Overview of Generative Software Development

Krzysztof Czarnecki

University of Waterloo, Canada
czarnecki@acm.org

Abstract. System family engineering seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way. In system family engineering, new system variants can be rapidly created based on a set of reusable assets (such as a common architecture, components, models, etc.). Generative software development aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages. This paper gives an overview of the basic concepts and ideas of generative software development including DSLs, domain and application engineering, generative domain models, networks of domains, and technology projections. The paper also discusses the relationship of generative software development to other emerging areas such as Model Driven Development and Aspect-Oriented Software Development.

1 Introduction

Object-orientation is recognized as an important advance in software technology, particularly in modeling complex phenomena more easily than its predecessors [1]. But the progress in reusability, maintainability, reliability, and even expressiveness has fallen short of expectations. As units of reuse, classes have proven too small. Frameworks are hard to compose, and their development remains an art. Components—as independently-deployable units of composition with contractually specified interfaces [2]—offer reuse, but the more functional the component, the larger and less reusable it becomes. And patterns, while intrinsically reusable, are not an implementation medium.

Current research and practical experience suggest that achieving significant progress with respect to software reuse requires a paradigm shift towards modeling and developing software system families rather than individual systems. *System-family engineering* (also known as *product-line engineering*) seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way [3–5]. In system-family engineering, new system variants can be rapidly created based on a set of reusable assets (such as a common architecture, components, models, etc.).[1]

[1] System-family engineering is mainly concerned with building systems from common assets, whereas product-line engineering additionally considers scoping and managing common product characteristics from the market perspective. In order to be more general, this paper adheres to system-family terminology.

Frameworks and components are still useful as implementation technologies, but the scope and shape of reusable abstractions is determined and managed through a system-family approach.

Generative software development is a system-family approach, which focuses on automating the creation of system-family members: a given system can be automatically generated from a specification written in one or more textual or graphical *domain-specific languages* [3, 6–11].

This paper gives an overview of the basic concepts and ideas of generative software development including DSLs, domain and application engineering, generative domain models, networks of domains, and technology projections. The paper closes by discussing the relationship of generative software development to other emerging areas such as Model Driven Development and Aspect-Oriented Software Development.

2 Domain-Specific Languages

A domain-specific language (DSL) is a language offering expressive power focused on a particular problem domain, such as a specific class of applications or application aspect. Whereas general-purpose programming languages such as Java or C++ were designed to be appropriate for virtually any kind of applications, DSLs simplify the development of applications in specialized domains at the cost of their generality.

DSLs are certainly not a new idea. In fact, before common programming abstractions were identified and packaged into general-purpose programming languages, many of the early computer languages were application-specific. For example, in his landmark paper "The Next 700 Hundred Programming Languages", Landin [12] cites a 1965 Prospectus of the American Mathematical Association: "... today... 1,700 special programming languages used to 'communicate' in over 700 application areas." Although many DSLs have been developed over the years, the systematic study of DSLs is more recent, e.g., [6, 13–15].

The domain specificity of a language is a matter of degree. While any language has a certain scope of applicability, some languages are more focused than others. Programming languages such as Fortran or Cobol, although designed with some application focus in mind, are still fairly general. For example, Fortran was designed to target mathematical applications, but it can be used to program anything from databases to user interfaces. When referring to DSLs, we consider much more focused languages, such as HTML or SQL. In fact, a great share of existing DSLs are not even programming languages [16].

Narrowing the application scope of a language allows us to provide better support for solving problems within the scope compared to what a general purpose programming language could offer. A DSL can offer several important advantages over a general-purpose language:

– *Domain-specific abstractions*: a DSL provides pre-defined abstractions to directly represent concepts from the application domain.

– *Domain-specific concrete syntax*: a DSL offers a natural notation for a given domain and avoids syntactic clutter that often results when using a general-purpose language.
– *Domain-specific error checking*: a DSL enables building static analyzers that can find more errors than similar analyzers for a general-purpose language and that can report the errors in a language familiar to the domain expert.
– *Domain-specific optimizations*: a DSL creates opportunities for generating optimized code based on domain-specific knowledge, which is usually not available to a compiler for a general-purpose language.
– *Domain-specific tool support*: a DSL creates opportunities to improve any tooling aspect of a development environment, including, editors, debuggers, version control, etc.; the domain-specific knowledge that is explicitly captured by a DSL can be used to provide more intelligent tool support for developers.

