The Journal of Systems and Software xxx (2014) xxx–xxx

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry

Rafael Capilla [a,*], Jan Bosch [b], Pablo Trinidad [c], Antonio Ruiz-Cortés [c], Mike Hinchey [d]

[a] Rey Juan Carlos University, Madrid, Spain
[b] Chalmers University of Technology, Gothenburg, Sweden
[c] University of Seville, Seville, Spain
[d] Lero – The Irish Software Engineering Research Centre, Limerick, Ireland

ARTICLE INFO

Article history:
Received 17 November 2012
Received in revised form
16 December 2013
Accepted 23 December 2013
Available online xxx

Keywords:
Dynamic Software Product Lines
Dynamic variability
Software architecture
Feature models

ABSTRACT

Over the last two decades, software product lines have been used successfully in industry for building families of systems of related products, maximizing reuse, and exploiting their variable and configurable options. In a changing world, modern software demands more and more adaptive features, many of them performed dynamically, and the requirements on the software architecture to support adaptation capabilities of systems are increasing in importance. Today, many embedded system families and application domains such as ecosystems, service-based applications, and self-adaptive systems demand runtime capabilities for flexible adaptation, reconfiguration, and post-deployment activities. However, as traditional software product line architectures fail to provide mechanisms for runtime adaptation and behavior of products, there is a shift toward designing more dynamic software architectures and building more adaptable software able to handle autonomous decision-making, according to varying conditions. Recent development approaches such as Dynamic Software Product Lines (DSPLs) attempt to face the challenges of the dynamic conditions of such systems but the state of these solution architectures is still immature. In order to provide a more comprehensive treatment of DSPL models and their solution architectures, in this research work we provide an overview of the state of the art and current techniques that, partially, attempt to face the many challenges of runtime variability mechanisms in the context of Dynamic Software Product Lines. We also provide an integrated view of the challenges and solutions that are necessary to support runtime variability mechanisms in DSPL models and software architectures.

1. Introduction

Nowadays, many organizations have adopted a Software Product Line (SPL) development approach for building their own product portfolio with higher quality and at lower cost. The reduction in time-to-market conditions and the launching of new products or releases more quickly is one of the key drivers for the adoption of an SPL strategy. However, as more and more systems require the adaptation to different context conditions or working under better quality conditions, a number of challenges have emerged that static or conventional SPL approaches cannot provide. Companies producing large-scale software and embedded systems that require highly configurable options, many of them configured by end-users at post-deployment time, are facing new challenges to adapt conventional SPLs to more dynamic approaches able to handle runtime concerns. For more than twenty years, companies have launched successful product lines to build software products faster and cheaper with higher quality. Several experiences in the product line hall of fame (Van der Linden et al., 2007) have demonstrated the benefits of the adoption of a product line approach.

Today, software-intensive embedded system families demand highly configurable and adaptable mechanisms many of them managed at runtime. Much of these systems are designed using software architectures that include runtime mechanisms for adaptation purposes, but stringent quality and business requirements drive the motivation for dynamic extensions, runtime reconfiguration, optimized performance, and autonomous behavior, amongst others. For instance, mobile and service-based applications have grown exponentially, affecting a large number of users that use software and devices that demand runtime post-configuration facilities. Customizable and context-aware services drive the current trend of cloud and SOA-based systems for selecting the most suitable service located anywhere, and context-ware properties play a role to adjust the service to new context conditions. Furthermore, binding services dynamically, demands rebinding and multiple binding capabilities by introducing variants for the dynamic selection of services, as this feature is not supported by conventional product lines. Combined approaches mixing SPL and SOA (Gomaa

* Corresponding author. Tel.: +34 91 4888119.
E-mail address: rafael.capilla@urjc.es (R. Capilla).

0164-1212/$ – see front matter © 2014 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.12.038

and Hashimoto, 2011) deal with the challenges to incorporate runtime mechanisms addressing context-awareness properties of services where family members need to evolve after deployment. In addition, the variety of self-adaptive, self-management, and autonomous systems (e.g., robots, unmanned vehicles) also require autonomous behavior and automatic decision-making, often based on a set of system options that can be activated and deactivated according to varying context conditions or user preferences.

During the past 15 years, traditional SPL approaches (Bosch, 2000; Clements and Northrop, 2001; Pohl et al., 2005) have successfully addressed the development of system families from a common architecture, maximizing reuse, and exploiting the variability of systems to produce cheaper and higher-quality products in less time. However, in a changing world, the increasing need for adaptive software demands autonomous behavior and self-management properties bringing new challenges for dynamic adaptation in system families. As variability becomes more dynamic, there is a clear to move to recent development approaches like Dynamic Software Product Lines – DSPLs (Hallsteinsen et al., 2008; Hinchey et al., 2012; Bencomo et al., 2012) as an emerging paradigm to handle variability at runtime and at any time.

1.1. Target audience and summary of contributions

The target audience of this work are researchers in the software architecture and product line engineering fields and also those professional SPL designers and engineers who need to migrate from a conventional SPL to a DSPL or develop software products that demand dynamic variability mechanisms from scratch. Therefore, we elaborate a list of major challenges and solutions that cover the full spectrum of a DSPL. Consequently, we provide our observations from the trenches from research and industry about the current state-of-the-art of the DSPL technologies, and more specifically the need for runtime variability mechanisms. As we have observed that there are partial solutions that work in isolation, one of the key contributions of this work is a framework that encompasses the following:

1. the required DSPL properties consisting of runtime variability support, multiple and dynamic binding times of products, and a way to model context properties,
2. the organizational changes a DSPL should have compared to conventional SPLs, and
3. the suggested solutions for each DSPL challenge we describe as relevant pieces required for launching a DSPL.

These relevant pieces or suggested solutions go from runtime variability mechanisms for context adaptation purposes to optimization mechanisms aimed to provide the best or the optimal solution in a given context.

The remainder of this paper as follows. Section 2 provides the related work regarding DSPLs. In Section 3 we characterize DSPL elements and processes in a framework. In Section 4 we outline the role and challenges of runtime variability mechanisms and other related issues of DSPL research that we will address in the remainder of the paper. Section 5 describes the set of technical solutions necessary to implement and use dynamic variability. In Section 6 we outline examples of use of the proposed solutions in various application domains. Section 7 summarizes the discussion the major results derived from this research and Section 8 draws the conclusions and future work.

2. Related work

The current limitations of today's SPL models rely on its inability to change the structural variability at runtime, provide the dynamic selection of variants, or handle the activation and deactivation of system features dynamically and/or autonomously. Because the development of runtime reconfigurable assets is still innovative and not fully investigated in the SPL area (Gomaa and Hussein, 2004; Lee and Kang, 2006; Schmid and Kröher, 2009), there is an important need to support the dynamic properties of systems and post-deployment capabilities as a new promising research and development area using DSPL models.

At present, emerging efforts that suggest the use of DSPL-based models focus on the implementation of runtime variability mechanisms and domain-specific languages for reconfiguring software system options (e.g., Cetina et al., 2009a uses PervML, a domain-specific language for pervasive systems for reconfiguring a smart home using a DSPL), in particular for specific application domains such as smart and autonomic homes, robots, mobile software, or service-based systems, amongst others. For instance, Cetina et al. (2010) address the challenges of triggering runtime reconfigurations and understanding the effects of such reconfiguration when prototyping a DSPL. The authors describe the case of a Smart Hotel and how Dynamic Software Product Lines can assist a system to determine the steps during reconfiguration operations at runtime, such as the activation and deactivation of system features dynamically and based on varying context conditions. Hallsteinsen et al. (2008) discuss the role of DSPLs in emerging domains, such as ubiquitous computing, robotics, and life-support services, amongst others. Because it is impossible to predict all the expected variability in a product line, Dynamic Software Product Lines should be able to produce adaptable software where runtime variations can be managed in a controlled manner.

Many software applications exhibit random behavior at runtime, and the service computing area is another promising research field where DSPLs have a niche to exploit runtime variability mechanisms for the dynamic selection of services, often based on changing QoS (Quality-of-Service) properties. Several authors (Hallsteinsen et al., 2009; Gomaa and Hashimoto, 2011; Istoan et al., 2009) highlight the importance of using a DSPL for modeling and implementing service-based systems by encoding dynamic variability as a decision model for the selection of the services at the latest binding. However, as the computational complexity during variant selection may be a drawback for performance in certain systems that have strong real-time requirements, a DSPL should be able to handle the necessary adaptations and current reconfiguration tasks after the original deployment. For instance, the Service Component Architecture (SCA) architecture style, which reconciles SOA and Component-Based Software Engineering (CBSE), and SCA platforms can be used to assemble assets dynamically in a running system (Parra et al., 2009). Bencomo et al. (2010) address the research theme "*when to adapt and how to adapt?*" in order to meet the demand of postponement of decisions on software adaptations required by dynamic environments and users.

Dynamic product lines are aimed also at binding features dynamically and to supporting multiple binding times and one or more variability mechanisms (Abbas et al., 2011). A DSPL nicely integrates the adaptation of assets and products dynamically in changing contexts, which helps products to evolve autonomously when the environment changes and provides self-adaptive and optimized reconfiguration. Additionally, a DSPL may exploit knowledge and context profiling as a learning capability for autonomic product evolution by enhancing self-adaptation (Abbas et al., 2011). However, in the era of post-deployment evolution (Malek et al., 2012), where embedded systems can change and be deployed several times, DSPLs offer a solution for software-intensive and embedded system families as they provide dynamic variability mechanisms. There are many embedded systems and application domains where runtime variability can play a key role in order to support the "autonomic" condition (Kephart and Chess, 2003) and

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

manage system capabilities autonomously and dynamically. For instance, sensor network system families that use context information can be represented using variability models to determine and manage those features that vary at runtime (Gámez et al., 2011).

2.1. Runtime variability techniques

The need to provide dynamic capabilities to system families pushes the demand for dynamic variability solutions where the many of the configurable options can be handled at runtime. At present, there are few proposals in support of runtime variability models but none of them cover the full spectrum of a complete solution to be used by a DSPL model. One of these solutions relies on the Common Variability Language (CVL) as an attempt for modeling runtime variability transformations, as it allows different types of substitutions to re-configure new versions of base models (Haugen et al., 2008). Cetina et al. (2009a) evaluate three alternative strategies for modeling runtime transformations using CVL. Synthesis and modification operations are used to generate a new base-model configuration or to compute the differences between two configurations and resolve the dynamic transformations in the variability model. Adding the necessary extensions to support dynamic variability in current feature models can be done using a meta-variability model, such as suggested by Helleboogh et al. (2009), where high-level constructs enable the addition and removal of variants on-the-fly to the base feature model. At present, most dynamic variability techniques deal with reconfiguration tasks for activating and deactivating features while the modifications to the structural variability at runtime is marginally treated.

When the existing variability model is modified, the structural and operational dependencies between variants and variation points must be also revisited. Traditional SPL practices use off-line tools to automate feature model checking (Batory et al., 2006) of hundreds of "*require*" and "*exclude*" dependencies between features. Lee and Kang's work (2004) addresses the need to check the runtime dependencies (i.e., operational dependencies) that drive the operational mode of a system while running (e.g., *what happens to feature X if we activate feature Y*?). The dynamic changes applied to the structural variability goes one step beyond the simple activation and deactivation of system features, as we need runtime checking facilities for both the changes in the states and the dependencies between features. Currently, there are several approaches but few tools supporting the automated analysis and checking of feature models for SPLs. Most of these approaches are described by Benavides et al. (2010), which nicely summarizes those scenarios where feature models must be checked if modified (e.g., valid product, valid partial configuration, anomaly detection like redundancies in the feature model, optimization, etc.). The changes performed over the features and dependencies often lead to multiple solutions, but in some cases we need an optimal solution as result of a reconfiguration process. Cetina et al. (2009a,b,c,d) propose a sequential procedure to collect and analyze the preferences of one or more users in a smart home environment. This information is used to produce an optimal product configuration which satisfies most of user's preferences. The FAMA toolset (http://www.isa.us.es/fama/) is used in this work to perform the optimization task. User preferences are collected in terms of features that are weighted, so the tool can compute and maximize the best configuration of features that satisfies the majority of the user's preferences. Also, many real-time systems that need a new system configuration may require optimization before deployment (Cuadrado et al., 2012) and optimization must be performed to satisfy stringent quality requirements. Sawyer et al. (2012) suggest representing the dynamic behavior of self-adaptive systems in a DSPL using goal models together with feature models to describe soft constraints for different levels of satisfaction. The optimization in this case is solved as a constraint-programming problem.

Another related issue to runtime variability mechanism is the capability of the DSPL to handle multiple binding times and re-binding of features at any time, but few works deal specifically with the property of binding time. An early work (Fritsch et al., 2002) describes a list of possible binding times for conventional SPLs. The authors mention the possibility for variants to be bound at three different runtime modes: at first startup, every startup, and dynamically. A more modern reference (Lee and Kang, 2006) discusses feature binding analysis as an activity to define binding units for each functional system unit that changes its behavior at runtime and how this technique can be used to manage variation points of dynamically reconfigurable products. Context analysis plays also a key role to model the decisions when to start a reconfiguration and to identify contextual parameters in the product line. The authors (Lee and Kang, 2006) raise also an important concern regarding the possible states of the binding units (e.g., Start, Suspend, Reconfiguring, Suspending, etc.), more specifically they describe a lifecycle with different possible states for the reconfiguration strategy applied. Therefore, several binding times are needed to switch between different operational modes or state of the variants.

2.2. Contextual Information

Another relevant aspect that influences DSPL techniques is the context information that is used by many context-aware, ubiquitous, and self-adaptive systems to manage context properties dynamically. Hence, such context properties can be also modeled as variants (a.k.a. contextual variability) that are managed in the feature model with other non-context features.

Like many self-adaptive systems[1] that exploit context information to adjust their behavior dynamically, recent research (Abbas et al., 2011) predicts an increasing development and use of DSPLs that want to share run-time knowledge and driven by the need of software products to improve quality faster and self-optimize in reaction to varying conditions. As DSPL are an emerging paradigm for the software industry, well-known mechanisms from the self-adaptation field can be incorporated to provide dynamic capabilities to conventional SPLs. One of these mechanisms able to support the autonomic condition of DSPL products is the Monitor–Analyze–Plan–Execute (MAPE) loop, which is used to reconfigure products of self-adaptive software dynamically. In addition, context profiling in conjunction with the MAPE loop is used to map contexts to variants, and offer then a better selection of the possible alternatives and according to the product line goals. For instance, sharing on-line knowledge at runtime in the context of a DSPL that builds Wireless Sensor Network (WSN) products can be used to self-optimize a particular WSN when and new node is added and reconfigure the network dynamically. Because context features become more and more relevant for DSPL, context analysis (Ali et al., 2009) is a technique that becomes relevant to identify and model the context itself through appropriate context features meaningful for DSPL products.

