1$

A key aspect to note is that a constraint can be applied at a specific step. In this case, $T_0 = T_1 = \ldots T_n$. That is, the constraint governs the selection state of a set of features all within a single time step. However, the constraints may also govern the selection state of features at different points in time, where $T_0 \neq T_1 \neq \ldots T_n$. Moreover, the features and time steps can arbitrarily cross-cut the steps where portions of the constraint govern feature selection at one step and other portions of the step relate to the selection state of features at other steps. For example, feature f_{aT_1} can have an exclusive or relationship with f_{bT_2} and f_cT_3. In this case, the constraint would dictate that if feature f_a is selected at step T_1, then either f_b has to be selected at step T_2 or f_c has to be selected at step T_3. The feature model constraints governing selection can apply both, as with existing approaches, within a single step, or span multiple steps. MUSCLES supports all of the standard feature model constraints but adds the added ability to specify that the constraint applies to the selection state of features at different steps.

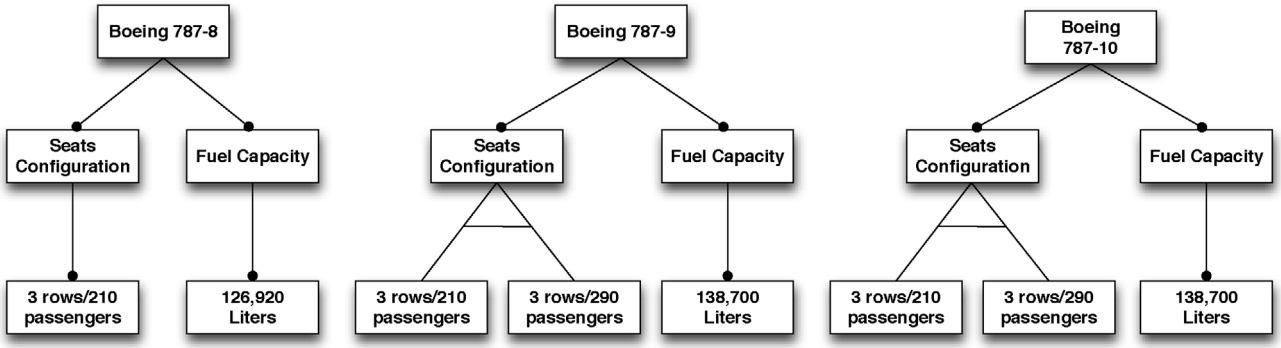


Fig. 6. Boeing 787 feature model drift.

5. Modeling feature model drift

When configuration occurs over multiple steps, the configuration process may span a substantial period of time. For example, the aeronautics development example from Section 1, where features are being added to a plane, spans several years. In most multi-step configuration problems, such as the Boeing 787, developers may need to reason about configuration over a span of days, months, or years in order to decide the best path. For example the Boeing 787 exhibited a number of feature model drifts during its configuration. Fig. 6 shows the feature model changes made to this aircraft since its first release.[1] Fig. 6 shows the changes that occurred over several years. In the period, from 2006 to 2014, the option of having 3 rows and 290 passengers was added. These improvements required the removal of the features "126.920L" and "3 rows/210 passengers" from the feature model, as well as the mandatory child relationship between "Seats Configuration" and "3 rows/210 passengers". Finally, two new features, "3 rows/290 passengers" and "138.700L," were added to the feature model. Further, the next planned release the plane in 2014 will remove the "3 rows/210 passengers" feature and the set relationship for seating and add the feature "3 rows/310 passengers" as the only seating option.

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 model relationships, after the initial configuration step. The allowed changes depend on the expressiveness of the models being used. For example, if cardinality based feature models are used, the changes can also incorporate changes in the cardinalities of relationships. If attributed feature models are used, changes to the values of attributes and attribute relations can be captured (Roos-Frantz et al., 2012). For example, aircraft 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. Note that, when a feature model drifts it is possible that errors may be introduced, such as contradictions that yield an unsatisfiable feature model. MUSCLES assumes that the feature models at each step are satisfiable and error-free. In this paper we do not focus on this problem but it can be easily addressed by using the error checking techniques listed by Benavides et al. (2010), and the parsers from FaMa (FaMa, 2013), TVL (Classen et al., 2011) or SPLOT (SPLOT, 2013) for lexer errors.

When using MUSCLES to analyze feature model drift, SPL managers need to be able to predict the changes ahead of time in order to reason about them, which is not always possible. Sometimes the drift occurs due to security flaws detected in advanced stages of development or due to market requirements. MUSCLES is useful for reasoning about how these changes can or will impact planned configurations.

In many cases, developers do know ahead of time which features or relations will be introduced, discontinued, or replaced. Moreover, developers often have an estimate of when the availability of the feature (and its relationships with the rest of the model) will change based on information provided by a supplier or other mechanism. This data on feature and constraint modification 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 4.4 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. An important point, however, is that this approach explicitly models the addition, removal of features and relationships in the future.