The traditional approach to providing domain-specific abstractions in programming languages is through libraries of user-defined functions, classes, and data structures. We consider the application programming interfaces (APIs) exposed by such libraries as a possible implementation form for DSLs. User-defined abstractions is a way to extend a language with domain-specific vocabulary, and library and API design is a form of language design. Of course, open-ended language design is more challenging than API design, which is constrained and guided by the host language. At the same time, while satisfying the first benefit in the list above, traditional libraries and APIs usually come short on the other items, such as domain-specific notation (beyond operator overloading, which may be available in the host language), error checking, and optimizations. Achieving the latter goals usually requires some form of metaprogramming.

DSLs come in a wide variety of forms, e.g., textual languages (stand-alone or embedded in a general-purpose programming language), diagrammatic languages, form-based languages, grid-based languages, etc. Section 6 lists different DSLs implementation technologies.

3 Domain Engineering and Application Engineering

System family engineering distinguishes between at least two kinds of development processes: *domain engineering* and *application engineering* (see Figure 1). Typically, there is also a third process, *management*, but this paper focuses on the two development processes (for more information on process issues see [3,4]). Generative software development, as a system-family approach, subscribes to the process model in Figure 1, too.

Domain engineering (also known as *product-line development* or *core asset development*) is "development for reuse". It is concerned with the development of reusable assets such as components, generators, DSLs, analysis and design models, user documentation, etc. Similar to single-system engineering, domain engineering also includes analysis, design, and implementation activities. How-

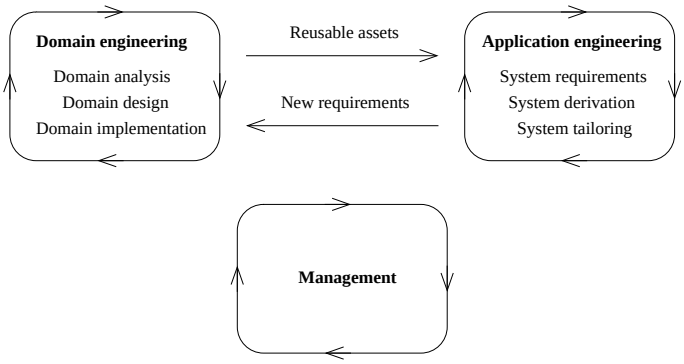

Fig. 1. Main processes in system-family engineering

ever, these are focused on a class of systems rather than just a single system.[2] *Domain analysis* involves determining the scope of the family to be built, identifying the common and variable features among the family members, and creating structural and behavioral specifications of the family. *Domain design* covers the development of a common architecture for all the members of the system family and a plan of how individual systems will be created based on the reusable assets. Finally, *domain implementation* involves implementing reusable assets such as components, generators, and DSLs.

Application engineering (also referred to as *product development*) is "development with reuse", where concrete applications are built using the reusable assets. Just as traditional system engineering, it starts with requirements elicitation, analysis, and specification; however, the requirements are specified as a delta from or configuration of some generic system requirements produced in domain engineering. The requirements specification is the main input for *system derivation*, which is the manual or automated construction of the system from the reusable assets.

Both processes feed on each other: domain-engineering supplies application engineering with the reusable assets, whereas application engineering feeds back new requirements to domain engineering. This is so because application engineers identify the requirements for each given system to be built and may be faced with requirements that are not covered by the existing reusable assets. Therefore, some amount of application-specific development or *tailoring* is often required in order to quickly respond to the customer's needs. However, the new requirements should be fed back into domain engineering in order to keep the reusable assets in sync with the product needs. Different models for setting up these processes

[2] Both terms "system family" and "domain" imply a class of systems; however, whereas the former denotes the actual set of systems, the latter refers more to the related area of knowledge. The use of the one or the other in compounds such as "domain engineering" is mostly historical.

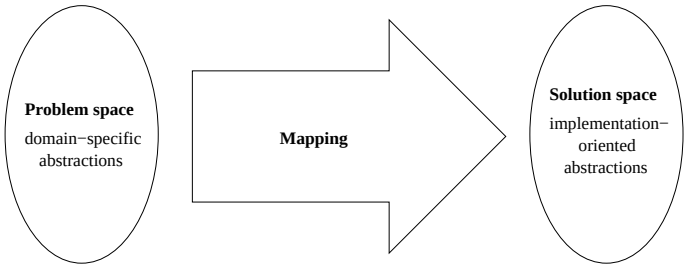

Fig. 2. Mapping between problem space and solution space

in an organization, e.g., separate or joint product-development and domain-engineering teams, are discussed in [17].