Therefore, the notion of *context variability*, introduced by Hartmann and Trew (2008), turns out to be key to identify those context features, as they suggest use classifiers (super-types enable

[1] In this work we do not focus on runtime architectures for self-*systems or reflective mechanisms commonly used by adaptive middleware (Bencomo et al., 2008). Even if context variability shares some commonalities with this technology, we will focus only on those aspects of interest for implementing and managing runtime variability mechanisms in a DSPL.

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

also a classification system for variants) to model and capture the common and variable aspects of the context.

However, our DSPL-enabled super-types models do not limit context features to only location properties, as we use this in a more general sense to organize groups of related system features. Also, we do not use a context variability model separate from the feature model like in Hartmann and Trew (2008). Rather, we annotate the variants with the super-types to specify a superseding classification. DSPLs are often considered to be an engineering approach for realizing adaptive systems based on the MAPE-K loop (Hinchey et al., 2012). In this way, Gámez et al. (2011) describe a feature model for WSNs under a SPL context, which uses also a MAPE-K (i.e., the MAPE loop with knowledge of the environment) adaptation mechanism. The family of middleware (FamiWare) for WSNs developed under the SPL aimed to support autonomic and dynamic reconfiguration of nodes for an optimal network functioning suggests the use of context variability to model context features and plans as two distinct branches in the feature model separated from those other features whose values are known at design time. From the analysis of recent experiences that suggest a DSPL and their related techniques, we informally observed that new elements, characteristics, and activities regarding the reconfiguration and late binding, context-awareness, dynamic selection, and runtime changes among others, highlight the difference versus conventional SPLs.

From our experience in research and industry and from an exhaustive review of current works in the topics of DSPLs, dynamic variability, self-adaptation and reconfiguration, we address a set of challenges relevant for DSPLs and how these connect to specific solutions. Consequently, in order to guide software engineers to launch a DSPL, we revisit the traditional SPL dual-lifecycle development process and reflect on it new activities concerning the dynamic nature of DSPLs, such as we characterize in next section.

3. A framework for DSPL characterization

The current state of DPSL technology suffers a lack of agreement about what a DSPL is, what activities it should encompass, and what degree of automation runtime variability mechanisms it should provide. From our recent work and observations from research and industry we have observed a set of properties implemented in products supporting dynamic variability and self-adaptation mechanisms. Hence, are contribution is based on the following points: (i) describe the properties required for launching a DSPL, (ii) provide a framework for interconnecting these desired properties with solutions using a DSPL, (iii) describe the organizational challenges that make the difference with conventional SPLs, and (iv) provide runtime variability solutions as central mechanism used by a DSPL. In this work we detail the properties we observed and we use these as a framework to identify the key challenges/technologies a DSPL should posses.

DSPL Property 1. Runtime variability support and management: A DSPL must support the activation and deactivation of features and changes in the structural variability that can be managed at runtime. Runtime reconfiguration is another property dynamic variability should encompass.
Related work: Helleboogh et al. (2009) and Bosch and Capilla (2012) suggest different ways to modify variants dynamically, while other authors dynamically (Lee and Kang, 2004; Cetina et al., 2009b,c,d) focus on the activation/deactivation of systems feature at runtime. In addition, reconfiguration strategies must be defined to select the means by which the reconfiguration is performed and according to different scenarios (Lee and Muthig, 2008).
Example: Automated transportation systems or Wireless Sensor Networks are application domains where new system features can be changed or reconfigured dynamically, which has a clear impact in the structural variability and state for variability models to handle such changes at runtime.

DSPL Property 2. Multiple and dynamic binding: Multiple binding times versus one single binding time is required, as when the software adapts its system properties to a new context, features can be bound several times and at different binding times (e.g., from deployment to runtime). Post-deployment capabilities might also need late binding.
Related work: Lee and Kang (2006) state the need to rebind to new services dynamically. In addition, Bosch and Capilla (2012) describe the relationships and transitions between multiple binding times under a DSPL.
Example: Systems that change their status dynamically (e.g., critical systems) and those that mach to different services at runtime (e.g., services bound dynamically) demand more than one binding time (e.g., from configuration to execution time).

DSPL Property 3. Context-awareness and self-adaptation for autonomic behavior: Dynamic SPLs must handle context-aware properties that are used as input data to change the values of system variants dynamically and/or to select new system options depending on the conditions of the environment.
The information that often changes in a DSPL is used to make new decisions or to select different system options on-the-fly. Runtime decisions often rely on context information, quality levels demanded by the running software, and user preferences among others. This context knowledge should be used in conjunction with adaptation mechanisms, context profiles, and data gathered during the execution of the system in order to select the best choices.
Related work: Cetina et al. (2008) classify different architectural techniques for DSPLs based according to their adaptation mechanism. The connection between the decision models and the reconfiguration activities is highlighted using a mixed approach. Self-*systems (Garlan et al., 2004; Raibulet, 2008; Georgas et al., 2009; Abbas et al., 2010; Cetina et al., 2009d; Raheja et al, 2009) commonly use adaptation mechanisms, but these should be tailored to support runtime variability. In addition, Hartmann and Trew (2008) and Abbas et al. (2011) highlight the importance of context variability to model context-aware properties that can be used in automatic decision making to modify the behavior of systems at runtime.
Example: Self-adaptive systems or smart home systems are clear example where context properties drive the behavior of the systems dynamically. Also, autonomous robots exploit context information to adapt its behavior to varying conditions. Other systems like mobile devices exploit location properties or user preferences to configure the device to different user needs.

3.1. DSPL organizational changes

In addition of the desired properties a DSPL should have, adding support for runtime changes may lead to some organizational changes in the DSPL versus conventional software product lines. From our observation we believe these organizational changes will modify and/or extend the traditional SPL activities in order to handle runtime activities, such as: runtime reconfiguration, runtime model checking, or transitioning between multiple binding times where re-deployment tasks are sometimes needed. From our perspective, the main challenges and activities that change from conventional SPL that affect the organizational issues of a DSPL are the following:

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

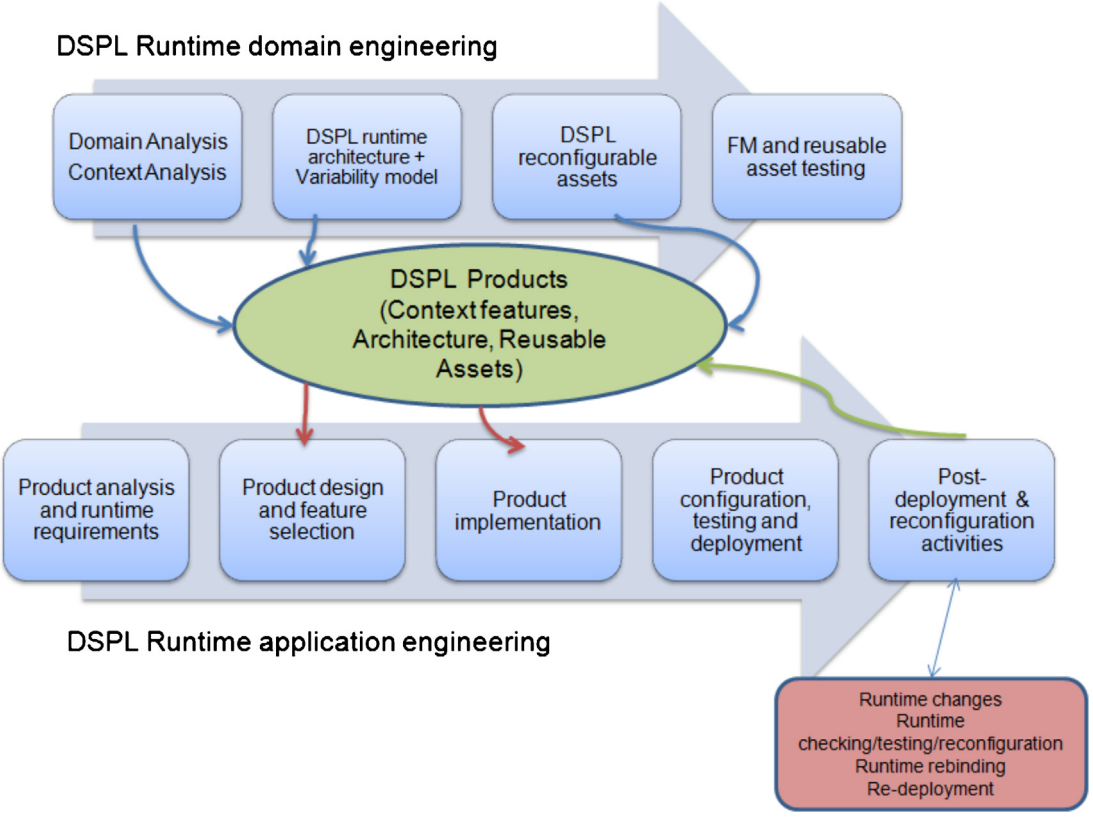

Fig. 1. Dual software development life-cycle to support DSPL activities.

- *Domain scoping/analysis*: Traditional domain analysis should include now modeling the context properties that are used as context properties by many systems that exploit environmental characteristics at runtime. Hence, while domain scoping still identifies the scope of the product family member, context analysis extends the domain scoping activity to identify those context features that are relevant for DSPL products.
- *Architecting the DSPL*: The traditional product line architecture (PLA) must include specific modules for supporting runtime reconfigurations and dynamic binding in order to automate the decision making process that may happen at execution time, and those runtime checking processes that ensure the consistency and validity of the modified feature model.
- *Product derivation*: The derivation of SPL products has many challenges when giving automated support (Elsner et al., 2009; Elsner, 2012). Some authors suggest semi-automatic derivation procedures using non-functional properties (Siegmund et al., 2008) or derivation techniques like model-driven engineering (Pleuss et al., 2010) in order to facilitate the generation of products. Such techniques may change the traditional application engineering phase as they entangle features, code, and automatic procedures to build the selected product more automatically. Semi-automatic derivation techniques are fully compatible with any DSPL derivation activity, as the DSPL products may demand the automation of redeployment and reconfiguration activities. Moreover, DSPL derivation and generation activities provide as much automation as possible and go a step beyond in order to enable the redeployment at the latest time, for which runtime checking of a reconfigured feature model results key.
- *Product reconfiguration*: The dynamic nature of DSPL products demand checking and reconfiguration activities that can be performed in runtime mode, which highlights the difference versus traditional SPL products.

Consequently, and based on the challenges described before, the organization of a DSPL varies slightly from the traditional SPL dual-lifecycle (e.g., Van der Linden et al., 2007) including new activities supporting the runtime nature of DSPL assets and products, such as we shown in Fig. 1.

The DSPL dual lifecycle of Fig. 2 extends the one traditionally used by current product lines with the following modifications. In the *DSPL runtime domain engineering* phase, we the traditional *domain analysis* activity is extended with a *context analysis* task, in which we identify not only the asset requirements and family member features but also those context-aware properties relevant for our DSPL products. Therefore, context variability and context-specific features are identified during the initial modeling phase. Also, multiple binding times and rebinding options of features can be defined as properties in the feature model, as well as the valid states for those context features. The runtime software architecture should include all those mechanisms able to handle the dynamic changes of DSPL assets and products, the activation and deactivation of features dynamically and the possible modification of the structural variability at runtime. This architecture should lead to the implementation of those reconfigurable and non-reconfigurable assets that will be tested at the end of the line, as well as the validity of the feature model. All these artifacts will populate the DSPL repository for the subsequent phase. During the *DSPL runtime application engineering* phase, runtime and non-runtime requirements drive the construction of DSPL family members. The runtime DSPL architecture will be customized alongside the feature model to produce the DSPL products that will exhibit the reconfigurable characteristics required. Once products are tested, configured, and deployed, if a new software configuration (e.g., due to the activation of features, changes in the quality level of the system, or the selection to new services) is required at runtime, our DSPL dual lifecycle support encompasses post-deployment and

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

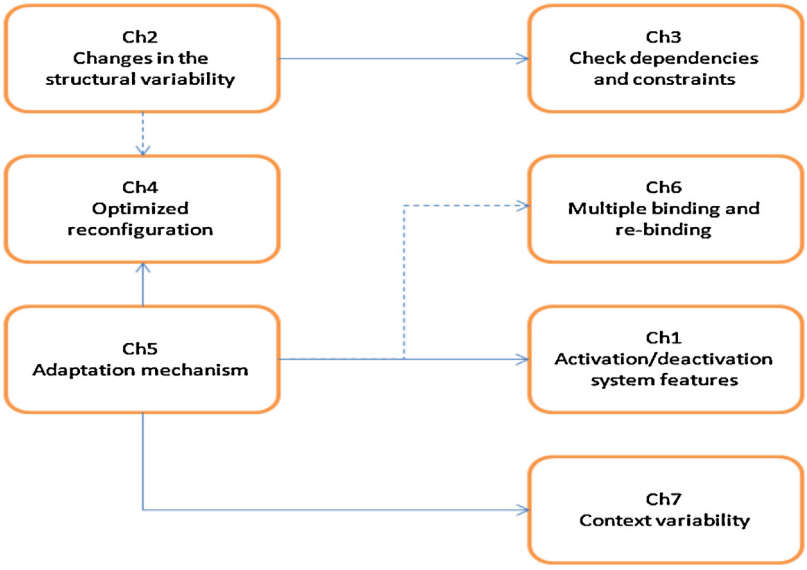

Fig. 2. Relationships between the challenges for runtime variability adoption.

reconfiguration task for handling all these changes. In some cases, certain runtime checking, testing and monitoring, and reconfiguration operations have to be made after the product is re-redeployed before it goes to its normal operational mode. Feedback from this task populates again the DSPL repository. As a result, these new activities extend the traditional application engineering phase with new post-reconfiguration tasks that are performed dynamically. Once we have described the major methodological changes[2] using a DSPL, in the following sections we focus on those runtime variability mechanisms that constitute the core of Dynamic Software Product Lines.

4. The role and challenges of runtime variability

From the characterization of a DSPL described in the previous section we can figure out that runtime variability is one of the basic mechanisms any DSPL must provide. The role that runtime variability plays for launching and evolving DSPL products becomes relevant when system variants change dynamically over time. Implementing such mechanisms is still challenging and difficult, but necessary to manage the diversity of real-time scenarios demanded by systems using context properties. At present, few authors deal with the challenge of runtime variability. Early approaches focused on traditional product lines; for example Goedicke et al. (2004) proposes a pattern language that provides domain-specific runtime variation point management facilities. Runtime variation points are encapsulated for dynamically configuring the patterns used in the construction of a Multimedia Home Platform (MHP). Lee and Muthig (2006) state "*the increasing demands for the postponement of on product variations to runtime to provide an operating-context relevant service*" (see also Gomaa and Saleh, 2006). Hence, features should be dynamically reconfigured and bound at runtime. Also, the COVAMOF software variability assessment method (COSVAM – Deelstra et al., 2009) mentions that runtime is the latest time variants should be bound, and this binding time is preferred over variability handled at predeployment when the system has to configure itself at runtime.

