Building Dynamic Software Product Lines

Mike Hinchey[1] and Sooyong Park[2] and Klaus Schmid[3]

[1]Lero?the Irish Software Engineering Research Centre, University of Limerick, Ireland
mike.hinchey@lero.ie

[2]Sogang University, South Korea
sypark@sogang.ac.kr

[3]Software Systems Engineering, Institute of Computer Science,
University of Hildesheim, Germany
schmid@sse.uni-hildesheim.de

Please cite this publication as follows:
Mike Hinchey, Sooyong Park, and Klaus Schmid. "Building Dynamic Software Product Lines". In: *IEEE Computer* 45.10 (2012), pp. 22–26. DOI: 10.1109/MC.2012.332.

The corresponding BibTeX-entry is:

```
@ARTICLE{HincheyParkSchmid12a,
  author = {Mike Hinchey and Sooyong Park and Klaus Schmid},
  title = {Building Dynamic Software Product Lines},
  journal = {IEEE Computer},
  year = {2012},
  volume = {45},
  pages = {22--26},
  number = {10},
  month = {October},
  doi = {10.1109/MC.2012.332},
}
```

This is the author-prepared version of the work. It is posted here by permission of the IEEE. It may differ significantly from the final version. The definitive version can be found in the IEEE Digital library; DOI: 10.1109/MC.2012.332

Building Dynamic Software Product Lines

Mike Hinchey[1], Sooyong Park[2], Klaus Schmid[3]

[1] Lero—the Irish Software Engineering Research Centre, University of Limerick, Ireland
[2] Sogang University, Seoul, South Korea
[3] University of Hildesheim, Germany

Dynamically Changing Software

Our world becomes increasingly complex and rapidly changing. This is not only a burden for people, but for software, too. Systems increasingly need to be capable to run continuously under adverse conditions (harsh environments, partial failures of subsystems, changing user needs, etc.). Often, they need to run unattended without interruption. Telecommunication systems and space systems are only rather extreme examples of this trend. More mundane applications like your smartphone, which ideally adapts its overall system behavior to energy levels, quality of network connection, etc. are also current challenges.

As a result of this type of challenges, we saw the advent of many different approaches to develop systems that are capable of adapting to changing needs, like self-adapting systems, agent-based systems, autonomous computing, emergent systems, bio-inspired systems, etc. While all these approaches share a fundamental theme: flexible adaptation of software to the needs, they also exhibit significant differences in terms of the range of adaptations supported, as well as the guarantees that can be expected for the various adapted systems.

A more recent addition to this category of approaches are *dynamic software product lines (DSPL)* [**Error! Reference source not found.**]. This kind of approaches extends existing product line engineering approaches by moving their capabilities to runtime. Due to the heritage of product lines one can rely on solid (and practicably) proven engineering foundations, which have been tested in numerous applications. This is a significant promise from an engineering perspective, as we need to be sure that adaptations of the system lead to desirable properties. So far DSPLs have been used in a wide range of application areas like household robotics, information systems, surveillance systems, industrial automation, etc.[1]

[1] Several such case studies were published at the International Workshop on Dynamic Software Product Lines. Information about this workshop series and the published papers can be found at dspl.lero.ie.

DSPL or not DSPL?

So what makes a DSPL special? In order to fully appreciate the concept, one needs to have a look at the concept of a product line first, as a DSPL is best characterized by its differences to the product line concept. Product line engineering (PLE) [2] is by now a well-known and widely used technology in industry [3]. Fundamentally, PLE is a reuse-oriented approach, which aims to develop assets in a way so that they can be reused successfully across a range of products: the so-called *software product line (SPL)*. A fundamental step towards this is to change the viewpoint from one product at a time to the product line as a whole. This required a set of innovations, among them:

- *Variability Modeling:* the differences among the systems in the product line are described explicitly. This also includes basic incompatibilities like certain characteristics of the system may not be present simultaneously. Typical approaches for variability modeling are feature models or decision models [4].
- *Reference Architecture:* a reference architecture is more than a standard software architecture. It is a software architecture, which covers the whole range of the product line by virtue of explicit customizations. Thus, it supports the variations as described by the variability model.
- *Business Scoping:* a product line architecture never addresses only a single customer or project, but rather a whole domain or business field. This needs to be sufficiently understood as a basis for product line design.
- *Two-Lifecycles:* a conceptual distinction is made between the development of capabilities *for reuse*, which need to be generic and easily composable to form a number of different systems, and development *with reuse*, which aims at creating products from the assets that have been created in the for-reuse-lifecycle.

