Dynamically Evolving the Structural Variability of Dynamic Software Product Lines

Luciano Baresi, Clément Quinton

Dynamically Evolving the Structural Variability of Dynamic Software Product Lines

Luciano Baresi and Clément Quinton
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Piazza L. Da Vinci, 32 - 20133 Milano, Italy
Email: {luciano.baresi | clement.quinton}@polimi.it

***Abstract*—A *Dynamic Software Product Line* (DSPL) is a widely used approach to handle variability at runtime, *e.g.*, by activating or deactivating features to adapt the running configuration. With the emergence of highly configurable and evolvable systems, DSPLs have to cope with the evolution of their structural variability, *i.e.*, the *Feature Model* (FM) used to derive the configuration. So far, little is known about the evolution of the FM while a configuration derived from this FM is running. In particular, such a dynamic evolution changes the DSPL configuration space, which is thus unsynchronized with the running configuration and its adaptation capabilities. In this position paper, we propose and describe an initial architecture to manage the dynamic evolution of DSPLs and their synchronization. In particular, we explain how this architecture supports the evolution of DSPLs based on FMs extended with cardinality and attributes, which, to the best of our knowledge, has never been addressed yet.**

I. Introduction

Deploying large-scale and highly distributed systems such as cloud-based or cyber-physical systems has recently emerged as a major trend in software development. While running, these systems have to cope with changes that may occur in their environment, and thus have to adapt dynamically their configuration at runtime to face those changes. A well-known approach to deal with runtime adaptations is by means of *Dynamic Software Product Lines* (DSPL) [1], [2]. DSPLs aim to bind features dynamically at runtime by activating or deactivating certain features according to the changing context. Those runtime reconfigurations are driven by the DSPL variability model, usually described as a *Feature Model* (FM), which defines the DSPL configuration space, *i.e.*, the possible and allowed reconfigurations.

DSPLs are built on the assumption that runtime adaptations can be foreseen at design time and thus rely on a predefined configuration (and reconfiguration) space, *i.e.*, the one limited by the set of configurations which can be derived from the FM. However, highly configurable systems have to cope with uncertainty, facing events that were not predicted at design time. To deal with these unforeseen events and propose new runtime adaptations, the DSPL configuration space has to evolve, *i.e.*, the DSPL FM must be edited. Evolving the structural variability of the DSPL is thus done dynamically, while derived configurations are running. Such dynamic structural evolutions lead to a synchronization issue, where running configurations are not bound to the correct configuration space when a reconfiguration is required to adapt to context changes. To deal with dynamic evolutions of DSPLs and address this synchronization issue, three main concerns must be taken into consideration. First, the *validation* of the dynamic evolution, to check whether or not the evolved FM is consistent after being edited and eventually apply the changes. Second, the *rebinding* stage, responsible of binding the new configuration space with the running system when required. Finally, the configuration *adaptation*, where the running configuration must be reconfigured according to a FM evolution or context changes.

The key contributions presented in this paper with respect to dynamic evolution of DSPLs are thus as follows.

- We propose a reference architecture for dynamically evolving DSPLs, which manages the three main stages of the DSPL evolution process described above. For each stage, we propose reasoning operations to avoid consistency and correctness issues, which cannot be properly handled manually by the DSPL maintainer due to their complexity.
- This architecture deals with FMs extended with both attributes and cardinalities, required to describe highly configurable systems. The dynamic reconfiguration is thus not only able to deal with feature activation/deactivation, but also with adaptation regarding quality attributes, as suggested in [3], number of instances of a feature or structural evolution involving constraints between features.

The rest of the paper is organized as follows: Section II introduces background information and motivates this work by highlighting the problem of evolving DSPLs. Section III presents the different kinds of evolution and explains their impact on configuration space synchronization. Section IV describes the architecture supporting the dynamic evolution of DSPLs, Section V discusses related work and Section VI concludes the paper.

II. Background and Motivation

This section introduces some background information about the FMs we use in our approach and illustrates the problems that might arise during their evolution at runtime by a motivating example.