One step beyond using a DSPL combines static and dynamic binding of features (Rosenmüller et al., 2011). The authors attempt to achieve a highly customizable SPL supporting self-configuration and using coarse-grained components to generate a DSPL that provides the runtime environment required by a particular scenario. The authors use an adaptation mechanism that reduces the computational effort for an optional configuration at runtime. For each DSPL instance, features can be activated and deactivated temporarily but they can also be added or removed from the instance (i.e., a feature no longer used is deleted and unloaded). This approach nicely merges static and dynamic techniques under a product line model. Because runtime variability is central to DSPLs, we extend the ideas described in Capilla and Bosch (2011) and Bosch and Capilla (2012) mainly referred to the modification of the structural variability at runtime and runtime binding, as we provide a more complete and concise list of challenges and needs that any runtime variability mechanism should support. Not all the following mechanisms must be supported concurrently by a DSPL, as many of them can be mixed to enhance each particular runtime variability mechanism. We also cite some related work for each challenge to justify its importance for DSPLs.

Challenge 1: Activation/deactivation of system features. The activation and deactivation of systems options (i.e., features) is one of the simplest forms of managing runtime variations, as a user or the system itself can autonomously switch on/off features dynamically. In some cases the system may need to be reconfigured if critical system features are changed, while in other cases the status mode of the system remains the same.
Example: A robot or an autonomous system may need to activate or deactivate certain system features to adapt its behavior to a varying environment conditions. Therefore, dynamic variability must support the activation/deactivation of system features.

Challenge 2: Automatic changes in the structural variability. As opposed to static SPLs, where variability models remain stable after design time, dynamic SPLs can modify their structural variability at runtime. Changing variants dynamically and automatically is technically feasible, but changing variation points at runtime requires some form of human intervention. Adding and

[2] Some authors consider inadequate the traditional dicothomy of early software product line methods, and they feel "*application engineering considered harmful*". In order to enable software mass customization they replace the application engineering phase by a software product line configurator, aimed to automate and instantiate product development (http://www.biglever.com/extras/BigLeverNewMethods.pdf).

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

removing variants has direct implications for the constraints and dependencies between the variants. In this research we will limit the changes to the structural variability only to single variants.
Example: The need to include a new functionality at runtime will require an automatic identification and reconfiguration of the new system feature which should be integrated dynamically in the feature model in order to reflect the change in the design.

Challenge 3: Runtime dependencies and constraints checking. Changes in the feature model performed at runtime may require, in certain cases, some kind of runtime checking to detect new incompatible product configurations, as variants that change may also introduce new changes in the dependencies and constraint rules that should be modified at runtime. Also, the validity of the feature model must be checked to avoid unfeasible products when, for instance, a new set of features is selected (Benavides et al., 2010).
Example: Any new feature added, removed, or changed in a feature model require to check at runtime the existing or new constraints in order to provide consistency and avoid incompatible configurations that may lead to a system fail.

Challenge 4: Dynamic and optimized reconfiguration. Dynamic reconfiguration is a key property for DSPLs where software products need to be reconfigured at runtime. As automated checking of feature models is often done in "off-line" mode, there is a need to provide optimized reconfiguration mechanisms that can be executed at runtime when variants change.
Example: Systems that demand automatic redeployment when a runtime reconfiguration happens often demand an optimized reconfiguration which has to be computed before the new deployment takes place.

Challenge 5: Adaptation mechanism. Many autonomous systems require an adaptation layer to adjust the system's behavior to the new conditions. The activation and change in a system feature often drive this behavior to adapt the system to new quality conditions or to a new environment.
Example: Robots and other autonomous systems are examples where system variants change their values according to varying conditions, and hence, an adaptation mechanism is need in the DSPL architecture to manage such kind of changes. Other systems exploit location and quality properties to modify its behavior (e.g., send tailored information to the user of a mobile phone who changes its location).

Challenge 6: Multiple binding and re-binding. DSPL products may require multiple binding times. As conventional product lines are designed to support one single binding time in most cases, the dynamicity of DSPL models must offer multiple binding times in order to switch between different operational modes.
Example: Systems that need to rebind to a different option or service dynamically will require to support more than one binding time (e.g., from configuration time to runtime). For instance, a critical system that goes to an automatic maintenance mode will change the values of some system properties until it passes all the tests and goes gain to the runtime mode, where the values of their variants change to the normal operational values.

Challenge 7: Context variability. Traditional feature models focus on system features and not on context features used to detect and manage context conditions. Therefore, the notion of context variability focuses on the identification and modeling on those context properties that are specifically designed to handle the diversity of the context that influences the dynamic behavior of systems.
Example: Most smart and autonomous systems that exploit context information to adapt their behavior at runtime support context-aware properties which are often modeled as context features (a.k.a. context variability).

In order to link the proposed framework characterizing a DSPL with the challenges that pertain to runtime variability mechanisms, we provide in next section the relationships to the solutions of these challenges in a DSPL context.

5. A runtime variability model

Based on the properties and challenges described in the previous sections, DSPL designers should decide what degree of automation they need in order to support the runtime concerns of the DSPL. For instance, product line designers or variability managers may need an automatic mechanism to activate and deactivate certain software features in order to handle autonomous reconfiguration capabilities, while it might not be necessary to support the inclusion of new system features (i.e., new functionality) automatically affecting the structural variability. Another scenario may refer to the possibility for providing on-line checking of the variability model when new requirements demand new system constraints that can limit the configuration of current products. As not all runtime variability mechanisms characterizing a DSPL might be necessary at the same time, we describe in this section the basic elements we believe are central for any runtime variability solution and for DSPLs as well. These elements encompass those aspects of runtime variability (RunVar) representation and management and those specific to context information, as relevant features used by many smart system families that managed such properties autonomously.

RunVar representation: Models, formalisms, and languages are needed to represent the dynamic characteristics of the variable options versus static approaches. Several alternatives exist to describe a feature model and the constraints relating the variants. For instance, propositional logic (CNF), constraint programming (CSP), ontologies, and conceptual graphs are formal solutions by which we can describe a variability model. Specific for DSPLs, we need to define the context features in our feature model, which of these may vary dynamically, and which can be bound at runtime. Hence, some extensions over the proposed notations may support the changes in the structural variability.
RunVar management: The modification of the structural variability at runtime requires managing not only the features that can be switched on/off dynamically but also the dependencies between features. Hence, when a new feature is added, removed, or changed dynamically we should support how these dependencies and constraints can be checked at runtime (e.g., during the installation of new system functionality at runtime, a system update without restart, or a hot-swap replacement) before the system reconfigures itself. Variability management operations must be extended from purely static approaches to runtime, and the changes must be reflected and visualized properly at the design level. Some key techniques useful for DSPL variability management are the following:

(a) *RunVar checking* is used to automate feature model checking, and testing is necessary to prove the viability of the current variability model and avoid inconsistencies before the system uses the new configuration. Off-line feature checking models are in common use, but on-line feature checking is still needed, in particular to test the incompatibility between features or a particular product configuration (e.g., during a software

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

Table 1
Properties, challenges, and solutions for a DSPL runtime variability model.

DSPL properties/challenges (Ch)	Runtime variability support	Multiple and dynamic binding	Context-aware and self-adaptation properties for autonomic decision making
Ch1: Activation/deactivation of system features	Context-aware capabilities Section 5.4	N/A	Context-aware capabilities Section 5.4
Ch2: Automatic changes in the structural variability	RunVar representation & management Section 5.1	N/A	N/A
Ch3: Runtime dependencies and constraints checking	RunVar checking Section 5.2	N/A	N/A
Ch4: Dynamic and optimized reconfiguration	RunVar reconfiguration Section 5.5	N/A	N/A
Ch5: Adaptation mechanism	N/A	N/A	Context-aware capabilities Section 5.4
Ch6: Multiple binding and re-binding	RunVar multiple binding Section 5.3	Multiple binding Section 5.3	N/A
Ch7: Context variability	RunVar representation Context-aware capabilities Section 5.4	N/A	Context-aware capabilities Section 5.4

update) using the constraints and dependencies between features.

(b) *RunVar reconfiguration*: As some systems demand an optimized configuration when features change and before the system is re-deployed, optimized configuration techniques of the new feature model must be provided and avoid incompatible configuration when the systems reconfigures itself at runtime.

(c) *RunVar multiple binding*: is key for DSPLs, as dynamic variability opens a window to support multiple binding times, where products may bind and re-bind to system options several times when system variants bind features between different system operational states.

Context-aware capabilities: Many runtime capabilities depend on context or quality information gathered by sensors and these sensing capabilities must be linked to features in order to identify which of the variable options use such information. Consequently, context variability play a key role in identifying those context features in the feature model, as the designer will know which set of features may vary dynamically in the DSPL. The identification of context-aware features and quality properties that often drive the modification of these context properties are key when context features change.

In Table 1 we connect the challenges and the solutions of DSPL runtime variability mechanisms using the framework described in Section 3. In the table, we provide the solution of the runtime variability model for each of the aforementioned challenges and how each challenge meets each part of the DSPL characterization. We also annotate the section where each solution is described.

As a DSPL can implement one or several of the technical solutions we provide in order to address more than one challenge. We do not perceive any conflicts between such solutions, as each one implements a different challenge. However, we found the following relationships between the challenges when more than one co-exists, such as Fig. 2 shows. The solid arrows means than in all or the majority of the cases a challenge needs another one. The dotted arrow means that one challenge does not necessarily need another challenge. For instance, challenge Ch2 needs challenge Ch3 when the structural variability is modified. In the case of the relationship between Ch5 and Ch7, the adaptation mechanism often uses context features for adapting the system at runtime.

In the following sections we will address five solutions which address the aforementioned challenges aimed at supporting runtime variability under the proposed DSPL framework. These solutions constitute proven ways to enable the technical aspects underpinning the notion of a DSPL but they should not be seen in isolation as any specific DSPL may implement one or several solutions at a given time. However, the proposed solutions are not unique and other approaches could be used to enable runtime variability.

5.1. Characterization of runtime variability

Since the first representation of a feature model (Kang et al., 1990) and other extensions proposed to enhance the description system features, runtime variability requires an additional characterization of current feature models. The need for runtime adaptation of embedded system families that demand new hardware and extensible software features motivates the need for open variability mechanisms. In this section we address Challenges 2 and 7 and we describe our solution for representing runtime variability.

Solution candidate: Our solution for Challenges 2 and 7 focuses on the automation of changes in the structural variability using the notion of super-types, such as described in our previous work (Órtiz et al., 2012; Bosch and Capilla, 2012). We focus on the structural modification of the variability rather than on those changes that modify the status of a feature (e.g., activation and deactivation). A super-type (st) extends the basic types (T) commonly used by system features. These super-types rely on the notion of context variability (Froschauer et al., 2009) as they act as classifiers in order to define a superseding classification system for features. Super-types enable software engineers to define categories for grouping variants that share a common functionality. We use super-types to automate the modification of variants in the feature model at runtime. In this work we do not change the variability defined by the variation points (they can also support the notion of super-types), as this requires some human intervention in most cases, which is harder to automate.

In our model, we define a super-type (denoted by st) as:st: a list of strings (names) defined by the user and aimed to categorize and classify a system feature.

Therefore, a variant (V) in our model is a set composed by:

- A list of allowed values belonging to the following basic types: integer, numerical, string, and Boolean. We can have ranges of numerical an integer values (e.g., [1. . .5]), lists of values (e.g., 1, 2, 3, 5.5, 8.7), and lists of strings (e.g., DVD, USB, CD).
- A type, denoted by T, assigned to the values (i.e., Boolean, string, numerical, integer).

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

- A list of compatible super-types (st). Variants may belong to one or several compatible super-types.

Automatic changes in system variants rely on the comparison of compatible super-types of related features. Therefore, we define a compatibility Boolean function to determine if a new variant can be added, removed, or changed in the feature model.

This compatibility function, we call *Comp()*, is defined as follows:

Comp(st_A, st_B) = 1, if st_A is equal to st_B; or if st_A and st_B belong to a list of compatible super-types
Comp(st_A, st_B) = 0, otherwise

Consequently, super-types organize system variants under common or compatible categories defined by the software designer, except for the root node. The automation of the comparison process between super-types can be easily done by reading a configuration file at runtime or performing queries on a database. The evolution scenarios we define to support the structural changes in the feature model using the super-types are as follows:

Adding a variant implies that we need to know the place where the new variant will be included in the feature model. To perform this operation, we check the compatibility of the super-type of the new variant with the super-types defined for its potential parents and siblings, as only variants with equal or compatible super-types with can be added.
Removing a variant means that the variant will be dropped from the feature tree and removed from any logical formula using this variant (e.g., from a variation point). Software variability management mechanisms should check how the dependency and constraint rules using that variant will be modified accordingly.
Changing a variant in its simplest form leads to change only their values or state, while the structural change refers to move the variant to a different location in the feature model. We implement this operation as a removal task followed by an addition of the variant we want to change to that place, but before performing this operation we check first the compatibility between the super-types.

Table 2
Example of super-types assigned to WSN features.

Feature	Types (T)	List of compatible super-types (st)
Temperature	Range of numerical values	Ambient
Smoke	Boolean	Ambient and security
Humidity	List of integer values	Ambient
Light	Boolean	Ambient
Communication type	List of integer values	Communication
Acoustic alarm	Boolean	Security

Example: In this example we suggest a simple scenario where a new variant is added to the feature model dynamically. We have chosen the variability of a Wireless Sensor Network (WSN) composed by sensors and actuators that react at runtime to different context conditions. A WSN can be modeled as a DSPL where various types of WSNs with different features can be engineered. A subset of the feature model of the WSN product family encompasses three different functional units that manage sensors and actuators for the following functionality: *communication type and node identity information, measurement of context information*, and *actuators*. Each functional unit can be organized around variation points that group sets of related features or variants. For each variant we define the list of compatible super-types, such as shown in Table 2.

