Evolution in Dynamic Software Product Lines

Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, Christian Schuhmayer

Received 8 November 2018; | Revised 24 April 2020; | Accepted 21 May 2020

DOI: xxx/xxxx

RESEARCH ARTICLE

Evolution in Dynamic Software Product Lines

Clément Quinton[1] | Michael Vierhauser[2] | Rick Rabiser[3] | Luciano Baresi[4] | Paul Grünbacher[5] | Christian Schumayer[5]

[1]University of Lille, CRIStAL UMR CNRS 9189, Inria Lille - Nord Europe. F-59000 Lille, France.
clement.quinton@univ-lille.fr

[2]Department of Business Informatics – Software Engineering, Johannes Kepler University Linz,
Altenberger Str. 69, 4040 Linz, Austria
michael.vierhauser@jku.at

[3]LIT CPS Lab, Johannes Kepler University Linz
Altenberger Str. 69, 4040 Linz, Austria
rick.rabiser@jku.at

[4]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Piazza L. Da Vinci 32, 20133 Milano, Italy
luciano.baresi@polimi.it

[5]CD Lab MEVSS, Institute for Software Systems Engineering, Johannes Kepler University Linz,
Altenberger Str. 69, 4040 Linz, Austria
paul.gruenbacher@jku.at,
schuhmayer.christian@gmx.at

Correspondence
C. Quinton, clement.quinton@univ-lille.fr

Summary

Many software systems today provide support for adaptation and reconfiguration at runtime, in response to changes in their environment. Such adaptive systems are designed to run continuously and may not be shut down for reconfiguration or maintenance tasks. The variability of such systems has to be explicitly managed, together with mechanisms that control their runtime adaptation and reconfiguration. Dynamic software product lines (DSPLs) can help to achieve this. However, dealing with evolution is particularly challenging in a DSPL, as changes made at runtime can easily lead to inconsistencies. This paper describes the challenges of evolving DSPLs using an example cyber-physical system for home automation. We discuss the shortcomings of existing work and present a reference architecture to support DSPL evolution. To demonstrate its feasibility and flexibility, we implemented the proposed reference architecture for two different DSPLs: the aforementioned cyber-physical system, which uses feature models to describe its variability, and a runtime monitoring infrastructure, which is based on decision models. To assess the industrial applicability of our approach, we also implemented the reference architecture for a real-world DSPL, an automation software system for injection molding machines. Our results provide evidence on the flexibility, performance and industrial applicability of our approach.

KEYWORDS:
Dynamic Software Product Lines, Evolution, Consistency

1 | INTRODUCTION

An increasing number of software systems today are adaptive systems that provide runtime adaptation and reconfiguration capabilities to react to changes in their environment. These systems usually run continuously and cannot be shut down for reconfiguration or maintenance tasks. For instance, cyber-physical systems must frequently reconfigure their software components at runtime to take into consideration the addition, removal or update of physical devices. *Dynamic Software Product Lines* (DSPLs) [1] provide the conceptual framework for managing the variability in such systems. A DSPL borrows the means to define and manage variability from conventional software product lines, but additionally supports system reconfiguration at runtime. Similar to conventional software product lines, variability models in a DSPL define the commonalities and variability of system artifacts with domain-specific properties and dependencies. They describe the possible variants of a system together with constraints and dependencies and can cover the system's problem space, i.e., stakeholder needs and desired features, as well as its solution space, i.e., the components realizing the solution architecture. The mappings between these two spaces then allows to assemble and configure a product based on customers' requirements [2,3]. In a DSPL, the variability model also has to describe how the system can be adapted at runtime. Models providing such support have been called open variability models [4].

Changes in the requirements and newly emerging technologies lead to the continuous evolution of DSPLs: for example, new features may need to be added or existing features may need to be removed. Managing evolution is particularly difficult in a DSPL context, as changes are made at

runtime, which can easily lead to inconsistencies among running components. Specifically, it is challenging to maintain the consistency between the problem space and the solution, the variability model and the running system, as well as the runtime adaptation mechanisms. Many approaches have been proposed for managing the evolution of software product lines [5], ranging from verification techniques to ensure consistent evolution, to model-based frameworks dedicated to the evolution of feature-based variability models [6]. For example, an interesting research thread proposes evolution templates for co-evolving a variability model and related software artifacts [3,7,8]. Model-checking approaches are used to guarantee the consistency of a variability model after evolution [9,10]. Furthermore, approaches for comparing the set of possible products before and after the evolution of a product line have been proposed [11,12]. These approaches, however, are limited regarding support for DSPL evolution, as they focus on guaranteeing the consistency of the evolved DSPL variability model but fail to ensure that the evolution is consistent with the DSPLs actual implementation and adaptation mechanisms.

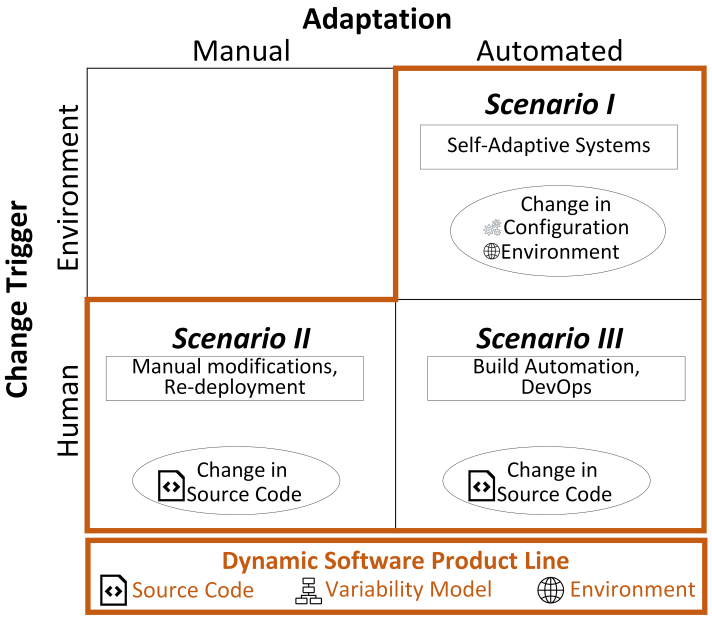


FIGURE 1 Overview of different types of changes (manual, or automated) and the respective triggers (triggered by a human, or by the environment) that can have an impact on the system. Certain scenarios are part of other research areas as well, but a DSPL needs to consider multiple different scenarios alongside with the PL variability model.

There are three evolution scenarios in a DSPL context (cf. Fig. 1). First, the change and the related adaptation are triggered manually, *e.g.*, when updating the code base. Second, the change can be performed manually, while the adaptation is automated *e.g.*, in a build automation or DevOps process. Finally, both the change and the adaptation can be triggered automatically, which is the case for self-adaptive systems where evolving environment implies a reconfiguration of the system. Besides the area of (dynamic) software product lines, considerable effort has been made in the domain of (self-)adaptive systems to efficiently adapt to changes in the environment, or recover from errors introduced in a running system. Examples range from autonomous vehicles or robots [13], to self-managing and self-adapting web services and IoT systems [14]. Self-adaptive systems (often using a MAPE-K [15] approach) react to changes in the environment to, for example, optimize throughput and reduce resource consumption by updating the configuration or architecture of the system. Automated build and deployment solutions (DevOps) on the other hand are triggered by (manual) changes in the program's source code and ensuring proper re-deployment of the system, sometimes in an incremental manner and/or at runtime. However, while these solutions provide important contributions to (self)-adaptivity and self-repair of systems at runtime, what sets them apart from a DSPL is their lack of explicitly modeling the *variability* of the system. This is important as the the running system must to be consistent with its counterpart described, for example, in a feature model (cf. Fig. 1).

Existing DSPL approaches are typically based on ad-hoc adaptation mechanisms that need to be maintained manually to keep them consistent with the problem and solution spaces during evolution [16]. Proper evolution support is, however, crucial to guarantee the consistency of the DSPL and of the (potential) adaptations of the running system [17]. Furthermore, existing approaches typically only support one particular variability management approach and are not flexible enough to allow their use in different domains and for different types of systems, using diverse implementation

techniques. A variability model-agnostic approach is still missing that facilitates the evolution of problem and solution spaces, together with the runtime adaptation mechanisms, and also checks the consistency of the resulting products. Based on our work on runtime reconfiguration of monitoring systems [18,19] and on the dynamic evolution of the architecture of DSPLs [20], we sketched some general issues related to DSPL evolution in a short paper [21]. The research described here builds on this short paper. Specifically, we developed *a flexible approach to support DSPL evolution* based on *a reference architecture, that can be implemented to support the evolution of a concrete (type of) DSPL*. The reference architecture describes the capabilities needed for detecting changes made to running systems, for automating the update of variability models, and for detecting inconsistencies among all the relevant elements of a DSPL. Specifically, our reference architecture guides the development of change detection, model update, and consistency checking solutions to support evolution in a concrete DSPL.

The key contributions of this paper are *(i)* a flexible approach, based on a reference architecture, to implement evolution support for a DSPL, *(ii)* implementations of the reference architecture for two different DSPLs in different domains – one cyber-physical system and one runtime monitoring system, both using different means for variability management, *(iii)* an evaluation of the feasibility and performance of our approach by simulating common evolution scenarios for both DSPLs to demonstrate that both implementations are capable of detecting inconsistencies introduced in a DSPL at runtime, and *(iv)* a demonstration of the industrial applicability of our approach by applying it to a real-world automation software system DSPL from the injection molding domain.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to DSPLs and introduces a running example. Section 3 discusses important evolution scenarios and their impact on the consistency of a DSPL and summarizes the challenges of DSPL evolution. Section 4 presents a reference architecture for implementing support for DSPL evolution. Section 5 describes our evaluation method. Section 6 describes the implementation of evolution support for two different DSPLs based on the reference architecture and performance experiments we conducted. Section 7 describes how our proposed reference architecture was implemented for a real-world automation software system from the injection molding domain by an engineer of MoldingCompany[a]. Section 8 discusses our results and elaborates threats to validity. Section 9 discusses related work, and Section 10 concludes the paper.

2 | BACKGROUND AND RUNNING EXAMPLE

Product line engineering aims at building software systems by reusing common software artifacts across a set of related products [22]. Managing the commonalities and variability of product lines is a cross-cutting concern [22,23]. Modelling product line variability concerns the problem space (*i.e.*, features and capabilities), the solution space (*i.e.*, components of the solution architecture), and the mapping space (*i.e.*, links between problem and solution space elements) [24,25].

2.1 | Background

A DSPL is a special form of a software product line designed to support runtime variability, that is, adaptation and reconfiguration at runtime. The DSPL engineering process (Fig. 2) extends the SPL process by adding post-deployment and reconfiguration activities [26]. In contrast to SPLs with variation points bound only at design time DSPLs allow to explicitly bind features at runtime. A variability model then describes all permissible reconfigurations [27], which enable the running system to adapt to changes in the environment, relying, *e.g.*, on context information [28]. This information in turn is captured and analyzed through the use of dedicated tools, which then trigger reconfigurations.

Fig. 2 depicts an abstract system whose variability is described in a feature model. Each feature in the feature model is realized by at least one related software artifact – *e.g.*, a service or a component – called *asset*. For the sake of simplicity, we do not include cross-tree constraints in this figure but want to point out that the architecture described further is well-suited to handle them. Also, we do not assume a 1-to-1 mapping of problem space features to solution space assets, *i.e.*, one feature can be realized by multiple assets and several features can be realized by one asset. This mapping of assets and features – here abstracted using a simple arrow – is typically specified using a dedicated language [24].

Adaptation rules state how the system can evolve. For example, *R4* specifies that feature *A* must be activated during reconfiguration *whenever condition N is met* (selected features are marked with a green check mark). After deploying the initial configuration, changes in the context of the running system may result in *condition N* to be met, leading to the inclusion of feature *A*. All phases of runtime adaptation (analysis, planning and execution) rely on the variability model. Ensuring its consistency at runtime is thus fundamental for DSPL engineering.

[a]The name has been changed to protect the company's privacy.

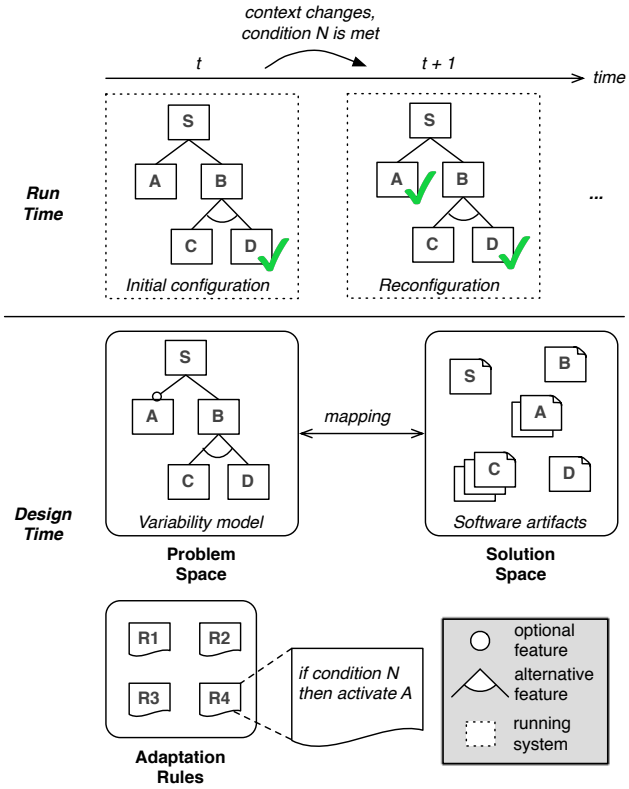


FIGURE 2 DSPL engineering.

2.2 | Running Example

To illustrate the challenges and issues faced when evolving a DSPL, we introduce a cyber-physical system (CPS) for home automation as an example of an adaptive system combining both hardware devices and software systems. The CPS consists of smart devices equipped with sensors and actuators interconnected through a software system. Sensors are used to retrieve information from the environment; reconfiguration plans are then carried out through the actuators. Fig. 3 depicts an excerpt of the variability model of our CPS, along with related adaptation rules.

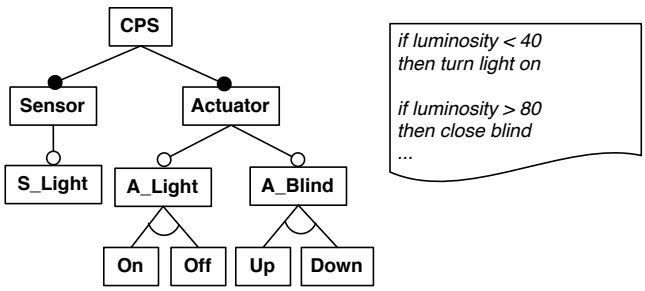


FIGURE 3 Excerpt of the variability model of a cyber-physical system and its related adaptation rules.