A. Extended Feature Models

In our approach, we extend the FODA notation [4], well-known for variability modeling by means of *Boolean* FMs, with cardinalities and attributes. In the rest of the paper, we refer to this kind of FMs as *extended* FMs. These extensions are required to describe the variability of complex and highly configurable systems such as cloud environments [5], *e.g.*, to express required resources or the number of instances. A cardinality-based FM supports feature cardinalities defined as an interval $[m..n]$ [6], as well as constraints on these cardinalities, *e.g.*, to express that a feature requires a certain number of instances of another feature [7]. A feature cardinality specifies how many instances of a feature can be included in the configuration.

Attribute-based FMs, on the other hand, are FMs whose additional information is defined in terms of feature *attributes* [8], [9]. Those attributes, mostly used to specify non-functional properties, *e.g.*, a size or a quantity, can be either booleans, integers, reals, or enumerations. In addition, the extended FMs used in this paper come with attribute-based constraints [5], *i.e.*, attributes are constrained to a given value under certain conditions. Figure 1 depicts such an extended FM.

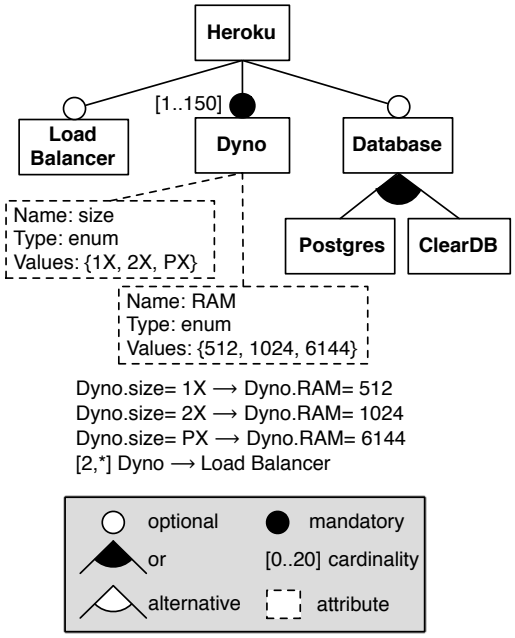


Fig. 1. Simplified Heroku PaaS Feature Model.

The FM describes an excerpt of the variability of Heroku[1], a Platform-as-a-Service cloud environment. Some configurations of this FM may involve up to 150 instances of the `Dyno` feature, which is a computational block that provides a certain amount of resources, *i.e.*, 512, 1024 or 6144 MB of memory depending on the `Dyno` size, as expressed by the three first constraints on attribute values. When at least two `Dynos` are part of the configuration, then a `Load Balancer` is required for balancing requests across these `Dynos`. This requirement is expressed with the fourth constraint, *i.e.*, a constraint over the `Dyno` feature cardinality. In addition, Heroku provides optional support to several databases such as `Postgres` and `ClearDB`.

In the remainder of the paper, all references to the architectural FM of the DSPL imply an extended FM.

B. Motivation and Examples

Current limitations of today's DSPLs rely on their inability to change the structural variability at runtime, and most of dynamic variability techniques deal with reconfiguration tasks such as activating or deactivating features without dealing with dynamic variability changes [10]. Yet, DSPLs (and SPLs) are likely to structurally evolve over time, for at least two main reasons. First, they are often a long-term investment, and they need to evolve to meet new requirements over many years. Second, dealing with a single, large, monolithic FM is not the right way: previous work has pointed out that dealing with such a large FM is problematic [11], [12], and it rather suggests to deal with incremental edits to the FM [13].

So far, little support exists for evolving the FM of a DSPL, that is, dynamically evolving its architectural FM while a derived system configuration is running and might require runtime adaptations. For example, cloud computing is a domain constantly evolving, and such a support is required to properly handle those evolutions, *e.g.*, availability of a new service, legacy service no more supported, new deployment plan and provided resources, etc. For instance, in February 2014, Heroku introduced a new kind of `Dyno`, the `PX` one, providing 12 times more RAM than the basic `1X` one, as depicted in Figure 1. Such an evolution is not functional and thus does not require any adaptation involving feature activation or deactivation. Nevertheless, it may require an adaptation of the running configuration, *e.g.*, to run a better-suited configuration regarding non-functional requirements (*e.g.*, a configuration with one `PX Dyno` can be cheaper than one with six `2X Dynos`, for the same amount of provided resources). Another issue was met by early adopters of `Postgres` instances based on the 32-bit version. Heroku then changed to 64-bit ones, leading to incompatibility issues for customers trying to upgrade to a larger database plan [14]. In other words, the running configurations relying on 32-bits versions were not synchronized with the Heroku architectural model providing 64-bits versions and adapting these configurations was impossible.