In the scenario shown in Fig. 3, we add new hardware to detect pollen (i.e., a pollen sensor), which is useful for warning people with allergies. The new variant called "Pollen" should be inserted in the feature tree under the "Measurement" variation point, as the new feature has the same super-type (i.e., Ambient) as his new parent and siblings. We have to remark that the new variant "Pollen" has two super-types because its functionality is concerned

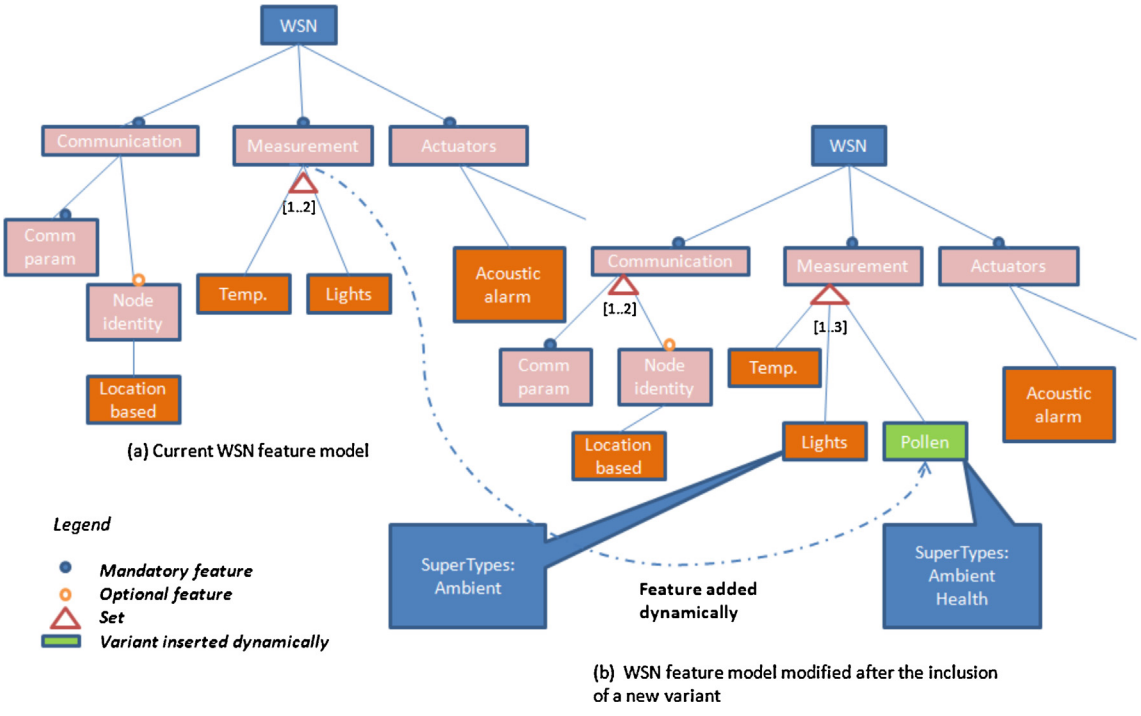

Fig. 3. The structural variability of a WSN feature model modified dynamically with the inclusion of a new variant.

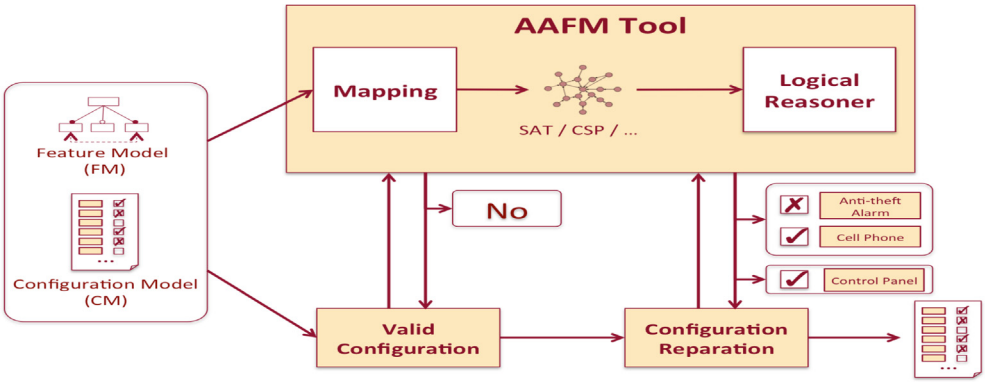

Fig. 4. Reconfiguration process assisted by AAFM tools.

with "Ambient" measures but also with "Health", as this super-type can be used to incorporate new functions related with the health of users and can use this to connect to, for instance, a medical service.

An automatic procedure can now insert the new variant dynamically in the correct place when the new hardware is added, and the new functionality is ready to use. The left side of Fig. 4 shows a subset of the feature model of the WSN product line, which comprises the following variation points: *communication, measures, and actuation*. In the right side of Fig. 2 we show the result when the "Pollen" feature is added to the feature model, as it shares a common super-type and also because the other functional units do not have compatible super-types with the new feature. In order to automate the inclusion of variants we suggest the following algorithm, but similar ones can be defined for removal and replacement operations.

The following algorithm describes the steps for inserting a variant in the feature tree at runtime, but similar algorithms can be defined when the variant is removed or changed from one location to another.

```
ALGORITHM FOR INSERTING A VARIANT AT RUNTIME USING SUPER-TYPES
Input: A feature model described as set of variants V
Output: A modified feature model as a set of variants V'
/* We compare the super-type of the variant (Vx) we want to insert with the super-types of
their siblings and parent. If the new variant has the same super-type with respect to his
new parent and any of the siblings it can be inserted as a child automatically. If the new
variant has no siblings, we only check the compatibility of the its super-type with the super-
type of its parent, as this new variant will be the first feature added under that parent*/
Foreach Vx in V {
 V'←V
 If Vx is Child(V) OR Sibling(V)
  If St(Vx) = St(V) OR St(Vx) is ListOfCompatibleSuperTypes[]
    Then InsertVariant(Vx, V');
/* In other case a new compatible super-type must be defined for the parent before the
variant can be included*/
    Else AddNewCompatibleSuperType(Vx);  }
/* If super-types are incompatible, then the new variant cannot be added except if the user
defines a new variation point with compatible super-types. */
  If St(Vx) != ListOfCompatibleSuperTypes[]
    If CreateVariationPoint = TRUE {
       Then CreateNewVariationPoint (St(Vx, V'));
       InsertVariant(Vx, V'); }
    Else exit(); }
return V';
```

To simulate the scenarios mentioned before we used a prototype tool (Variability Modeling Web Tool – Capilla et al., 2007) for which we developed a runtime module called Alter Product Runtime Mechanism (APRM) to simulate the dynamic changes in the feature model. The APRM module checks the super-type of the variants when a runtime change is requested and modifies the structural variability accordingly using a parallel environment where the changes are applied in a secured form without affecting the main system during its execution. When adding a new variant, the APRM can simulate the addition of new dependency and constraint rules, but automatic online constraint checking to detect possible inconsistencies is not yet supported. However, new constraints can be pre-checked before they are uploaded into the feature model. In addition, for removing variants we only need to run automatic scripts to check potential inconsistencies in the dependency rules when we drop a variant. Finally, changes in the values of variants is easy to simulate as we can upload files dynamically with the new configuration, and then check incompatible states of related features. As mentioned, moving a variant to a different location can be performed as a removal operation followed by an adding operation. This solution does not prescribe those cases where a variant can be inserted in two different places, as in most situations super-types are defined for functional units in the feature model that manage a specific functionality for a system family. In such cases, we can define specific policies to resolve conflicts or warn DSPL managers and then the DSPL

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

designer can decide on the best place to locate or change a variant.

Applicability: Any long-lived system that demands a continuous evolution of their features dynamically, often because of stringent adaptation requirements, is a good candidate to incorporate a runtime variability mechanism. Increasingly, self-adaptive and autonomous systems need to adapt their behavior at runtime and provide runtime adaptation and reconfiguration of their system features, in particular for those remote reconfiguration tasks when the system operates in unattended mode. For example, CVL constructs have been successfully used in embedded software-intensive systems like railway signaling and communications (Haugen et al., 2008). Helleboogh et al. (2009) apply the notion of changing variants on-the-fly to an Automated Transportation System (ATS) in order to support the dynamic variability. An ATS consists of a number of fully Automated Guided Vehicles (AGVs) that more and more require dynamic adjustable features. The smart home systems domain is another case of autonomic computing where features are reconfigured dynamically to support the activation and deactivation of system features to automate reconfiguration operations and reuse feature models at runtime (Cetina et al., 2009b,c,d).

5.2. Runtime variability management

Today, feature models provide a comprehensive set of analysis operations to achieve what is known as Automated Analysis of Feature Models (AAFM) (Benavides et al., 2010). In DSPL products, because certain runtime changes in deployed variability models may have impact both in the structure of the feature model or in the current configuration supported by system features, there is a need for variability management operations before the system shifts again to its normal operational mode. In this section we address the problem of runtime checking of feature models (Challenge 3) in two ways: (i) check feature constraints and dependencies when the structural variability is modified, and (ii) check the dependencies among features when these are activated or deactivated. Hence, we provide a solution that enables real-time model checking and reconfiguration capabilities in feature models.

Potential solution: In large feature models, modeling all the changes that may occur in the variability model at runtime could be an intractable problem, as products that dynamically adapt their behavior or perform reconfiguration operations need to change their valid states or switch between different operational models. Therefore, we add a configuration model (CM) module or database aimed to support all valid states of a deployed feature model or configured product. The CM represents those features that are activated and deactivated and also those features that are "damaged". A damaged feature is that one in an invalid state that cannot be activated anymore (e.g., due to a hardware or software failure). The feature model (FM) is used in combination with the configuration model to describe not only the variability deployed in the product but also the allowed configurations that may change dynamically in the DSPL product. In addition, quality attributes can be used to determine the best or the optimized product configuration when one feature requires another feature or if a specific product configuration requires a specific value or another feature. To achieve the proposed goals we devise the following two scenarios:

Scenario 1. Changes in the feature model: When a new feature is added or removed dynamically from the feature model, we need to check at runtime the constraints and dependencies of the modified or new feature. This task has been often carried out in off-line mode, but in DSPL products the dynamic behavior of the system demands an on-line checking in order to reconfigure the product dynamically. A feature which is added will be included dynamically in the CM with the "deactivated" state by default. A feature that is removed will be simply dropped in the CM. After, all dependencies between the features that have been added or removed will be checked dynamically using any of the existing automatic checking model techniques, such as Benavides et al. (2005). This solution complements the solution suggested in the previous section for changing the structural variability, as AAFM approaches will ensure the validity of the changes made on the feature model.

Scenario 2. Changes in the configuration model: The second scenario addresses the possible changes in the states of the features, often caused by demand to activate or deactivated system features or because a need of a certain quality level. In this case, the new CM must be checked if it satisfies all the restrictions imposed by the FM. Among these operations, a "valid configuration" operation determines whether a CM is valid (i.e., the CM meets all the constraints with respect to a given FM). In order to determine which changes should be made to obtain a valid CM, we use what we call the "configuration reparation" task (White et al., 2010; Trinidad, 2012) which computes the new CM satisfying all the constraints defined in the FM. We achieve the new CM with the smallest number of changes with respect to the previous CM. This operation can be described in terms of constraint optimization or SAT problems that can be solved on behalf of a logical reasoner software, such as Fig. 4 describes.

In order to incorporate runtime variability management facilities to the DSPL, we advocate the use of AAFMs that can be supported by well-known declarative paradigms such as SAT[3] or constraint programming problems[4] (Benavides et al., 2010). Moreover, DSPL solutions can exploit off-the-shelf logical solvers (e.g., SPLOT – http://www.splot-research.org/ – and FAMA) are AAFM solutions that can manage the dynamic checking and diagnosis of variability models. SPLOT is a web application analyzes feature models, while. FAMA (Trinidad et al., 2008a,b) is an off-line tool that can be integrated in on-line mode for feature model checking operations.

Example: As an example of scenario 2, let us consider a DSPL product of a smart home system (SHS) whose variability model is shown in the feature model of Fig. 5. The current state for each feature is also shown in the DSPL product. In our example, lighting and control system functionalities can be activated via Cell Phone. In case an end-user wants to use the video on-demand feature, the current configuration is not valid until the video on-demand can activate the Internet connection (currently deactivated). Consequently, a "configuration explanation" operation uses diagnosis techniques to determine which features must change their state to reach a valid product configuration at runtime. In this situation four possible reconfigurations can be obtained, one for each kind of Internet connection: wifi-b/g, wifi-n, Ethernet, and 3G protocols. The tiebreak criterion to choose the best configuration can be based on quality attributes demanded by the user or desired system configurations such as the available the bandwidth rate. Once the algorithm determines the optimal or the right configuration, the CM features change their values to the valid states once the constraints and dependencies have been satisfied.

[3] A SAT solver is a software that takes a propositional formula usually expressed in conjunctive normal form (CNF) and used to determine its satisfiability when the formula evaluates to true.

[4] Constraint programming analysis is a set of techniques that deal with constraint satisfaction problems (CSP) consisting on a set of variables, a set of finite domains for the variables, and a set of constraints that limit the values of the variables. A CSP solution attempts to find valid states where all constraints are satisfied.

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

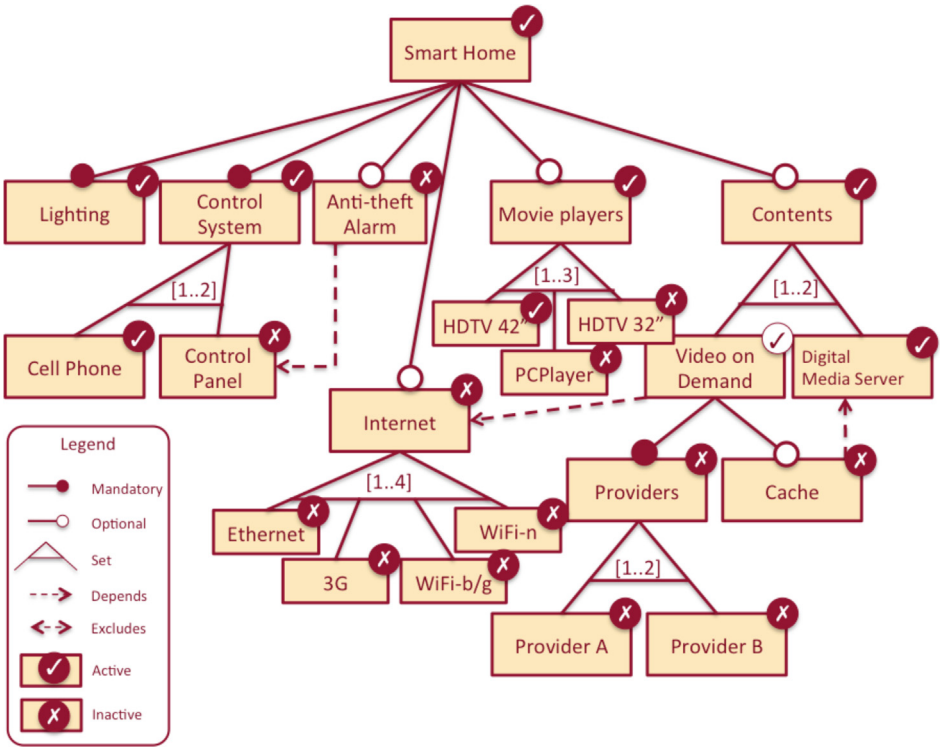

Fig. 5. A smart home system DSPL described in terms of a FM and its DSPL state in terms of activated and deactivated features.