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. If the jth feature is an optional child of the ith feature (the software package) at step T and at step K, the jth feature becomes mandatory, we can model this as:

$$(f_{jT} = 1) \Rightarrow (f_{iT} = 1)$$

At step K, the jth feature becomes mandatory, changing the constraints on selection of the feature:

$$(f_{iK} = 1) \Rightarrow (f_{jK} = 1)$$
$$(f_{jK} = 1) \Rightarrow (f_{iK} = 1)$$

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 constrant that ($f_{iK} = 1$) $\Rightarrow$ ($f_{jK} = 1$), which makes f_j mandatory.

Examples of other feature model drifts as CSP constraints are shown in Fig. 7.

The approach described above can handle arbitrary modifications to a feature model as long as the modifications yield a new feature model with at least one valid product. If a contradiction is introduced via feature model drift and no valid products are present, the solver will not be able to derive a configuration path.

[1] http://www.boeing.com/commercial/787family.

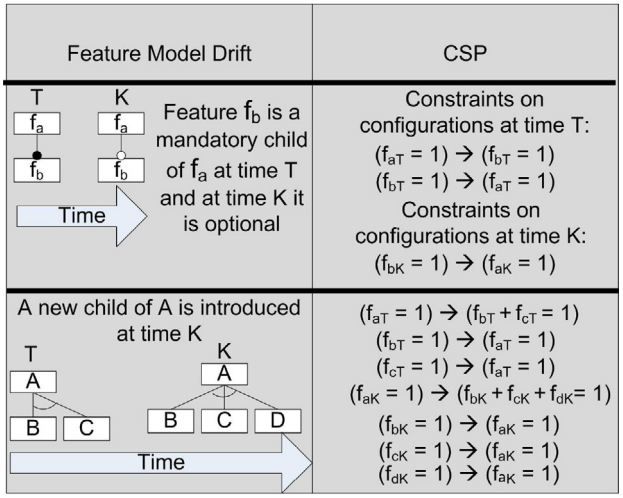


Fig. 7. A CSP model of feature model drift.

Another possible contradiction is if the edge or point configuration constraints contradict the changes introduced by feature model drift. For example, if a feature that is mandated by a point configuration constraint is removed by feature model drift, a contradiction occurs. The approach requires that neither type of contradiction be present.

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 ith 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^{start} and $3 \leq E_i^{start} \leq 5$. The duration of the epoch, E_i^{dur}, is $E_i^{dur} = \infty$.

To express feature model epochs, constraints must be added to bound the values for E_i^{start} and E_i^{dur}. We introduce the function,

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

to determine the begining 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:

$$3 \geq S(E_i^{start}, E_i^{dur}, F_0, F_1, \ldots, F_{end}) \geq 5$$

Similarly, a separate function,

$$W(E_i^{dur}, E_i^{dur}, F_0, F_1, \ldots, F_{end})$$

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:

$$W(E_i^{dur}, E_i^{dur}, F_0, F_1, \ldots, F_{end}) = \infty$$

An important note is that this approach assumes that the changes that are applied to the feature model during a change epoch are assumed to be correct. For example, if a feature is removed in a particular step, any other modifications to the feature model needed to bring it to a valid state (*e.g.*, removing dependent cross-tree constraints, adding replacement features, etc.) are also applied so that the feature model does not have inconsistent or unsatisfiable constraints. Moreover, the approach also assumes that objective functions for the optimization process are not specified in a manner that they are undefined when one or more features are added or removed. At all steps, it is assumed that the objective function is defined and all features needed to calculate its value are present.

5.3. Epoch-based feature model constraints

The feature model drift epochs make it possible to model situations in which the exact step in which a change will occur to a feature model 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 jth feature is an optional child of the ith feature (the software package) at step T and at step K, the jth feature becomes mandatory, we can model this as:

$$(f_{jT} = 1) \Rightarrow (f_{iT} = 1)$$

At step K, the jth 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 jth feature is an optional child of the ith feature (the software package) at the start and at some step, K, where $3 \leq K \leq 5$, the jth 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:

$$(f_{jT} = 1) \Rightarrow (f_{iT} = 1)$$

If step K is within the time period of the feature drift epoch, the jth feature becomes mandatory, changing the constraints on selection of the feature:

$$((f_{iK} = 1) \Rightarrow (f_{jK} = 1)) \Leftrightarrow (E_i^{start} \leq K \leq E_i^{start} + E_i^{dur})$$
$$((f_{jK} = 1) \Rightarrow (f_{iK} = 1)) \Leftrightarrow (E_i^{start} \leq K \leq E_i^{start} + E_i^{dur})$$

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 aircraft media server. Within 1 year from the time that this new mobile phone platform is developed, they will be able to develop a video player that streams media from the aircraft media server on the same mobile platform.

In this scenario, the development of the mobile phone server video streamer feature is dependent upon the occurrence of the mobile platform's server comunication 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 epoch by constraining an epoch's start, E_j^{start}, and duration, E_j^{dur}, in terms of another epoch, E_i. For example, if we wish to define the epoch, E_j, as occuring at least two steps after the epoch, E_i, we can say:

$$E_j^{start} \geq E_i^{start} + 2$$

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 introduce branching into the occurrence of epochs. For example, developers might have a single physical connector on a in-seat screen that they plan to use either to consume streaming video from a media server or connect to controls for managing the above seat lighting and vents, but not both.