Evolving the architectural FM, although necessary, may nevertheless lead to two main issues. First, inconsistencies may arise in the new FM which has to be checked prior to be used as the DSPL architectural FM. In DSPL approaches, model driven engineering has been widely used to address this issue, *e.g.*, relying on type conformance [15]. However, this is not sufficient in the presence of extended FMs, where additional checking operations must be performed to ensure the consistency of the model (see Section IV-B). Second, once

[1]https://www.heroku.com

evolved, the architectural FM provides a configuration space that is different from the one provided by the previous FM, thus leading to a running configuration that is not synchronized anymore with the DSPL FM when trying to adapt.

III. Desynchronized Configuration Spaces

In this section, we provide a big picture of the problem of configuration space desynchronization. Then, we analyze the different kinds of dynamic evolution regarding the DSPL FM and whether they require a rebinding of the configuration space.

A. Structural Variability Evolution and Desynchronization

Evolving the architectural FM of a DSPL while a derived configuration is running leads to a desynchronization between this running configuration and the DSPL FM, and causes issues when an adaptation is triggered. Figure 2 illustrates such a desynchronization.

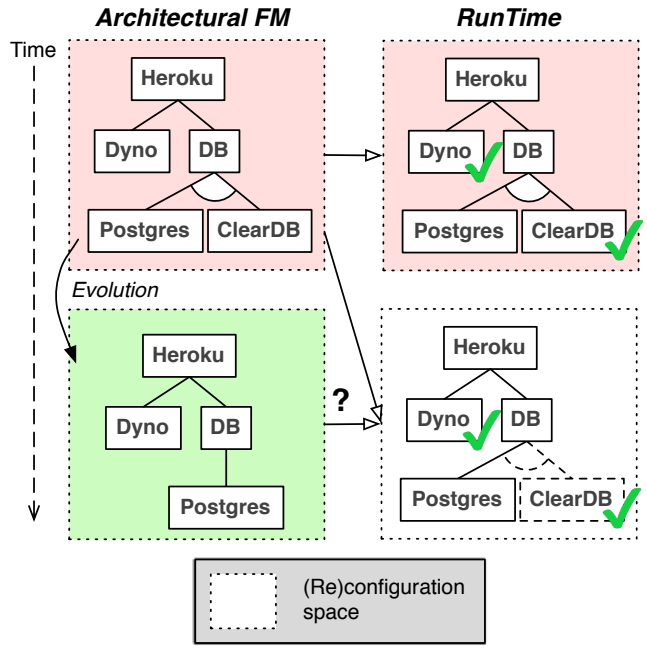


Fig. 2. Evolving the DSPL FM and related (re)configuration spaces.

The running system, *i.e.*, the derived configuration used at runtime, is based on the original architectural FM, as depicted in the top part of the figure. If required, *e.g.*, because of context changes, the running configuration can be adapted and changed accordingly by activating or deactivating a feature, or by reducing the number of running feature instances. This adaptation is done within a given configuration space, *i.e.*, the one provided by the architectural FM the running configuration has been derived from. Then, for any possible reason, *e.g.*, unmodeled features because they were unknown *a priori* or because of mistakes in the FM, the architectural FM of the DSPL has to evolve, while the existing configuration is still running and adapting itself according to context changes. The new architectural FM (bottom-left corner of the figure) now comes with a new configuration space, *i.e.*, without the `ClearDB` feature. In this situation, the running configuration is said *desynchronized* with respect to the new reference architectural FM (bottom part of the figure), since originally bound and derived from the previous FM configuration space (top part of the figure).

The main issue with such a desynchronization is to determine and bind the correct configuration space to the running system for further possible adaptations. In the depicted example, should the running system still rely on `ClearDB` and, if not, should it be adapted according to the former or the new configuration space? In the next section, we discuss the different evolution scenarios about the DSPL architectural FM and their impact on the synchronization.

B. Synchronizing Configuration Spaces

This section investigates edits (manual or automated) done to extended FMs and their impact on the configuration space evolution. We consider here atomic model edits only, *i.e.*, to add, remove or update a model element where updating a model element means changing one of its properties. Here, relevant model elements are features and cross-tree constraints, and relevant feature properties are cardinalities and attributes. The different edits discussed in this paper are listed in Table I and Table II.