Domain engineering can be applied at different levels of maturity. At minimum, domain analysis activities can be used to establish a common terminology among different product-development teams. The next level is to introduce a common architecture for a set of systems. Further advancement is to provide a set of components covering parts or all of the systems in the system family. Finally, the assembly of these components can be partially or fully automated using generators and/or configurators. The last level represents the focus of generative software development. In general, the generated products may also contain non-software artifacts, such as test plans, manuals, tutorials, maintenance guidelines, etc.

4 Mapping Between Problem Space and Solution Space

A key concept in generative software development is that of a mapping between *problem space* and *solution space* (see Figure 2), which is also referred to as a *generative domain model*. Problem space is a set of domain-specific abstractions that can be used to specify the desired system-family member. By "domain-specific" we mean that these abstractions are specialized to allow application engineers to express their needs in a way that is natural for their domain. For example, we might want to be able to specify payment methods for an electronic commerce system or matrix shapes in matrix calculations. The solution space, on the other hand, consists of implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space. For example, payment methods can be implemented as calls to appropriate web services, and different matrix shapes may be realized using different data structures. The mapping between the spaces takes a specification and returns the corresponding implementation.

4.1 Configuration and Transformation

There are at least two different views at the mapping between problem space and solution space in generative software development: *configuration* view and *transformational* view.

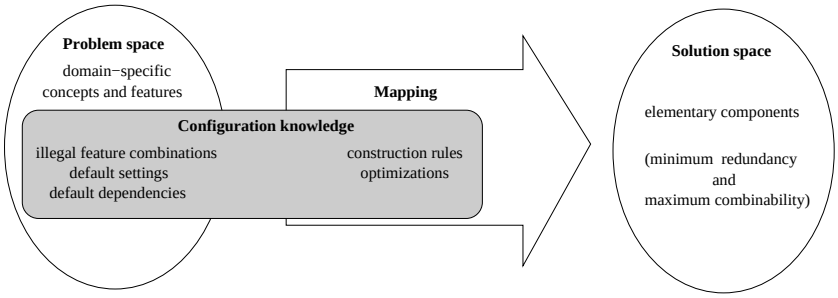

Fig. 3. Configuration view on the mapping between problem space and solution space

In the configuration view, the problem space consists of domain-specific concepts and their features (see Figure 3). The specification of a given system requires the selection of features that the desired system should have. The problem space also defines illegal feature combinations, default settings, and default dependencies (some defaults may be computed based on some other features). The solution space consists of a set of implementation components, which can be composed to create system implementations. A system-family architecture sets out the rules how the components can be composed. In the configuration view, an application programmer creates a configuration of features by selecting the desired ones, which then is mapped to a configuration of components. The mapping between both spaces is defined by construction rules (certain configurations of features translate into certain configurations of implementation components) and optimizations (some component configurations may have better non-functional properties then others). The mapping plus the illegal feature combinations, default settings, and default dependencies collectively constitute *configuration knowledge*. Observe that the separation between problem and solution space affords us the freedom to structure abstractions in both spaces differently. In particular, we can focus on optimally supporting application programmers in the problem space, while achieving reuse and flexibility in the solution space.

In the transformational view, a problem space is represented by a domain-specific language, whereas the solution space is represented by an implementation language (see Figure 4). The mapping between the spaces is a transformation that takes a program in a domain-specific language and yields its implementation in the implementation language. A domain-specific language is a language specialized for a given class of problems. Of course, the implementation language

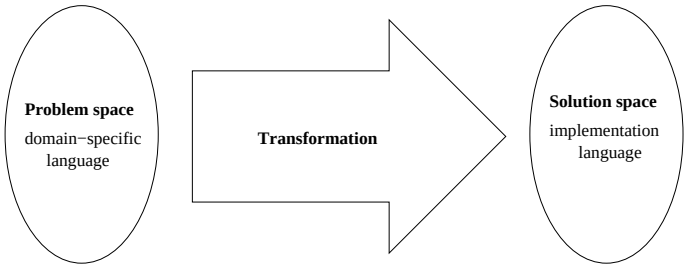

Fig. 4. Transformational view on the mapping between problem space and solution space

may be a domain-specific language exposed by another domain. The transformational view directly corresponds to the Draco model of domains and software generation [6].