To encode branching constraints into feature model drift, developers can use the E_i^{start} variable to encode branching constraints. For example, if the changes described by the ith feature model drift are mutually exclusive with the changes in jth feature model drift, this constraint can be encoded as:

$$E_i^{start} \geq 0 \Leftrightarrow E_j^{start} = -1$$

$$E_j^{start} \geq 0 \Leftrightarrow E_i^{start} = -1$$

where, $E_j^{start} = -1$ indicates that the jth 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. We tested a number of hypotheses related to the scalability of MUSCLES using various SPL configuration parameters, such as the total number of configuration steps.

6.1. Experimental platform

Our first experiment was performed with an implementation of 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 2.4 GHz CPU, 2 GB 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 MB (-Xms40m) and a maximum memory size of 256 MB (-Xmx256m).

The second experiment was performed with an implementation of the MUSCLES provided by the open-source FAMA tool suite. FAMA is also built using the Java Choco open-source CSP solver. The experiments were performed on a rack-mounted DELL PowerEdge server with 12 cores, 2GB of RAM, and running Ubuntu. The JVM was run in server mode using a heap size of 40 MB (-Xms40m) and a maximum memory size of 256 MB (-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. A CSP solver's performance can vary widely, from extremely fast to exponential time, depending on the constraints of a particular problem characteristic. In practice, CSP solvers tend to perform very well. To be thorough, we wanted to test the technique on a large number of models to get an accurate picture of the solving time. To solve this problem, we used a random feature model generator developed in prior work (White et al., 2008a). The feature model generator and code for these experiments is also available in open-source form along with the Ascent Design Studio. The feature model generator takes as input the desired total number of features, maximum branching factor, total number of cross-tree constraints, and maximum depth for the feature model tree. This feature model generator is based on the techniques developed by Thum et al. (2009). The generator produces a random feature model that meets the requirements. 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.[2]

We also needed the ability to produce valid starting and ending configurations that the solver could derive a configuration path between. To produce these configurations, we used the CSP technique developed by Benavides et al. (2005) to derive valid configurations of the feature model. If the CSP technique could not derive at least two different configurations from the feature model, it was considered void and thrown out.

Our experiments uncovered trends similar to what observed in prior work (White et al., 2008a). In particular, the branching factor, depth, and cross-tree constraints had little effect on configuration time. The key indicator of the solving complexity was the number of XOR-feature groups in a model. The other key indicators of solving complexity where whether or not optimization was used and the total number of time steps involved in the configuration.

6.2. Experiment: multi-step configuration scalability

6.2.1. Hypothesis

We hypothesized that MUSCLES could scale up to hundreds of features and 10 or more time steps, having this hypothesis we designed this experiment to prove it, believing that a CSP solver would be fast enough to derive a configuration path in a few seconds. On the other hand, the null hypothesis of this experiment is that the solver can't find a suitable solution for the problem in a reasonable amount of time.

6.2.2. Experiment design

We measured the solving time of MUSCLES by generating random multi-step configuration problems and solving for

[2] 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).

Table 1
Hypotheses and experiment design.

Hypotheses			
Null hypothesis (H_0)	MUSCLES can rapidly identify the configuration paths from the starting configuration to the final configuration with hundreds features and 10 or more time steps.		
Alt. hypothesis (H_1)	The complexity of the problem will prevent MUSCLES from finding solutions for problems with hundreds of features and 10 steps.		
Design			
Dependent variable	Time required by MUSCLES to derive a configuration path		
Independent variable I	Technique used for FM generation	Levels:	Random models produced by Thum et al.'s technique(Thum et al., 2009)
	Technique for selecting point and edge constraints	Levels:	Random (Java function applied to the list of model features traversed inorder.)
Blocking variables	Number of steps	Levels:	3, 4, 5, 6, 7, 8, 9, 10
Constants	CSP solver	Value:	ChocoReasoner
	Heuristic for variable selection in the CSP solver	Value:	*Default*
	Percentage of cross-tree constraints (constraints/features)	Levels:	%10

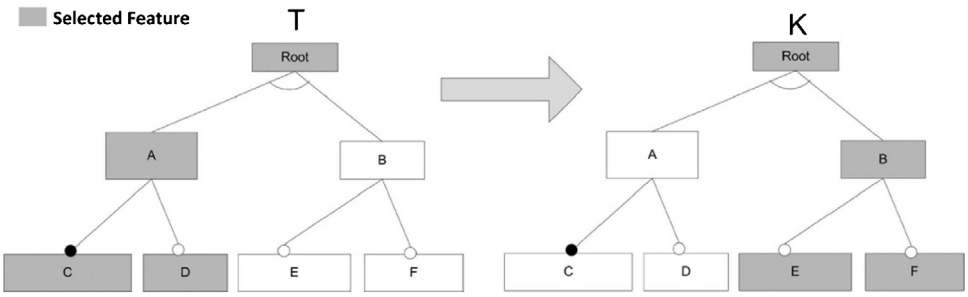


Fig. 8. Changing between two XOR subtrees.

configuration paths that involved larger and larger numbers of steps. The problems were created by generating semi-random feature models with 500 features as well as starting and ending configurations for each model. The models were obtained using the tool Betty, which provides an implementation of the Thum et al.(Thum et al., 2009) proposal. The point configuration constraints were derived by using a constraint solver to derive a valid configuration for each step. Once a point configuration was chosen for a step, an edge constraint was added limiting the sum of the feature selection and deselection costs to be exactly the sum needed to reach the randomly chosen point configuration. Table 1 shows the details of the experiment executed. MUSCLES was used to derive a configuration path between the starting and ending 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 ~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.