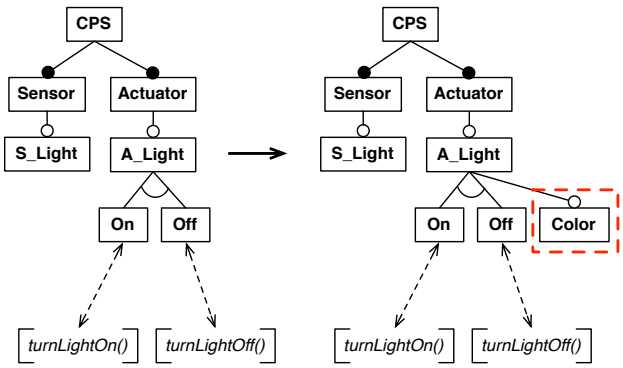
In this case, a possible reconfiguration action is to switch on light through the actuator *A_Light* whenever the luminosity in a given room, captured by sensor *S_Light*, is lower than 40 lumens, or to close the roller blinds of the window, whenever it is higher than 80 lumens. Obviously, a CPS automatically managing the luminosity, temperature, humidity, and energy consumption of a home relies on many such adaptation rules for different rooms and devices. Each adaptation rule is thus related to the variability model describing possible reconfigurations of the CPS.

3 | DSPL EVOLUTION

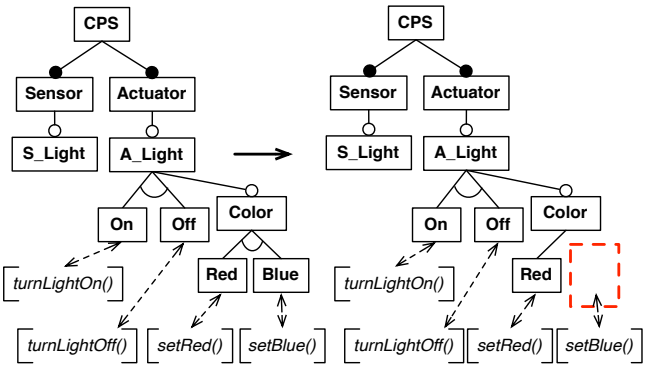
A DSPL is a long-term investment and is meant to be used over a long period of time. As any other software system, it needs to evolve to meet new requirements. For instance, new devices may be configured to support new kinds of adaptations or existing reconfiguration mechanisms may be changed or removed. When evolving a DSPL, developers have to cope with the variability of the different spaces, while ensuring that adaptations that may occur within the running system are still feasible. Intra-space changes affect only one space while inter-space changes involve different spaces [3]. In addition, runtime changes of DSPL elements defined at design time often have an impact on the reconfigurability of the deployed system. Evolving a DSPL is thus an error-prone and challenging task. This section *discusses all the possible changes that can occur in the three modeling spaces of the DSPL during evolution and their potential effect on the running system*. We present examples of these effects on the runtime adaptations of the CPS DSPL, and then discuss the related challenges.

3.1 | Impact of Evolution

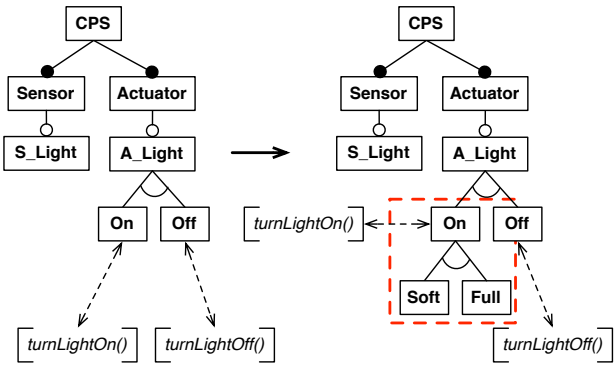
In the following, the variability of the CPS is described using a feature model, the most common approach to model product lines. Although we describe only one change per space in our examples, the changes can affect the consistency of the overall DSPL. The left-hand side of each (sub-)figure shows the initial DSPL, and the right-hand side the one after evolution. Elements depicted with square brackets are software artifacts from the solution space, *e.g.*, components or services. Throughout our examples, we assume that an operation, whose signature is given between the square brackets, is implemented by these software artifacts. Dashed lines with arrows on both sides represent the mapping between problem and solution spaces. Rectangles in dashed lines highlight the resulting inconsistencies.



(a) Adding an element to the problem space.



(b) Removing an element from the problem space.



(c) Updating an element of the problem space.

FIGURE 4 Problem space evolutions.

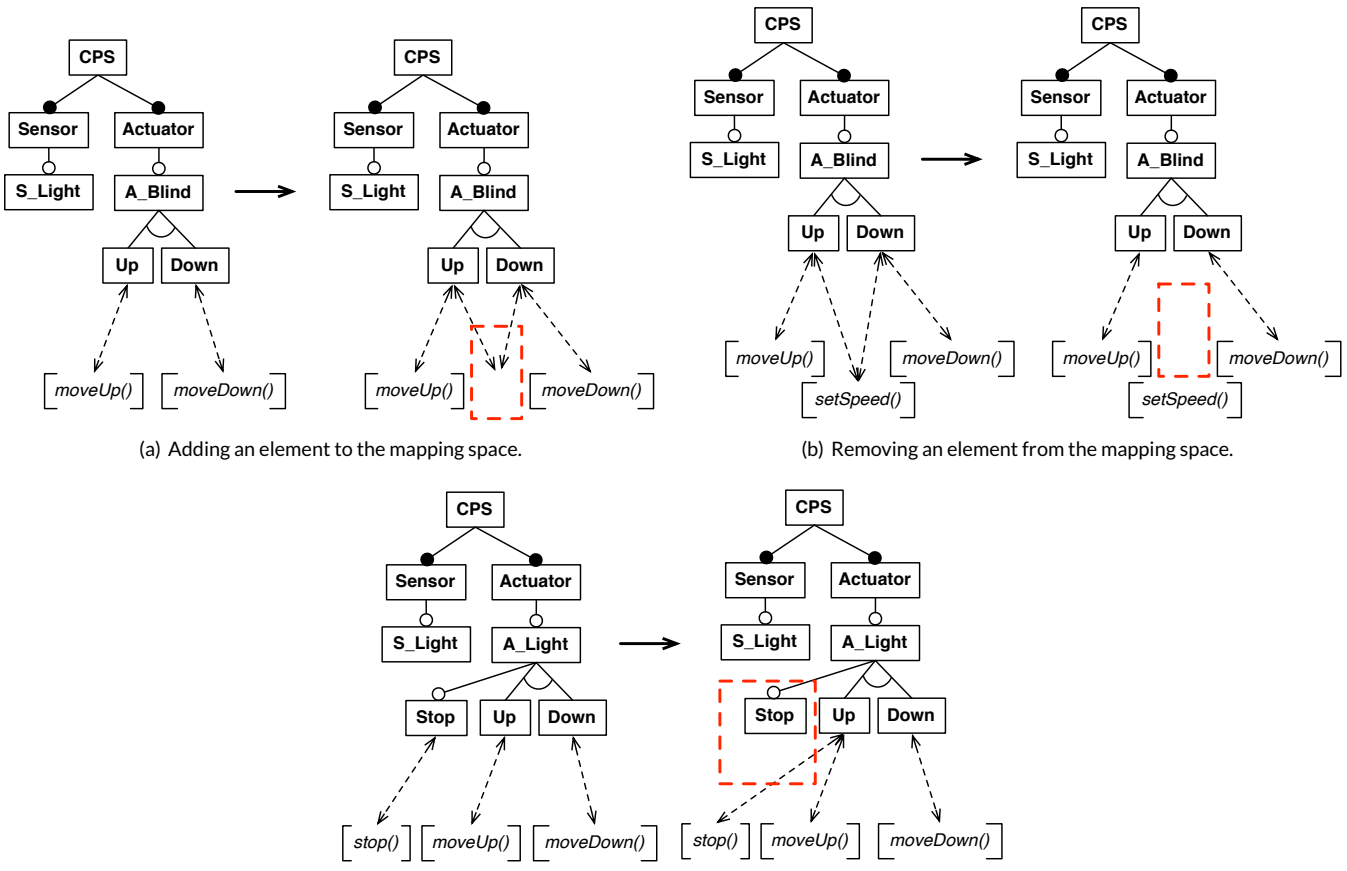
Problem Space

For instance, in Fig. 4(a) the optional feature *Color* is added to the system to make the light actuator now configurable by allowing the selection of a light color. This feature may require specific colored light bulbs. The problem in this case is caused by the missing software artifact that needs to be mapped to the new feature. In our example, *Color* must be defined and included in the adaptation rule, since otherwise the *Color* feature is empty (has no associated operation), i.e., the reconfiguration will have no effect[b]. The problem occurs immediately if the feature is mandatory, or when activating an optional feature.

Removing a feature can easily impact the consistency of the DSPL. Let us consider a case where the color of the light could have been red or blue, but the latter cannot be set anymore after evolution, *e.g.*, because of a hardware problem (Fig. 4(b)). If feature *Color* is involved in an adaptation rule, *e.g.*, turn on the blue light when it is 07:00 in the morning, then the reconfiguration would fail whenever this condition is met, even though the related solution space element is available.

A feature, or any other element of the solution space, *e.g.*, a constraint, can also just be updated[c]. For example, one can think of a different way of switching lights on and off. The variability model is updated when feature *On* is upgraded, as a dimmer now allows the light to be turned on completely or partially (*Full* and *Soft* modes in Fig. 4(c)). Adaptation rules that involve feature *On* (Fig. 3) need to be updated to reflect this change and to avoid an inconsistency. Updating the rules is done by setting the correct action to perform, *e.g.*, when the luminosity becomes lower than 40 lumen, turn on the light in soft mode.

Mapping Space



(a) Adding an element to the mapping space.

(b) Removing an element from the mapping space.

(c) Updating an element of the mapping space.

FIGURE 5 Mapping space evolutions.

[b] In the context of this paper, we consider that a feature is empty when its selection has no effect on the expected adaptation.

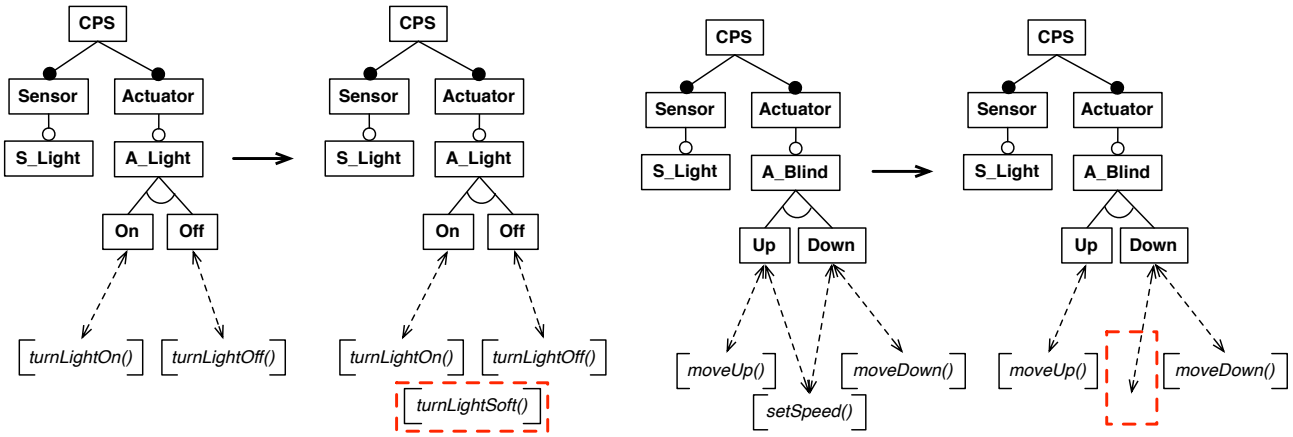
[c] An update is considered as a transaction, that is, a sequence of atomic changes treated as one change (*e.g.*, *Add* + *Rem*). The DSPL is thus checked only once (after the update) instead both after the *Add* and after the *Rem*.

A new mapping can be added when a feature requires an extension and is then realized by composing two different software artifacts. For example, Fig. 5(a) considers the case in which the implementation of features *Up* and *Down* for blinds requires two software artifacts after evolution: one for controlling the speed and another one for the direction (we assume both were handled by the same artifact before). If the mapping is not correctly defined and does not point to any artifact, the adaptation is partially inconsistent, *e.g.*, one can still control the direction (up/down) of the blinds but not their speed since the related artifact is not mapped correctly.

Removing a mapping element can prevent the proper reconfiguration of the running system. In Fig. 5(b), conversely to the previous scenario, artifacts *moveUp()* and *moveDown()* also control the speed after evolution, so the mapping related to speed is removed. In that case, if the software artifacts are not properly updated, the element of the solution space that controls the speed (*setSpeed()*) becomes a dead asset.

When updating a mapping, the reference to the element needs to be changed either in the problem space or in the solution space. For example, Fig. 5(c) illustrates the case where artifact *Stop()*, which was initially bound to feature *Stop*, is now bound to feature *Up*: if one pushes button *Up* when the blind moves down, it stops, instead of having a dedicated button *Stop*. Adaptations related to feature *Stop* would now fail, as software artifacts are no longer bound to this feature. For instance, such an adaptation could require that the blind stops when there is too much wind or rain.

Solution Space



(a) Adding an element to the solution space.

(b) Removing an element from the solution space.

(c) Updating an element of the solution space.

FIGURE 6 Solution space evolutions.

Fig. 6(a) considers a new artifact that implements a soft new way of turning on the light. If this new software artifact should be taken into consideration for the reconfiguration at runtime, related elements in the mapping as well as in the problem space must also be added.

Removing a software artifact when evolving the solution space can lead to problems when different features share the same artifact. Fig. 6(b) illustrates this situation as artifact *setSpeed()* is removed. In this scenario, feature *Up* no longer needs such an artifact (we suppose this is now implemented by artifacts *moveUp()* and *moveDown()*). This can result in a partially inconsistent adaptation, *i.e.*, the blind can still move down but the speed cannot be controlled, which can be an issue.