For each edit, the tables indicate whether the considered edit requires a rebinding to synchronize the FM and the running configuration $\mathcal{C}$ or not. In addition, it indicates whenever an adaptation is required. For both operations, we distinguish between a soft and a hard requirement. More precisely, we consider that a rebinding or an adaptation can be either *required* or *suggested*. The former is meant to be mandatory and refers to functional adaptations, while the latter is optional but may improve the running configuration by providing best-suited non-functional properties. This distinction is denoted in the table as R and r for a required and suggested rebinding, respectively (A and a for an adaptation). Please not that a rebinding, whatever required or suggested, is always followed by an adaptation, thus ensuring the running configuration $\mathcal{C}$ to be synchronized with the new FM.

Feature		Optional	Mandatory
`Add`		r	R
`Remove`	$f \notin \mathcal{C}$	-	N.A.
	$f \in \mathcal{C}$	A	R

Feature		Opt. → Mand.	Mand. → Opt.
`Update`	$f \notin \mathcal{C}$	R	N.A.
	$f \in \mathcal{C}$	-	a

TABLE I
Edits on features.

Table I describes the edits that can be performed on features. A feature f can be added or removed from the FM, or

can be updated from optional to mandatory (Opt. $\rightarrow$ Mand.) or conversely (other cardinality updates, *e.g.*, from [1..4] to [1..2], are discussed in TABLE II). Adding an optional feature in the DSPL FM does not require a rebinding. However, such a rebinding is suggested, since a more suitable configuration involving the added optional feature may be derived. On the other hand, when a new mandatory feature f is added, the rebinding is required to derive a new configuration including f, *e.g.*, to fix a bug in the current configuration $\mathcal{C}$.

Removing an optional feature f that is not involved in the current running configuration ($f \notin \mathcal{C}$) does not lead to any change, while removing a mandatory feature not involved in $\mathcal{C}$ cannot happen (if f is mandatory, $f \in \mathcal{C}$). On the other hand, removing an optional feature involved in $\mathcal{C}$ requires an adaptation of $\mathcal{C}$, *e.g.*, deactivate the feature. If such a feature is mandatory, then a rebinding is required. The reader may notice that, for the removal, we assume that the involved feature does not have dangling references in cross-tree constraints anymore (in such a case, the involved constraint must be removed before removing the feature).

Updating a feature f not involved in $\mathcal{C}$ from optional to mandatory implies $\mathcal{C}$ to be bound to the new configuration space and be adapted to include the mandatory feature. The reverse update is not applicable as a mandatory feature is always included in $\mathcal{C}$. Considering $f \in \mathcal{C}$, updating f from optional to mandatory does not change $\mathcal{C}$ as f is already included in $\mathcal{C}$. Finally, switching f from mandatory to optional does not prevent $\mathcal{C}$ to run properly, but an adaptation is suggested as alternative configurations may be more suitable.

Constraint	$\delta \rightarrow \beta$	
`Add` `Update` `Remove`	$\delta \notin \mathcal{C}$	-
	$\delta \in \mathcal{C}$	*A*
Cardinality / Attribute		
`Update`	$f \notin \mathcal{C}$	*a*
	$f \in \mathcal{C}$	*A*

TABLE II
EDITS TO CONSTRAINTS, CARDINALITIES AND ATTRIBUTES.

TABLE II describes the edits that can be performed on cross-tree constraints, whether Boolean or non-Boolean ones, *e.g.*, constraints on the number of instances of a given feature [16]. δ and β can thus be either features or attribute/cardinality values. Whatever the considered edit, an adaptation is required if $\delta \in \mathcal{C}$ is supposed to include, exclude or update β. If $\delta \notin \mathcal{C}$, then $\mathcal{C}$ does not have to change. The last edit discussed in this paper is about updating some properties of the FM, *i.e.*, the value of a feature cardinality or attribute. If the feature is not part of the running configuration, then an adaptation is suggested as the new value may enable more suitable alternative configurations. If the feature is part of the configuration, then the adaptation is required as the new value may have consequences on the whole configuration. For example, regarding a cloud environment DSPL, two application server nodes require a load balancer to be configured while it is not mandatory when only one node is configured [7].