Fig. 8 shows an example of a large-scale configuration path problem where the solver must derive a configuration path that switches from including feature `A` to feature `B`.

With this type of configuration problem, the solver was forced to change every feature selection in the starting configuration to reach the end state, *i.e.*, these experiments maximized the difference between the starting and ending configurations.

We generated and solved temporal configuration path problems for feature models with 500 features. We successively increased the number of time steps involved in the configuration path to produce larger and larger configuration paths. The maximum number of changes per configuration checkpoint were bounded to 1/4 of the total number of features. We solved 100 randomly generated configuration path problems per feature model size.

6.2.3. Results and analysis

The results from the experiment are shown in Fig. 9.

This figure shows the solving time in milliseconds for the configuration path derivation versus the total number of time steps in the configuration problem. As shown in Fig. 9, the solving time scales roughly linearly with the number of time steps.

The apparent linear scaling of the technique with respect to the number of time steps is a promising result. Although more work is needed to show that this linear scaling continues for different configuration path properties, these results indicate that the technique may scale well as the number of time steps grows. Our future work will further investigate the scalability of the technique and improve MUSCLES's CSP formulation. We also note that standard CSP solving algorithms, such as branch and bound appear to work well for these problems even though they have exponential worst case time complexity. However, it may be possible to develop new solving algorithms that provide better performance for special classes of multi-step configuration problems. Based on the results, we ended agreeing that the null hypothesis we proposed have been proved and MUSCLES can rapidly identify the

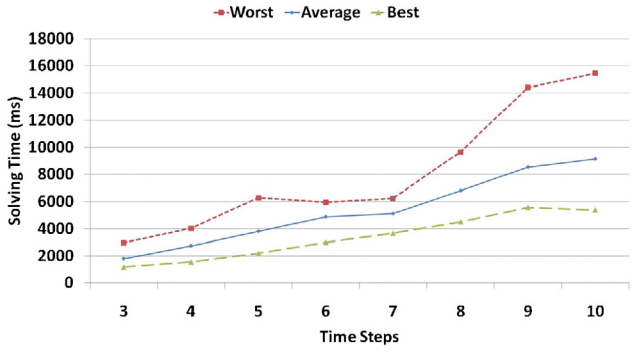


Fig. 9. Automated configuration time for varying numbers of time steps.

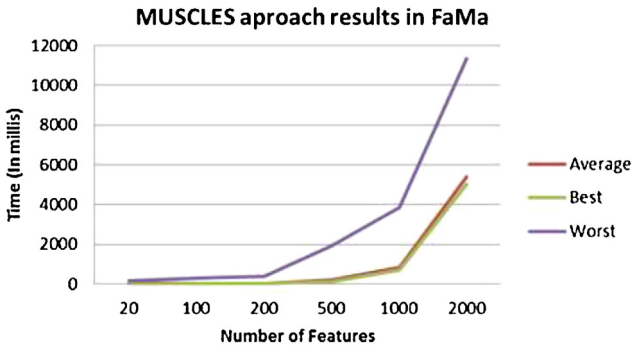


Fig. 10. Automated configuration time for feature model drift problems.

configuration path with hundreds of features and 10 or more steps.

6.3. Experiment: feature model drift scalability

6.3.1. Hypothesis

In this experiment we proposed as a null hypothesis, that MUSCLES could solve for configuration paths that included feature model drift in several seconds. Given that null hypothesis, the alternative hypothesis is that MUSCLES can not solve problems with drifts in a few seconds.

6.3.2. Experiment design

As in the first experiment, 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. In this second experiment, we introduced changes to the feature model at each step. At each step, one feature was added or removed. The feature model was then checked to ensure that it included one or more valid products using CSP analysis. If the new feature model did not contain any valid products, the feature change was reversed and another random change attempted. The feature models were semi-randomly generated with 20–2000 features as well as starting and ending configurations for each model. MUSCLES was used to derive a configuration path between the two configurations over multiple steps. The properties of the feature models described in Experiment 1 were also used for this experiment.

Table 2 shows the details of Experiment 2.

6.3.3. Results and analysis

The results from the experiment are shown in Fig. 10. This figure shows the solving time in milliseconds for the configuration path derivation versus the total number of features. Overall, the approach scaled well for large feature models. At 1000 features, a solution could be found in 4 seconds or less. We believe that for the majority of industry problems, being able to deal with feature models with 1000 features will be sufficient. Therefore, given those results, the null hypothesis proved to be correct.

6.4. Experiment limitations and threats to validity

Even though the experiments presented in this paper provide evidence that the solution proposed is valid, there are some conditions that may affect the validity of those experiments. In this section we show the different validity threads that could affect the experiments.

