Evolution in Dynamic Software Product Lines: Challenges and Perspectives

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

Evolution in Dynamic Software Product Lines: Challenges and Perspectives

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

Evolution in Dynamic Software Product Lines: Challenges and Perspectives

Clément Quinton*, Rick Rabiser†, Michael Vierhauser†,
Paul Grünbacher† and Luciano Baresi*
*Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy
{clement.quinton | luciano.baresi}@polimi.it
†Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz, Austria
{rick.rabiser | michael.vierhauser | paul.gruenbacher}@jku.at

ABSTRACT

In many domains systems need to run continuously and cannot be shut down for reconfiguration or maintenance tasks. Cyber-physical or cloud-based systems, for instance, thus often provide means to support their adaptation at runtime. The required flexibility and adaptability of systems suggests the application of Software Product Line (SPL) principles to manage their variability and to support their reconfiguration. Specifically, Dynamic Software Product Lines (DSPL) have been proposed to support the management and binding of variability at runtime. While SPL evolution has been widely studied, it has so far not been investigated in detail in a DSPL context. Variability models that are used in a DSPL have to co-evolve and be kept consistent with the systems they represent to support reconfiguration even after changes to the systems at runtime. In this short paper we present a classification of the required operations for jointly evolving problem and solution space in a DSPL. We analyze the impact of such operations on the consistency of a DSPL and propose an approach to deal with the described issues. We describe a runtime monitoring system used in the domain of industrial automation software as an example of a DSPL evolving at runtime to motivate and explain our work.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies, Representation*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering, Reuse models*

Keywords

Dynamic Software Product Lines, Evolution, Consistency

1. INTRODUCTION

Dynamic software product line engineering [6, 11] has gained increasing popularity in recent years with the advent of software requiring runtime adaptation and reconfiguration capabilities, such as cyber-physical systems, runtime monitoring systems, cloud-based systems, or service-based systems. In conventional SPL engineering variation points are typically bound at design time. DSPL engineering on the other hand supports binding variability at runtime, thus enabling a system to reconfigure itself during operation to adapt to changes in its environment, *e.g.*, by relying on context-based information [11].

A DSPL, like any SPL, is a long-term investment that is in use for many years and needs to be continuously evolved *e.g.*, to meet new requirements [5] or to adopt new technologies [12]. Evolving a DSPL poses significant challenges as both problem and solution space must co-evolve with the system they describe to avoid inconsistencies during runtime adaptation. Most existing work on SPL evolution has focused on the evolution of the *problem space* [5]. However, evolving a variability model may also affect the related assets (*i.e.*, the *solution space*) and vice versa. Yet, limited work has been conducted to support such co-evolution. For instance, a set of evolutions of the problem space and of the solution space as well as remapping operators to avoid inconsistencies between the two spaces are described in [17, 4]. Other authors [13, 18] study co-evolution within the Linux kernel, to extract co-evolution patterns for the Linux kernel variability model and keep it in sync with the actually implemented variability. Existing research only recently started to investigate evolution in a DSPL context, and especially its impact on the running system. Helleboogh *et al.* [10] proposed the notion of super-types to describe the evolution of variability models at runtime. Capilla *et al.* [6] use super-types to automate the modification of variants in a feature model at runtime. However, these approaches are limited to a given set of changes, *e.g.*, the addition of a variant, as other kinds of changes cannot be automated [9]. Based on our work on runtime reconfiguration of monitoring systems [20, 16] and on dynamic architecture evolution in DSPL engineering [1], we investigate the issues related to DSPL evolution in more detail.