If we look at today's successful industrial product lines, we invariably find most, if not all of these principles applied in practice. These principles cover a broad range of typical engineering activities, including modeling, technology, processes and planning.

The key idea of product line engineering is to have a well-designed set of assets that fit together (by means of the common reference architecture) as described by the variability model. Thus, the variability model takes a leading role, by describing in domain engineering the potential range of variations, while in application engineering it provides a basis for selecting the specific characteristics of the intended product. That all selectable combinations actually correspond to correctly working products is ensured by the reference architecture. This also provides the mechanisms for adequately composing the various parts at different points in time during the development, distribution and use of the systems (so-called *binding time*).

While the above explains what a SPL is, it does not tell us what a DSPL is. So, is a DSPL just a specific form of a SPL? Not quite. A main observation is that in a SPL the binding time typically is before runtime. It is typically during compile time, link time, perhaps even load time, when different modules are loaded to compose the system tailored to specific needs. The key question is now: can we use the same kinds of concepts and just move the binding time to runtime? Can we perhaps even bind and

rebind variability at runtime? This kind of ideas leads us directly to the concept of dynamic software product lines. In this case, we are not necessarily dealing with a whole product line in the traditional sense, but we may perceive the whole DSPL as a single system, adapting its behavior when variability is rebound during operation.

This simple transition, from a product line resulting in many different systems where variability is bound at development time to a system, which adapts by binding variability at runtime, has a number of repercussions. The variability is suddenly not only an engineering artifact, which is no longer present at runtime, but the DSPL needs to be able to consult – in one form or the other – the variability model at runtime to identify adaptations. This is needed, as the variability model is the core artifact to guide system adaptation. In SPL a reference architecture is a common framework and a software architecture only supports some parts of it (the selected variability). In a DSPL this becomes the DSPL architecture, which essentially is a system architecture, but supports the whole range of adaptation in an explicit and deterministic way. What is called a (business) scoping in a SPL also has a correspondence in a DSPL, albeit a more indirect one: we need to identify the range of potential adaptations of the DSPL (Adaptability scoping). This typically means we need to identify potential contexts and determine the potential triggers for adaptation and the set of potential reactions that shall be supported by the system. Even for the two-lifecycle approach, we can identify a correspondence: the domain engineering lifecycle, where the focus is on all possible variations corresponds to system construction in a DSPL, while the application engineering is no longer a typical engineering lifecycle, but rather corresponds to the use of the system. So, in part system configuration is done by the system itself. Table 11 summarizes this relationship.

(Classical) Software Product Line	**Dynamic Software Product Line**
Variability management describes different possible systems	Variability management describes different adaptations of the system
Reference architecture provides a common framework for a set of individual product architectures	DSPL architecture is a single system architecture, which provides a basis for all possible adaptations of the system
Business scoping identifies the common market for the set of products	Adaptability scoping identifies the range of adaptation supported by the DSPL
Two-lifecycle approach describes two engineering lifecycles, one for domain engineering and one for application engineering	Two lifecycles: the (DSPL) engineering lifecycle aims at a systematic development of the adaptive system and the usage lifecycle exploits the adaptability in use

Table 1: Relationship between SPL and DSPL

So, is a DSPL also a SPL? The answer depends. Of course, DSPLs can also be built as DSPLs. Some approaches support the combination of variabilities with different binding times (runtime & development time) within a single infrastructure. Some approaches even support different binding times for the same variability [7]. Thus, the border between a DSPL and a traditional SPL can be narrow to non-

existent. However, there is no need for a DSPL to lead to a number of identifiable, distinct systems at development time.