Finally, Fig. 6(c) illustrates an evolution in the solution space, where the implementation of feature *On* is changed. After the change, activating this feature turns on the light in soft mode. Although the resulting reconfiguration turns the light on (in soft mode, but on anyway), this may not be the expected behavior with respect to the semantics of feature *On*. The evolved software artifact could also be an implementation that is completely unrelated to the light system and turns on the TV, thus leading to another inconsistent adaptation.

Table 1 summarizes the different evolution scenarios in the problem and solution spaces and the mapping between the two, with respect to atomic tasks/changes as discussed above. It further summarizes the possible effects of each change on the consistency of the CPS DSPL and the impact on the running configuration.

TABLE 1 Changes in different modeling spaces and examples of their potential effect on the CPS DSPL. `Add` stands for addiction, `Rem` for removal and `Upd` for update.

	Change	Potential impact on the CPS DSPL
Problem Space	`Add`	**feature selection will have no effect** : the added *Color* feature (cf. Fig 4(a)) is empty[2], *i.e.*, not related to any solution space element, and selecting it will thus have no effect.
	`Rem`	**artifact cannot be activated anymore** : *setBlue()* (cf. Fig 4(b)) cannot be activated as it is no longer related to any problem space element.
	`Upd`	**feature selection will have no effect** : *Soft* and *Full* (cf. Fig 4(c)) are empty features, *i.e.*, not related to any solution space element, and selecting them will thus have no effect.
Mapping Space	`Add`	**artifact cannot be activated anymore** : the Blind speed cannot be controlled (cf. Fig 5(a)) as the new mapping does not point to a solution space element.
	`Rem`	**artifact cannot be activated anymore** : *setSpeed()* (cf. Fig 5(b)) cannot be activated as after removing the mapping, no feature is related with *setSpeed()* anymore.
	`Upd`	**feature selection will have no effect** : feature *Stop* (cf. Fig 5(c)) is empty, *i.e.*, not related with any solution space element, and selecting this feature now will not have any effect anymore.
Solution Space	`Add`	**artifact cannot be activated anymore** : *turnLightSoft()* (cf. Fig 6(a)) cannot be activated, as no feature is related to the new solution space element.
	`Rem`	**runtime adaptation will partly fail** : feature *Down* (cf. Fig 6(b)) is not fully implemented, *i.e.*, the speed can no longer be controlled as this solution space element has been removed.
	`Upd`	**unexpected runtime behavior** : wrong implementation (asset) for feature *On* (cf. Fig 6(c)) leads to unexpected behavior.

3.2 | Challenges

The discussed cases show that evolution in DSPLs requires the consideration of multiple aspects [21]. We see three main challenges:

C_1: *Supporting evolution independently of the domain, implementation technique, or modeling approach.* Existing DSPL evolution approaches focus on a particular variability modeling approach, *e.g.*, feature models [26]. In practice, however, different approaches are used [29]. Also, different ways of modeling adaptation rules can be utilized in relation to variability models. Furthermore, systems are often implemented using various technologies. The success of an approach for the evolution of DSPLs in practice thus depends on its flexibility to support different modeling and implementation techniques. This suggests a generic architecture that guides the implementation of concrete solutions, which use specific variability modeling approaches and adaptation mechanisms.

C_2*: Ensuring consistency of models and the running system.* The consistency of the DSPL must be checked whenever at least one of the spaces evolves. Consistency must thus be checked for each space (intra-space consistency) and across spaces (inter-space consistency). For instance, related elements from different spaces can become inconsistent with the actual software or hardware and prevent the derivation of products, despite the validity of these products in the variability model. Although patterns for keeping co-evolving both modeling spaces consistent have been studied [30,31], most approaches so far have focused on the consistency of either the problem space [11,32,33] or the solution space [34]. Also, existing work focuses on checking the consistency (of models) in software product lines without considering dynamic software product lines: while model consistency is similar in a SPL and a DSPL, consistency checking in a DSPL also needs to take into account the *running* system. Thus, it must be ensured that its adaptation rules are still consistent with the variability model and do not violate any possible reconfiguration. Detecting such inconsistencies is not straightforward as adaptation rules are typically defined independently of the variability model using diverse *ad-hoc* approaches such as event-condition-action rules [16,35,26]. When the model evolves, the rules should be updated automatically. Also, when evolving rules, the model must potentially also be updated.

C_3*: Supporting evolution triggered by the running system or the model.* The evolution of a DSPL can be driven from two different perspectives. First, the different spaces can evolve, *e.g.*, a feature might be added to the problem space or a new component might be developed in the solution space to address new requirements. Whenever one space evolves, the other space must evolve accordingly. Only then a configuration defined in the problem space can be materialized by composing elements from the solution space (*e.g.*, Fig. 4(a)–4(c)), and vice versa (*e.g.*, Fig. 6(a)–6(c)). Updating a running system – to reflect changes made to the modeling spaces also in the running system – can be challenging depending on the technologies used. Second, the evolution[d] of a DSPL can also be driven by the running system: if the system changes both modeling spaces may need to evolve to reflect these changes. Approaches have been proposed for dealing with the co-evolution of different modeling spaces [3,36], for reflecting system evolution in the models [37,38], or vice versa [39]. However, they are typically only capable of dealing with a particular set of changes and cannot handle both evolution between modeling spaces and evolution driven by changes in the system.

4 | REFERENCE ARCHITECTURE

To address the aforementioned challenges, we present a reference architecture [40] that supports DSPL evolution.

Reference architectures

A number of approaches exist to inform the development of reference architectures [41,42]. Following the types of reference architecture proposed by Angelov et al. [43] we regard our reference architecture as a Type 5. Such architectures are designed to facilitate the design of systems that will become needed in the future. Our reference architecture defines the key components required in a system implementing it, discusses algorithms supporting the operation of the components, and presents protocols demonstrating the interactions among the components.

A reference architecture aims at *providing guidance (i.e., instructions) on how to actually design a system* [44]. Our reference architecture *(i)* is independent of specific domains, implementation techniques, or modeling approaches for DSPLs (cf. challenge C_1); *(ii)* it focuses on ensuring the consistency of the running system and of the models representing the DSPL (cf. challenge C_2); and *(iii)* supporting evolution triggered from different perspectives (cf. challenge C_3).

The reference architecture is divided into four parts, as depicted in Fig. 7. *DSPL Adaptation* comprises the adaptation rules defined to automatically adapt the system at runtime. *Change Detection and Propagation* comprises a component that listens to the running system to detect any (relevant) change made to the system and subsequently propagates this change to the *Model Evolution* component. *Model Evolution* comprises components to evolve the variability model and/or the adaptation rules based on the changes received from Change Detection and Propagation, given the specific scenario (see Section 3). *Model Consistency* uses a consistency checker to ensure that performed evolution operations, especially if based on human decisions, do not introduce inconsistencies in the DSPL. The parts, each responsible for a given concern in the DSPL evolution process, and the components they comprise, are described independently of any concrete (modeling) approach and can be implemented for any systems. Small squares in the components indicate where such specific implementations are needed, e.g., to let the Model Updater update the Variability Model for a concrete variability modeling approach. In Table 2, we list generic operations for each component, *e.g.*, to listen to changes made to the running system and to update the variability models accordingly. These operations can be implemented when creating a concrete solution for a DSPL. To this end, we describe two different implementations in Section 6 . Below, we describe each part of the reference architecture and the components it comprises, again referring to the CPS example.

[d] Please note that in our approach, an evolution results in a new version of the DSPL. The previous version of the variability model is replaced with the evolved one and earlier configurations may no longer be derivable.

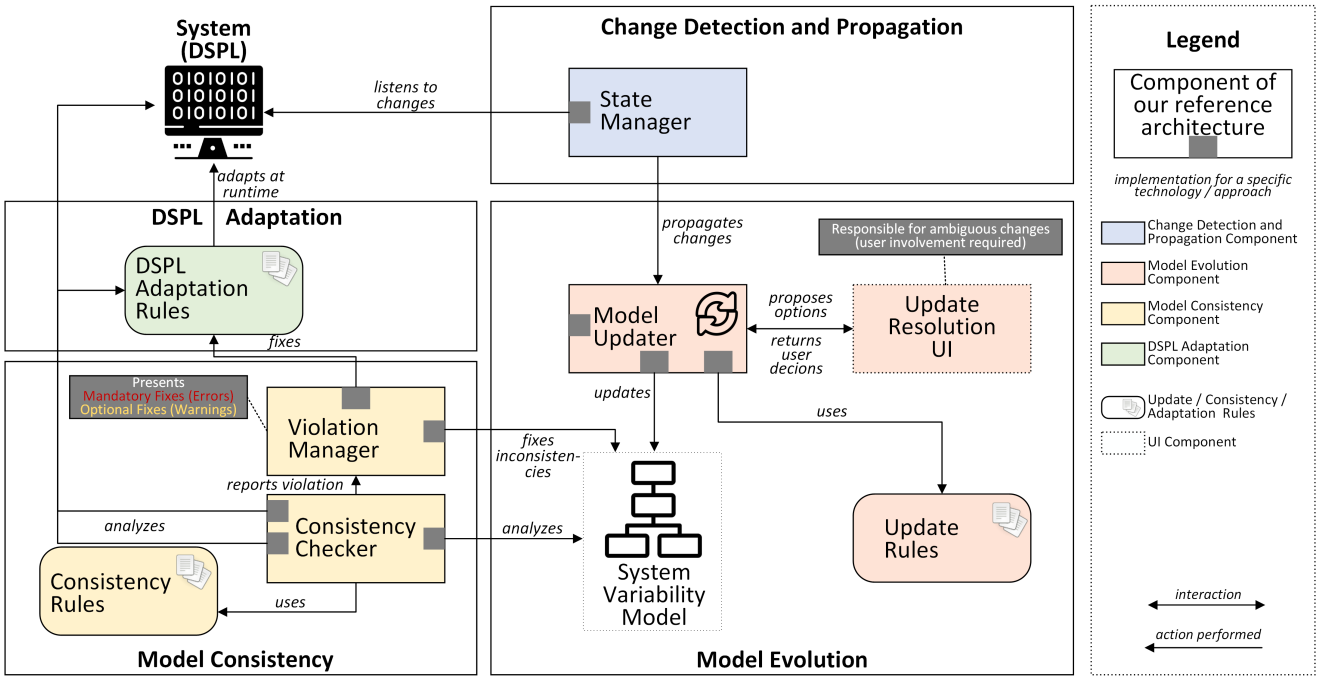


FIGURE 7 Reference architecture for DSPL evolution.

DSPL Adaptation is responsible for managing all adaptation rules related to the DSPL. The specific way adaptation rules are described depends on the domain and the implementation of the DSPL. For instance, the rules may be encoded using a domain-specific language that describes event-condition-action rules [16,35,26]. When a change occurs in the running system, the adaptation rules are likely to evolve together with the rest of the DSPL, e.g., as an adaptation rule may involve a feature that is no longer present in the variability model after the change. For example, an adaptation rule defining the color of the light to be turned on under certain conditions will no longer be valid after the DSPL evolution scenario described in Fig. 6(b), and thus the rule must be adapted or removed from this component to properly evolve the DSPL (cf. challenges C_1 and C_2).

Change Detection and Propagation is responsible for detecting and propagating changes in the running system. A *State Manager* monitors the running system and receives notifications whenever changes occur. This could be achieved either by actively listening to system changes, or by periodically querying the system and calculating changes with respect to a previously observed state. When a change is detected by the *State Manager*, it is passed to the *Model Updater*, which determines relevant update actions for the variability model (cf. challenge C_3).

In **Model Evolution**, the variability model defines all possible configurations for the software managed as DSPL, and thus guides all adaptations, that is, it switches from one configuration to another given a particular adaptation rule. The variability model can be defined using any available variability modeling approach [29]. Our architecture distinguishes between two types of changes on the variability model: in case of *unambiguous* changes – *e.g.*, the CPS system gets a new feature to turn on a green light – the variability model can be updated immediately and automatically: a new feature *Green*, can be added as a child of feature *Color* (cf. Table 1: PS_Add). In practice, however, multiple stakeholders may maintain a DSPL. This can lead to ambiguous changes, which cannot be resolved automatically. The DSPL maintainer needs to be prompted through an interface – the *Update Resolution UI* – to decide on how to react and how to update the variability model by selecting among suggested possible actions or performing a custom one. For instance, the scenario described in Fig. 4(a) is not trivial, as feature *Color* can be added as a child of feature *A_Light* or of feature *On*. An ambiguous change can also occur when a new component with a different name substitutes a running component, leading to semantic issues that could only be solved using ontologies to map components together, *e.g.*, as in [45]. *Update Rules* help automate model updates (cf. challenge C_3). For instance, an update rule may specify that an optional feature must be added to the variability model as a child of the root feature if a new optional functionality is available in the system. Whenever a change is ambiguous, the DSPL maintainer performs a manual evolution using the *Update Resolution UI* and can then define a new rule based on the change she has just made. Once stored with the other rules, the rule can be used to automate future evolution scenarios to avoid possible ambiguity. *Update Rules* and *Model Updater* depend on the used approach for modeling variability. Our architecture provides generic operations (cf. Table 2) that can be implemented for a specific approach, to specify update operations and update rules, as described in Section 6. The list of possible changes in different modeling spaces (cf. Table 1) is a useful basis to develop update rules, *i.e.*, changes in problem, mapping, and solution spaces can be automated in reaction to the evolution of the DSPL.

TABLE 2 Generic operations of our DSPL evolution reference architecture.

Component/Operation	Description
State Manager	
`detectChange(changeEvent)`	Monitors the running system and listens to any `changeEvent`, *e.g.*, new component deployed to the running system.
`notifyChanges(change[])`	Periodically sends the list of `change[]` to the *Model Updater*.
Model Updater	
`analyzeChanges()`	Reacts to `notifyChanges(change[])` and analyzes the required changes. Calls update methods for variability model and/or DSPL adaptation rules accordingly.
`updateVariabilityModel(modelChange)`	Searches for an update rule matching the required `modelChange`, *e.g.*, add feature. If found, it parses the rule to update the model automatically, *e.g.*, by adding a feature. If an automatic update is not possible, *e.g.*, in case of ambiguous changes, the operation prompts the user.
`readUpdateRule(rule)`	Parses the related update `rule` to infer the proper automated model evolution to be applied.
`updateAdaptationRule(rule, ruleChange)`	Searches for the DSPL adaptation `rule` to be evolved and performs the required `ruleChange`. *E.g.*, when a feature has been renamed in the model, the referring adaptation rule needs to be modified too.
Consistency Checker	
`parse(model)`	Loads and reads the variability `model`.
`parse(adaptationRules)`	Loads and reads the DSPL `adaptationRules`.
`checkConsistency(model)`	Checks the consistency of the variability `model`, *i.e.*, checks for issues in different modeling spaces and also compares variability model and running system with each other according to the defined consistency rules.
`checkConsistency(adaptationRules)`	Checks the consistency of the DSPL `adaptationRules`.
`checkConsistency(model, adaptationRules)`	Checks if the variability `model` is consistent with the DSPL `adaptationRules`
Violation Manager	
`fixInconsistencyInModel(modelChange)`	Updates the model to fix an inconsistency. Prompts the user if required.
`fixInconsistencyInAdaptationRule(ruleChange)`	Updates the adaptation rule to fix an inconsistency. Prompts the user if required.