In this paper we describe our first results, providing the following contributions: we illustrate DSPL evolution using an example of runtime monitoring systems for industrial automation. We then present a *classification of required operations for jointly evolving problem and solution spaces in a* DSPL. We do not want to emphasize one particular variability modeling approach (*e.g.*, feature-based, decision-oriented, UML-based, or orthogonal variability models) but we use the general term variability model to describe any

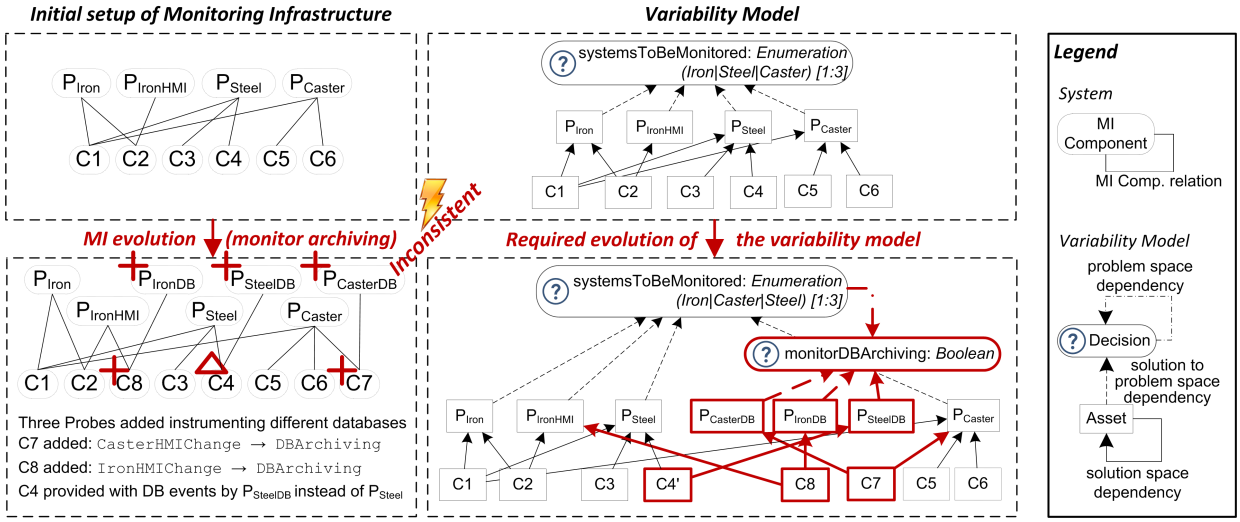

Figure 1: Evolution of the MI DSPL (left) and its impacts on the variability model supporting reconfiguration of the MI (right).

model of the variability of a software system. Such a model may be created using any approach that provides advanced modeling capabilities such as cardinalities and attributes [7, 15] necessary in a DSPL context to describe additional information like the number of instances of a feature or component at runtime. We also *analyze the impact of evolution operations on the consistency of the* DSPL: the evolution of problem or solution space can lead to inconsistencies within the given space, between spaces, and with respect to rules for the runtime adaptation of the system. Finally, we outline an *architecture of a tool-supported approach that addresses the described issues and supports the evolution of a* DSPL.

2. DSPL EVOLUTION EXAMPLE

2.1 A Monitoring Infrastructure DSPL

In previous work, we developed a *Monitoring Infrastructure* (MI) providing support for analyzing the behavior of a system of systems (SoS) in the domain of industrial plant automation [20]. The MI relies on probes instrumenting a system and providing events and data. Constraints are then used to check the behavior of the system at runtime based on these events and data. To support the reconfiguration of our MI at runtime we have developed a variability management approach [16], i.e., we describe the variability of the key components of the MI – *probes*, *events*, and *constraints* – using variability models. The MI is a DSPL and allows us to illustrate the co-evolution challenges outlined above.

Fig. 1 (top left) shows a simplified representation of a MI developed to monitor several systems of a plant automation SoS: four probes instrument an ironmaking automation system and its human-machine interface (HMI), a steelmaking automation system, as well as a continuous casting automation system. The probes provide events that are checked by six constraints (C1-C6) at runtime. We do not present details on all constraints here for the sake of simplicity (see [16] for details). Constraint `C4`, for instance, checks that after a plant operator changed a steel production plan, these changes are archived in the database. This constraint relies on events provided by probe `PSteel`. Both, probes and constraints are components of the MI DSPL and can be activated and deactivated at runtime. Furthermore, the implementation of constraints (*i.e.*, the code written in a constraint language) can be modified at runtime. Fig. 1 (top right) depicts a partial variability model of the MI that allows one to select the systems to be monitored. When resolving variability (*e.g.*, after the decision of a user), the probes and constraints are activated or deactivated accordingly at runtime. In this example we use a simplified version of a decision model [7].