So, what are the core principles of a DSPL? Of course, as in any maturing field there is not yet a final agreement, but it seems the following principles are currently widely accepted. A DSPL ..

- contains an explicit representation of the configuration space and constraints that describe permissible runtime reconfigurations on the level of intended capabilities of the system (as opposed to a technical level). This is a variability model in the sense that discrete options are identified (as opposed to continuous models. This differentiates a DSPL clearly from a number of other adaptation approaches.
- is capable of performing the necessary reconfiguration of the system autonomously. Thus, the derivation of different system instances needs to be autonomous, once the intended configuration is known. The environment monitoring and subsequent identification of a desired configuration is explicitly not necessary to qualify as a DSPL.
- is developed based on an explicit engineering process that takes the scope of variability explicitly into account. Here, the principles and approaches of the engineering of "classical" SPL can be exploited.

What is not required for a system to qualify as a DSPL?

- It need not be built as part of a (development time) SPL. This may seem puzzling at first, but it simply means the whole variability may be taken care of at runtime.
- Its adaptation space is at least implicitly pre-defined by its variability model and as variability models are discrete models any problem requiring some form of analogous or continuous modeling is not appropriate for them.
- Purely parameterized adaptation is not considered a DSPL; rather a modification of the actually executing system implementation is required. Ideally, a reconfiguration of architectural components is performed.
- It need not be self-adapting. It is completely valid that the decision to perform a reconfiguration is made on an abstract level by a human. However, all aspects of realizing this reconfiguration must be handled by the DSPL without human intervention at runtime.

While autonomous behavior is not required for a DSPL, most DSPLs today actually aim at closing the control-loop by addressing also monitoring and decision-making. Thus, while DSPLs are often used as a systematic engineering approach to realize self-adaptive systems, this need not be so. There exist DSPLs, which are not self-adaptive systems and self-adaptive systems, which are not DSPLs.

DSPLs as Adaptive Systems

While not necessarily adaptive systems, DSPLs are often taken as a systematic engineering approach to realize adaptive systems. It thus makes sense, to compare DSPLs with other approaches to developing adaptive systems. In [6], Bencomo et al. compare a number of existing DSPL realizations to the MAPE-K cycle, showing that some of them successfully realize this model. As a consequence some DSPLs also

qualify as an autonomous system. However, usually, they will not qualify as emergent systems, as the basic approach underlying adaptivity in a DSPL is the systematic engineering of adaptation options (variability). Thus, DSPLs are operating in a predefined (even though potentially infinite) configuration space. However, some research work also addresses lifting the lid on this one: they aim to even address the evolution of the DSPL autonomously. In this case, the system aims to improve even its own variability model. As a consequence while DSPL is per se not an approach leading to emergent behavior, it can be combined with approaches to support exactly this. This can then be interpreted as autonomous evolution.

The strength of DSPL, however, is the systematic engineering foundations that can be brought to bear on adaptive and self-adaptive systems. Thus leading to higher reliability and predictability for adaptive systems.

As a result DSPLs share also a weakness with more traditionally engineered systems: their evolution capabilities. If a modification beyond the initially provided configuration space is needed, the implementation of the system itself needs to be modified. However, even this can become easier using a DSPL-approach: by exploiting the variability model and augmenting it (so-called open variability), it is possible to integrate further adaptation capabilities.

Research Issues

As an evolving field, there are still many open research issues [7]. We only want to point out some of them:

- As described above, so far little support exists for DSPL-evolution. Here approaches are needed that explicitly exploit the characteristics of a DSPL and leverage it, to become more successful at evolution. This may include automated (learning-based) evolution
- While approaches to the implementation of development-time configuration are today well understood, the DSPL-approaches for runtime reconfiguration are still an important research area, especially, with respect to their relation to the overall system architecture.
- While the core variability model focuses on the modeling of the reconfiguration options, ongoing research in the DSPL-area attempts to enlarge these approaches to capture the context description and decision making part as well.
- Similar to “classical” product line engineering DSPL-reconfiguration focused so far mainly functional capabilities, while addressing quality characteristics only to a limited extend.
- How to deal at runtime with over-constrained situations? For example, we might identify that several available capabilities are simultaneously required, but at the same time the variability model defines them as mutually exclusive.
- How to best extend variability modeling, so that it can be used as a basis for context interpretation and thus supports the fully autonomous case.