Model Consistency is essential when dealing with evolution in a DSPL, especially when different people are responsible for different parts of the system and or variability models. In our approach, different possible inconsistencies are defined as *Consistency Rules* (or constraints) that can be fed to a dedicated *Consistency Checker* (cf. challenges C_2 and C_3). The *Consistency Checker* analyzes the running system together with the variability model and the adaptation rules to detect any inconsistencies. The *Consistency Checker* thus needs to interface with the variability model but also requires status updates of the running system.A consistency rule could, for instance, specify that for each problem space feature there must be at least one solution space asset realizing that feature, *i.e.*, *for each feature at least one component must exist in the system that realizes that feature*. Another rule could define that *for each feature selected for the DSPL (i.e., 'activated'), the respective component realizing the feature must be currently running ('active') in the system*. The concrete set of consistency rules will depend on the system and on the modeling approach used, as we will show in our evaluation.

In case of any detected inconsistencies, a *Violation Manager* prompts the user through a dedicated interface. This could involve critical errors that must be fixed (*e.g.*, if the removal of an element from the variability model breaks an existing adaptation rule), but also minor inconsistencies needing attention (*e.g.*, if the addition of an asset duplicates an existing one). The *Consistency Checker* is thus in charge of detecting structural inconsistencies after they occurred, while semantic ones (*e.g.*, mapping to the wrong elements) are left to the user. Again, however, ontologies could be applied

to also cover at least basic semantic inconsistencies. The different parts of our reference architecture are independent of each other, leaving the concrete implementation and application scenario to decide the technique or modeling approach to be used.

5 | EVALUATION METHOD

We investigate three research questions regarding the feasibility and applicability of our approach. Specifically, we assess the general feasibility of our approach by implementing it for two different DSPLs, and determine for the two concrete implementations of the reference architecture if inconsistencies arising from different evolution scenarios are correctly managed. We further show how our reference architecture can be applied to a real-world DSPL in the domain of automation software for injection molding machines.

5.1 | Application Examples

The first DSPL is a cyber-physical system providing capabilities for controlling and managing home automation devices such as sensors and actuators under certain conditions, *e.g.*, turning off the TV in case no one is watching it [46]. To deal with the context-awareness of the devices in use, *e.g.*, a Belkin WeMo thermostat, adaptation rules are defined and managed by controllers deployed on set-top boxes. For instance, an adaptation rule for this device is *if someone turns on the thermostat, send an SMS to registered user #1*. Such systems are highly configurable and likely to evolve, as running devices may face failures, or new devices may be added to an existing system to provide new functionality. For more details about the different spaces of the CPS DSPL, refer to Romero *et al.* [46].

The second DSPL is an event-based runtime monitoring infrastructure REMINDS [18]. REMINDS has a client-server architecture: systems are instrumented using probes that send events and data from the systems to the REMINDS server, which aggregates and distributes these events and data to registered clients. Clients are, for instance, tools for checking constraints [47] on the expected behavior based on the monitored events or visualization components explaining constraint violations to facilitate diagnosis. In previous work, we emphasized the need for sophisticated runtime variability management mechanisms [19] and support for automated evolution in REMINDS [21], since a monitoring infrastructure must co-evolve with the underlying system it monitors. In the context of this paper, however, we focus on the evolution of the components of REMINDS itself: the probes, monitored events and data, and constraints being added, modified, or removed at runtime in the REMINDS monitoring infrastructure.

5.2 | Research Questions

RQ1. Is the reference architecture flexible enough to support different DSPL implementations?

To assess the feasibility of the proposed architecture, we implemented it for the two different DSPLs described above. For both DSPLs, we implemented the components and operations of our reference architecture and created variability models, partly based on existing ones [19]. The two implementations use different technologies and different kinds of variability models, yet they both comply with the reference architecture. For example, the CPS DSPL uses Eclipse EMF [48], Java, and extended feature models [32,49], while REMINDS uses Eclipse, Java, and DOPLER [50] decision models.

RQ2. How well do the two reference architecture implementations perform?

To assess how well the two implementations manage inconsistencies to support DSPL evolution, we simulated scenarios that are likely to happen in the two applications and evaluated their impact on the respective DSPL, to find out whether each implemented approach is able to detect inconsistencies and to react appropriately/fast enough. Table 3 provides an overview of our evaluation setup. We used problem space, mapping space, and solution space operations to cover a fair range of different possible inconsistencies and picked the following representative scenarios:

1. Scenario 1 (SC1) – A problem space element is removed to constrain possible (re-)configurations of the DSPL. This can result in dead elements (assets) in the solution space that were related to the removed element of the problem space.
2. Scenario 2 (SC2) – The mapping space is updated after merging two existing assets into a single one. This can result in a problem space element with no effect (no relation to any solution space element).
3. Scenario 3 (SC3) – A solution space element is added to evolve the system, i.e., a new component can now be configured. If not mapped to an existing or new problem space element, the new solution space element is dead and cannot be activated at runtime.

For each of these three scenarios we perform 100 changes that lead to an automated update of the respective variability model of the CPS and REMINDS DSPLs. 99 out of 100 changes lead to correct updates while 1 change (randomly, *e.g.*, the 67th change) leads to an incorrect update resulting in an inconsistency. For example, an incorrect update operation for scenario SC3 would add the solution space element without also relating it

TABLE 3 Overview of our evaluation setup.

CPS DSPL [46]	**REMINDS DSPL** [18]
Feature Model	Decision Model
2000 features, 200 adaptation rules, 3000 assets	400 decisions, 1000 assets
SC1: A feature is removed from the feature model	SC1: A decision is removed from the model
SC2: An existing mapping (feature to asset) is updated	SC2: An existing mapping (asset to decision) is updated
SC3: A new asset is added	SC3: A new asset is added to the model
Runs per scenario: 10	
Total number of changes: 1000	
Seeded inconsistencies: 1 random defect introduced in every 100 changes	

to any problem space element or other element. This simulates an error a user might actually make during DSPL evolution. Indeed, while adding a solution space element can be automated, the mapping of the new solution space element to an existing or a new problem space element requires user involvement or at least an update rule as described earlier, and errors can thus easily be introduced.

To evaluate the performance, we measured for the two DSPL implementations the time required to check the consistency after we seeded one inconsistency. We based our evaluation on significant variability models, *e.g.*, feature models with 2000 features, considered by Berger *et al.* [51] as large feature models. For the concrete performance measurements, which are specific for each implementation and were performed on different machines, please refer to Section 6.

RQ3. Industrial applicability: can the reference architecture be used to support a real-world DSPL implementation?

To assess the applicability of the proposed architecture, we implemented it for a real-world DSPL, i.e., an automation software system for injection molding machines by an Upper Austrian company[e]. More specifically, as part of an ongoing project, MoldingCompany is extending the architecture of its automation software to allow plugging external devices into machines at runtime. The DSPL approach discussed in this paper was regarded as promising to deal with the identified adaptation and reconfiguration scenarios and to better support new devices and vendors in the future. The main benefit of the DSPL architecture is the development of a feature base on the machine, which dynamically configures available features of connected peripheral devices. The project thereby shifts the process of connecting new unknown peripheral devices to machines from design time (pre-deployment) to runtime (post-deployment). Depending on the capabilities of the connected devices features are enabled or disabled. Furthermore, the approach allows to activate or deactivate features based on results from monitoring an injection molding machine. Specifically, an engineer from MoldingCompany developed capabilities for runtime adaptation of the automation software including adaptation rules guided by the reference architecture presented in this paper and with advice from the authors. To support the (re-)configuration of the system at runtime the engineer developed a feature model and a component model to represent the system components that can be (de-)activated or reconfigured at runtime. In Section 7, we present the created architecture instance, the models, and the DSPL adaptation rules. We further describe how our architecture supports future evolution scenarios (e.g., adding new devices) in this domain and report initial feedback from MoldingCompany engineers.

6 | FLEXIBILITY AND PERFORMANCE (RQ1 AND RQ2)

We implemented and customized our reference architecture for the CPS and REMINDS DSPL to demonstrate that it is sufficiently flexible to support different implementations of the various components. Both implementations use different technologies and different kinds of variability models but comply with the architecture. To find out whether our approach is capable of properly detecting inconsistencies, we performed experiments that reflect the three previously described evolution scenarios and measured the time [f] required to check the consistency of each DSPL for which we implemented our reference architecture.

[e] Due to non-disclosure agreements we refer to this company as "MoldingCompany"

[f] Please note that the numbers regarding the average evaluation times for the two approaches cannot be compared with each other since the approaches are implemented using different technologies for constraint solving and error reporting. However, the numbers provide a hint on the general performance of both approaches.

6.1 | Reference Architecture Implementation for the CPS DSPL

Implementation (RQ1)

We first implemented the reference architecture for the CPS DSPL, which relies on feature models to manage runtime variability and adaptation rules as described below. Implementing and customizing the reference architecture for the CPS DSPL took about two person-weeks and was done by one developer. He could reuse initial variability models of this DSPL created in earlier work [46] and a consistency checker developed for a different project [52]. As depicted in Fig. 8(a) , each component of the CPS DSPL conforms to its respective meta-model. It thus provides a flexible means to define DSPLs for different domains or evolving a given DSPL, by switching model components whenever needed. The CPS DSPL is thus an instance of the DSPL meta-models, and is managed through Eclipse and EMF.

Adaptation. Adaptation rules are described in the feature model, such as the one shown in Fig. 8(a). In particular, we rely on *extended* feature models, *i.e.*, feature models whose additional information is defined in terms of feature *attributes* [32,49]. Specifically, adaptation rules are defined as attribute-based constraints. To distinguish between design-time model constraints and runtime adaptation rules, we rely on a slight extension of the feature meta-model proposed in [52]. This extension is twofold and consists of a Boolean attribute, *runtime*, added to the meta-classes *Constraint* and *Attribute*. The former enables the definition of runtime constraints, *i.e.*, adaptation rules, while the latter is used to define runtime attributes, *i.e.*, their value will only be taken into consideration at runtime. In practice, attributes and constraints flagged with *runtime* are not used during the initial configuration. They become active only once the system is up and running. Runtime attribute values remain unset at design time, and are then updated at runtime relying on an event listener that listens to changes that occur in the environment of the system.

Change Detection and Propagation. We developed an interface that handles change detection through a monitoring system implementing the operations proposed in Table 2. Once a change is detected, it is propagated to *State Manager* that translates this change into two models, *(i)* a model fragment that comprises the model element(s) to be updated and *(ii)* a change model, which is an instance of the change meta-model. The change meta-model enables the definition of all types of changes, *i.e.*, the addition, update or removal of elements in any space. Once translated, the two models are used by *Model Updater* to evolve the related models accordingly.

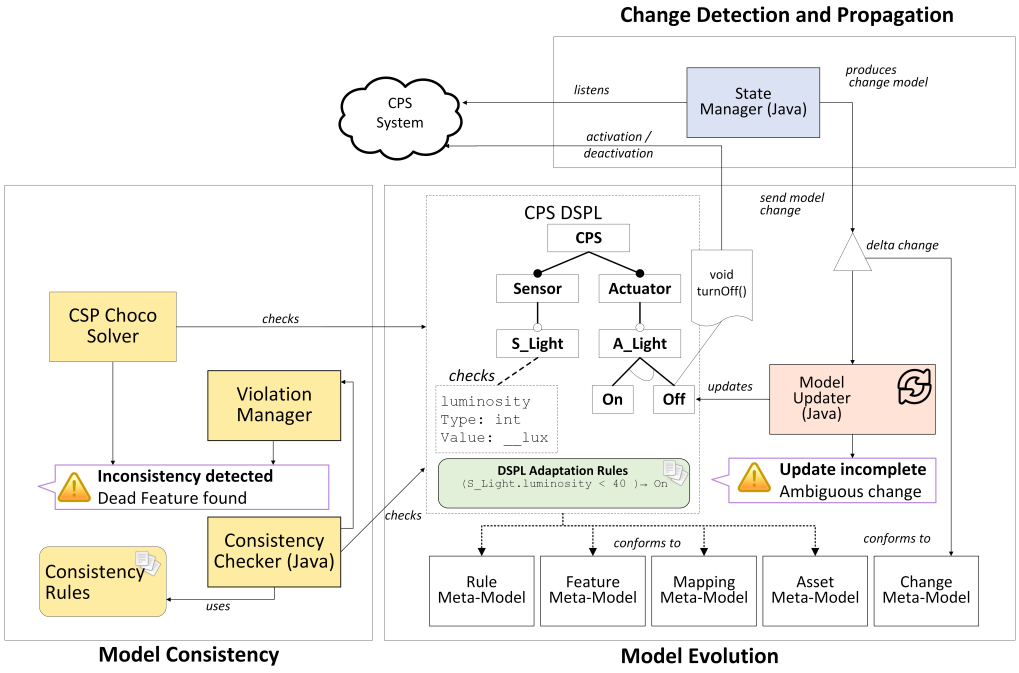
Model Evolution. The *Model Updater* retrieves the models related to the change, and evolves the main models by relying on update rules. It parses the change model together with the adaptation rules, the mapping space model or the feature model depending on the scenario, and performs the change. For instance, the change model can describe the removal of an element in the solution space (cf. Table 1: SS_Rem). Whenever the *Model Updater* retrieves such a model, it looks for an update rule named *removeSSElement* in the rules repository. This rule describes the guidelines of a proper removal, *i.e.*, it removes the solution space element itself and the related mapping space elements.