2.2 DSPL Evolution

Whenever the systems monitored by the MI change, the probes and constraints defined and managed by the MI potentially need to be modified as well (at runtime). For example, if probes no longer provide certain events after changes, constraints on these events may detect violations even if the adapted system works correctly. Furthermore, the monitoring needs of the user of the MI may also change over time. Whenever the MI evolves, the variability model supporting its reconfiguration at runtime has to co-evolve as well.

Fig. 1 (bottom) depicts such an evolution scenario, where the MI evolves at runtime to enable monitoring databases that archive events for all systems. The user of the MI develops new probes for instrumenting each monitored system's database component (`PIronDB`, `PSteelDB`, `PCasterDB`) to receive events whenever plans for producing iron or steel are archived in the database. The existing probes `PIronHMI`, `PSteel`, and `PCaster` already provide events whenever such a plan is changed via the user interface (HMI). The user of the MI defines constraints to check that HMI plan change events lead to database archiving events. For the ironmaking and continuous casting systems she/he defines constraints `C7` and `C8` and relates them to the respective probes. For the steelmaking system, constraint `C4` already checks archiving events for steel production and needs to be modified and related with the new probe `PSteelDB`.

As a result of these changes the variability model of the MI is inconsistent with the new version of the MI and both, problem and solution space, need to be updated to reflect the modified MI system. In the solution space, assets for the new probes and constraints need to be defined together with their

Evolution Operation (✓) and *Potential Impact*			Potential Impacts on DSPL Consistency
`Problem Space`	`Mapping`	`Solution Space`	
✓: Add Element	*Add/Update Element*	*Add Element*	Reconfiguration choice without effect (dead feature)
✓: Remove Element	*Remove Element*	*Remove Element*	Reconfiguration fails (dead asset, can't be activated)
✓: Update Element		*Update Element*	Reconfiguration fails (false optional or dead feature)
Add Element	✓: Add Element	*Add Element*	Missing relation among choices and components
Remove Element	✓: Remove Element	*Remove Element*	Reconfiguration fails (see above)
	✓: Update Element		Reconfiguration fails (see above)
Add Element	*Add/Update Element*	✓: Add Element	Component can't be activated during reconfiguration
Remove Element	*Remove Element*	✓: Remove Element	Reconfiguration fails (see above)
Update Element	*Update Element*	✓: Update Element	Reconfiguration fails (see above)

Table 1: Changes in different spaces and their impact on other spaces and the DSPL consistency.

relations, and the existing asset representing constraint `C4` and its relations need to be updated. In the problem space, a new decision needs to be added to allow reconfiguring the MI for monitoring database archiving and be mapped to the new assets.

3. DSPL EVOLUTION OPERATIONS

Evolving a DSPL is challenging as changes in different modeling spaces have different impacts on the way the running system can be reconfigured. Table 1 describes the performed intraspacial evolution operations [17] (marked with ✓) as well as the potential (interspacial) impacts on other spaces.

Problem space.

Adding a new model element to the problem space has an impact on the mapping with the solution space. The added element must either be mapped to an existing asset or a new asset needs to be added and mapped. For instance, when adding a decision regarding archiving in our MI example, assets `PCasterDB`, `PIronDB`, and `PSteelDB` need to be related with this new decision. Removing a model element from the problem space similarly has interspatial effects. The element has to be removed either together with its mapping to avoid a dangling reference or together with both its mapping(s) and related asset(s). Updating a model element in the problem space may involve different properties, *e.g.*, the element name, the range of an element's cardinality, or one of its attribute values. It may be either problem space-specific or result in required changes in the solution space. For example, changing the name of a decision does not necessarily require updating assets, whereas changing the range of possible decision values can require modifying assets mapped to the original range.