External validity: The inputs for the experiments presented in this paper have been inspired by industry problems. However, it is possible that the feature models that we have experimented with do not properly reflect real-world models. The major threats to the external validity of are experiments are: (1) *Population validity*, the models used are randomly created and may not mirror realistic feature models seen in industry. The complexity of the constraints used and the size of the problem may vary with real projects. To try to minimize this effect we have relied on the Thum et al.'s feature model generation approach (Thum et al., 2009) and its implementation provided by the BeTTy tool (BeTTy, 2013). (2) *Ecological validity*, MUSCLES analyses were run individually to minimize the impact of third-party threads in the time being measured. However, there might be other threads such as operating system threads that could impact execution time. To minimize this effect, we carried out not one execution of a model, but a hundred executions and used the average as the result.

Internal validity: The time required to analyze a feature model depends on the number of features and percentage of cross-tree constraint and deriving SPL configurations has been proven to be an NP problem in previous research (Benavides et al., 2010; Kang et al., 1998). Multi-step configuration problems add other inputs that might affect the performance, such as the number of the steps required to reach the final configuration. The conducted experiments were designed to not exceed a maximum budget between successive steps. However, if we add attributes to the experiments, more complex functions than the sum of costs by selecting and deselecting features can be employed. For example, a more complex function over the numeric delta between attributes of successive steps could be used, such as combining carbon emissions with monetary costs. To improve the internal validity of the experiments we experimented with multiple variations on the number of features and a variety of step counts.

Construct validity: The first results looks promising in terms of time required to solve problems with 1000 features. We assume that most real-world problems will be of similar scale. However, because the tests were not exhaustive, more analysis of which solver heuristics provide the best results are needed.

7. Related work

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

7.1. Feature model semantics

Prior research has laid out the formal semantics of feature models, variability, and configuration (Schobbens et al., 2007; Benavides et al., 2010). MUSCLES builds upon these previously described semantics and introduces new approaches for dealing with configuration over multiple steps. Both the prior semantics and MUSCLES are complementary research.

7.2. Constraint optimization techniques and scheduling problems

MUSCLES builds upon extensive prior work on constraint satisfaction problems and optimization (Van Hentenryck, 1989). Constraint satisfaction programming techniques have been used for a wide variety of related problems in artificial intelligence, process improvement, operations research, and other areas (Van Hentenryck, 1989). In particular, the *scheduling problem* is a well-known constraint optimization problem that looks at how to schedule a finite set of resources to complete a task in order to maximize or minimize an objective function. This problem is related to

Table 2
Hypotheses and experiment design.

Hypotheses			
Null hypothesis (H_0)	MUSCLES can find solutions for problems that incorporate feature model drift and have a high number of features in several seconds.		
Alt. hypothesis (H_1)	MUSCLES can not solve for solutions within a few seconds if feature model drift occurs.		
Design			
Dependent variable	Time required by MUSCLES to configure a model with varying drifts.		
Independent variable	Technique used for FM generation	Levels:	Random models generated using Thum et al.'s technique (Thum et al., 2009)
	Technique for selecting point and edge constraints	Levels:	Random (Java function applied to the list of model features traversed inorder)
Blocking variables	Number of features	Levels:	20, 100, 200, 500, 1000, 2000
Constants	CSP solver	Value:	ChocoReasoner
	Heuristic for variable selection in the CSP solver	Value:	*Default*
	Percentage of cross-tree constraints (constraints/features)	Levels:	%10
	Number of steps	Levels:	3
	Selection of initial and final configurations	Levels:	Random (from the set of valid configurations)

MUSCLES but not specific to the multi-step configuration derivation problem for feature models that MUSCLES focuses on.

7.3. Automated single-step configuration

Several single-step feature model configuration and validation techniques have been proposed (Benavides et al., 2013; Mannion, 2002; Batory, 2005; Beuche, 2004; Buhrdorf et al., 2004; White et al., 2007). 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 avionics 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 $K = 1$.

Bosch (2002) describe several evolution patterns that appear in the configuration of SPLs. These patterns incorporate the effects of manipulating variation points in regards to time and resource consumption. In contrast, MUSCLES accounts for this evolution variability by spreading feature selections over multiple stages, such that an introduction of a new variation point can be taken into account.

7.4. Configuration workflows

Hubaux et al. (2009) presented a formalism to determine the work-flow required to configure a feature model in multiple steps. MUSCLES approach also focuses on configuring a model in multiple steps. However, Hubaux et al.'s work does not investigate feature model drifts or automated derivation of a configuration path between a starting and final configuration. Moreover, MUSCLES also supports optimizations, such as minimizing the number of changes in the configuration path.

7.5. Feature edits

In the work presented by Thum et al., the authors present a catalog of possible changes that can be applied to a feature model in the process of model evolution. The authors also discuss approaches for determining if the new model is a subset or a super-set of the original feature model. This work differs from ours in many aspects. Thum et al. (2009) do not cover multi-step changes nor identify paths between configurations across multiple steps. However, the random generator used in this research's experimentation is the same used by Thum et al.