Applicability: The proposed solution for checking feature and configuration models on-line provides DSPL products autonomous reconfiguration capabilities. Such facility becomes useful for self-healing systems but limited by the physical access to the system, such as smart home and SCADA systems, or spacecraft missions (except in those cases where remote reconfiguration is possible). In those cases where the system demands human intervention (e.g., due to a replacement of a physical module), our suggested solution enables the activation and deactivation using the previous known configuration and the constraints defined in the feature model. Remote updates are possible launching a reconfiguration process that requires only update the feature model that represents the overall set of allowed configurations of the system.

5.3. Runtime binding and re-binding

For years, conventional SPLs have exploited the notion of binding time to delay the design decisions to a later stage, what is often known as variability in time. Traditionally, SPL models and products limit this delay to only one binding at a time (e.g., a product knows the values of its variants at, for instance, design or configuration time), but more than one binding time concurrently is rarely seen.

Today, in the era of post-deployment, more and more systems push to support runtime binding modes In this way systems need to be reconfigured and redeployed several times or periodically. Therefore, the increasing demand of self-adaptation properties, late binding of variants, and the need to re-bind system options several times has motivated the necessity to support: (i) *multiple binding times*, and (ii) *re-binding* capabilities in order to provide highly adaptable products. In line with the concept of any-time variability "*the ability of a software artifact to vary its behavior at any point in the life cycle*" (Van der Hoek, 2004), this section addresses Challenge 6.

Potential solution: Dynamic Software Product Lines often demand multiple binding times and bind variants at a very late stage in the software lifecycle, as the flexibility and dynamicity of DSPL products depends more on the possible times where variants can bind to their values. In our DSPL reference architecture, a binder (i.e., a dynamic binding manager) is a module that manages multiple and late binding times, but variants can be bound differently. For instance, an autonomous system that modifies its behavior using exclusively context information may implement a binder that can upload different values from local or remote dynamic files, while a service-based system may implement a binder that selects new services using system variants on behalf of a proxy server which redirects the requests to the right service. In mobile software, build-in variants are integrated into the software device when the end-user selects a configurable option, and some system options are bound locally in the device while others need a remote connection to a service provider. Hence, different binding solutions for different dynamic systems might be possible. Additionally, enabling multiple binding and re-binding capabilities lead also to move from one binding time to another, but not all the transitions are allowed (e.g., we cannot bind a variant from design or compilation time to runtime, but moving from configuration to runtime is possible). Likewise binding time is a property of feature models in conventional SPLs that could be annotated separately, we could have a database or text files supporting a set of allowed binding times for each bunch of related features in the DSPL. The transition from one binding time to another will depend of the capabilities of the binder and concrete DSPL product.

From a recent work (Bosch and Capilla, 2012), we provide a finer classification for multiple binding times and the possible transition from one binding time to another. We describe this in Table 3, which guides software designers on the selection of static or dynamic binding times and on the possible transitions between them. The explanation of the table is as follows:

- *Static/dynamic binding*: The first row of the Table describes for each of the proposed binding times which of them can be considered static, dynamic or both.
- *Configurability*: The second row describes how difficult or easy we perceive implementing each binding time solution will be. As

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

Table 3
Transition between multiple binding times closer to runtime operations in variability models.

	Configuration binding time (Cf)	Deploy (Dpl) re-deploy binding times (Rdpl)	Runtime (start-up) binding time (RT)	Pure runtime (operational mode) binding time (PRT)
Static/dynamic binding	Static or dynamic	Static or dynamic	Dynamic	Dynamic
Configurability	Medium/high	Medium/high	High	Very high
Single/multiple binding times allowed	Single and multiple	Single and multiple	Single and multiple	Single and multiple
Binding in the developer/customer side	Developer and customer	Developer and customer	Only customer	Only customer
Transition for multiple binding times	Cf → Dpl Cf → RT	Dpl → Rdpl Rdpl → Cf	Dpl → RT Rdpl → RT RT → Rdpl RT → Cf	RT → PRT PRT → Cf Cf → PRT PRT → PRT

more close to runtime the binding mode the complexity increases because we need a binder for selecting the values of the variants at runtime and if needed, provide the transition from one binding mode to another.

- *Single or multiple binding times allowed*: The third row shows if multiple or single binding times are allowed. For instance, those systems that pose a certain degree of dynamicity can support more than one binding time concurrently, as opposite to static binding times like compilation.
- *Binding in the developer/customer side*: The fourth row describes if the binding time can occur in the developer or customer side or both. For instance, a DSPL product can be configured at the customer side before deployment while end-users can configure some system options in the product (i.e., customer side). By contrary, products that implement runtime variability are expected to bind their variants at the customer side. However, we do not take into account those testing operations that the developer may run or special remote configurations (e.g., the case of unmanaged vehicles or systems) for classifying the binding time in one site or another.
- *Transition between multiple binding times*: The fifth row shows the possible transitions from one binding time to another. A transition happens when we use a static or dynamic binding and the designer allows changing the moment in which the variants are bound, but from the point of view of a DSPL, we limit the transitions only to dynamic binding. The arrows between two binding modes mean the allowed transitions for each of the binding times described in the columns. For instance, for configuration binding time (Cf), variants using a dynamic strategy can bind to their values at configuration time but if the system implements, for instance, automatic deployment procedures, certain variants can bind their values at deployment time (Dpl). Another example that happens at the pure runtime binding time is the transition "PRT → PRT", which means that the product running in fully operational mode can be reconfigured dynamically and it will go again to the same operational mode where variants at bound at the same stage.

Example: To motivate our example, we focus on the simulation of the dynamic binding of single services as part of work developed at the *Rey Juan Carlos University* in 2010, where we developed prototype software to select services dynamically using two operation modes: *manually and automatically*. Our example does not show any transition between possible binding times but it serves to illustrate a possible binding solution for service-oriented systems. The platform uses a set of quality attributes which can be flexibly grouped in a quality formula using a set of variants. A Web server acting as a broker stores local services while third-party providers can also put the information of their services using a Web form (i.e., these services are stored remotely on the service provider infrastructure). Therefore, each broker stores in a SQL database the information of their local and remote services available for the users of the platform. Once a user selects a service of a specific type and functionality, the broker compares the quality of the current service (i.e., the quality is determined by a formula using different quality parameters) with those others stored in the database. If the broker finds a new service with better quality, it binds to the new service found at runtime. In manual mode, the system warns the user if he/she would like to switch to a better service while in automatic mode this task is done transparently to the user. The binder also manages the states of the services and the PID to stop and start services, as well as a URL to bind to remote services. Software variability is used to manage the parameters of the quality formula and part of the information used by the binder. As expected, the broker acts as a proxy server to locate and bind distributed services stored in different machines. Fig. 6 shows an example of the modules supporting the functionality for the dynamic selection of single Web services that use QoS properties and context information (e.g., GPS location info) as variants for the selection of the best service.

Applicability: As there many types of systems where dynamic binding is possible, in Table 4 we provide different types of binding solutions that enable dynamic binding or allow the transition between different binding times.

Table 4
Systems make use of dynamic binding solutions.

Types of system suitable for dynamic binding	Binding technology solutions
Self-adaptive and autonomous systems	Variants bind to values using dynamic configuration files. The binder binds values at runtime downloading a remote file or using data from the environment
Service-based systems	A local or remote proxy server selects services to bind variants defined in service-based application. The binder is often implemented as a proxy
Mobile software	A local binder in the device selects in-built system options. The binder access services remotely through a proxy server in the service provider or third-party provider. Variants are instantiated using context information like the in-built positioning system
Stand-alone systems that demand periodical reconfiguration tasks	Variants bind to values using dynamic configuration files. The system connects to a remote server to select new configurable options remotely
Cloud-base systems Mobile cloud-based software	Cloud applications bind remotely to service using domain-specific platforms like SCA (System Component Architectures). Cloud providers offer specific hardware-software integrated solutions where users get resources dynamically on demand on behalf of virtualized solutions offered by the SaaS-PaaS-IaaS model or using virtual machine-based cloudlets for mobile devices

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

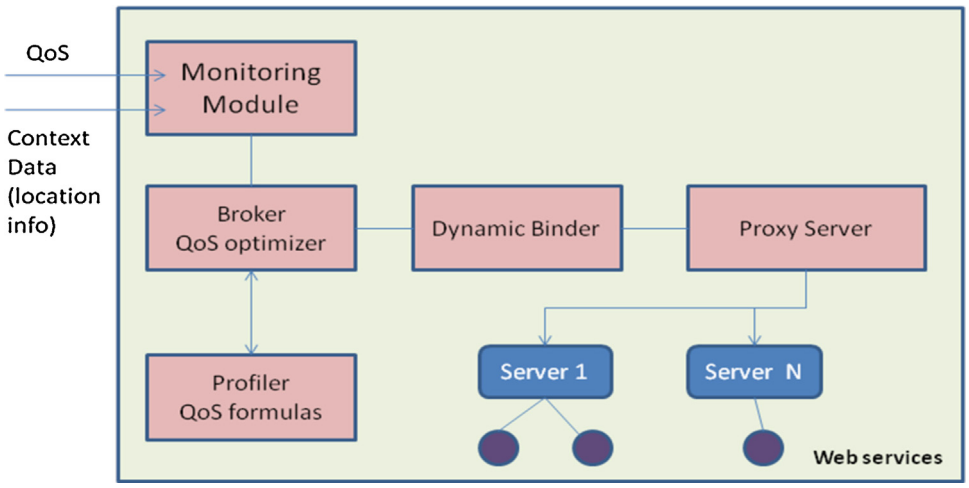

Fig. 6. Functional modules for managing binding in the selection of Web services.

5.4. Context variability for adaptation purposes

Many embedded systems use context information to adjust their behavior at runtime. For instance, hardware sensors detect the changes in the environment and the system reacts to the new conditions. In other cases, software detects changes in the quality conditions and selects a new alternative or service at the latest binding time in order to perform the same operation better. From our point of view, systems that exploit context awareness must gather data from three different sources: (i) hardware sensors, (ii) software sensors able measure the qualities of the running system (e.g., Internet, services, and scripts), and (iii) user input (e.g., mobile software users that activate a system feature). Consequently, the context-awareness conditions for DSPLs should enable this multi-data entry capability to activate or switch system features dynamically. For DSPLs, runtime variability mechanisms should use features specifically related to manage context information on-demand and offer runtime reconfiguration alternatives for highly configurable products. In this section we address Challenges 1, 5, and 7.

Potential solution: Following the strategy of annotating DSPL features with super-types, we suggest an integrated solution for context features rather duplicating feature models (i.e., a feature model and a context variability model). This way provides a simpler solution and reduces the number of dependencies between features. Like other FODA extensions proposed in the past (e.g., annotate FODA with quantitative values), we label those context-aware features in the FODA tree with "C" and we specify some additional information that will be used by a MAPE manager for our DSPL autonomic context-aware products. We assume those features not labeled with "C" do not have influence in context-aware properties, and the software designer is the responsible to model those context features in the variability model. We believe that context features provide the necessary capability of systems to reconfigure and manage themselves when features bind dynamically to their allowed options during several binding times.

As shown in Table 5, we define supplementary information useful for a context manager to control the information described in the context features. This information is managed in our reference architecture by the variability manager and other reconfiguration and binding mechanisms. The second column of Table 5 shows the current "State" of the feature, that is when a feature is activated or deactivated. The third and fourth columns describe the names of the policies and reconfiguration strategies that apply for each context feature. For instance, the reconfiguration of a particular feature is often driven by one policy which may comprise one or several reconfiguration strategies. These algorithms are used by the variability manager to adjust the behavior of the system to different situations. Finally, we include the binding time of the features, as more than one binding times are possible for DSPL products.

The detection of incompatibilities between active and non-active features or between different reconfiguration strategies must be solved by adequate policies and by the variability manager module. In addition, reconfiguration plans must be tested safely before the reconfiguration process happens and to avoid inconsistent states in the system, but in most cases a rollback mechanism is also provided. However, our goal with this solution is to provide a model to define context features linking these to reconfiguration strategies, and not to provide specific reconfiguration actions dependable of a particular hardware and/or system functionality.

Our proposed solution is straightforward and simple but it eases the way to annotate context features without having two distinct feature models, as the proposed table can be easily handled in order to select the best runtime reconfiguration and adaptation strategy.

Table 5
Examples of context features using different strategies for different binding times.

Context features	State	Policies	Reconfiguration strategies	Binding time
Temperature	On	Policy 1	Strategy 1	Configuration Runtime
Smoke	Off	Policy 2	Strategy 1 Strategy 2	Runtime
Humidity	On	Policy 1	Strategy 1	Runtime
Light	Off	Policy 1	Strategy 1	Runtime
Comm. node identity	On	Policy 3	Strategy 3	Deployment Redeployment Runtime

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

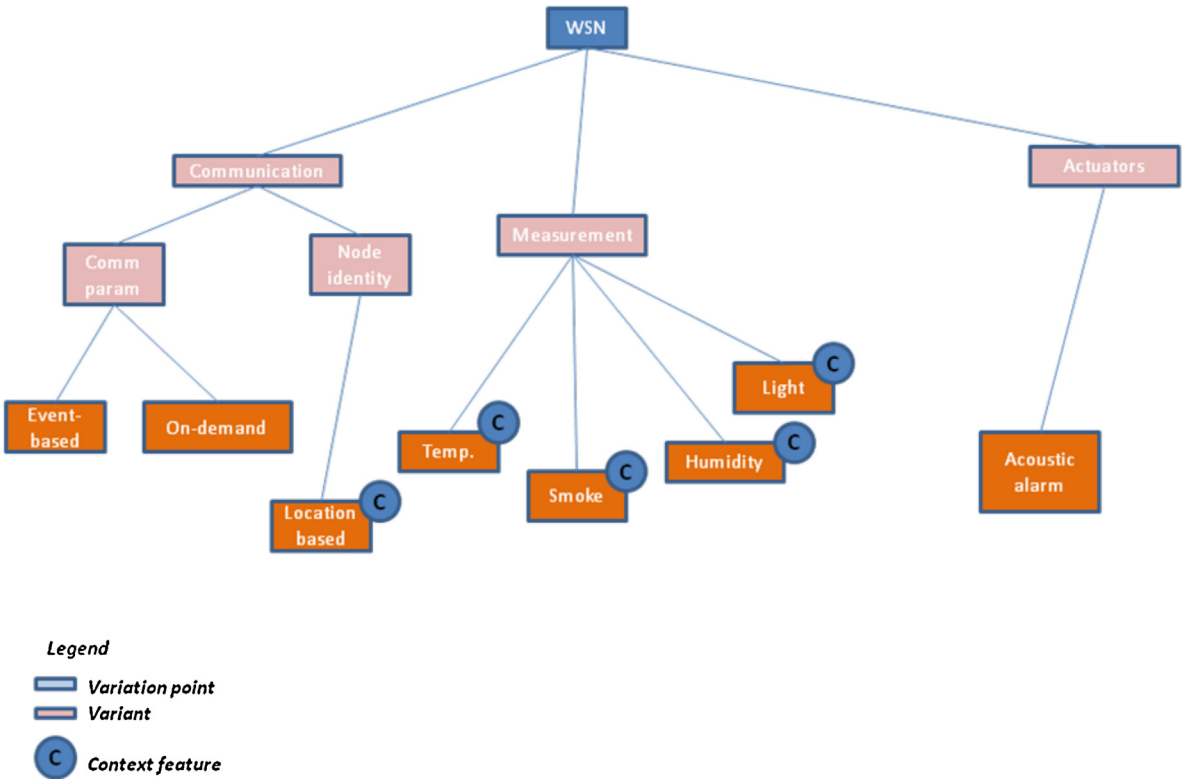

Fig. 7. Context features labeled in the WSN feature model.