Problem-to-solution space mappings.

Evolution can also be necessary after changing the mappings between problem and solution spaces. For instance, this can mean creating a relation between existing problem and solution space elements (intraspatial evolution), or creating the mapping together with a new element in the problem space, solution space, or both (interspatial evolution). In our example, mappings from the new probe assets regarding archiving to the new archiving decision have to be created. Removing a mapping may similarly be possible without modifying elements of the problem or solution space, or it may require deleting elements from the problem and/or solution space. Updating a mapping means changing its own reference to an existing model element or asset, and is thus intraspatial.

Solution space.

Finally, evolution operations can be triggered by changing the solution space, as depicted in Fig. 1. Adding or removing an element from the solution space is similar to adding or removing a problem space element, and thus represents interspatial evolution. It requires adding or updating mappings and elements of the problem space depending on the kind of relationship (1:1 or n:m). In our MI example, three probe assets and two constraint assets are added to the solution space, which requires adding several new mappings. The evolution of the solution space can either be intraspatial (*e.g.*, changing the implementation of an existing asset) or interspatial (*e.g.*, splitting or merging existing assets and updating related mappings and problem space elements). In our MI example, an asset (`C4`) is also updated, *i.e.*, certain attributes change and a new relation needs to be defined.

In DSPLs these evolution operations have to be performed at runtime, without introducing inconsistencies, to enable successful system reconfiguration after DSPL evolution.

4. IMPACTS ON DSPL CONSISTENCY

As shown in Table 1 some evolution operations may lead to inconsistencies in the DSPL. For instance, in our running example, adding constraint assets `C7` and `C8` without relating them to any probe assets would make them dead assets. The consistency of the DSPL must be checked when performing changes, regardless of whether the evolution is intraspatial or interspatial [17].

Intraspatial consistency.

Foreseeing the impact of a certain change in one modeling space on the overall model can be difficult. Analyses [2] can be used after evolution to detect potential anomalies such as *dead or false-optional features* (see Table 1). Addi-

tional analyses are required to check the consistency of the variability model with respect to cardinalities [14]. Existing approaches and tools can also be applied to check the consistency of the solution space, where similar anomalies can arise [12]. For instance, dependencies between assets can prevent one of them being used in any reconfigured product, thus being considered as a *dead asset*. An evolution operation may also remove an asset without evolving the problem space accordingly, thus leading to correct products in the solution space that cannot be reconfigured correctly anymore using the problem space model. Finally, all mappings between both spaces must be consistent, *i.e.*, references between elements in both spaces must refer to existing elements in each space.

Interspatial consistency.

Although each space may be consistent if considered alone, evolution can lead to an inconsistency when considering the two spaces together. Let $\mathcal{V}$ be the set of products that can be derived from the DSPL variability model, and $\mathcal{A}$ the set of products that can be composed with existing assets and their dependencies. Then the product line is considered as interspatial consistent *iff* $\mathcal{V} \subseteq \mathcal{A}$. In our running example (cf. Fig. 1), new assets, a new decision and a mapping between new assets and the new decision are added, leading to a consistent problem space, solution space, and mappings between them. However, this evolution can still introduce an inconsistency in the DSPL. For example, the newly introduced `PSteelDB` asset provides database archiving events, which were already provided by `PSteel` before the evolution. Constraint `C4` is based on these archiving events, which are provided by two different probes now, `PSteel` and `PSteelDB`. Thus, either `PSteel` needs to be modified to no longer provide archiving events or the mapping must be updated to avoid that both `PSteel` and `PSteelDB` send the same events at once.

Evolution and Runtime Adaptations.