When the change model and the update rule make it feasible, the change is performed automatically, otherwise an output message is displayed within the Eclipse console to involve the developer. For instance, adding a new option (cf. Table 1: PS_Add) for an existing functionality can be handled automatically (*i.e.*, an update rule defines that a new variant feature f2 is added as a direct child feature of the existing one f1: *Add* f2 *childOf* f1), while moving features may not be straightforward (*e.g.*, the user has to define where to insert the moved feature and move sub-features accordingly). Once the evolution is performed (either manually by the user – *e.g.*, moving features; or automatically by the framework based on update rules – *e.g.*, new option), the consistency of the DSPL is checked.

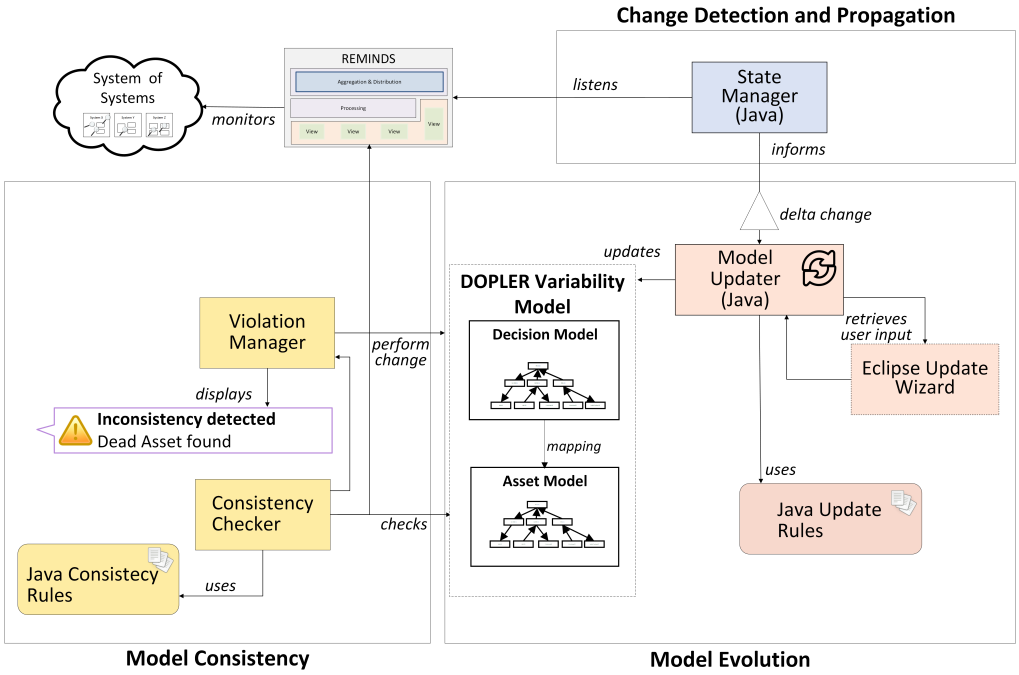
Model Consistency. Once evolved, the feature model is translated into a *Constraint Satisfaction Problem* (CSP). We then use the Choco CSP solver [53] (but any Java-based solver would also integrate smoothly, *e.g.*, SAT4J) to check the consistency of the feature model. The proposed model-based approach helps ensuring the consistency of the overall DSPL and acts as *Violation Manager*. First, changes are translated into model fragments that conform to the related meta-model and are thus consistent with the rest of the model. Second, models directly refer to other models/model fragments, ensuring all models are consistent with each other. Indeed, the main issue with most existing feature-based DSPL approaches is that adaptation rules are defined independently of the feature model. The evolution of a DSPL is thus often error-prone as for such approaches there is no way to check whether adaptation rules and the feature model are consistent after evolution. Instead, we define adaptation rules as models and make them refer to features in the feature model, similar, *e.g.*, to Gamez and Fuentes [54] who define adaptation rules in a reconfiguration plan and relate it to a feature model.

Performance evaluation (RQ2)

To assess the performance of our approach we programmatically generated models larger than the models manually created for the CPS. The generation process produces *(i)* random features (2000 features) and adaptation rules (200), *(ii)* 1-2 assets per feature (*i.e.*, a total of 3000 assets), and thus *(iii)* one or two mappings per feature. While these models are randomly generated, their size and structure can be compared with real feature models, *e.g.*, from the operating system domain as reported by Berger *et al.* [51]. For each evolution scenario SC1-SC3, we then performed 99 change operations leading to valid model updates, and 1 change operation leading to an inconsistency. In addition to checking whether all inconsistencies were indeed detected we measured the average evaluation time for evaluating consistency after each change. More precisely, we measured the



(a) Implementation for CPS



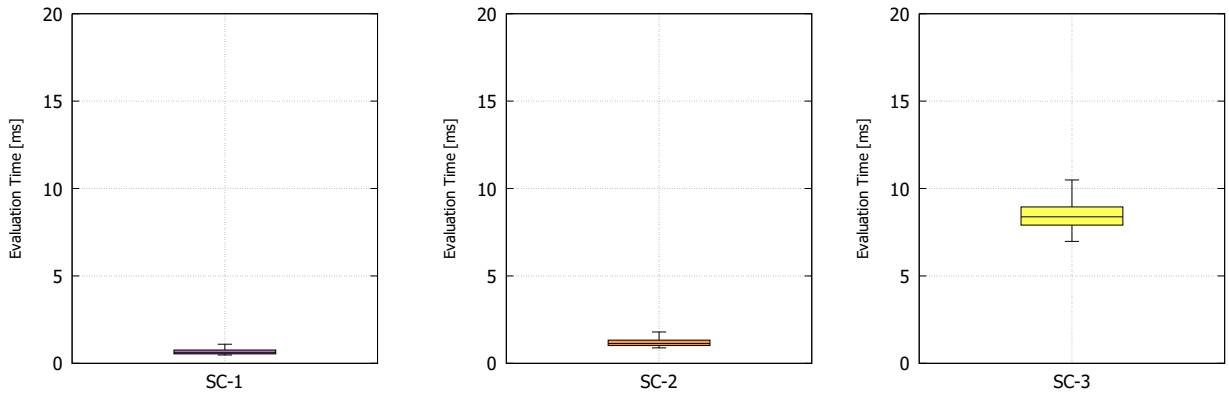
(b) Implementation for REMINDS

FIGURE 8 Implementations of our reference architecture for the CPS DSPL and the REMINDS DSPL.

time required to retrieve the content of all involved EMF models, and to check whether all references of a given model are present in other models or not. All experiments were performed on a MacBook Pro with a 2,6 GHz Intel® Core™ i5 processor and 8 GB of DDR3 RAM.

By relying on the model-based approach used in the CPS DSPL, all generated inconsistencies were detected and their cause was explained. Depending on the scenario, the time required to check the consistency of the CPS DSPL varies from 0.75ms up to 8.5ms on average (0.75ms for SC1, 1.3ms for SC2, and 8.5ms for SC3). Details on the evaluation times for the three scenarios are presented in Figure 9 (a–c). We observed that the time

required to check the consistency is negligible and is not a threat to the scalability of the model-based support in the CPS implementation of our reference architecture. The main advantage of our implementation is that the *Consistency Checker* can rely on methods from the EMF API to load and parse the models. While for SC1 and SC2 only 2 elements of the DSPL are involved (feature model and adaptation rules for SC1, feature model and mappings for SC2), SC3 requires the feature model, the mappings, and the assets to be loaded, parsed and checked, which explains the increased time for checking the consistency for SC3 (Fig. 9 (c)). Overall, our empirical evaluation indicates that our approach is well-suited for dealing with DSPLs with a substantial number of features, adaptation rules, and assets.



(a) CPS - SC1 (b) CPS - SC2 (c) CPS - SC3

FIGURE 9 Average consistency constraint evaluation times when detecting seeded inconsistencies (1 out of 100 change operations) for 1000 performed changes for 3 scenarios for the CPS implementation of our reference architecture.

6.2 | Reference Architecture Implementation for the REMINDS DSPL

Implementation (RQ1)

To support DSPL evolution for REMINDS, which uses decision models to manage variability, we have implemented our reference architecture as described below (cf. Fig. 8(b)). Implementing and customizing the reference architecture for the REMINDS DSPL also took about two person-weeks and was done by one of the main developers of REMINDS. He could reuse initial variability models of REMINDS created in earlier work [19] and a consistency checker developed for a different project [55].

Adaptation. In our earlier work, we developed an approach for variability management that provided the starting point for our implementation of the reference architecture to support the reconfiguration of REMINDS at runtime [19]. Specifically, we describe the variability of the key components of REMINDS (probes, event types, constraints) using decision-oriented DOPLER variability models [50].

In DOPLER, decisions define configuration options (to be set by end users or programmatically via the DOPLER API). The decision type defines what values can be set on decisions (Boolean, string, number, or enumeration). A decision can depend on other decisions hierarchically, if it needs to be made before other decisions, or logically, if the answer affects other decisions. Decisions are related to assets in DOPLER models (this is the problem to solution space mapping): answering decision questions allows selecting and (re-)configuring components. Assets in DOPLER models represent the core product line artifacts (*e.g.*, software components) in the solution space. Assets can depend on each other functionally (*e.g.*, one asset requires another asset) or structurally (*e.g.*, if an asset is part of a another asset), *i.e.*, assets can have solution space dependencies. Using DOPLER, users can create domain-specific meta-models to define the asset types, attributes, and dependencies for their domain or system. In the REMINDS case study, for instance, the asset types are probe, event type, and constraint. They have different attributes and are related to each other, specifically, probes provide events of different types and constraints check events.

Users or programs can set decision values for decisions defined in the DOPLER variability model, thereby (de-)activating probes, constraints, and related event types at runtime through an interface that connects DOPLER and REMINDS. Decisions thus represent the possible adaptations that can be made at runtime, *i.e.*, the adaptation rules. If, for instance, a REMINDS probe is used to instrument an archiving component (persistence management) of a system, the decision could be called *monitoring_archiving*, with the question 'Do you want to monitor the archiving process?', and related to the archiving probe represented by an asset in the DOPLER model with name *archiving_probe*. More examples can be found in [19].

Change Detection and Propagation. The REMINDS framework provides interfaces for retrieving state information of the elements of the running monitoring infrastructure (*i.e.*, probes, constraints, event types) to determine whether the elements are active or inactive, and also for retrieving information on elements being added, removed or modified. We developed the component *State Manager* of our reference architecture to implement these interfaces. It keeps track of the current state of the elements in the running REMINDS infrastructure, periodically checks which elements have been added, removed or modified, and forwards this information to component *Model Updater*.

Model Evolution. We implemented the *Model Updater* as an Eclipse Plug-in in the DOPLER variability modeling IDE [50]. It is triggered by *State Manager* whenever a change in REMINDS occurs. Based on *Update Rules*, *Model Updater* checks whether update actions can be performed automatically or user input is required. More specifically, our implementation of the *Model Updater* distinguishes unambiguous and ambiguous changes. In case of unambiguous changes, an update is triggered directly by the *Model Updater* to automate the changes to the variability model. For ambiguous changes, user feedback is required. In such cases, the *Update UI* (a simple Eclipse Wizard) is triggered, allowing the user to select a particular resolution strategy. For example, when a probe is removed from REMINDS, an update rule could specify to simply remove the respective asset from the variability model. However, as the asset might be mapped to one or more other model elements, *e.g.*, decisions, it will typically make sense to involve the user via the *Update UI* and ask her to decide whether to really remove the asset together with all mappings to it.

Model Consistency. To check model consistency, we rely on an existing *Consistency Checker* [55] that allows to check the consistency of a decision model and arbitrary artifacts based on consistency rules (not to be confused with the constraints used by REMINDS to check system behavior at runtime). For the purpose of the REMINDS case study we extended this existing *Consistency Checker* to check the conformance between the DOPLER variability model and REMINDS. We implemented the consistency rules for REMINDS as an extension of the existing *Consistency Checker* in Java. Facades provide access to the DOPLER variability model on the one hand – to retrieve model elements such as assets and decisions – and to REMINDS on the other hand - to retrieve registered and running probes and constraints. Internally, the *Consistency Checker* employs an incremental approach [56] thus reducing the overhead and providing instant feedback to users on emerging violations.

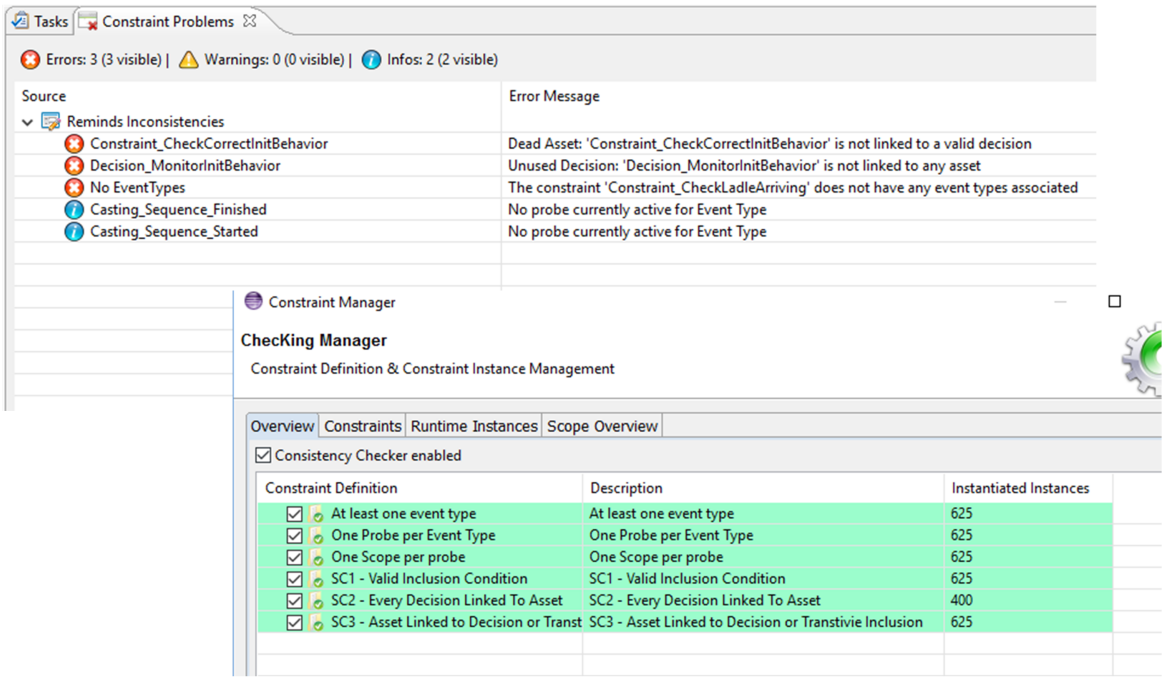


FIGURE 10 DOPLER IDE showing constraint violations (upper part) and currently activated constraints (lower part).