7.6. Solving product derivation problems

Mendonca et al. (2007) addressed the problem of dealing with different actors with different interests configuring the model at the same time by translating the model to a process model. This process, consists of 4 principal stages: (i) identify the conflicts; (ii) use the validated model and conflicts sets, and actors decisions to identify and resolve decision conflicts; (iii) build the process model; and finally, and (iv) validate the generated process.

7.7. Managing changes using dynamic SPLs

Rosenmüller et al. (2011) investigate unforeseen changes due to the run-time swapping of components in a SPL. This works differs from MUSCLES because it focuses on evolution of variants within the same SPL. MUSCLES' feature model drift allows modifications to the structure of the SPL which is not covered in the research by Rosenmuller et al.

7.8. Staged configuration

Czarnecki et al. (2004) 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 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: $\forall T \in (0, \ldots, K-1), F_T \subset F_{T+1}$.

Hwan et al. (2005) have looked at mechanisms for synchronizing specializations of feature models as changes occur over time. This problem is similar to the feature model drift problem outlined in this paper. MUSCLES focuses on a different and complementary aspect of the problem, which is reasoning in the face of changes to the feature model over time. Both synchronization and automated reasoning in the face of changes to the underlying feature model are needed and each approach addresses a different aspect of the problem.

Classen et al. (2009) 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.

7.9. Supply-chain product-lines

Hartmann et al. (2009) investigate methods of building models that incorporate the variability and constraints of multiple suppliers into a product-line feature model. The approach described by Hartmann et al. is orthogonal to MUSCLES. Hartmann's work focuses on the modeling aspects related to capturing and maintaining the constraints from multiple suppliers whereas MUSCLES provides a mechanism to reason about the constraints over time.

7.10. Understanding configuration over time

Elsner et al. (2010) have looked at variability over spans of time and the issues related to understanding when and how variability points relate to each other. MUSCLES focuses on automating three key tasks that Elsner et al. identify as needed for managing variability over time. Specifically, MUSCLES provides capabilities for automating and optimizing tasks that Elsner et al. term: (1) proactive planning, (2) tracking, and (3) analysis. Whereas Elsner et al. focus on general identification of the issues in managing variability over time, MUSCLES focuses on providing a framework for automating the specific tasks that Elsner et al. identify as needed in this space.

7.11. Model-driven feature model evolution

A number of approaches have looked at the development of modeling tools to support feature model evolution. Pleuss et al. (2012), Botterweck et al. (2009) model coherent sets of changes to a feature model as model fragments and allow modelers to describe evolved versions of feature models at future points in time. Further, the underlying model-driven tooling allows developers to check the correctness of the evolved models or interactively evolve the model. Whereas these existing approaches focus on the user-interface modeling and constraint-checking aspects, MUSCLES focuses on complementary automated mechanisms for optimizing the planning steps of future evolutions of configurations. For example, Pleuss et al.'s techniques do not provide configuration evolution optimization capabilities or automated non-interactive evolution based on objective functions, which the MUSCLES technique provides. MUSCLES can be used to augment model-driven approaches, such as Pleuss et al.'s with automated optimization and configuration evolution derivation capabilities.

7.12. Quality attribute evaluation

Several techniques have been proposed for evaluating quality attributes (Etxeberria and Sagardui, 2008; Immonen, 2006; Olumofin and Misic, 2005) 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.

7.13. Step-wise refinement

Batory (2004) describes AHEAD, a technique for the configuration of SPLs. AHEAD utilizes step-wise refinement, in which SPLs are configured iteratively. Our technique is similar in that it also selects additional features over the course of multiple-steps in order to reach a target configuration.

7.14. Re-configuration approaches

Other research has looked at re-configuration or SPL error correction in a number of different scenarios. For example, Xiong et al. (2012) researched eCos configuration problems, proposing a set of solutions for a set of concrete configuration errors. White et al. (2010) also have investigated automated techniques for fixing feature model configuration conflicts. These prior works differs significantly from this MUSCLES because they focus on single-step configuration, where as this paper focuses on reasoning about multi-step configuration.

8. Concluding remarks and future work

Many SPL scenarios 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 aircraft'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 aircraft), 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.

In future work, we plan to investigate a number of research questions related to multi-step configuration:

- *Real-time configuration process monitoring* → Unplanned changes are almost inevitable in long-term configuration processes. In future work, we plan to investigate approaches for monitoring configuration processes in real-time and automatically adjusting the starting and end configurations, as well as feature model constraints, as changes occur. Being able to monitor and identify changes to the configuration process will allow SPL users to reason about unforeseen changes in real-time.
- *Feature-model drifts extended validation* → An extended validation of feature model drifts within a real-world project should be performed. While the current validation is guided by the expertise of

some authors in the aeronautics industry, an experience report of feature-model drift should be performed to provide a better validation of the proposal presented in this paper. Moreover, some experimentation should be extended. For example, by executing our experiments with attributes and more complex functions.

- *Model transformation approaches for turning existing models into multi-step configuration problems* → Although creating a separate DSML for MUSCLES is one approach to leveraging it, this approach requires organizations to adopt new modeling languages and tools. Instead, an alternate approach is to investigate how to transform arbitrary existing domain models into multi-step configuration problems and document the required expressiveness of those domain models.
- *Feature model epoch and epoch ordering validation* → In order to test model epochs, we are working to extend the BeTTY Framework (BeTTy, 2013), to be able to test feature model drift operations with ordered epochs.