Reconfigurations rely *(i)* on the DSPL variability model that must be consulted at runtime to find adaptations [11], and *(ii)* on predefined adaptation rules, *e.g.*, goal policies or Event-Condition-Action (ECA) rules [3]. While evolving the DSPL, the consistency between the variability model and such adaptation rules must thus be ensured. Considering our running example, after the evolution of the variability model the user can reconfigure the MI to monitor database archiving. However, in practice database archiving only makes sense to be activated when a database is actually up and running. The MI knows which systems are up and running and can (de-)activate probes and constraints automatically depending on the system configuration. However, the variability model is not aware of such changes and thus will offer the reconfiguration option for database monitoring regardless of the status of the monitored system. Reconfiguration rules of the MI must thus be updated together with the variability model to avoid such inconsistencies.

5. SOLUTION ARCHITECTURE

We are currently working on a tool-based approach supporting evolution in a DSPL to put the ideas described in this paper to practical use. We intend to automate the described evolution operations as well as to provide support for managing inconsistencies. Fig. 2 depicts the key components of our solution architecture.

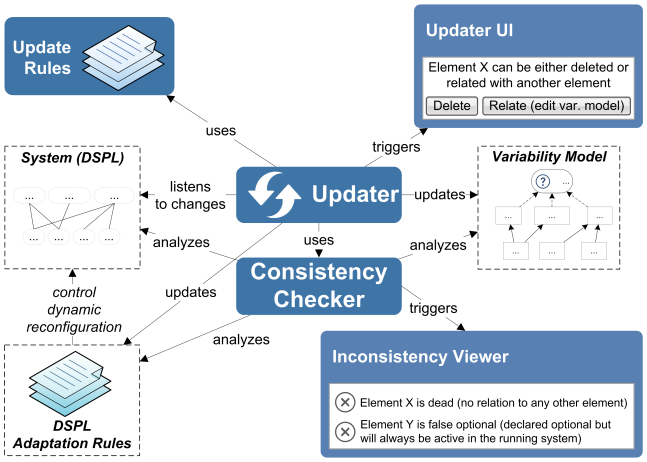

Figure 2: Solution architecture to support DSPL evolution.

Each change of the running system in a DSPL should lead to an automated update of *(i)* the variability model specifying reconfiguration options and *(ii)* the DSPL adaptation rules controlling automatic runtime reconfiguration based on these reconfiguration options. Changes in the solution space can be handled in a systematic and automated manner, because information about which components are added or removed and how components are related with each other can be extracted from the running system. This information can then be used to automatically update the variability model. More specifically, the `Updater` component listens to all changes made in the running system (*e.g.*, component added, component deleted, or dependency among components created) and updates the variability model accordingly (*e.g.*, adds an asset, removes an asset, or creates asset relations in the asset model).

The `Updater` works on top of several `Update Rules` to automate the evolution process as far as possible. These rules are based on the described evolution operations. For instance, for each new asset (in our MI example, the archiving probes and the two new constraints), a rule specifies to create a default decision together with a mapping to allow selecting the new asset during reconfiguration. Several rules can also be specified for updating problem space dependencies (decision relations) and attributes (cardinality). For example, whenever an asset related to a decision with a cardinality is removed, the cardinality maximum has to be reduced by 1.

However, evolving the problem space as well as the mappings can also require human intervention. Considering our MI example, the new probe and constraint assets could either be related to an existing decision or a new decision can be added. The `Updater UI` is triggered in such cases and allows the user to make a choice before the `Updater` component updates the variability model.

The `Updater` also relies on a `Consistency Checker` that analyzes the variability model and compares it with the running system as well as with the DSPL adaptation rules. Based on the `Update Rules`, some inconsistencies can be automatically resolved, *e.g.*, a default decision can be created in the variability model for a detected dead asset and an adaptation rule related to a deleted decision can be removed. Inconsis-

tencies that cannot be resolved automatically are displayed to the user in an `Inconsistency Viewer` to allow her/him to manually resolve them by modifying the variability model and/or the DSPL adaptation rules.

At the time of writing, we have implemented a prototype and tested it on evolution scenarios from the MI domain [16]. The `Updater` component already works and several Java-based `Update Rules` have been implemented as part of an Eclipse plug-in for the decision-oriented DOPLER tool [8]. We are currently working on the `Updater UI`, the `Consistency Checker`, and the `Inconsistency Viewer`. For the latter two we adapt the approach proposed in [14] and combine it with the consistency checker described in [19]. Our current prototype already supports identifying problem space elements (decisions) without relation to any asset and resolves these inconsistencies automatically.