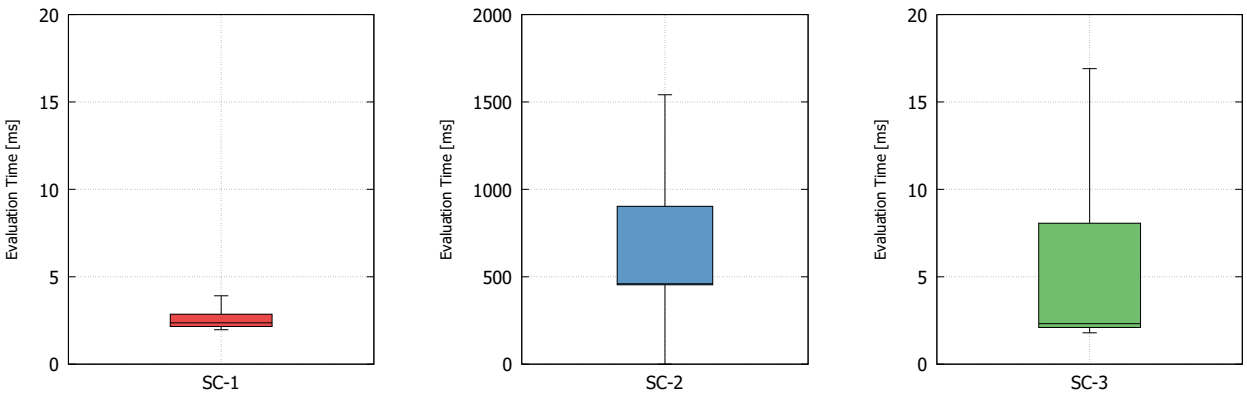
The *Violation Manager* is integrated within the DOPLER IDE and retrieves information on occurring inconsistencies from the *Consistency Checker*. It reports details to the user on the violation of each consistency rule, *e.g.*, the origin within the model and the cause of the violation. In case of a possible resolution a (semi-)automated fix can be applied to the model. In the mentioned example of adding a new asset to the variability model due to a new probe added to REMINDS, the *Consistency Checker (i)* immediately detects this asset as dead if it has not also been mapped to a decision as described above, and *(ii)* informs the user about this inconsistency via the *Violation Manager*. Fig. 10 depicts a screenshot of the DOPLER modeling IDE showing constraint violations (upper part) and currently active consistency rule (lower part). Different types of violation are

highlighted differently depending on their severity (*e.g.*, Errors vs. Warnings). The engineer can review each violation and navigate to its origin for an in-depth inspection, *e.g.*, of the model element not defined properly. Up to this point we have not implemented support for automatically fixing detected model inconsistencies. This is part of our future work.

Performance evaluation (RQ2)

For the simulation, we extended the existing DOPLER variability model[19] with additional elements through duplication of existing elements, *i.e.*, we generated additional decisions and assets to produce a model with overall 1000 assets and 400 decisions, which however has the same structure as the manually created original model. This by far exceeds the typical size of decision models which can be expected in a typical industrial scenario[50]. We assessed whether the inconsistencies where detected, instrumented the *Consistency Checker* to measure evaluation times and eventually calculated the average evaluation time per scenario. In this case, the evaluation time is the time required to evaluate a single constraint instance, more precisely, the time the method `evaluate()` needs to complete, return 'consistent' or 'inconsistent', and generate violations which are then forwarded to the user interface. We performed the evaluation runs using the latest version of the DOPLER IDE and Eclipse 3.8 on a standard Desktop machine with an Intel® Core™ i5 CPU @2.60GHz 16GB RAM running Windows 10 64-Bit.

For all three scenarios all seeded inconsistencies were detected. Depending on the type of check performed, the evaluation times vary between 3ms and 600ms on average (3.0ms for SC-1, 609ms for SC-2, and 6.74ms for SC-3). Details on the evaluation times for the three scenarios are presented in Figure 11 (a–c). For SC-1 and SC-3 the average evaluation times per constraint instance are below 7ms while for SC-2 the average evaluation time rises above 600ms. This increased evaluation time is due to the fact that the constraint for this scenario requires evaluating all mappings between problem and solution space, *i.e.*, all assets linked to decisions. DOPLER was not optimized for querying such mappings in the first place resulting in the need to iterate over all assets contained in the variability model to evaluate the constraint. This is the reason for the higher evaluation time for this constraint. Optimizing the access to decisions and assets in DOPLER would greatly reduce the resulting evaluation time and could easily resolve this issue. However, in all the three scenarios, the time needed to report the inconsistency to the user is still acceptable and allows for (almost) instant feedback.



(a) REMINDS - SC1 (b) REMINDS - SC2 (c) REMINDS - SC3

FIGURE 11 Average consistency constraint evaluation times when detecting seeded inconsistencies (1 out of 100 change operations) for 1000 performed changes for 3 scenarios for the REMINDS implementation of our reference architecture.

7 | INDUSTRIAL APPLICABILITY (RQ3)

To assess the applicability of our proposed architecture, we instantiated it for a real-world DSPL, *i.e.*, MoldingCompany's automation software system for injection molding machines. Plastic products range from small everyday life-products such as toothbrushes or toys, up to big products such as water pipes or garbage containers. Injection molding is a manufacturing method where material (*e.g.*, thermoplastic polymer) is heated until it is molten and injected into a mold cavity, where the part is cooled and hardened. Injection molding machines are widely used to produce plastic

products for many different markets. Example areas include automotive (car body, interior, glazing), packaging (containers, buckets, pallets), medical (healthcare, diagnostics) or teletronics (mobile communication, displays).

In the injection molding industry, various peripheral devices with different capabilities are mounted on a single machine to satisfy different requirements. For instance, in the temperature control process, specific parts of the machine have to be heated to a defined temperature, while other machine parts have to be cooled down. This can be achieved with peripheral temperature control devices provided by external manufacturers. For many years, the approach to connect peripheral devices has been to use a serial interface and a standardized, manufacturer-dependent protocol. Recently, the Euromap council proposed the standardized interface based on the OPC Unified Architecture (OPC UA) to better cope with this situation. OPC UA is a machine-to-machine communication protocol for industrial automation developed by the OPC Foundation[g]. The standard Euromap 82 and Euromap 82.1 define an OPC UA Information model, *e.g.*, for temperature control devices and peripheral devices in general. MoldingCompany uses OPC UA to connect and switch among multiple peripheral devices via an Ethernet connection. This allows for dynamic variability (*i.e.*, reconfiguration at runtime) and provides the technical foundation for applying our DSPL architecture in an industrial context. As part of our evaluation, an engineer of MoldingCompany created a feature model describing the Temperature Control Device (TCD) capabilities for MoldingCompany's injection molding machines. According to this model (and its mapping to OPC UA nodes) we can dynamically identify which OPC UA nodes are available on a connected device (nodes are defined as mandatory/optional). When specific nodes are present, features can be enabled. The values of OPC UA nodes are the basis for adaptation rule checks performed during runtime.

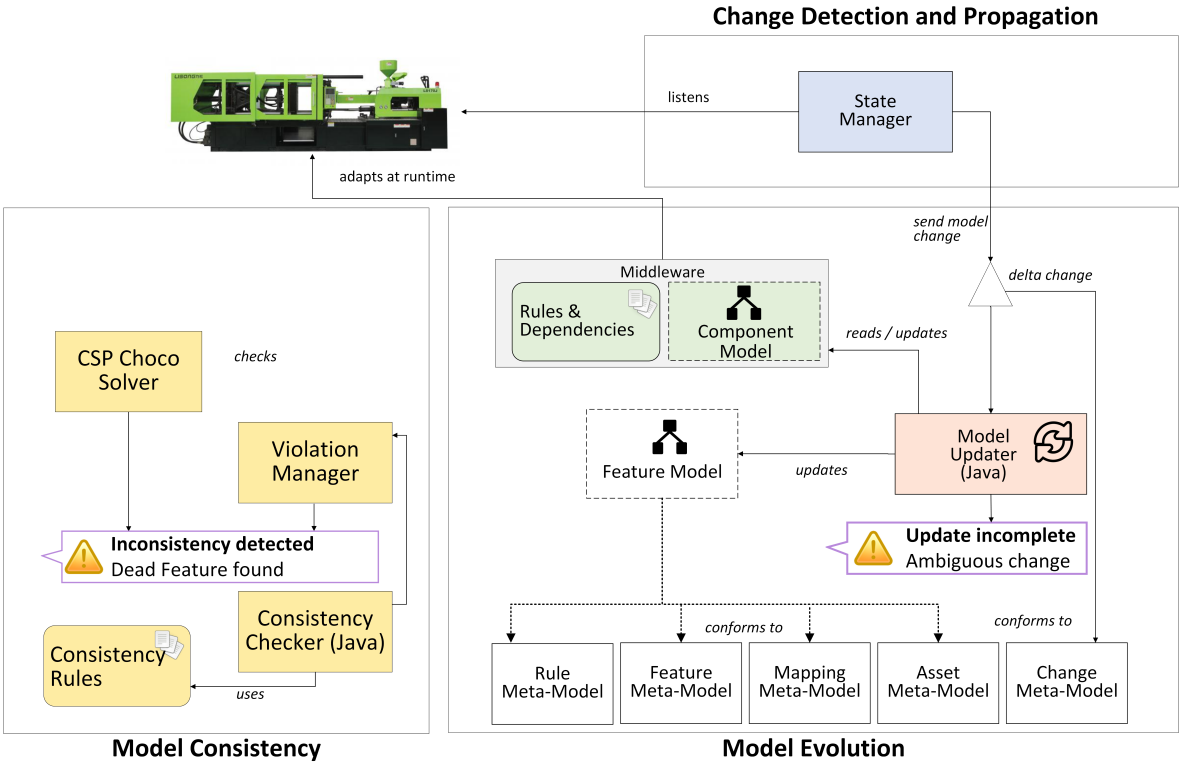


FIGURE 12 Implementation of our reference architecture for the injection molding automation system of MoldingCompany.

Figure 12 depicts our DSPL reference architecture instantiated for the MoldingCompany case study. The engineer used the same approach as we used for the CPS DSPL described above in Section 6.1. We describe the feature model and the adaptation rules the industrial engineer developed using our reference architecture. We cannot provide details of the component model and the mapping of features to components due to non-disclosure agreements, but we present general numbers and specific examples below. The middleware implemented by the engineer uses the adaptation rules and the component model to perform (re-)configuration of the system at runtime. The other parts of the reference architecture are the same as for the CPS instance described above.

[g]https://opcfoundation.org/

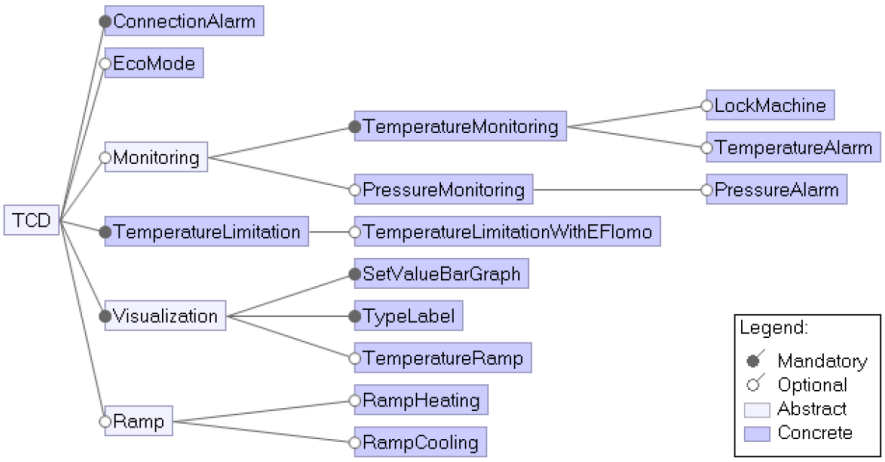


FIGURE 13 Feature Model for injection molding automation system temperature control devices DSPL.

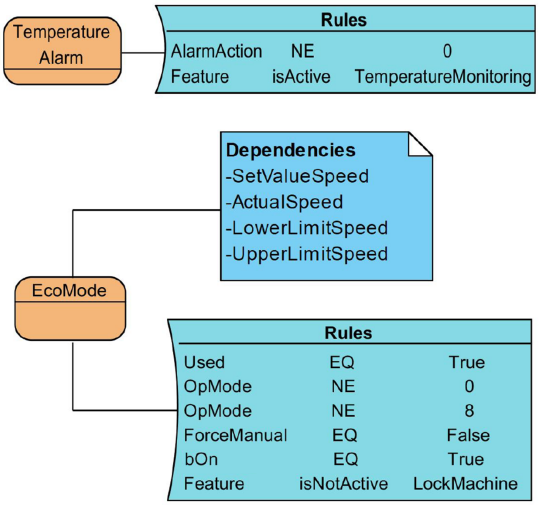


FIGURE 14 Two example adaptation rules based on features of the temperature control devices feature model.

Figure 13 depicts the feature model for the temperature control device and Figure 14 shows two example adaptation rules. The first rule is for the temperature alarm feature. Given that temperature monitoring is available, it can trigger alarms of different severity (*i.e.*, a positive integer value greater than zero). The second rule is related to the EcoMode feature, which allows to dynamically determine the best valve position for the water distribution system and the TCD pump speed set value to ensure optimal power usage. When the EcoMode feature is activated, the pump speed is determined automatically and the input field on the visualization is deactivated, indicating that it can currently not be changed manually. The adaptation rule defines that the connected TCD must be set to Used (activated by the operator) and the OperatingMode must not be 0 (switched off/ready) or 8 (connection problem/undefined). The ForceManual Boolean, which would deactivate the automatic calculation if set to true, must be false. Finally, the water distribution system component must be switched on, which is determined by the Boolean value bOn. Overall, the engineer created 25 adaptation rules (for 15 features), which are in turn mapped to 78 solution space components.

While we did not use the entire reference architecture in this example due to the focus on the modeling and adaptation capabilities, we were able to reuse the consistency checking and model update parts from the CPS example. We received initial positive feedback, regarding the approach, from MoldingCompany. Specifically, they confirmed that without the DSPL approach it took developers several days until a prototype was running for every new device. The DSPL architecture, on the other hand, reduced the time necessary to implement a similar device of an established product line (*e.g.*, a new TCD of a different manufacturer) to approximately one working day. Our experience with our industrial partner thus shows the

usefulness and applicability of our approach in a real-world scenario from the injection molding machines domain, in addition to the two DSPLs previously described.