The Ascent Design Studio (ascent-design-studio.googlecode.com) and FAMA (famats.googlecode.com/svn/branches/multistep) provide open-source implementations of MUSCLES.

Acknowledgements

This work has been partially supported by the National Science Foundation (NSF), the Air Force Research Lab (AFRL RI), the European commission (FEDER), by the Spanish government under TAPAS (TIN2012-32273) project and the Andalusian government under Talentia program and THEOS (TIC-5906) projects.

References

Abran, A., Bourque, P., Dupuis, R., Moore, J.W. (Eds.), 2001. Guide to the Software Engineering Body of Knowledge – SWEBOK. IEEE Press, Piscataway, NJ, USA, ISBN 0769510000.

Batory, D.,2004. Feature-oriented programming and the ahead tool suite. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, pp. 702–703.

Batory, D., 2005. Feature models, grammars, and propositional formulas. Software Product Lines, 7–20.

Benavides, D., Trinidad, P., Ruiz-Cortés, A., 2005. Automated reasoning on feature models. In: Pastor, O., Falcão e Cunha, J. (Eds.), Advanced Information Systems Engineering, vol. 3520. Springer Berlin Heidelberg, pp. 491–503, http://dx.doi.org/10.1007/11431855_34.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: a literature review. Information Systems 35 (6), 615–636.

Benavides, D., Trinidad, P., Cortés, A.R., Segura, S., 2013. FaMa. Springer, Berlin/Heidelberg, pp. 163–171, http://dx.doi.org/10.1007/978-3-642-36583-6-11 (Chapter FaMa).

BeTTy Framework. http://www.isa.us.es/fama (accessed 02.01.13).

Beuche, D., 2004. Variant management with pure:: variants, Pure-systems GmbH, Tech. Rep.

Boehm, B., 1975. The high cost of software. Practical Strategies for Developing Large Software Systems, 3–15.

Bosch, J.,2002. Maturity and evolution in software product lines: approaches, artefacts and organization. In: Proceedings of the Second Conference Software Product Line Conference (SPLC2). Springer-Verlag, pp. 257–271.

Botterweck, G., Pleuss, A., Polzer, A., Kowalewski, S.,2009. Towards feature-driven planning of product-line evolution. In: Proceedings of the First International Workshop on Feature-oriented Software Development. ACM, pp. 109–116.

Buhrdorf, R., Churchett, D., Krueger, C., 2004. Salion's experience with a reactive software product line approach. Software Product-Family Engineering, 317–322.

Classen, A., Hubaux, A., Heymans, P., 2009. A formal semantics for multi-level staged configuration. In: Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pp. 51–60.

Classen, A., Boucher, Q., Heymans, P., 2011. A text-based approach to feature modelling: syntax and semantics of tvl. Science of Computer Programming 76 (12), 1130–1143, http://dx.doi.org/10.1016/j.scico.2010.10.005 http://www.sciencedirect.com/science/article/pii/S0167642310001899

Clements, P., Northrop, L., Boehm, B., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Coplien, J., Hoffman, D., Weiss, D., 1998. Commonality and variability in software engineering. Software, IEEE 15 (6), 37–45.

Czarnecki, K., Wasowski, A.,2007. Feature diagrams and logics: there and back again. In: Software Product Line Conference, 2007. SPLC 2007. 11th International. IEEE, pp. 23–34.

Czarnecki, K., Helsen, S., Eisenecker, U., 2004. Staged configuration using feature models. Software Product Lines Conference, 162–164.

Deng, G., Schmidt, D.C., Gokhale, A., Gray, J., Lin, Y., Lenz, G., 2008. Evolution in model-driven software product-line architectures. In: Tiako, P.F. (Ed.), Designing Software-Intensive Systems. IGI Global, http://dx.doi.org/10.4018/978-1-59904-699-0.

Dougherty, B., White, J., Kegley, R., Preston, J., Schmidt, D., Gokhale, A., 2011. Optimizing integrated application performance with cache-aware metascheduling. On the Move to Meaningful Internet Systems: OTM 2011, 432–450.

Elsner, C., Botterweck, G., Lohmann, D., Schröder-Preikschat, W., 2010. Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems. In: Variability in Time-Product Line Variability and Evolution Revisited.

Etxeberria, L., Sagardui, G.,2008. Variability driven quality evaluation in software product lines. In: Software Product Line Conference, 2008. SPLC'08. 12th International. IEEE, pp. 243–252.

FaMa Framework. http://www.isa.us.es/fama (accessed 02.01.13).

Gill, C.D., Loyall, J.P., Schantz, R.E., Atighetch, M., Gossett, J.M., Gorman, D., Schmidt, D.C.,2004. Integrated adaptive qos management in middleware: a case study. In: Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE. IEEE, pp. 276–285.

Hartmann, H., Trew, T., Matsinger, A.,2009. Supplier independent feature modelling. In: Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University, pp. 191–200.