In this Issue

The various contributions in this special issue illustrate the topic of dynamic software product lines from a wide range of perspectives. We start with a paper by Capilla and Bosch, which discusses current industry trends and needs and why this leads to a general need for more dynamic software product lines in practice. They also aim to provide a categorization of different types of variability that may occur in a DSPL.

A different form of characterization of DSPLs is provided by Bencomo et al. They provide an overview of the existing landscape of the work on dynamic software product lines, based on a study of the existing literature in this area. In particular, they map this work in terms of its types of its contributions and its maturity.

An important theme in DSPL is the convergence between DSPL-approaches and other approaches that aim at more dynamic systems. Two of the contributions in this special issue aim into this direction and focus explicitly at combining work in service-oriented computing with product line engineering and DSPL work.

The paper by Baresi et al. aims at dynamic adaptations of workflows, for which they define the DyBPEL-extension to the Business Process Execution Language (BPEL). They combine this with a variability model based on the CVL (common variability language) and use it to manage variability of workflows in service-oriented systems at runtime.

While the input for a new configuration comes in the previous case from a variability designer who explicitly updates the selected configuration, the work by Lee at al. is closer to the vision of autonomic computing and aims at performing reconfiguration autonomously based on available services with a focus on the provided quality of service. Interesting in this case is that services may be added or removed from the context at runtime and this is dynamically taken into account.

Finally, the work by Sawyer et al. takes a rather unusual route towards the realization of a DSPL. While most work focuses on typical variability models, like feature models, here the configuration space is described by a goal model (based on KAOS). They do also aim at autonomous reconfiguration, where the optimal configuration is identified using a constraint solver. Their approach is illustrated by using a wireless sensor network.

While this issue with its selected contributions provides a broad overview of current work on DSPL, the picture is by no means complete or exhaustive. Moreover, as dynamically adaptive systems become more and more important, approaches to construct such systems like the DSPL-approach, will become more important and more widely used as well.

Acknowledgements

Sooyong Park was partly supported by the Next-Generation Information Computing Development Program through the NRF, funded by the MEST 2012M3C4A7033348. The research by Klaus Schmid was partly supported by the EU-project INDENICA.

6. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero—the Irish Software Engineering Research Centre (www.lero.ie).

References

1. S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid. *Dynamic Software Product Lines.* IEEE Computer, Vol. 41, No. 4, pp. 93-95, 2008.
2. P. Clements, L. Northrop. *Software Product Lines*, Addison Wesley, 2002.
3. *The product line hall of fame*. Online available at: splc.net/fame.html (checked: 3.9.12)
4. K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski. *Cool features and tough decisions: a comparison of variability modeling approaches*. Proceedings of the 6th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12), ACM, pp. 173-182, 2012.
5. K. Schmid, H. Eichelberger. *Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects*, Second International Workshop on Variability Modeling of Software-Intensive Systems (VAMOS), ICB-Research Report No. 22, ISSN 1860-2770, pp. 63-71, 2008
6. N. Bencomo, J. Lee, S. Hallsteinsen: *How dynamic is your Dynamic Software Product Line?* Workshop on Dynamic Software Product Lines, 61-68, 2010.
7. A. Classen, A. Hubaux, F. Sanen, E. Truyen, J. Vallejos, P. Costanza, W. Meuter, P. Heymans, W. Joosen. *Modelling Variability in Self-Adaptive systems: Towards a Research Agenda.* Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE) in conjuction with GPCE / OOPSLA, 2008.