8 | DISCUSSION AND THREATS TO VALIDITY

Our evaluation demonstrates that our reference architecture was a useful basis to implement support for DSPL evolution in at least two different cases (RQ1) in two different domains. Both implementations exhibit significant differences: one uses a model-driven, feature-based approach (CPS) and the other one a tool-driven, decision-oriented approach (REMINDS). However, both implementations rely on Eclipse (RCP or EMF) and Java, and different technologies – specifically, technologies not supporting component-based or object-oriented development – could make the implementation of our reference architecture significantly more difficult. In the two presented cases, based on our reference architecture and the proposed generic operations, each implementation has been done by one person in a rather short time, *i.e.*, around two weeks.

We also provide initial evidence on the practicality of our reference architecture by implementing a real-world DSPL from the domain of automation software for injection molding machines, which was not done by the developers of the reference architecture, but by an industrial engineer. He could reuse several components from the CPS DSPL, which further demonstrates its feasibility. Initial feedback is very promising but more work is needed, particularly to better demonstrate the evolution support of our approach in practical scenarios.

Overall, the reference architecture was very useful to guide the implementation of evolution support for three different DSPLs in different domains. This was only possible because we kept the description of the reference architecture components, their interactions, and the operations rather abstract. However, this also has the drawback, that the implementation effort for each DSPL is significant, even if one can reuse existing models and tools as described above. It would thus make sense to further automate this process of implementing the reference architecture, e.g., by providing several template implementations for different types of variability models. Also, the actual reference architecture implementation process needs to be better formalized. Even though the reference architecture components and the generic operations support the implementation, the concrete activities and their inputs and outputs should be made more explicit.

Internal Validity. Since both approaches build on prior work and tools, which had partly been implemented by the authors, we cannot claim that all parts have been implemented from scratch. However, both the original solutions have been adapted and extended significantly to fit the needs of the CPS and REMINDS DSPLs. In both cases, it took less than two weeks to implement the components *State Manager* and *Model Updater*, and the facades to interact with the running systems. Two of the authors implemented extensions to their own architecture basing their work on existing tool environments that have been published before [52,50] and using the reference architecture to guide their extensions of these existing environments. While other developers could follow a different implementation approach, we still think we could sufficiently demonstrate the flexibility and practicality of our approach. Our reference architecture worked for two quite different modeling approaches. While we cannot guarantee it would work for any given approach or technology, we believe that following the architecture can guide developers in creating DSPL evolution support as we could initially demonstrate with our industrial applicability study.

Considering the evaluation runs, we measured how well our consistency checker implementations work, which indicates their scalability. However, we randomly generated large variability models – though based on real, smaller models such as the one described in [19] – which might still not correctly reflect how such models would really look like in practice. Also, we randomly seeded selected inconsistencies in the three evolution scenarios representing one edit per space. In practice, many more scenarios are possible and might occur in all the three spaces (even in combination), making the exhaustiveness of the experiments difficult to reach. We thus cannot claim our approach scales in any possible case, but we can still argue that we found initial evidence for its scalability and flexibility.

External validity refers to how well data, processes, theories can be applied to other domains and application scenarios and how generalizeable the results and findings are. With the presented reference architecture we aim to provide a generic architecture that can be applied to different domains and technologies. Applying our reference architecture to other approaches will require detailed knowledge about the specific system and the respective variability management approach. In terms of industrial applications, are confident that the three instances of the reference architecture (one of which was a large industrial system) provide evidence that the reference architecture can be easily adopted for different types of systems and different types DSPLs, meaning different types of variability modeling and management approaches. Regarding our evaluation, we have performed lab experiments for two different implementations of the reference architecture both representing large-scale systems. While we can not claim full generalizability in terms of performance and scaleability of the approach, we however, are confident think that this provides a solid basis for the applicability of the approach. Further work will be needed assessing the applicability and performance characteristics for different types of systems.

9 | RELATED WORK

We discuss related research on DSPLs, on the evolution of SPLs and variability models, on checking the consistency of (product line) models, and particularly on evolution in DSPLs. Table 4 summarizes the different concerns addressed by the approaches discussed in this section and shows whether they are addressed in terms of problem or solution space (*PS* and *SS*), *Traceability* during SPL evolution or comparison of the set of *Products* or *Configurations* before and after evolution. A check mark indicates whether the approach proposes solutions or deals with the different criteria.

TABLE 4 Overview of the discussed approaches.

Approach	SPL Evolution			Model-Checking			DSPL evolution	
	PS	SS	Traceability	PS	SS	Products	PS	Configurations
Arcega *et al.* [57]							✓	✓
Capilla *et al.* [58]							✓	✓
Czarnecki and Pietroszek [59]				✓	✓			
Deng *et al.* [60]	✓	✓						
Elsner *et al.* [61]				✓	✓			
Font *et al.* [39]	✓							
Gamez and Fuentes [54,62]	✓	✓	✓					
Guo *et al.* [9]				✓				
Helleboogh *et al.* [63]							✓	
Mauro *et al.* [64]			✓	✓			✓	✓
Mende *et al.* [65]		✓						
Murta *et al.* [66]	✓	✓	✓					
Neves *et al.* [12,8]	✓	✓	✓					
Passos *et al.* [7]	✓	✓						
Quinton *et al.* [10]				✓				
Seidl *et al.* [3,67]	✓	✓	✓					
Thüm *et al.* [11]						✓		

Dynamic software product lines. As modern systems demand more and more post-deployment activities and runtime capabilities, several DSPL approaches have emerged in the last decade to deal with these requirements [26]. For instance, some authors have proposed to leverage techniques from service-oriented architectures to build service-oriented DSPLs, *i.e.*, DSPLs built by composing services [68,69] or, similarly, aspect-oriented DSPLs [70]. Recently, Bashari *et al.* [71] proposed a classification of various DSPL implementations and compared their adaptation mechanisms. Bencomo *et al.* [16] discuss these different approaches for building a DSPL and show that these supposed DSPLs were not as 'dynamic' as expected. They also point out the need to cope with uncertainty at runtime by providing support for the DSPL evolution. This motivates our own work to provide such evolution support. In a similar study, Bencomo *et al.* [72] discuss different techniques for modeling variability at runtime together with existing approaches for adapting system configurations at runtime (*e.g.*, using adaptation rules or goals). We also make use of adaptation rules, *e.g.*, described as extended feature models in our CPS DSPL case study and described as DOPLER decisions in our ReMinds DSPL case study.

Evolution of software product lines and variability models. Dynamic software product line approaches often build on traditional software product line solutions. Naturally, it makes sense to discuss the support for evolution in traditional SPLs as a basis to discuss support for the evolution of DSPLs (see below). Many product line approaches assume that activities in domain and application engineering can take a fairly stable product line for granted. However, real-world product lines inevitably and continuously evolve. Managing evolution is thus success-critical, particularly in model-based approaches to ensure consistency after changes to meta-models, models, and actual artifacts. Several authors [73,74] have stressed the importance of approaches for product line evolution to avoid the erosion of a product line, *i.e.*, the deviation from the product line model up to the point where key properties no longer hold.

Several approaches have been proposed for managing the evolution of software product lines [5], ranging from verification techniques to ensure consistent evolution, to model-based frameworks dedicated to the evolution of feature-based variability models [6]. For example, an interesting research thread proposes evolution templates for co-evolving a variability model and related software artifacts [3,7,8]. Such templates can be compared with the update rules we have developed in our approach. Deng *et al.* [60] describe a model-driven product line approach that focuses on the issue of domain evolution and product line architectures. They discuss several challenges for the evolution of model-driven software product line architectures and present their solution for supporting evolution with automated domain model transformations. Such transformations could also be useful in our context to realize the update rules to support the evolution of the variability models in our DSPLs when applying model-driven techniques. Another example is the work by Mende *et al.* [65], who present tool support for the evolution of software product lines based on the grow-and-prune model. They support identifying and refactoring code that has been created by copy and paste and which might be moved from product level to product line level. Refactoring of a DSPL is not the scope of our work, and should rather be avoided at runtime, if at all possible, as it can too easily lead to many inconsistencies. However, if necessary, the work and tool by Mende *et al.* could be useful to support refactoring the DSPL code. Font *et al.* [39] propose an approach that compares, aggregates and resolves input models (models describing products) to create a CVL variability model. This approach can also be used to evolve the existing variability model by taking into consideration a new input model. Our CPS DSPL case study also describes changes as models, however, not as CVL models but as two feature models: a model fragment that comprises the model element to be updated and a change model, which is an instance of the change meta-model.

Another important research thread deals with the evolution of the variability defined for a software product line. Some authors surveyed different approaches and terminologies [75], while others propose a taxonomy of evolution operations in different modeling spaces [76]. We built on this work when describing the evolution operations in DSPLs. Neves *et al.* [12] propose a SPL evolution approach that preserves the original behavior of evolving product lines, *i.e.*, products that could be generated before evolution can still be generated after the evolution. This of course is only possible if restricting the removal of certain needed features, which makes the process easier but also constitutes a limitation of this approach. To keep a configuration consistent with a feature model even after evolution of the latter, Gámez *et al.* [54] present an approach that automatically evolves the configuration with respect to the changes performed in the model while also taking into consideration the possible cardinalities. Such an approach is also useful in a DSPL context, with the main difference being that automatically evolving the configuration would have to be done (also) at runtime. Seidl *et al.* [67] introduce Hyper feature models, capable of versioning the features and their constraints to maintain evolution traceability over time and guarantee the compatibility of one version of a feature with versions of another one. Feature traceability is thus a central concern in SPL evolution approaches, and has been shown to be essential in a feature-oriented project [77]. Our work was largely inspired by this earlier work on evolving software product lines, and we extended this work by considering runtime management of such evolution.

Consistency checking. Approaches for model-checking are often used to guarantee the consistency of a variability model after evolution [9,10]. For example, Czarnecki *et al.* [59] present a feature-based approach using model templates that supports defining consistency constraints using OCL. Elsner *et al.* [61] present an incremental approach for checking consistency during derivation in multi product line environments. We also support defining consistency constraints in different ways, depending on the implementation of our reference architecture. Approaches such as the ones by Czarnecki *et al.* and Elsner *et al.* could thus also be applied in our context. Due to the fact that we have to deal with consistency checking at runtime, an incremental approach is preferable (cf. our REMINDS case study).

Existing work also addresses the issue of consistency between models and code in product lines. For instance, Murta *et al.* [66] present an approach for ensuring consistency of architectural models and the corresponding implementation during evolution. The approach supports arbitrary evolution policies and is based on recording changes in a configuration management system. In our context, we also have to analyze the actual DSPL's code to detect inconsistencies (cf. our reference architecture) and an approach such as the one by Murta *et al.* could be adapted to our DSPL context.

Approaches for comparing the set of possible products before and after the evolution of a product line have also been proposed [11,12]. In addition to these 'classic' consistency-checking approaches, we also provide in our architecture a means to check the validity of adaptation rules together with the different models and thus ensure proper runtime reconfigurations.

Evolution of dynamic software product lines. What Knauber [78] pointed out over 12 years ago – that it is easier to plan and evolve a product line for anticipated variability proactively, but it is rather complicated and hard to support unanticipated reactive product line evolution – still holds today,

particularly for dynamic software product lines. Indeed, current approaches supporting product line evolution [5] have not specifically been designed to support the evolution of a DSPL and to maintain its consistency at both design and runtime. Existing research only recently started to investigate evolution in a DSPL context, and especially its impact on the running system.

Helleboogh *et al.* [63] proposed the notion of meta-variability to support evolution in DSPLs. A meta-variability model is used to describe the way the variability can evolve at runtime, while a meta-variability meta-model defines the relationships between the variability and the meta-variability models. The implementation of a reference architecture for the CPS DSPL also relies on change meta-models. Capilla *et al.* [26,58] use super-types to automate the modification of variants in a feature model at runtime. A new variant can be added to an existing variation point as long as it belongs to the same super type as the variant to be replaced. Such rules are very useful to support DSPL evolution and could be realized in our context via our update rules concept. Arcega *et al.* [57] propose evolution strategies to migrate from the current version of a configuration space to an evolved one in a DSPL. Their approach thus enables the current running configuration to match the new configuration space.

However, these approaches are often limited to a given set of changes and focus on evolving the configuration space and related models without considering other concerns (*e.g.*, system-driven evolution, solution space elements, constraint-checking at runtime), and also do not deal with adaptation rules. Mauro *et al.* [64] propose a metamodel to keep track of the feature model versions when evolving the DSPL. They rely on the DarwinSPL tool suite to handle evolutions made by the DSPL maintainer. That is, they propose their own *Model Updater* component and the related GUI to deal with non-trivial evolution scenario. Their approach is thus compliant with our proposed reference architecture.

10 | CONCLUSIONS AND FUTURE WORK

In many domains, systems need to run continuously and may not be shut down for reconfiguration or maintenance tasks even if they require constant adaptations to react to changes in their environment. To manage the variability and support the runtime reconfiguration of such adaptive systems, DSPL approaches have been proposed. Usually built as a long-term investment, a DSPL has to evolve continuously to meet new requirements. While the evolution of SPLs has been widely studied, little support exists for the challenging process of evolving a DSPL while maintaining its consistency and preserving the runtime adaptation capabilities of the managed system.

In this paper, we proposed a reference architecture for dealing with the evolution of DSPLs. We first described the possible evolution scenarios, explained what inconsistencies could result from such scenarios, and then presented the main components of our reference architecture, along with generic operations that can be implemented for a concrete DSPL. To assess the flexibility of our approach, we investigated the use of the reference architecture in two concrete DSPLs (a cyber-physical system and a runtime monitoring infrastructure) and we conducted experiments to provide evidence on the performance of our approach for these two implementations regarding detection of inconsistencies. Through an industrial collaboration, we also demonstrated the usefulness of our approach in real-world scenarios. Further studies should be conducted to demonstrate the approach is suitable for diverse DSPLs.

Although the reference architecture presented in this paper is well-suited to detect inconsistencies resulting from the evolution of both the DSPL models or the running system, we only considered in our evaluation changes that originate from monitoring the running system. A possible improvement for future work would thus be to also deal with unsynchronized configurations, *i.e.*, running configurations derived from a previous version of the DSPL architecture. We also plan to improve our approach by leveraging learning-based mechanisms for those ambiguous changes that require human intervention. Indeed, such manual changes could be 'learned' and be used later to automatically support similar evolutions. Our approach could also be extended to enable previous configurations to be (re-)derived even after evolution, *e.g.*, by exploiting traceability mechanisms from version control management tools. Also, we plan to involve users responsible for maintaining DSPLs and get their feedback on the usefulness of our approach.