Despite the superficial differences, there is a close correspondence between both views. The problem space with its common and variable features and constraints in the configuration view defines a domain-specific language, and the components in the solution space can also be viewed as an implementation language. For example, in the case of generic components, we can specify this target language as a GenVoca grammar with additional well-formedness constraints [8,18]. Thus, the configuration view can also be interpreted as a mapping between languages.

The two views relate and integrate several powerful concepts from software engineering, such as domain-specific languages, system families, feature modeling, generators, components, and software architecture. Furthermore, the translation view provides a theoretical foundation for generative software development by connecting it to a large body of existing knowledge on language theory and language translation.

4.2 Network of Domains

Observe that Figure 2 can be viewed recursively, i.e., someone's problem space may be someone else's solution space. Thus, we can have chaining of mappings (see Figure 5 a). Furthermore, a mapping could take two or more specifications and map them to one (or more) solution space (see Figure 5 b). This is common when different aspects of a system are represented using different DSLs. A mapping can also implement a problem space in terms of two or more solution spaces (see Figure 5 c). Finally, different alternative DSLs (e.g., one for beginners and one for expert users) can be mapped to the same solution space (see Figure 5 d), and the same DSL can have alternative implementations by mappings to different solution spaces (e.g., alternative implementation platforms; see Figure 5 e).

In general, spaces and mappings may form a hypergraph, which can even contain cycles. This graph corresponds to the idea of a *network of domains*

by Jim Neighbors [6], where each implementation of a domain exposes a DSL, which may be implemented by transformations to DSLs exposed by other domain implementations.

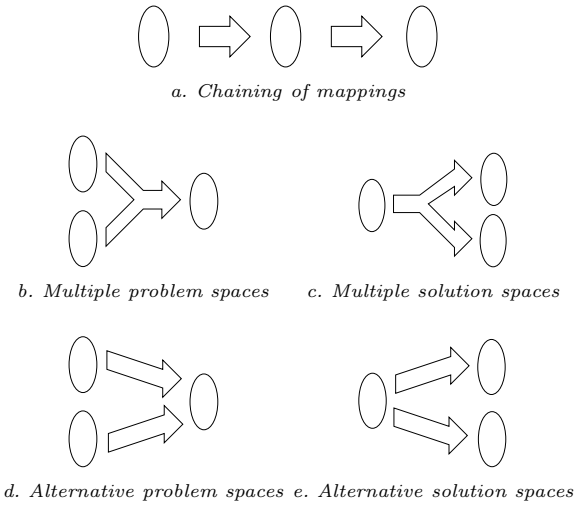

Fig. 5. Different arrangements of mappings between problem and solution spaces

5 Feature Modeling and Feature-Oriented Approach

Feature modeling is a method and notation to elicit and represent common and variable features of the systems in a system family. Feature modeling was first proposed by Kang et al in [19] and since then has been extended with several concepts, e.g., feature and group cardinalities, attributes, and diagram references [20].

An example of a feature model is shown in Figure 6. The model expresses that an electronic commerce system supports one or more different payment methods; it provides tax calculation taking into account either the street-level address, or postal code, or just the country; and it may or may not support shipment of physical goods. A feature diagram such as in Figure 6 may be supplemented with additional information including constraints (selecting a certain feature may require or exclude the selection of another feature), binding times (features may be intended to be selected at certain points in time), default attribute values and default features, stakeholders interested in a given feature, priorities, and more. Features may or may not correspond to concrete software modules. In general, we distinguish the following four cases:

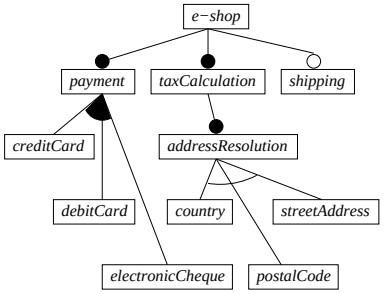

Fig. 6. Example of a feature diagram

- *Concrete* features such as data storage or sorting may be realized as individual components.
- *Aspectual* features such as logging, synchronization, or persistency may affect a number of components and can be modularized using aspect technologies.
- *Abstract* features such as performance requirements usually map to some configuration of components and/or aspects.
- *Grouping* features may represent a variation point and map to a common interface of plug-compatible components, or they may have a purely organizational purpose with no requirements implied.