Example: Fig. 7 extends the WSN feature model of Fig. 3 to label only those context features with "C". This gives the designer a quick view about the context features without having separate branches in the feature model and to organize the context variability. We could also apply the same solution to annotate the state (i.e., On and Off) of the features and show the dynamic changes graphically.

Regarding the states of the features given in Table 5, one runtime scenario could be triggering events when certain threshold values are exceeded, as this may require the activation and/or deactivation of system features based on the events received by the sensors and analyzed by a MAPE loop. The modification to any of the nodes in the network may cause an update to the software of the rest of the nodes, as the information propagates across the network. A similar scenario has been used in previous work (Órtiz et al., 2012). Any high-level notation can be used to formalize the changes in the states and implemented using a high-level programming language. In the following pseudo-code we describe how features can be activated according to a certain event and the current configuration of the threshold values. All the suggested changes are checked and applied in case there is no conflict and all constraint rules are satisfied.

FOR all features in "ST:Ambient" **Configure** (threshold values)

Activate_Feature (Comm_type.Event-based)

IF **Check_Feature_States** (All_features)=True

THEN **Apply_changes** (constraint_rule_set)

Applicability: Self-*systems, pervasive, and autonomous systems among others are the most suitable candidates to benefit from the usage of context features. Experiences with autonomic computing in the Wireless Sensor Network field, where sensor nodes adapt themselves to context changes with minimum human intervention and exploiting context features (Gámez et al., 2011), or the case of a smart home system (SHS) where a feature model is embedded as a system component to guide the selection of system variants in response to changes in the environment. In this solution, features are associated to reconfiguration plans to provide self-reconfiguration capabilities at runtime (Cetina et al., 2009b,c,d).

5.5. Dynamic and optimized reconfiguration

Some dynamic systems take into account the preferences of different users in terms of desirable features and measurable criteria such as memory and power consumption or feature attributes such as Internet bandwidth or screen resolution. When customer's preferences change at runtime, the system must be reconfigured to satisfy as much preferences as possible. Since user's preferences might be contradictory, only some of them can be partially satisfied and a criterion is needed to compute the most suitable reconfiguration. This section addresses Challenge 4.

Potential solution: Although these proposals deal with user decisions, they are limited those that affect features, being unable to represent any decision on quality attributes further than the five satisfaction levels proposed by Sawyer et al. (2012). In this solution we propose the use of the so-called Stateful Feature Models, SFMs (Trinidad, 2012) which represent user preferences in terms of features and attributes. SFMs constitute a natural evolution of traditional FMs that include the set of user's configurations in the model itself, as they allow the representation of user decisions with attributes and cardinalities. The FAMA toolset is able to deal with FM attributes but the current version does not provide a means of representing user decisions about attributes. Trinidad (2012) develops STEAm, an MDE-based prototype tool for the automated analysis of SFMs. Since SFMs enable decisions on attributes, STEAm may prove to be an interesting tool for dynamic and optimized configuration.

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

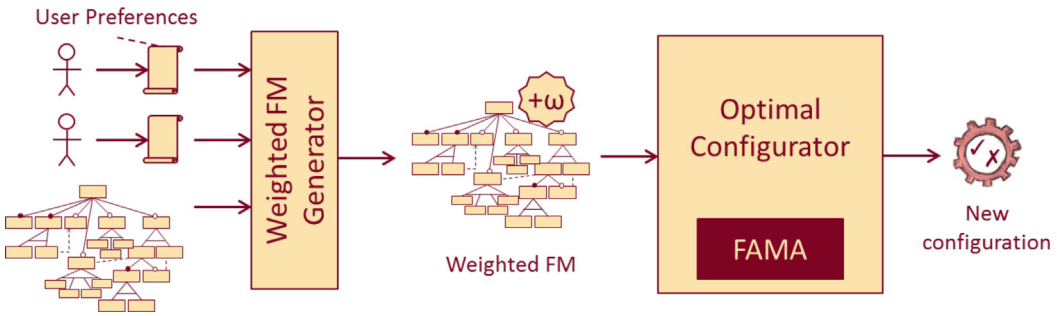

Fig. 8. Optimized configuration procedure for multiple users.

Example: Let us consider the FM in Fig. 3 that describes a smart home system. There are two users at home whose preferences are: User (A) Watching video on demand on the 42″ TV using any Internet connection and; User (B) watching a video from the DMS on a laptop using a wireless Internet connection, either Wi-Fi-n or Wi-Fi-b/g. We assign a weight that depends on the importance of the final configuration which is close to user preferences. For instance, let us consider a user A whose weight is $w(A) = 2$ and $w(B) = 1$, which means that two decisions satisfy user A and one decision satisfies user B. Then, a weighted FM is generated from this information, resulting that every feature is given a weight if it is selected or removed depending on user preferences.

Let us consider that U_S is the set of users that have selected a feature, and U_R is the set of users that have removed that feature. The weight for a feature can be defined as using the proposed formula:

$$\omega(F) = \sum_{\omega \in \varepsilon U_S} \omega(u_i) - \sum_{\omega \in \varepsilon U_R} \omega(u_i)$$

In our example, the Internet connection feature (F_{IC}) has an assigned weight $w(F_{IC}) = 3$ when the feature is selected, and $w(F_{IC}) = -3$ if the feature is removed. In addition, Wi-Fi-n $w(F_{wn})$ and Wi-Fi-b/g $w(F_{bg})$ features have the following values $w(F_{wn}) = w(F_{bg}) = \pm 1$ for adding and removal operations when user B specify such need. The configurator is in charge of searching for an optimal configuration which maximizes the sum of weights and satisfies as much user preferences as possible. In this case, the most weighted solution uses Wi-Fi-n or Wi-Fi-b/g as Internet connections which satisfy both users. A quality criterion can be used to represent other constraints, such as pay-per-use cost for internet connections, which is bound to the download rate required by the video on demand provider and depending on different screen resolutions. User A can be worried about reducing the cost as much as possible while user B does not care about this issue and maybe wants to obtain the highest resolution screen. In that case, the weights are used to find the best solution which satisfies both users. Fig. 8 shows the inputs and outputs using the FAMA configurator tool using weighted feature models and according the user preferences.

Applicability: Self-*systems and pervasive systems that interact with several users at the same time are suitable candidates that demand optimized solutions, users of these systems often demand the activation and deactivation of system features based on a quality criterion. In addition, service-based system continuously demand changes in the quality level when new services with better quality are found or when external conditions change.

6. Application domains

Once we have described the possible solutions for implementing a runtime variability mechanism, in this section we provide additional discussion and examples of specific application domains where DSPLs can harvest best results.

6.1. Service-oriented systems

Service-based and cloud systems are driving the SaaS (Software-as-a-Service) trend to provide flexible and low cost services to replace part of the core functionality of systems. Customizable services often rely on quality factors and other context properties (e.g., the user selects a new service as a configurable option in a mobile device), and software variability can help to make better decisions during the selection of a new service. Context-aware service engineering becomes an essential part of modern service computing as it can be extensively used to control the behavior of services. Many of the efforts using contextual information for service engineering can be found in Kapitsaki et al. (2009). Consequently, extensible variability models for dynamic services are necessary to provide higher flexibility at lower cost. Several experiences suggest the use of variability to select services dynamically. Koning et al. (2009) proposes the VxBPEL language, as an adaptation of BPEL (Business Process Execution Language) using variants to determine the selection of the best choice for composite services in dynamic business environments. The replacement and re-binding of services dynamically when QoS conditions change is supported by different types of variability implemented in the VxBPEL language. However, the majority of the dynamic solutions to bind and re-bind services dynamically when QoS conditions change use a proxy solution to redirect the flow to the right service (Canfora et al., 2008; Erradi and Maheshwari, 2008). These proxy solutions monitor QoS parameters and provide a binder mechanism to select the most appropriate service available transparently to the user.

Regarding the use of DSPLs in service-oriented computing, Shokry and Ali Babar (2008) introduce SOA (service-oriented architecture) for the creation of dynamically reconfigurable product line architectures and runtime variability techniques for the development of embedded vehicle electronics, as a kind of Distributed Real-time Embedded Systems (DRES). Hallsteinsen et al. (2009) argue that DSPLs offer an adequate development model for service implementations and where runtime variability becomes an important concern for SLA (Service Level Agreement) negotiation. The synergy between service-oriented architectures and Dynamic Software Product Lines have been also investigated by Istoan et al. (2009), and applied to a service-based middleware designed for solving house automation solutions. Parra et al. (2009) propose a context-awareness dynamic service-oriented product line (CAPucine), with the goal to combine service-oriented and context-aware to integrate DSPL assets dynamically into a running system. The approach uses a SCA[5]

[5] Open SOA. Service component architecture specifications. 2007. Available at: http://www.osoa.org/display/Main/Service+Component+Architecture+Home.

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

(Service Component Architecture) platform (FraSCAti) to enable dynamic binding and assembly of components at runtime (i.e., context-aware assets). Gomaa and Hashimoto (2011) state also the importance of DSPLs for dynamic adaptation of service-oriented systems, and they use a dynamic feature model for dynamically adapt SOA members at runtime. The SASSY (Self-architecting Software Systems) runtime architecture is the proposed solution to monitor and replace services using the dynamic feature selection mechanism, which dynamically determines the changes to the application feature model.

The importance of combining the SOA and DSPL worlds is discussed by Lee and Kontoya (2010) and Lee et al. (2012), as most SPLE approaches lack mechanisms for supporting the runtime conditions of service-based systems, where services are monitored and selected dynamically when the context, quality conditions, or user preferences change. Pre- and post-conditions also drive the dynamic binding of services in the DSPL and using QoS specifications to monitor and negotiate the system quality for workflow reconfigurations. With the appearance of cloud computing and third-party providers, the delocalization and selection of services dynamically on-demand become more important for DSPLs in order to build more adaptive and evolvable service-based systems.

6.2. Mobile software

Another application domain suitable for implementing runtime variability using a DSPL is the large variety of mobile software products that currently exist in the market. The increasing trend of end-customers that use smart phones to connect to Internet services is pushes to download and use more and more third-party applications (e.g., downloaded from Google Play Store) and other services (e.g., News, Weather, Navigation, Maps, etc.). Smart phones and other related products increase user experience and satisfaction as they allow a high degree of personalization, as users can customize many of the services and applications installed in the device (e.g., profile configuration of latest WhatsApp versions). In addition, much of this software uses context information to locate new services or local information (e.g., Samsung's local app to locate restaurants, shops, etc.), while other application rely on strong QoS requirements like downloading video at a sustainable bit rate or accessing 3G+ networks with enough power signal. Mobile software enables high customization and personalization of software through multi-data entry, in particular for those applications that use context information and enable users to set their own preferences. Therefore, in the era of post-deployment evolution where such devices can be configured and re-configured at any time, runtime variability becomes a good choice to enable the selection of different software options, and access infrastructure's services more dynamically. Regarding the dynamic binding problem in mobile devices, this can be done after deployment locally to the device for in-built variants that are user configurable, or services and software can be bound remotely at runtime (e.g., a software update). This remote binding can be realized through the telecommunication operator as service provider or directly to third-party providers.

Some experiences using dynamic variability in mobile devices are the following. Brataas et al. (2011) use variability for context-aware mobile applications using an adaptation middleware to reason about different application variants that may be reconfigured dynamically at runtime. As expected, the selection of resources on-demand in the proposed model is driven by a QoS model. Also, as mobile and other pervasive devices pose limited hardware capabilities, automatic variant selection is also limited by non-functional properties of the device like memory consumption or the data transfer rate of the communication protocol used, which often hampers the selection time of the variant in the device (White et al., 2007).

6.3. Ecosystems

Successful software product lines experience an expanding scope in terms of products, features and customer bases (Bosch, 2006). The scope of a software product line typically evolves because it receives broader adoption within the company. However, there is no reason why a software product line would have to stop expanding at the organizational boundary. The product line architecture and shared components, typically referred to as the platform, can also be made available to parties external to the company. Once the company decides to make its platform available outside the organizational boundary, the company transitions from a software product line to a software ecosystem.

There are, at least, two reasons why a company would be interested in moving toward a software ecosystem. First, the company may realize that the amount of functionality that needs to be developed to satisfy the needs of their customers is far more than what can be built in a reasonable amount of time and with an R&D investment that offers an acceptable return on investment. For web services companies, as well as the software industry as a whole, the market often operates based on the "winner takes all" principle. Consequently, building a large customer base as rapidly as possible is a key strategy for long-term success.

Secondly, the mass customization trend drives the need for a significant R&D investment for successful software applications. Especially on the web, e.g., web service mashups, but also in other domains, e.g., mobile devices, users demand a significant degree of customization or even compositionality that allows each user to create a potentially unique configuration that addresses her or his specific needs and desires. Extending the product (which includes the platform) with externally developed components or applications provide an effective mechanism for facilitating mass customization.

The aforementioned developments are some of the driving forces behind the emergence of software ecosystems (Bosch, 2009). For instance, companies that have initially built success with their web application are compelled to platformize their application and open it up for contributions by third party developers. These developers can provide functionality that addresses the needs of user segments that the company providing the platform would be unable to develop by itself.

The transition to a software ecosystem has significant implications for software variability. The primary implication is that because composition of the base product with the solutions provided by ecosystem developers is frequently performed by customers, any variation points that are used need to be bound or rebound at run-time. Second, third party applications may offer new variants for existing variation points in the base product as well as new constraints for variant points as well as variants. Finally, there often are interactions between the solutions provided by different ecosystem developers that require significant run-time dynamism from the base platform to address potential conflicts.