In the next section, we describe an architecture that supports the dynamic evolution of the architectural FM of a DSPL, and provides related rebinding and adaptation mechanisms.

IV. PROPOSED ARCHITECTURE

To support dynamic architectural evolution in a DSPL engineering process, we propose an architecture that relies on three main components. The proposed architecture is depicted in Figure 3.

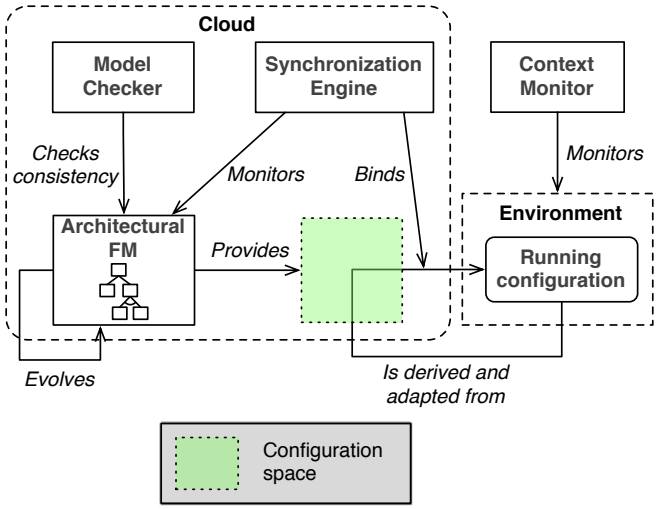


Fig. 3. Overview of the Architecture.

The initial running configuration is derived from the initial architectural FM, *i.e.*, with respect to the configuration space provided by this FM. Once the system is running, the runtime environment is monitored through a `context monitoring` engine and adaptations can be triggered (Section IV-A). These adaptations are performed with respect to a given configuration space. This configuration space may evolve if the architectural FM has evolved as well. In such a case, the consistency of the new FM is checked by the `model checker` before using it as the new reference model (Section IV-B). Once the consistency of the model is ensured, the `synchronization engine` is responsible for binding the running system with the correct configuration space if a rebinding is required (Section IV-C).

A. Runtime Adaptations

While existing DSPL architectures are focused on activation and deactivation of features to handle the required adaptation, our approach supports in addition new kinds of adaptation mechanisms, related to the FM extensions: adaptation with respect to *(i)* the number of feature instances (cardinalities) and *(ii)* the non-functional properties (attributes).

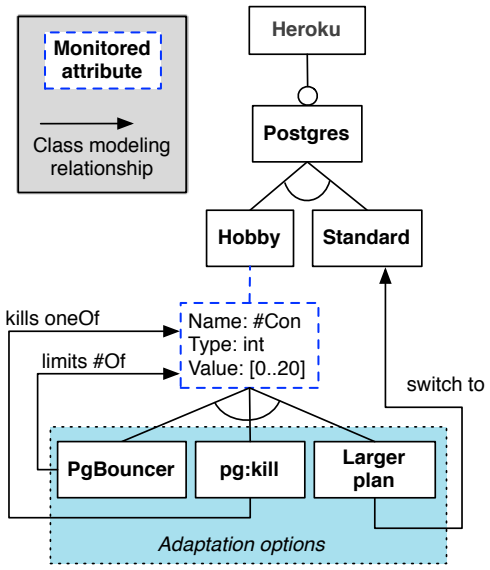


Fig. 4. Adaptation with respect to a changing attribute value.

Figure 4 illustrates an adaptation triggered by an evolving attribute value. Another excerpt of the Heroku FM is depicted, focusing on the `Postgres` database service. Heroku provides such a `Postgres` database support with different configuration plans, *i.e.*, different amount of resources and different prices. In this example, the `Hobby` configuration enables up to 20 active connections, which is lower than the `Standard` one that is more expensive (prices and attributes related to the `Standard` feature are not depicted for the sake of clarity). Once the database has reached the maximum number of active connections, it will no longer accept new connections. This will result in connection timeouts and will likely cause exceptions. In this case, there are three different adaptation options, depicted in the bottom part of the figure by mixing feature and class modeling as suggested in [17]. First, one can migrate to a `larger plan`. Second, we may consider killing the long running queries by the `pg:kill` command provided by the Heroku SDK. Finally, we may use the `PgBouncer` connection pooler which limits the number of connections before reaching the database connection limit.