Feature modeling gives rise to a feature-oriented approach to generative software developement [8]. In the early stages of software family development, feature models provide the basis for scoping a system family by recording and assessing information such as which features are important to enter a new market or remain in an existing market, which features incur a technological risk, what is the projected development cost of each feature, and so forth [21]. Subsequently, feature models created in domain analysis are the starting point in the development of both system-family architecture and DSLs (see Figure 7). Architecture development takes a solution-space perspective at the feature models: it concentrates on the concrete and aspectual features that need to be implemented as components and aspects. Familiar architectural patterns, such as in [22, 23], can be applied, but with the special consideration that the variation points expressed in the feature models need to be realized in the architecture. During subsequent DSL development, a problem-space perspective concentrating on features that should be exposed to application developers determines the required DSL scope, possibly requiring additional abstract features.

6 Technology Projections and Structure of DSLs

Each of the elements of a generative domain model can be implemented using different technologies, which gives rise to different *technology projections*:

- DSLs can be implemented as new textual languages (using traditional compiler building tools), embedded in a programming language (e.g., template

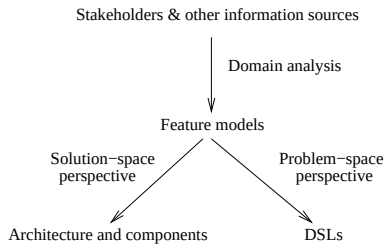

Fig. 7. Feature-oriented approach

metaprogramming in C++ or Template Haskell [24], OpenC++ [26], OpenJava [25], Metaborg [27]), graphical languages (e.g., UML profiles [28], GME [29], MetaEdit+ [30], or Microsoft's DSL Technology in VisualStudio), wizards and interactive GUIs (e.g., feature-based configurators such as FeatureModelingPlugin [31], Pure::Consul [32], or CaptainFeature [33]), or some combination of the previous. The appropriate structure of a DSL and the implementation technology depend on the range of variation that needs to be supported (see Figure 8). The spectrum ranges from routine configuration using wizards to programming using graphical or textual graph-like languages.

- Mappings can be realized using product configurators (e.g., Pure::Consul) or generators. The latter can be implemented using template and frame processors (e.g., TL [9], XVCL [34], or ANGIE [35]), transformation systems (e.g., DMS [36], StrategoXT [37], or TXL [38]), multi-staged programming [39], program specialization [40–42], or built-in metaprogramming capabilities of a language (e.g., template metaprogramming in C++ or Template Haskell).
- Components can be implemented using simply functions or classes, generic components (such as in the C++ Standard Template Library), component models (e.g., JavaBeans, ActiveX, or CORBA), or aspect-oriented programming approaches (e.g., AspectJ [43], HyperJ [44], or Caesar [45]).

While some technologies cover all elements of a generative domain model in one piece (e.g., OpenJava or template metaprogramming in C++), a more flexible approach is to use an intermediate program representation to allow using different DSL renderings (e.g., textual or graphical) with different generator back-ends (e.g., TL or StrategoXT).

The choice of a specific technology depends on its technical suitability for a given problem domain and target users. For example, in the case of DSLs, concise textual languages may be best appropriate for expert users, but wizards may be better suited for novices and infrequent users. In the case of generator technologies, the need for complex, algebraic transformations may require using a transformation system instead of a template processor. Furthermore, there may be non-technical selection criteria such as mandated programming languages, existing infrastructure, familiarity of the developers with the technology, political and other considerations.