6.4. Autonomous and self-adaptive systems

Following decades of successful use in industrial automation, public attention has turned again to robots to replace humans and perform tasks where continual human operation would be impossible. This includes the use of robots to facilitate independent living for aged, ill or infirm patients (Brady et al., 2012). Of interest too is the use of robots for excavating natural resources and for surveillance and exploration. The latter has, no doubt, piqued the public

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

attention in no small means due to NASA's successful robotic exploration program, most recently the Curiosity rover.

The use of SPLs in such applications makes a lot of sense: many of the applications possess many of the same properties. For example, in the case of independent living, houses may have different configurations and vary greatly in size and location, but most will have the same major room-types to monitor (kitchen, one or more bathrooms, one or more bedrooms) and standard "hurdles" (staircases, etc.) although they may vary in location and shape. For this reason, a product-line offers many benefits, with each house-monitoring system being a different instance of the same product.

However, the use of SPLs is even of more interest when the system is adaptive. NASA has long realized that there are benefits accruing from the use of SPLs. In addition to the cost reductions and decreases in development time, the preservation of organizational memory has been a particular benefit for NASA where mission development can long outpace the work-life of any individual. Moreover, NASA missions are by their very nature adaptive. They must adjust to unforeseen (and unforeseeable) circumstances. For example, a rover landing on the moon or other planetary surface, must be able to adapt to an unstable landing, to the loss of instruments or circuitry due to sand and foreign bodies, etc. Adaptation to every possible eventuality cannot be planned a priori, but significant amounts of adaptation are possible.

An adaptive system, particularly one that adapts over a significant period of time rather than continuously, can be viewed as a product line with each "product" in the line being a particular instance of the system that is evolving (Peña et al., 2007). NASA has realized this and applied it, for example, in its GRAIL missions. A similar approach is being taken with the concept mission ANTS (Autonomous Nano-Technology Swarm). The ANTS mission foresees the use of a large number (as many as 1000) self-similar devices which will either explore planetary surfaces or fly together in a swarm to collect data from the asteroid belt (the PAM submission), the rings of Saturn (the SARA submission) or the lunar surface (LARA). The hardware for each submission will vary slightly but will be built around the same architecture and with heavy reliance on adaptation for mission preservation and resilience (Vassev et al., 2012). Adaptation is essential as the mission is simply too far from Earth to be tended by humans. In the case of the PAM submission, for example, the round-trip delay from Earth to the asteroid belt exceeds 40 min. Even if the mission could identify an issue and relay it to Earth, and even if ground control could immediately diagnose the problem and issue instructions, 40 min would have elapsed. Dynamic variation enables the mission to take on whatever "role" is necessary in dealing with potential hazards and unplanned events. It is by being able to change at run-time to another instance of the SPL that resilience and long-term protection is retained by the mission.

For example, the PAM submission envisions 3 types of spacecraft: workers, which will have specialized miniaturized instruments for collection of various types of data (X-ray, magnetoscopes, etc.), communication specialists which will relay messages between the low-power workers, and rulers which will coordinate and make decisions on behalf of the sub-swarm based on the data collected. The mission, being so far from Earth, cannot realistically replace damaged components in the short term. However, dynamic variability enables damaged spacecraft to change role. For example, a worker with a damaged telescope due to a collision with an asteroid or another spacecraft in the swarm (collisions are very likely as the low-power spacecraft can only move short distances at a time as they carry no propulsion mechanisms) is no longer able to take images, however it may change role to function as a communication specialist because it has the necessary physical functionality and simply needs to employ dynamic variation to alter its software.

Similarly, dynamic variability can enable different devices to perform other roles. In various ANTS submissions, there are different (physical) hazards due to the environment. In SARA, which explores the rings of Saturn, a significant issue is dealing with exposure to gases and radiation from the planet's atmosphere. In PAM, a major issue, in addition to collisions, is the prospect of solar storms, where the Sun ejects both physical debris and electrical radiation which can damage the solar panels (or sails) of the ANTS spacecraft. Dynamic variability enables the spacecraft to change role to enter a "protection" mode without the small spacecraft having to have all of the software loaded at launch time. Instead, a change of the software is possible to enable a change in behavior. Fig. 9 illustrates possible roles that the spacecraft may engage in Peña et al. (2007).

7. Discussion

In this section we discuss our results from the solutions proposed in Section 5 and derived from the challenges described in Table 1. The current maturity of the DSPL technology and processes showed us that we cannot provide a full validation of the DSPL dual lifecycle, but only partial solutions, some of them being proven in small-medium systems and cases studies. However, the lack of tools covering the full spectrum of DSPL solutions and the low adoption in industry still hampers this goal. From our observations we found that not all of the seven desired properties in our DSPL model need to be supported at the same time. To recap, we summarize in the following five points the major characteristics of the problems we addressed and the solutions given. We believe in the validity of the proposed solutions as a result of our experience in the development of some of these and their application to sample applications.

1. The *characterization of runtime variability* models must support the activation and deactivation of features dynamically and the possible changes in the structural variability on-the-fly. Our notion of super-types enables the modification of the structural variability and describes how context features can be modified dynamically. Changes to the structural variability at runtime enable designers to add, remove, or change a specific system functionality at runtime and to make the designer of it.
2. *Runtime variability management operations* are needed to check and ensure the validity of the new system configuration and feature models. Constraints and dependency rules among features must be validated, and automated analysis tools such as FAMA help in this validation. Changes in the states of features are necessary to support, for instance, the variety of systems that exploit context information. Consequently, any change in states of features in a DSPL context must be monitored and checked at runtime.
3. *A DSPL must provide late and multiple binding times*, as many of the activities performed dynamically happen at post-deployment time. Compared to previous work, we have provided a finer classification of possible binding times and the transitions between them. In addition, and depending on the type of systems we want to build, the implementation of the binding time mechanism can be different, and it may vary, for instance, from a service-based system to self-adaptive software. As there are many critical systems that demand changes between their different operational modes, using multiple binding times for those critical features we can address the transition between the states of such critical features.
4. *Context variability* now plays a key role for DSPLs, as software designers must identify those context features as system options that may vary according to changes in the context. Therefore, an explicit identification and representation of context features in

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

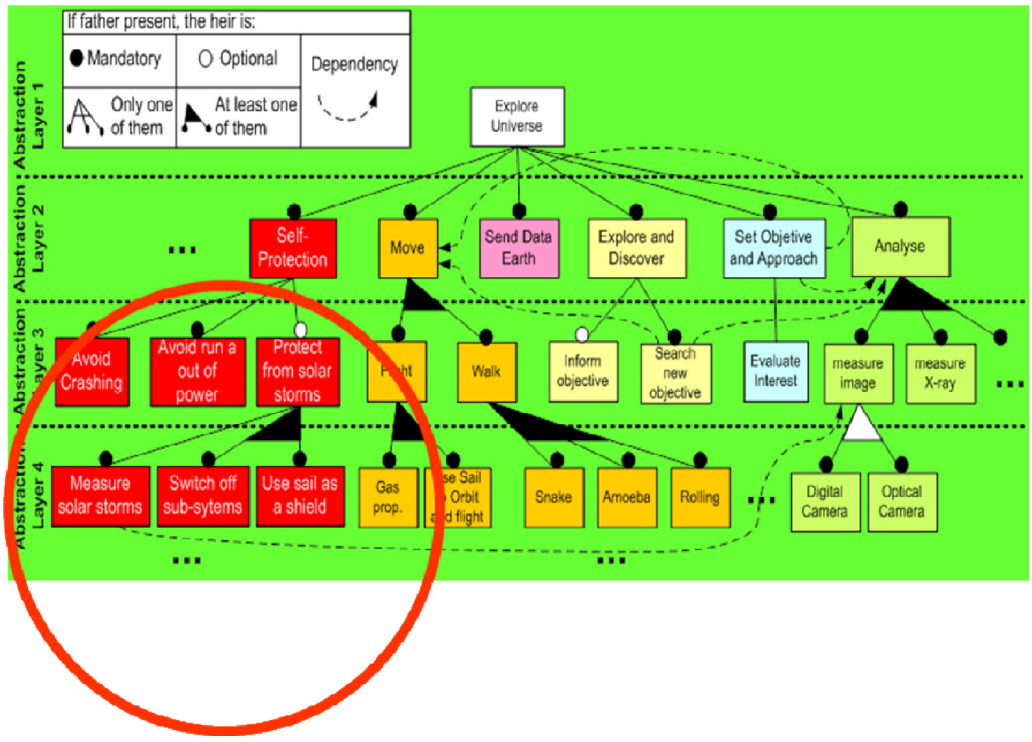

Fig. 9. A feature model illustrating the possible behavior of an ANTS spacecraft in protecting from a solar storm by powering down sub-systems to avoid electrical damage and using its solar sail (panel) as a shield.

the feature model helps DSPL designers to manage them more efficiently, and associate specific policies and strategies for the activation and deactivation of those context features.

5. *Dynamic and optimized reconfiguration mechanisms* are key for many mission-critical systems that demand an optimal solution for the behavior of the system that is support by variants that change at runtime. Our proposed stateful feature models constitute a first attempt to provide on-line optimization solutions with tool support able to compute the best choice based on user preferences and other quality information.

8. Conclusions and future work

Many of the concepts and ideas described in this paper can be found in a recent work (Capilla et al., 2013). However, in this research we go a step beyond connecting the problems and solutions that Dynamic Software Product Lines demand versus the more traditional SPL practices. The challenge of achieving the dynamic adaptation and reconfiguration of DSPL products implies in many cases that feature models can be changed on the fly or the activation and deactivation of features on demand or when context conditions vary.

From an architecture point of view, we have highlighted the main functional parts any DSPL runtime architecture should pose, but also how this architecture can be adapted to different DSPL products (e.g., service-based systems). From the characterization of a DSPL we learned that the traditional dual lifecycle used in conventional SPLs must be extended to support post-deployment reconfiguration operations and is motivated because DSPL products should support the latest binding time of their system options, sometimes performed by end users. Therefore, new activities aimed at handling runtime operations establish the difference from conventional approaches where the binding time and the realization of the variability are often performed before deployment time.

Dynamic Software Product Lines offer a good choice for those systems that modify their configuration dynamically but because of the cost of introducing runtime architecture and instrumentation in code, DSPLs are not affordable for all types of systems. This is true also because of the computational overhead required by some of the technical solutions described in this research. We have also provided examples of application domains where DSPLs may succeed or become more useful, as the increasing market of mobile and smart systems demand more and more adaptation, evolution, and reconfigurable solutions. These examples demonstrate the need that applications with stringent runtime requirements demand runtime mechanisms that reduce human intervention when certain system features change.

Future research is still necessary to provide more efficient mechanisms able to manage the dynamic and adaptive characteristics of modern software and also to determine how systems can be deployed and re-deployed automatically using variability mechanisms and multiple binding times, reducing the effort required by systems engineering operations. Regarding our runtime variability characterization, new hierarchies of super-types can be defined and also priority lists to define where a new variant can be inserted in case we could have several possibilities to anchor new functionality in the system. As an emerging topic, we expect that promising new research will bring better and integrated solutions for DSPL products.

References

Abbas, N., Andersson, J., Weyns, D.,2011. Knowledge evolution in autonomic product lines. In: 5th International Workshop on Dynamic Software Product Lines (DSPL), 15th International Software Product Line Conference (SPLC 2011). ACM DL, 36.

Abbas, N., Andersson, J., Löwe, W.,2010. Autonomic Software Product Lines (ASPL). In: 1st Workshop on Variability in Software Product Line Architectures, 4th European Conference on Software Architecture (ECSA 2010). ACM DL, pp. 324–331.

Ali, R., Chitchyan, R., Giorgini, P., 2009. Context for goal-level product line derivation. In: Proceedings of 3rd Dynamic Software Product Lines (DSPL), Limerick, Ireland.

Batory, D., Benavides, D., Ruiz-Cortés, A., 2006. Automated analysis of feature models: challenges ahead. Communications of the ACM 49 (12), 45–47.