ACKNOWLEDGMENTS

The work by the Italian authors (when the work described in this paper started, the first author was still working at Politecnico di Milano) has been supported by project EEB – Edifici A Zero Consumo Energetico In Distretti Urbani Intelligenti (Italian Technology Cluster For Smart Communities) – CTN01_00034_594053. Regarding the work by the Austrian authors the financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Primetals Technologies is gratefully acknowledged. Michael Vierhauser's work has also been supported by the Austrian Science Fund (FWF) under Grant No. J3998- N31.

References

1. Hinchey M, Park S, Schmid K. Building Dynamic Software Product Lines. *IEEE Computer* 2012; 45(10): 22–26.
2. Berg K, Bishop J, Muthig D. Tracing Software Product Line Variability: From Problem to Solution Space. In: SAICSIT '05. South African Institute for Computer Scientists and Information Technologists; 2005; Republic of South Africa: 182–191.
3. Seidl C, Heidenreich F, Aßmann U. Co-evolution of Models and Feature Mapping in Software Product Lines. In: ACM; 2012; Salvador, Brazil: 76–85.
4. Capilla R, Bosch J, Kang K. *Systems and Software Variability Management: Concepts, Tools and Experiences.* Springer . 2013.
5. Marques M, Simmonds J, Rossel PO, Bastarrica MC. Software product line evolution: A systematic literature review. *Information and Software Technology* 2019; 105: 190–208.
6. Pleuss A, Botterweck G, Dhungana D, Polzer A, Kowalewski S. Model-driven Support for Product Line Evolution on Feature Level. *Journal of Systems and Software* 2012; 85(10): 2261–2274.
7. Passos L, Guo J, Teixeira L, Czarnecki K, Wąsowski A, Borba P. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In: ACM; 2013; Tokyo, Japan: 91–100.
8. Neves L, Borba P, Alves V, et al. Safe Evolution Templates for Software Product Lines. *Journal of Systems and Software* 2015; 106(C): 42–58.
9. Guo J, Wang Y, Trinidad P, Benavides D. Consistency Maintenance for Evolving Feature Models. *Expert System Applications* 2012; 39(5): 4987–4998.
10. Quinton C, Pleuss A, Le Berre D, Duchien L, Botterweck G. Consistency Checking for the Evolution of Cardinality-based Feature Models. In: ACM; 2014; Florence, Italy: 122–131.
11. Thüm T, Batory D, Kastner C. Reasoning About Edits to Feature Models. In: IEEE Computer Society; 2009; Vancouver, British Columbia, Canada: 254–264.
12. Neves L, Teixeira L, Sena D, Alves V, Kulezsa U, Borba P. Investigating the Safe Evolution of Software Product Lines. In: ACM; 2011; Portland, Oregon, USA: 33–42.
13. Cheng BH, Eder KI, Gogolla M, et al. Using models at runtime to address assurance for self-adaptive systems. In: Springer. 2014 (pp. 101–136).
14. Provoost M, Weyns D. DingNet: a self-adaptive internet-of-things exemplar. In: IEEE. ; 2019: 195–201.
15. Corporation I. An Architectural Blueprint for Autonomic Computing. tech. rep., IBM; 2005.
16. Bencomo N, Lee J, Hallsteinsen SO. How Dynamic is your Dynamic Software Product Line?. In: ; 2010; Jeju Island, Korea: 61–68.
17. Sharifloo AM, Metzger A, Quinton C, Baresi L, Pohl K. Learning and Evolution in Dynamic Software Product Lines. In: ACM; 2016; Austin, Texas: 158–164.
18. Vierhauser M, Rabiser R, Grünbacher P, Seyerlehner K, Wallner S, Zeisel H. ReMinds: A Flexible Runtime Monitoring Framework for Systems of Systems. *Journal of Systems and Software* 2016; 112: 123–136.
19. Rabiser R, Vierhauser M, Grünbacher P. Variability Management for a Runtime Monitoring Infrastructure. In: ACM; 2015; Hildesheim, Germany: 35–42.
20. Baresi L, Quinton C. Dynamically Evolving the Structural Variability of Dynamic Software Product Lines. In: Inverardi P, Schmerl BR., eds. *Proceedings of the 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*IEEE; 2015; Florence, Italy: 57–63.
21. Quinton C, Rabiser R, Vierhauser M, Grünbacher P, Baresi L. Evolution in Dynamic Software Product Lines: Challenges and Perspectives. In: ACM; 2015; Nashville, USA: 126–130.
22. Clements P, Northrop L. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional . 2001.

23. Pohl K, Böckle G, Linden v. dFJ. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag . 2005.

24. Zschaler S, Sánchez P, Santos JP, et al. VML* - A Family of Languages for Variability Management in Software Product Lines. In: Springer; 2009; Denver, CO, USA: 82–102.

25. Apel S, Kästner C. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 2009; 8(5): 49–84.

26. Capilla R, Bosch J, Trinidad P, Ruiz-Cortés A, Hinchey M. An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry. *Journal of Systems and Software* 2014; 91: 3–23.

27. Hinchey M, Park S, Schmid K. Building Dynamic Software Product Lines. *Computer* 2012; 45(10): 22–26.

28. Hallsteinsen S, Hinchey M, Park S, Schmid K. Dynamic Software Product Lines. *Computer* 2008; 41(4): 93–95.

29. Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wąsowski A. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In: ACM; 2012; Leipzig, Germany: 173–182.

30. Tartler R, Lohmann D, Sincero J, Schröder-Preikschat W. Feature consistency in compile-time-configurable system software: facing the linux 10, 000 feature problem. In: ; 2011; Salzburg, Austria: 47–60.

31. Passos L, Teixeira L, Dintzner N, et al. Coevolution of variability models and related software artifacts. *Empirical Software Engineering* 2015: 1–50.

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

33. Botterweck G, Pleuss A. Evolution of Software Product Lines. In: Mens T, Serebrenik A, Cleve A., eds. *Evolving Software Systems*Springer Berlin Heidelberg. 2014 (pp. 265–295).

34. McGregor J. The Evolution of Product Line Assets. Tech. Rep. CMU/SEI-2003-TR-005, Software Engineering Institute, Carnegie Mellon University; Pittsburgh, PA: 2003.

35. Lemos dR, others . Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: Lemos dR, Giese H, Müller H, Shaw M., eds. *Software Engineering for Self-Adaptive Systems II*Springer Berlin Heidelberg. 2013 (pp. 1–32).

36. Borba P, Teixeira L, Gheyi R. A Theory of Software Product Line Refinement. *Theor. Comput. Sci.* 2012; 455: 2–30.

37. Acher M, Cleve A, Collet P, Merle P, Duchien L, Lahire P. Reverse Engineering Architectural Feature Models. In: Springer-Verlag; 2011; Essen, Germany: 220–235.

38. She S, Lotufo R, Berger T, Wąsowski A, Czarnecki K. Reverse Engineering Feature Models. In: ACM; 2011; Waikiki, Honolulu, HI, USA: 461–470.

39. Font J, Ballarín M, Haugen Ø, Cetina C. Automating the Variability Formalization of a Model Family by Means of Common Variability Language. In: ; 2015; Nashville, Tennessee, USA: 411–418.

40. Bass L, Clement P, Kazman R. *Software Architecture in Practice (2nd Edition).* Addison-Wesley Professional . 1998.

41. Nakagawa EY, Oquendo F, Becker M. RAModel: A Reference Model for Reference Architectures. In: ; 2012: 297–301.

42. Galster M, Avgeriou P. Empirically-grounded Reference Architectures: A Proposal. In: ACM; 2011: 153–158.

43. Angelov S, Grefen PWPJ, Greefhorst D. A classification of software reference architectures: Analyzing their success and effectiveness. In: ; 2009: 141–150.

44. Weinreich R, Buchgeher G. Automatic reference architecture conformance checking for soa-based software systems. In: IEEE; 2014; Sydney, Australia: 95–104.

45. Quinton C, Haderer N, Rouvoy R, Duchien L. Towards Multi-cloud Configurations Using Feature Models and Ontologies. In: ACM; 2013; Prague, Czech Republic: 21–26.

46. Romero D, Quinton C, Duchien L, Seinturier L, Valdez C. SmartyCo: Managing Cyber-Physical Systems for Smart Environments. In: Weyns D, Mirandola R, Crnkovic I., eds. *Proceedings of the 9th European Conference on Software Architecture (ECSA 2015)*Springer; 2015; Dubrovnik/Cavtat: 294–302.

47. Vierhauser M, Rabiser R, Grünbacher P, Egyed A. Developing a DSL-Based Approach for Event-Based Monitoring of Systems of Systems: Experiences and Lessons Learned. In: ACM; 2015; Lincoln, Nebraska, USA: 715–725.

48. Steinberg D, Budinsky F, Paternostro M, Merks E. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional. 2nd ed. 2009.

49. Bąk K, Czarnecki K, Wąsowski A. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In: Malloy B, Staab S, Brand v. dM., eds. *Proceedings of the 4th International Conference on Software Language Engineering (SLE 2011)*Braga, Portugal: Springer Berlin Heidelberg. 2011 (pp. 102–122).

50. Dhungana D, Grünbacher P, Rabiser R. The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study. *Automated Software Engineering* 2011; 18(1): 77–114.

51. Berger T, She S, Lotufo R, Wąsowski A, Czarnecki K. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 2013; 39(12): 1611–1640.

52. Quinton C, Romero D, Duchien L. SALOON: a Platform for Selecting and Configuring Cloud Environments. *Software: Practice and Experience* 2016; 46(1): 55–78.

53. Prud'homme C, Fages JG, Lorca X. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.; 2014.

54. Gamez N, Fuentes L. Software Product Line Evolution with Cardinality-Based Feature Models. In: Springer Berlin Heidelberg; 2011; Pohang, South Korea: 102–118.

55. Vierhauser M, Grünbacher P, Heider W, Holl G, Lettner D. Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines. In: ; 2012; Innsbruck, Austria: 531–545.

56. Egyed A. Instant consistency checking for the UML. In: ACM. ; 2006; Shanghai, China: 381–390.

57. Arcega L, Font J, Haugen Ø, Cetina C. Achieving Knowledge Evolution in Dynamic Software Product Lines. In: IEEE; 2016; Osaka, Japan: 505–516.

58. Capilla R, Valdezate A, Díaz FJ. A Runtime Variability Mechanism Based on Supertypes. In: ; 2016: 6-11

59. Czarnecki K, Pietroszek K. Verifying feature-based model templates against well-formedness OCL constraints. In: ACM; 2006; Portland, Oregon, USA: 211–220.

60. Deng G, Schmidt DC, Gokhale A, Gray J, Lin Y, Lenz G. Evolution in model-driven software product-line architectures. In: Tiako P., ed. *Designing Software-intensive Systems*Idea Group Inc. (IGI). 2008 (pp. 1280–1312).

61. Elsner C, Lohmann D, Schröder-Preikschat W. Fixing Configuration Inconsistencies across File Type Boundaries. In: IEEE CS; 2011; Oulu, Finland: 116–123.

62. Gamez N, Fuentes L. Architectural evolution of FamiWare using cardinality-based feature models. *Information and Software Technology* 2013; 55(3): 563 - 580. Special Issue on Software Reuse and Product Linesdoi: https://doi.org/10.1016/j.infsof.2012.06.012

63. Helleboogh A, Weyns D, Schmid K, Holvoet T, Schelfthout K, Van Betsbrugge W. Adding Variants on-the-fly: Modeling Meta-Variability in Dynamic Software Product Lines. In: ; 2009; San Francisco, CA, USA: 18–27.

64. Mauro J, Nieke M, Seidl C, Yu IC. Context-aware reconfiguration in evolving software product lines. *Science of Computer Programming* 2018; 163: 139 - 159. doi: https://doi.org/10.1016/j.scico.2018.05.002

65. Mende T, Beckwermert F, Koschke R, Meier G. Supporting the grow-and-prune model in software product lines evolution using clone detection. In: IEEE CS; 2008: 163–172.

66. Murta LG, Van Der Hoek A, Werner CM. ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links. In: ACM; 2006; Tokyo, Japan: 135–144.

67. Seidl C, Schaefer I, Aßmann U. Integrated Management of Variability in Space and Time in Software Families. In: ACM; 2014; Florence, Italy: 22–31.

68. Gomaa H, Hashimoto K. Dynamic Software Adaptation for Service-oriented Product Lines. In: ACM; 2011; Munich, Germany: 35:1–35:8.

69. Baresi L, Guinea S, Pasquale L. Service-Oriented Dynamic Software Product Lines. *Computer* 2012; 45(10): 42–48.

70. Parra C. *Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations*. Thesis. Université des Sciences et Technologie de Lille - Lille I, 2011.

71. Bashari M, Bagheri E, Du W. Dynamic Software Product Line Engineering: A Reference Framework. *International Journal of Software Engineering and Knowledge Engineering* 2017; 27: 191-234. doi: 10.1142/S0218194017500085

72. Bencomo N, Hallsteinsen S, De Almeida ES. A View of the Dynamic Software Product Line Landscape. *Computer* 2012; 45(10): 36–41.

73. Deelstra S, Sinnema M, Bosch J. Variability assessment in software product families. *Information and Software Technology* 2009; 51(1): 195–218.

74. Johnsson S, Bosch J. Quantifying software product line ageing. In: ; 2000; Limerick, Ireland: 27–32.

75. Elsner C, Botterweck G, Lohmann D, Schröder-Preikschat W. Variability in Time - Product Line Variability and Evolution Revisited. In: Universität Duisburg-Essen; 2010; Linz, Austria: 131–137.

76. Schmid K, Eichelberger H. A Requirements-Based Taxonomy of Software Product Line Evolution. *Electronic Communications of the EASST* 2007; 8: 1–13.

77. Passos L, Czarnecki K, Apel S, Wąsowski A, Kästner C, Guo J. Feature-oriented Software Evolution. In: ACM; 2013; Pisa, Italy: 17:1–17:8.

78. Knauber P. Managing the Evolution of Software Product Lines. In: Springer; 2004; Madrid, Spain.

How to cite this article: C. Quinton., M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, and C. Schumayer (2018), Evolution in Dynamic Software Product Lines, *Journal of Software: Evolution and Process* , *2020;00:1–6*.