Handling such adaptation mechanisms allows our architecture to support *soft* adaptations, *i.e.*, adaptations according to non-functional properties such as the number of active connections. These adaptations avoid the derivation of a new configuration with different functional features activated, for instance switching to a `ClearDB` database.

B. Checking the Consistency of the Evolving Feature Model

Evolving a model, whatever the model and evolution might be done dynamically or not, can lead to a new but inconsistent model. In our approach, once the architectural FM has evolved, the consistency of the new FM is checked through a two-step procedure. First, our architecture relies on model driven engineering principles, and each FM is defined as an instance of the extended FM metamodel depicted in Figure 5 (a complete description of this metamodel is given in [18]). Relying on this metamodel, a type-conformance checking procedure is executed to ensure the correctness of the evolved FM, *i.e.*, that the new FM conforms to the FM metamodel.

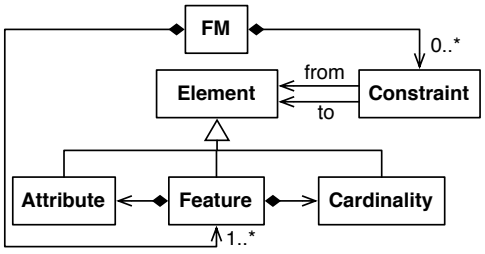


Fig. 5. Excerpt of the Extended FM Metamodel.

Second, we perform additional consistency checking. Indeed, dealing with extended FMs poses additional complexity, in particular when handling cardinalities into FMs and their constraints. Figure 6 depicts such a situation, the FM on the left-hand side of the figure being evolved to become the one on the right-hand side.

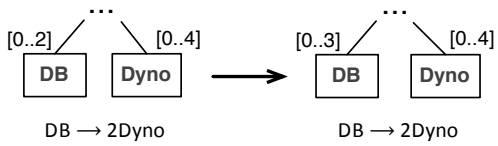


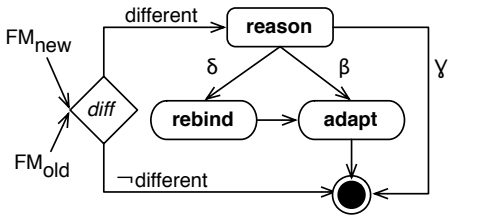
Fig. 6. A cardinality update leading to an inconsistency.

In this evolution scenario, one more Postgres database instance can now be configured, and the cardinality of the optional database feature `DB` is updated from [0..2] to [0..3]. However, the cross-tree constraint specifies that each configured instance of `DB` requires 2 instances of `Dyno` to run properly. As the upper bound of `Dyno` is still 4, it is not possible to configure more than 2 instances of `DB` since not enough `Dynos` are provided, and the evolved FM is said inconsistent. Although rather simplistic, this example shows that evolving extended FMs while ensuring their consistency is a complex task. In our previous work [16], we described the different kinds of inconsistencies that may arise in such evolution scenarios and showed that detecting and understanding why the FM is inconsistent is not trivial. We thus provided an automated support to detect such inconsistencies and explain the cause of the inconsistency and where it is located in the model. In our architecture, we rely on this automated support to perform the second consistency checking.

C. Configuration Space Synchronizer

The configuration space synchronizer is responsible for monitoring the changes of the architectural FM and for binding the running configuration with the new configuration space, if needed. Thus, once the FM has evolved and has been evaluated consistent, it might trigger a rebinding between the running system and the new configuration space. To determine whether

such a rebinding is required or not, the architecture provides a support to identify which kind of edit has been performed (see Section III-B). This identification is based on the computation of the differences between the FM before the evolution and the evolved one. In particular, it relies on a syntactic difference *diff* operator that takes two FMs as input and computes their differences [19]. Figure 7 depicts the integration of this operator in the synchronization decision process. Thus, a rebinding, an adaptation or nothing is performed given the result of the *diff* operator.



From TABLE I and TABLE II :

δ = {Add mand. f ; Remove mand. f, $f \in C$; ...}

β = {Add constraint ; Update cardinality ; ...}

ɣ = {Remove opt. f, $f \notin C$; ...}

Fig. 7. Synchronization decision process.