Hubaux, A., Classen, A., Heymans, P.,2009. Formal modelling of feature configuration workflows. In: Proceedings of the 13th International Software Product Line Conference, SPLC 's09. Carnegie Mellon University, Pittsburgh, PA, USA, pp. 221–230 http://dl.acm.org/citation.cfm?id=1753235.1753266

Hwan, C., Kim, P., Czarnecki, K., 2005. Synchronizing cardinality-based feature models and their specializations. In: Model Driven Architecture-Foundations and Applications. Springer, pp. 331–348.

Immonen, A., 2006. A method for predicting reliability and availability at the architectural level. Software Product Lines, 373–422.

Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. Form A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5 (1), 143–168.

Lehman, M.M., 1996. Laws of software evolution revisited. In: Software Process Technology, In: Montangero, C. (Ed.), Lecture Notes in Computer Science, vol. 1149. Springer Berlin Heidelberg, pp. 108–124.

Madl, G., Abdelwahed, S., Schmidt, D.C., 2006. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. Real-Time Systems 33 (1), 77–100.

Mannion, M., 2002. Using first-order logic for product line model validation. Software Product Lines, 149–202.

Mendonca, M., Cowan, D., Oliveira, T., 2007. A process-centric approach for coordinating product configuration decisions. In: 40th Annual Hawaii International Conference on System Sciences, 2007. HICSS 2007, p. 283a.

Metzger, A., Heymans, P., Pohl, K., Schobbens, P., Saval, G.,2007. Disambiguating the documentation of variability in software product lines: a separation of concerns, formalization and automated analysis. In: Requirements Engineering Conference, 2007. RE'07. 15th IEEE International. IEEE, pp. 243–253.

Nechypurenko, A., Wuchner, E., White, J., Schmidt, D.C., 2007. Application of aspect-based modeling and weaving for complexity reduction in the development of automotive distributed realtime embedded system. In: Proceedings of the Sixth International Conference on Aspect-Oriented Software Development.

Olumofin, F., Misic, V.,2005. Extending the atam architecture evaluation to product line architectures. In: 5th Working IEEE/IFIP Conference on Software Architecture, 2005. WICSA 2005. IEEE, pp. 45–56.

Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S., 2012. Model-driven support for product line evolution on feature level. Journal of Systems and Software 85 (10), 2261–2274, http://dx.doi.org/10.1016/j.jss.2011.08.008.

Pohl, K., Böckle, G., Linden, F.J., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Roos-Frantz, F., Benavides, D., Ruiz-Cortés, A., Heuer, A., Lauenroth, K., 2012. Quality-aware analysis in product line engineering with the orthogonal variability model. Software Quality Journal 20 (3/4), 519–565, http://dx.doi.org/10.1007/s11219-011-9156-5.

Rosenmüller, M., Siegmund, N., Pukall, M., Apel, S., 2011. Tailoring dynamic software product lines. SIGPLAN Not. 47 (3), 3–12, http://dx.doi.org/10.1145/2189751.2047866.

Schobbens, P., Heymans, P., Trigaux, J., Bontemps, Y., 2007. Generic semantics of feature diagrams. Computer Networks 51 (2), 456–479.

Shaw, R., 1999. Boeing 737-300 to 800. MBI Publishing Company, Osceola, Wisconsin, ISBN 0-7603-0699-0.

SPLOT. http://www.splot-research.org/ (accessed 02.01.13).

Thum, T., Batory, D., Kastner, C.,2009. Reasoning about edits to feature models. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 's09. IEEE Computer Society, Washington, DC, USA, pp. 254–264, http://dx.doi.org/10.1109/ICSE.2009.5070526.

Van Hentenryck, P., 1989. Constraint satisfaction in logic programming. Logic programming, MIT Press, ISBN 978-0-262-08181-8, I–XXVI, 1–224.

White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A.,2007. Automating product-line variant selection for mobile devices. In: Software Product Line Conference, 2007. SPLC 2007. 11th International. IEEE, pp. 129–140.

White, J., Schmidt, D., Benavides, D., Trinidad, P., Ruiz-Cortés, A.,2008a. Automated diagnosis of product-line configuration errors in feature models. In: Software Product Line Conference, 2008. SPLC'08. 12th International. IEEE, pp. 225–234.

White, J., Schmidt, D.C., Nechypurenko, A., Wuchner, E., 2008b. Model intelligence: an approach to modeling guidance. UPGRADE Journal 9 (2), 22–28.

White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A., 2008b. Automatically composing reusable software components for mobile devices. Journal of the Brazilian Computer Society 14 (1), 25–44.

White, J., Dougherty, B., Schmidt, D., Benavides, D.,2009. Automated reasoning for multi-step feature model configuration problems. In: Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University, pp. 11–20 http://dl.acm.org/citation.cfm?id=1753235.1753266

White, J., Benavides, D., Schmidt, D., Trinidad, P., Dougherty, B., Ruiz-Cortes, A., 2010. Automated diagnosis of feature model configurations. Journal of Systems and Software 83 (7), 1094–1107 http://www.sciencedirect.com/science/article/pii/S016412121000049X

Xiong, Y., Hubaux, A., She, S., Czarnecki, K.,2012. Generating range fixes for software configuration. In: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012. IEEE Press, Piscataway, NJ, USA, pp. 58–68 http://dl.acm.org/citation.cfm?id=2337223.2337231