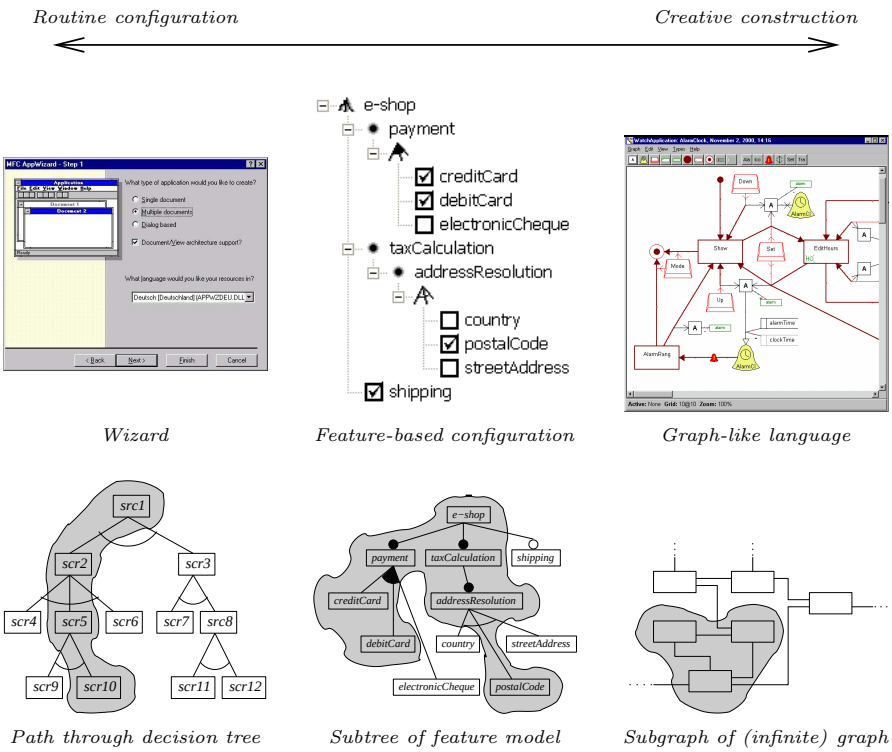

Fig. 8. Spectrum of DSL structures

7 Model Driven Development

Perhaps the closest related area to generative software development is model-driven development (MDD), which aims at capturing every important aspect of a software system through appropriate models. A model is an abstract representation of a system and the portion of the world that interacts with it. Models allow answering questions about the software system and its world portion that are of interest to the stakeholders. They are better than the implementing code for answering these questions because they capture the intentions of the stakeholders more directly, are freer from accidental implementation details, and are more amenable to analysis. In MDD, models are not just auxiliary documentation artifacts; rather, models can be compiled directly into executable code that can be deployed at the customer's site.

There has been a trend in MDD towards representing models using appropriate DSLs, which makes MDD and generative software development closely related. Perhaps the main difference between MDD and generative software development is the focus of the latter on system families. While system families can be of interest to MDD, they are not regarded as a necessity.

Model-Driven Architecture (MDA) is a framework for MDD proposed by the Object Management Group (OMG) [46]. While still being defined, the main goal of MDA is to allow developers to express applications independently of specific implementation platforms (such as a given programming language or middleware). In MDA, an application is represented as a Platform Independent Model (PIM) that later gets successively transformed into series of Platform Specific Models (PSMs), finally arriving at the executable code for a given platform. The models are expressed using UML and the framework uses other related OMG standards such as MOF, CWM, XMI, etc. A standard for model transformations is work in progress in response to the Request for Proposals "MOF 2.0 Query/Views/Transformations" issued by OMG.

MDA concepts can be mapped directly onto concepts from generative software development: a mapping from PIM to PSM corresponds to a mapping from problem space to solution space. Beyond the similarities, there are interesting synergies. On the one hand, benefits of MDA include a set of standards for defining and manipulating modeling languages and the popularization of generative concepts in practice. Thanks to MDA, current UML modeling tools are likely to evolve towards low-cost DSL construction tools. On the other hand, the MDA efforts until now have been focusing on achieving platform independence, i.e., system families with respect to technology variation. However, generative software development addresses both technical and application-domain variability, and it may provide valuable contributions to MDA in this respect (see Figure 9). Often asked questions in the MDA context are (1) what UML profiles or DSLs should be used to represent PIMs and (2) what is a platform in a given context. Domain analysis and domain scoping can help us to address these questions.