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

Benavides, D., Ruiz-Cortés, A., Trinidad, P., 2005. Automated reasoning on feature models. In: LNCS 3520, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, pp. 491–503, ISSN 0302-9743.
Benavides, D., Segura, S., Ruiz Cortés, A., 2010. Automated analysis of feature models 20 years later: a literature review. Information Systems 35 (6), 615–636.
Bencomo, N., Blair, G., Flores, C., Sawyer, P., 2008. Reflective component-based technologies to support dynamic variability. In: 2nd International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008), Essen, Germany, pp. 141–150.
Bencomo, N., Lee, J., Hallsteinsen, S., 2010. How dynamic is your dynamic software product line? In: 4th International Workshop on Dynamic Software Product Lines (DSPL), 14th International Software Product Line Conference (SPLC 2011), pp. 61–68.
Bencomo, N., Hallsteinsen, S., Santana de Almeida, E., 2012. A view of the dynamic software product line landscape. IEEE Computer 45 (10), 36–41.
Bosch, J., 2000. Design and Use of Software Architectures. Adopting and Evolving a Product Line Approach. Addison-Wesley, Harlow.
Bosch, J., 2006. The challenges of broadening the scope of software product families. Communications of the ACM 49 (December (12)), 41–44.
Bosch, J., 2009. From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference (SPLC 2009).
Bosch, J., Capilla, R., 2012. From static to dynamic variability in software-intensive embedded system families. IEEE Computer 45 (10), 28–35.
Brady, G., Sterritt, R., Monekosso, D., 2012. Autonomic robot for assisted living: supporting people with mild dementia. In: Engineering of Autonomic and Autonomous Systems (EASe), IEEE International Conference and Workshops, Novi Sad, Serbia.
Brataas, G., Jiang, S., Reichle, R., Geihs, K.,2011. Performance property prediction supporting variability for adaptive mobile systems. In: 5th International Workshop on Dynamic Software Product Lines (DSPL), 15th International Software Product Line Conference (SPLC 2011). ACM DL, 37.
Canfora, G., Di Penta, M., Esposito, R., Villani, M., 2008. A framework for QoS-aware binding and re-binding of composite services. Journal of Systems and Software 81 (10), 1754–1769.
Capilla, R., Sánchez, A., Dueñas, J.C., 2007. An analysis of variability modeling and management tools for product line development. In: Proceedings of the Software and Services Variability Management Workshop: Concepts, Techniques, and Tools, Helsinki University of Technology, Helsinki, Finland, April 19–20, pp. 32–47.
Capilla, R., Bosch, J., 2011. The promise and challenge of runtime variability. IEEE Computer 44 (12), 93–95.
Capilla, R., Bosch, J., Kang, K.-C., 2013. Systems and Software Variability Management: Concepts. Tools and Experiences. Springer, Berlin.
Cetina, C., Trinidad, P., Pelechano, V., Ruiz-Cortés, A., 2008. An architectural discussion on DSPL. In: 2th International Workshop on Dynamic Software Product Lines (DSPL), 12th International Software Product Line Conference (SPLC 2008), pp. 59–68.
Cetina, C., Haugen, Ø., Zhang, X., Fleurey, F., Pelechano, V., 2009a. Strategies for variability transformation at run-time. In: SPLC 2009, ACM Proceedings Series 446, San Francisco, CA, USA, pp. 61–70.
Cetina, C., Giner, P., Fons, J., Pelechano, V.,2009b. Using feature models for developing self-configuring smart homes. In: 5th International Conference on Autonomic and Autonomous Systems (ICAS 2009). IEEE CS, pp. 179–188.
Cetina, C., Giner, P., Fons, J., Pelechano, V., 2009c. Autonomic computing through reuse of variability models at runtime: the case of smart homes. IEEE Computer 42 (10), 37–43.
Cetina, C., Trinidad, P., Pelechano, V., Ruiz-Cortés, A., 2009d. Mass customization along lifecycle of autonomic homes. In: Proceedings of 3rd Dynamic Software Product Lines (DSPL), Limerick, Ireland.
Cetina, C., Giner, P., Fons, J., Pelechano, V., 2010. Designing and prototyping dynamic software product lines: techniques and guidelines. In: Software Product Line Conference (SPLC), pp. 331–345.
Clements, P., Northrop, L., 2001. Software Product Lines. Practices and Patterns. Addison-Wesley, Boston.
Cuadrado, F., Dueñas, J.C., García-Carmona, R., 2012. An autonomous engine for services configuration and deployment. IEEE Transactions on Software Engineering 38 (3), 520–536.
Deelstra, S., Sinnema, M., Bosch, J., 2009. Variability assessment in software product families. Information and Software Technology 51 (1), 195–218.
Elsner, C., Lohmann, D., Schröeder-Preikschat, W., 2009. Product derivation for solution-driven product line engineering. In: FOSD 2009, pp. 35–41.
Elsner, C., (Ph.D. dissertation) 2012. Automating Staged Product Derivation for Heterogeneous Multi-product Lines. Universität Erlangen-Nürnberg, Available at: http://www.opus.ub.unierlangen.de/opus/volltexte/2012/3265/pdf/Christoph ElsnerDissertation.pdf
Erradi, A., Maheshwari, P.,2008. Dynamic binding framework for adaptive web services. In: 3rd International Conference on Internet and Web Applications and Services. IEEE CS, pp. 162–167.
Fritsch, C., Lehn, A., Strohm, T., Bosch, R., 2002. Evaluating variability implementation mechanisms. In: Proceedings of International Workshop on Product Line Engineering (PLEES), pp. 59–64.
Froschauer, R., Zoitl, A., Grünbacher, P., 2009. Development and adaptation of IEC 61499 Automation and control applications with runtime variability models. In: 7th IEEE International Conference on Industrial Informatics (INDIN 2009), pp. 905–910.
Gámez, N., Fuentes, L., Aragüez, M.A.,2011. Autonomic computing driven by feature models and architecture in FamiWare. In: 5th European Conference on Software Architecture (ECSA), LNCS 6903. Springer-Verlag, pp. 164–179.
Garlan, D., Cheng, S.W., Huang, A.C., Schemerl, B.R., Steenkiste, P., 2004. Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37 (10), 46–54.
Georgas, J.C., van der Hoek, A., Taylor, R.N., 2009. Using architectural models to manage and visualize runtime adaptation. IEEE Computer 42 (10), 52–60.
Goedicke, M., Köllmann, C., Zdun, U., 2004. Designing runtime variation points in product line architectures: three cases. Science of Computer Programming 53 (3), 353–380.
Gomaa, H., Hussein, M., 2004. Dynamic software reconfiguration in software product families. In: van der Linden, F. (Ed.), Software Product Family Engineering. Lecture Notes in Computer Science 3014. Springer-Verlag, Berlin, Heidelberg, pp. 435–444.
Gomaa, H., Saleh, M., 2006. Feature-driven dynamic customization of software product lines. In: Morisio, M. (Ed.), Reuse of Off-the-Shelf Components. LNCS 4039. Springer-Verlag, pp. 58–72.
Gomaa, H., Hashimoto, K.,2011. Dynamic software adaptation for service-oriented product lines. In: 5th International Workshop on Dynamic Software Product Lines (DSPL), 15th International Software Product Line Conference (SPLC 2011). ACM DL, p. 35.
Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2008. Dynamic software product lines. IEEE Computer 41 (4), 93–95.
Hallsteinsen, S., Jiang, S., Sanders, R., 2009. Dynamic service product lines in service oriented computing. In: 3rd International Workshop on Dynamic Software Product Lines (DSPL), 13th International Software Product Line Conference (SPLC 2009).
Hartmann, H., Trew, T., 2008. Using feature diagrams with context variability to model multiple product lines for software supply chains. In: 12th International Software Product Line Conference, pp. 12–21.
Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A., 2008. Adding standardized variability to domain specific languages. In: Geppert, B., Pohl, K. (Eds.), Software Product Lines, 12th International Conference, SPLC 2008, vol. 1. Limerick, Ireland, September 8–12, 2008, pp. 139–148.
Helleboogh, A., Weyns, D., Schmid, K., Holvoet, T., Scheltfhout, K., van Betsbrugge, W., 2009. Adding variants on-the-fly: modeling meta-variability in dynamic software product lines. In: Proceedings of 3rd International Workshop on Dynamic Software Product Lines (DSPL 2009), San Francisco, CA, USA.
Hinchey, M., Park, S., Schmid, K., 2012. Building dynamic software product lines. IEEE Computer 45 (10), 22–26.
Istoan, P., Nain, G., Perrouin, G., Jézéquel, J.M.,2009. Dynamic software product lines for service-based systems. In: 9th International Conference on Computer and Information Technology. IEEE CS, pp. 193–198.
Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.,1990. Feature-oriented domain analysis (FODA) feasibility study. In: Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, CMU.
Kapitsaki, G.M., Prezerakos, G.N., Tselikas, N.D., Venieris, I.S., 2009. Context-aware service engineering: a survey. Journal of Systems and Software 82 (8), 1285–1297.
Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36 (1), 41–50.
Koning, M., Sun, C., Sinnema, M., Avgeriou, P., 2009. VxBPEL: supporting variability for web services in BPEL. Information and Software Technology 51 (2), 258–269.
Lee, J., Kang, K.,2004. Feature dependency analysis for product line component design. In: International Conference on Software Reuse, LNCS 3107. Springer-Verlag, pp. 69–85.
Lee, J., Kang, K., 2006. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: Proceeding of the 10th International Software Product Line Conference (SPLC), August 21–24, pp. 131–140.
Lee, J., Muthig, D., 2006. Feature-oriented variability management in product line engineering. Communications of the ACM 49 (12), 55–59.
Lee, J., Muthig, D.,2008. Feature-oriented analysis and specification of dynamic product reconfiguration. In: International Conference on Software Reuse (ICSR 2008), LNCS 5030. Springer-Verlag, pp. 154–165.
Lee, J., Kontoya, G., 2010. Combining service orientation with product line engineering. IEEE Software 27 (3), 35–41.
Lee, J., Kontoya, G., Robinson, D., 2012. Engineering service-based dynamic software product lines. IEEE Computer 45 (10), 28–35.
Malek, S., Medvidovic, N., Mikic-Rakic, M., 2012. An extensible framework for improving a distributed software system's deployment architecture. IEEE Transactions on Software Engineering 38 (1), 73–100.
Órtiz, O., García, A.B., Capilla, R., Bosch, J., Hinchey, M., 2012. Runtime variability for dynamic reconfiguration in wireless sensor product lines. In: 6th International Workshop on Dynamic Software Product Lines (DSPL), 16th International Software Product Line Conference (SPLC 2012).
Parra, C., Blanc, X., Duchien, L.,2009. Context awareness for dynamic service-oriented product lines. In: 13rd International Software Product Line Conference. ACM, pp. 131–140.
Peña, J., Hinchey, M.G., Resinas, M., Sterritt, R., Rash, J.L., 2007. Designing and managing evolving systems using a MAS product line approach. Journal of Science of Computing Programming 66 (1), 71–86.
Pleuss, A., Botterweck, G., Dhungana, D., 2010. Integrating automated product derivation and individual user interface design. In: 4th International

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038

Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), pp. 69–76.
Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering. Foundations, Principles, and Techniques. Springer, New York.
Raheja, R., Weng, S. -H., Garlan, D., Schmerl, B., 2009. Improving architecture-based self-adaptation using preemption. In: First International Workshop in Self-Organizing Architectures (SOAR 2009), pp. 21–37.
Raibulet, C., 2008. Facets of adaptivity. In: Proceedings of the 2nd European Conference on Software Architecture, LNCS 5292, pp. 342–345.
Rosenmüller, M., Siegmund, N., Apel, S.,2011. Tailoring dynamic software product lines. In: 10th International Conference on Generative Programming and Component Engineering (GPCE 20111). ACM, pp. 3–12.
Sawyer, P., Mazo, R., Díaz, D., Salinesi, C., Hughes, D., 2012. Using constraint programming to manage configurations in self-adaptative systems. IEEE Computer 45 (10), 56–63.
Schmid, K., Kröher, C., 2009. An analysis of existing configuration software systems. In: 3rd International Workshop on Dynamic Software Product Lines (DSPL), 13th International Software Product Line Conference (SPLC 2009).
Shokry, H., Ali Babar, M., 2008. Dynamic Software Product Line Architectures Using Service-based Computing for Automotive Systems.
Siegmund, N., Kuhleman, M., Rosenmüller, M., Kaestner, C., Saake, G., 2008. Integrated product line model for semi-automated product derivation using non-functional properties. In: 2nd Variability Modelling of Software Intensive Systems (VaMoS 2008), pp. 25–32.
Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., Toro, M., 2008a. Automated error analysis for the agilization of feature modelling. Journal of Systems and Software 81 (6), 883–896.
Trinidad, P., Benavides, D., Ruiz-Cortés, A., Segura, S., Jimenez, A., 2008b. Fama framework. In: Software Product Lines Conference, ISBN 978-1-905952-06-9.
Trinidad, P., (Ph.D. dissertation) 2012. Automating the Analysis of Stateful Feature Models. University of Seville.
Van der Hoek, A., 2004. Design-time product line architectures for anytime variability. Science of Computer Programming. Special Issue on Software Variability Management 53 (3), 285–304.
Van der Linden, F., Schmid, K., Rommes, E.,2007. Software product lines in action. In: The Best Industrial Practice in Product Line Engineering. Springer.
Vassev, E., Sterritt, R., Rouff, C., Hinchey, M.G., 2012. Swarm technology at NASA: building resilient systems. IT Professional 14 (2), 36–42.
White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A.,2007. Automating product-line variant selection for mobile devices. In: 11th International Software Product Line Conference (SPLC 2007). IEEE CS, pp. 129–140.
White, J., Benavides, D., Schmidt, D., Trinidad, P., Ruiz-Cortés, A., Dougherty, B., 2010. Automated diagnosis of feature model configurations. Journal of Systems and Software 83 (7), 1094–1107.

Rafael Capilla is an associate professor of the Faculty of Informatics at the Rey Juan Carlos University of Madrid and formerly in the Technical University of Madrid, Spain. Before joining the Rey Juan Carlos University, he worked for a Telecommunication company as software engineer and in a Computing Center at the University of Seville, Spain for more than nine years. He received a PhD in Computer Science from the Rey Juan Carlos University of Madrid. His research activities include software architecture, product line engineering, variability management, technical debt and Web technologies. He is co-author of a book "Systems and Software Variability Management: Concepts, Tools and Experiences" published by Springer, and co-author of several journal and conference research papers. He has been guest editor of journal issues and he actively participates in many conference program committees and reviewer in prestigious international journals. He also co-organized several workshops, general chair of the CSMR 2010 conference and participated in many Spanish and European R+D projects. Currently, he heads the Software Architecture & Internet Technologies (SAIT) research group which combines professional and research activities with the development of mobile software applications for companies.

Jan Bosch is professor of software engineering and director of the software research center at Chalmers University Technology in Gothenburg, Sweden. Earlier, he worked as Vice President Engineering Process at Intuit Inc where he also lead Intuit's Open Innovation efforts and headed the central mobile technologies team. Before Intuit, he was head of the Software and Application Technologies Laboratory at Nokia Research Center, Finland. Before joining Nokia, he headed the software engineering research group at the University of Groningen, The Netherlands, where he holds a professorship in software engineering. He received a MSc degree from the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden. His research activities include open innovation, innovation experiment systems, compositional software engineering, software ecosystems, software architecture, software product families and software variability management. He is the author of a book "Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach" published by Pearson Education (Addison-Wesley & ACM Press), (co-)editor of several books and volumes in, among others, the Springer LNCS series and (co-)author of a significant number of research articles. He is editor for Science of Computer Programming, has been guest editor for journal issues, chaired several conferences as general and program chair, served on many program committees and organized numerous workshops. In the startup space, Jan is chairman of the board of Fidesmo in Stockholm, and Remente, in Gothenburg, Sweden. He serves on the advisory board of Assia Inc. in Redwood City, CA, as well as the advisory board of Burt, in Gothenburg, Sweden. Also, he acts as an external business advisor for the School of Entrepreneurship at Chalmers University of Technology, Gothenburg, Sweden. As a consultant, as a professor and as an employee, Jan has worked with and for many companies on innovation and R&D management including Philips, Thales Naval Netherlands, Robert Bosch GmbH, Siemens, Nokia, Ericsson, Grundfos, Tellabs, Avaya, Tieto Enator and Det Norska Veritas. More information about his background can be found at his website: www.janbosch.com.

Pablo Trinidad is a senior lecturer at the University of Seville, where he received his PhD and MSc in computer science (with honours). He worked for several companies and worked as a freelance engineer before joining the academia in 2004. Dr. Trinidad is a member of the Applied Software Engineering research group, in which he researches on software product lines with an special attention to the automated analysis of variability models and the automated deployment and maintenance of dynamic products.

Antonio Ruiz-Cortés is accredited full professor and head of the Applied Software Engineering Group (ISA, www.isa.us.es) at University of Sevilla, Spain. He obtained his PhD (with Honours) in Computer Science from this University. His current research lines include service oriented computing, software product lines, and business process management. Previously, he worked on requirements engineering and multiparty interaction.

Mike Hinchey is Director of Lero - the Irish Software Engineering Research Centre and Professor of Software Engineering at University of Limerick, Ireland. He is also currently Vice President of the International Federation for Information Processing (IFIP) and a guest professor at University of Potsdam, Germany. He received a PhD in Computer Science from University of Cambridge, England; a M.Sc. in Computation from University of Oxford, England, and a B.Sc. in Computer Systems from University of Limerick, Ireland.

Please cite this article in press as: Capilla, R., et al., An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2013.12.038