Binding the running configuration with a new configuration space does not only imply the notification to the running configuration when the new configuration space is available, but also the provision of all related software artifacts. Thus, these artifacts must be easily accessible by the running configuration to let it use them when reconfiguring. Although embedding them within the running configuration is possible, it is a threat to the system behavior, as it may increase memory footprint or computation time while decreasing overall performance. In the proposed solution, the FM and related artifacts are thus provided in the cloud and accessible as any other cloud service. When an evolution requires a rebinding, the synchronization engine notifies the running configuration that, in turn, uses provided artifacts for the reconfiguration.

V. Related Work

As pointed out by Capilla *et al.* [10] in their survey on DSPL practices, there are two main limitations in existing DSPL approaches regarding runtime variability management, both being addressed with the solution described in this paper. The first limitation refers to handling structural changes dynamically. So far, little support exists for evolving the DSPL variability model. For instance, Helleboogh *et al.* [20] propose to explicitly document the way the variability model of the DSPL may evolve using a so-called *meta-variability* model. This approach, however, is limited to specific evolution scenarios, *e.g.*, adding variants on-the-fly, and assumes that all possible evolutions are known and modeled at design time.

The second limitation is related to checking the consistency of the evolved structural variability model at runtime. Recently, Weyns [21] reported an extensive survey on the use of variability in software systems. In particular, this survey highlights the lack of support in current variability modeling approaches for guaranteeing correctness and consistency at runtime [22]. Morin *et al.* [23] described a model-driven approach to support adaptation in dynamically adaptive systems, and rely on metamodels to check the consistency of the designed models. We also perform such a checking, but the models involved in large-scale and highly configurable systems may be more complex than the ones they manage, and require additional consistency checking.

Among existing solutions for model checking FMs, few provide automated support and most of them only manage Boolean FMs, *i.e.*, FMs without extensions [24]. Within the proposed architecture, we leverage the automated consistency checking support described in [16] at runtime, by providing it as a *model checker-as-a-service*. Assessing the correctness of the DSPL and supporting the dynamic evolution of its structural variability provide very valuable means for the specification and analysis of robust and reliable self-adaptive systems [25]. Perrouin *et al.* [26], for instance, pointed out the synchronization issue when evolving the variability model at runtime and highlighted the need for a runtime checker ensuring the validity of dynamically adaptive systems.

VI. Conclusion & Research Agenda

In this paper, we have presented a solution for dealing with the dynamic evolution of DSPLs, *i.e.*, the evolution of the architectural FM of the DSPL while derived configurations are running. In particular, this architecture is able to handle the evolution of FMs extended with both attributes and cardinalities. These extended FMs are required to describe highly configurable systems such as cloud environments, which we used as running example to motivate the need for such an architecture. We have shown that dynamically evolving the FM of a DSPL leads to configuration space synchronization issues, and we have explained the different automated approaches and tools required to cope with these issues.

As future work, we envision to apply and evaluate our approach on different domains, *e.g.*, on cyber-physical systems. The aim of having a wider range of analyzed domains is to assess the validity of the proposed solution. In particular, synchronization rules might be slightly different whether the DSPL is used in the area of information systems or industrial automation, *e.g.*, delaying a rebinding or applying it only on a subset of running configurations. Further investigation could also consider traceability and history of DSPL evolutions to support the rollback of faulty reconfigurations or apply proactive adaptations.

Acknowledgment

The work presented in this paper has been partially supported by project Giocoso: GIOchi pediatrici per la COmunicazione e la SOcializzazione (Regione Lombardia).

REFERENCES

[1] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic Software Product Lines," *Computer*, vol. 41, no. 4, pp. 93–95, April 2008.

[2] M. Hinchey, S. Park, and K. Schmid, "Building Dynamic Software Product Lines," *Computer*, vol. 45, no. 10, pp. 22–26, Oct 2012.