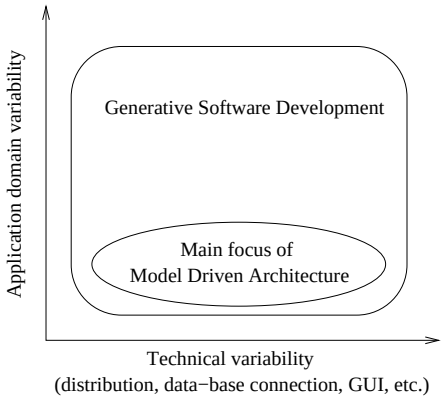

Fig. 9. Relationship between generative software development and MDA

8 Other Related Fields

Figure 10 classifies a number of related fields by casting them against the elements of a generative domain model. Components, architectures, and generic programming are primarily related to the solution space. Aspect-oriented programming provides more powerful localization and encapsulation mechanisms than traditional component technologies. In particular, it allows us to replace many "little, scattered components" (such as those needed for logging or synchronization) and the configuration knowledge related to these components by well encapsulated aspectual modules. However, we still need to configure aspects and other components to implement abstract features such as performance properties. Therefore, aspect-oriented programming technologies such as AspectJ cover the solution space and only a part of the configuration knowledge. But aspects can also be found in the problem space, esp. in the context of DSLs used to described different aspects of a single system. Areas such as DSLs, feature modeling, and feature interactions address the problem space and the front part of the configuration knowledge. Finally, system-family and product-line engineering span across the entire generative domain model because they provide the overall structure of the development process (including domain and application engineering).

9 Concluding Remarks

Generative software development builds upon and exploits the synergies among several key concepts:

1. Software system families are the key to achieving systematic software reuse.
2. Domain-specific languages are about providing optimal support for application developers.

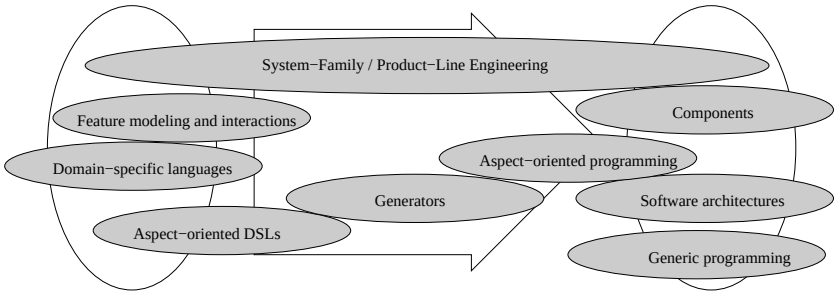

Fig. 10. Relationship between generative software development and other fields (from [47])

3. Mappings enable design knowledge capture.
4. Aspect-oriented development provides better separation of concerns and composition mechanisms.
5. Feature modeling aids family scoping, and DSL and architecture development.

References

1. Meyer, B.: Object-Oriented Software Construction. Second edn. Prentice Hall, Upper Saddle River, NJ (1997)
2. Szyperski, C.: Component Software—Beyond Object-Oriented Programming. Second edn. Addison-Wesley / ACM Press, Boston, MA (2002)
3. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley (1999)
4. Clements, P., Northrop, L., eds.: Software Product Lines: Practices and Patterns. International Series in Computer Science. Addison-Wesley (2001)
5. Parnas, D.: On the design and development of program families. IEEE Transactions on Software Engineering **SE-2** (1976) 1–9
6. Neighbors, J.M.: Software Construction using Components. PhD thesis, Department of Information and Computer Science, University of California, Irvine (1980) Technical Report UCI-ICS-TR160. Available from `http://www.bayfronttechnologies.com/thesis.pdf`.
7. Cleaveland, J.C.: Building application generators. IEEE Software **9** (1988) 25–33
8. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
9. Cleaveland, C.: Program Generators with XML and Java. Prentice-Hall (2001)
10. Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: A case study. ACM Transactions on Software Engineering and Methodology (TOSEM) **11** (2002) 191–214
11. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Indianapolis, IN (2004)
12. Landin, P.J.: The next 700 programming languages. Commun. ACM **9** (1966) 157–166
13. Bentley, J.L.: Little languages. Communications og the ACM **29** (1986) 711–721

14. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35** (2000) 26–36
15. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. Technical Report SEN-E0309, CWI, Amsterdam (2003) Available from `http://www.cwi.nl/ftp/CWIreports/SEN/SEN-E0309.pdf`.
16. Wile, D.S.: Supporting the dsl spectrum. CIT Journal of Computing and Information Technology **9** (2001) 263–287
17. Bosch, J.: Software product lines: Organizational alternatives. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE). (2001)
18. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology **1** (1992) 355–398
19. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR -21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
20. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In Nord, R.L., ed.: Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings. Volume 3154 of Lecture Notes in Computer Science., Springer-Verlag (2004) 266–283
21. DeBaud, J.M., Schmid, K.: A systematic approach to derive the scope of software product lines. In: Proceedings of the 21st International Conference on Software Engineering (ICSE), IEEE Computer Society Press (1999) 34–43
22. Buschmann, F., Jkel, C., Meunier, R., Rohnert, H., Stahl, M., eds.: Pattern-Oriented Software Architecture – A System of Patterns. International Series in Computer Science. John Wiley & Sons (1996)
23. Bosch, J.: Design and Use of Software Architecture: Adopting and evolving a product-line approach. Addison-Wesley (2000)
24. Czarnecki, K., O'Donnel, J., Striegnitz, J., Taha, W.: Dsl implementation in metaocaml, template haskell, and c++. [48] 50–71
25. M. Tatsubori: OpenJava: An extensible Java (2004) Available at `http://sourceforge.net/projects/openjava/`.
26. Sigeru Chiba: OpenC++ (2004) Available at `http://opencxx.sourceforge.net/index.shtml`.
27. Bravenboer, M., Visser, E.: Concrete syntax for objects. domain-specific language embedding and assimilation without restrictions. In C.Schmidt, D., ed.: Proceedings of the 19th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04). Vancouver, Canada. October 2004, ACM SIGPLAN (2004)
28. Jeff Grey et al.: OOPSLA'02 Workshop on Domain-Specific Visual Languages (2002) Online proceedings at `http://www.cis.uab.edu/info/OOPSLA-DSVL2/`.
29. Lédeczi, Á., Árpád Bakay, Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Computer **34** (2001) 44–51
30. MetaCase, Jyväskylä, Finland: MetaEdit+ User Manual. (2004) Available from `http://www.metacase.com`.
31. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature modeling plug-in for Eclipse. In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop. (2004) Paper available from `http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf`. Software available from `gp.uwaterloo.ca/fmp`.
32. pure-systems GmbH: Variant management with pure::consul. Technical White Paper. Available from `http://web.pure-systems.com` (2003)

33. Bednasch, T., Endler, C., Lang, M.: CaptainFeature (2002-2004) Tool available on SourceForge at `https://sourceforge.net/projects/captainfeature/`.
34. Wong, T., Jarzabek, S., Swe, S.M., Shen, R., Zhang, H.: Xml implementation of frame processor. In: Proceedings of the ACM Symposium on Software Reusability (SSR'01), Toronto, Canada, May 2001. (2001) 164–172 `http://fxvcl.sourceforge.net/`.
35. Delta Software Technology GmbH: ANGIE - A New Generator Engine (2004) Available at `http://www.delta-software-technology.com/GP/gptop.htm`.
36. Baxter, I., Pidgeon, P., Mehlich, M.: Dms: Program transformations for practical scalable software evolution. In: Proceedings of the International Conference on Software Engineering (ICSE'04), IEEE Press (2004)
37. Visser, E.: Program transformation with stratego/xt: Rules, strategies, tools, and systems. [48]
38. Cordy, J., Dean, T., Malton, A., Schneider, K.: Source transformation in software engineering using the txl transformation system. Information and Software Technology **44** (2002)
39. Taha, W.: A gentle introduction to multi-stage programming. [48]
40. Jones, N., Gomard, C., , Sestoft, P., eds.: Partial Evaluation and Automatic Program Generation. International Series in Computer Science. Prentice-Hall (1993)
41. Consel, C., Danvy, O.: Tutorial notes on partial evaluation. In: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, Charleston, SC, USA, ACM Press (1993) 493–501
42. Consel, C.: From a program family to a domain-specific language. [48] 19–29
43. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: Proceedings of ECOOP'01. Lecture Notes in Computer Science, Springer-Verlag (2001)
44. Tarr, P., Ossher, H., Harrison, W., , Sutton, S.M.: N degrees of separation: Multi-dimensional separation of concerns. In: Proceedings International Conference on Software Engineering (ICSE) '99, ACM Press (1999) 107–119
45. Mezini, M., Ostermann, K.: Variability management with feature-oriented programming and aspects. In: Foundations of Software Engineering (FSE-12), ACM SIGSOFT (2004)
46. Object Management Group: Model-Driven Architecture (2004) `www.omg.com/mda`.
47. Barth, B., Butler, G., Czarnecki, K., Eisenecker, U.: Report on the ecoop'2001 workshop on generative programming. In: ECOOP 2001 Workshops, Panels and Posters (Budapest, Hungary, June 18-22, 2001). Volume 2323 of Lecture Notes in Computer Science., Springer-Verlag (2001)
48. Christian Lengauer, D.B., Consel, C., Odersky, M., eds.: Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers. Volume 3016 of Lecture Notes in Computer Science. Springer-Verlag (2004)