6. CONCLUSIONS AND FUTURE WORK

While evolution is a fundamental concern, the DSPL community has not considered it as a priority yet. We have discussed the different challenges related to the evolution of a DSPL as well as different inconsistencies that can arise during evolution. We have presented evolution operations regarding the problem and solution space, as well as the mappings of these spaces in a DSPL. We have also described how these evolution operations may propagate from one space to the other. We have further described how these operations can introduce inconsistencies that can impact one or both spaces and prevent proper runtime adaptations. We have outlined a tool-based approach for supporting evolution in a DSPL. We are working on this approach and we plan to generalize it to different types of variability models. Finally, we would like to evaluate our approach in the context of case studies from different domains, *e.g.*, cloud-based systems or cyber-physical systems.

Acknowledgments

This work has been partially supported by project EEB – Edifici A Zero Consumo Energetico In Distretti Urbani Intelligenti (Italian Technology Cluster For Smart Communities) – CTN01_00034_594053, the Christian Doppler Forschungsgesellschaft Austria, and Primetals Technologies.

7. REFERENCES

[1] L. Baresi and C. Quinton. Dynamically Evolving the Structural Variability of Dynamic Software Product Lines. In *SEAMS*, 2015.

[2] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[3] N. Bencomo, J. Lee, and S. O. Hallsteinsen. How Dynamic is your Dynamic Software Product Line? In *SPLC (vol. 2)*, pages 61–68, 2010.

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

[5] G. Botterweck and A. Pleuss. Evolution of software product lines. In *Evolving Software Systems, Mens, T., Serebrenik, A., and Cleve, A. (eds.)*, pages 265–295. Springer, 2014.

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

[7] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *VaMoS*, pages 173–182. ACM, 2012.

[8] D. Dhungana, P. Grünbacher, and R. Rabiser. The dopler meta-tool for decision-oriented variability modeling: A multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011.

[9] W. Heider, R. Rabiser, and P. Grünbacher. Facilitating the Evolution of Products in Product Line Engineering by Capturing and Replaying Configuration Decisions. *STTT*, 14(5):613–630, 2012.

[10] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. Van Betsbrugge. Adding Variants on-the-fly: Modeling Meta-Variability in Dynamic Software Product Lines. In *DSPL, SPLC*, pages 18–27, 2009.

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

[12] J. McGregor. The Evolution of Product Line Assets. Technical Report CMU/SEI-2003-TR-005, 2003.

[13] L. Passos, T. Leopoldo, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel. *Empirical Software Engineering*, To appear 2015.

[14] C. Quinton, A. Pleuss, D. Le Berre, L. Duchien, and G. Botterweck. Consistency Checking for the Evolution of Cardinality-based Feature Models. In *SPLC*, pages 122–131. ACM, 2014.

[15] C. Quinton, D. Romero, and L. Duchien. SALOON: a platform for selecting and configuring cloud environments. *Software:Practice and Experience*, 2015.

[16] R. Rabiser, M. Vierhauser, and P. Grünbacher. Variability management for a runtime monitoring infrastructure. In *VaMoS*, pages 35–42. ACM, 2015.

[17] C. Seidl, F. Heidenreich, and U. Aßmann. Co-evolution of Models and Feature Mapping in Software Product Lines. In *SPLC*, pages 76–85. ACM, 2012.

[18] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem. In *EuroSys*, pages 47–60. ACM, 2011.

[19] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider. Flexible and scalable consistency checking on product line variability models. In *ASE*, pages 63–72. ACM, 2010.

[20] M. Vierhauser, R. Rabiser, P. Grünbacher, C. Danner, S. Wallner, and H. Zeisel. A flexible framework for runtime monitoring of system-of-systems architectures. In *WICSA*, pages 57–66. IEEE, 2014.