[3] A. Metzger and K. Pohl, "Software Product Line Engineering and Variability Management: Achievements and Challenges," in *Proceedings of the Future of Software Engineering*, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 70–84. [Online]. Available: http://doi.acm.org/10.1145/2593882.2593888

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) - Feasibility Study," The Software Engineering Institute, Tech. Rep., 1990. [Online]. Available: http://www.sei.cmu.edu/reports/90tr021.pdf

[5] C. Quinton, D. Romero, and L. Duchien, "Automated selection and configuration of cloud environments using software product lines principles," in *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*. IEEE, 2014, pp. 144–151. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2014.29

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing Cardinality-based Feature Models and their Specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[7] C. Quinton, D. Romero, and L. Duchien, "Cardinality-based Feature Models with Constraints: A Pragmatic Approach," in *Proceedings of the 17th International Software Product Line Conference*, ser. SPLC'13. New York, NY, USA: ACM, 2013, pp. 162–166. [Online]. Available: http://doi.acm.org/10.1145/2491627.2491638

[8] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated Reasoning on Feature Models," in *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, ser. CAiSE'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 491–503. [Online]. Available: http://dx.doi.org/10.1007/11431855_34

[9] D. Batory, D. Benavides, and A. Ruiz-Cortes, "Automated Analysis of Feature Models: Challenges Ahead," *Commun. ACM*, vol. 49, no. 12, pp. 45–47, Dec. 2006. [Online]. Available: http://doi.acm.org/10.1145/1183236.1183264

[10] 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," *J. Syst. Softw.*, vol. 91, pp. 3–23, May 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.12.038

[11] M.-O. Reiser and M. Weber, "Multi-level feature trees: A pragmatic approach to managing highly complex product families," *Requir. Eng.*, vol. 12, no. 2, pp. 57–75, May 2007. [Online]. Available: http://dx.doi.org/10.1007/s00766-007-0046-0

[12] M. Acher, P. Collet, P. Lahire, and R. France, "Composing Feature Models," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, M. van den Brand, D. Gašević, and J. Gray, Eds. Springer Berlin Heidelberg, 2010, vol. 5969, pp. 62–81. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12107-4_6

[13] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged Configuration through Specialization and Multilevel Configuration of Feature Models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005. [Online]. Available: http://dblp.uni-trier.de/db/journals/sopr/sopr10.html#CzarneckiHE05a

[14] "How to Upgrade a Legacy Heroku Database," 2013, accessed 15.01.2015. [Online]. Available: http://blog.sendhub.com/post/30041247598/how-to-upgrade-a-legacy-heroku-database

[15] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-based Model Conformance and Multiview Consistency Checking," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 3, Jul. 2007. [Online]. Available: http://doi.acm.org/10.1145/1243987.1243989

[16] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck, "Consistency Checking for the Evolution of Cardinality-based Feature Models," in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, ser. SPLC'14. New York, NY, USA: ACM, 2014, pp. 122–131. [Online]. Available: http://doi.acm.org/10.1145/2648511.2648524

[17] K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski, "Clafer: Unifying class and feature modeling," *Software & Systems Modeling*, pp. 1–35, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10270-014-0441-1

[18] C. Quinton, D. Romero, and L. Duchien, "SALOON: a Platform for Selecting and Configuring Cloud Environments," *Software: Practice and Experience*, 2015. [Online]. Available: http://dx.doi.org/10.1002/spe.2311

[19] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle, "Feature Model Differences," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, Eds. Springer Berlin Heidelberg, 2012, vol. 7328, pp. 629–645. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31095-9_41

[20] 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 *Proceedings of the 3rd International Workshop on Dynamic Software Product Lines (DSPL'09)*, D. Muthig and J. D. McGregor, Eds. ACM, 2009, pp. 18–27.

[21] D. Weyns, "Variability: From software product lines to self-adaptive systems," in *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, ser. SPLC '14. New York, NY, USA: ACM, 2014, pp. 12–12. [Online]. Available: http://doi.acm.org/10.1145/2647908.2655959

[22] "Variability: From Software Product Lines to Self-Adaptive Systems," 2014, accessed 15.01.2015. [Online]. Available: http://homepage.lnu.se/staff/daweaa/papers/2014DSPL.pdf

[23] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ Run.time to Support Dynamic Adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[24] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2010.01.001

[25] L. Baresi, "Self-adaptive systems, services, and product lines," in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, ser. SPLC '14. New York, NY, USA: ACM, 2014, pp. 2–4. [Online]. Available: http://doi.acm.org/10.1145/2648511.2648512

[26] G. Perrouin, B. Morin, F. Chauvel, F. Fleurey, J. Klein, Y. Le Traon, O. Barais, and J.-M. Jézéquel, "Towards Flexible Evolution of Dynamically Adaptive Systems," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1353–1356. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337416