Software Product Family Architectures - Engineering Run-Time Variability Dependencies in an FPGA-based Signal Processing Board

MICHEL JARING
Department of Mathematics and Computing Science
University of Groningen
P.O. Box 800, 9700 AV Groningen
THE NETHERLANDS

Abstract: - In a software product family context, software architects anticipate product diversification and design architectures that support variants in both space (multiple contexts) and time (changing contexts). Product diversification is based on the concept of variability: a single architecture and a set of components support a family of products. Software product families need to support increasing amounts of variability, leading to a situation where variability dependencies become of primary concern. This paper discusses (1) a formalization of variability dependencies in the context of product family engineering and (2) a case study in developing a Real-Time Adaptive Signal Processing (RASP) system. RASP is a run-time reconfigurable signal processing board based on Field Programmable Gate Array (FPGA) technology. The formalization of variability dependencies has been used to design, implement and document RASP.

Key words: - Software product families; Variability; Generic software architecting; Dynamically reconfigurable systems; Configurable system on-chip.

1 Introduction

Composing software products from software components is not a recent concept. Early work on software components already appeared three and a half decades ago [1], followed by the idea to develop so-called program families [2]. This has evolved into practical software engineering approaches that share the ability to promote software reuse across many products. An example of software reuse in practice is the successful adoption of software product families in industry. The goal of the software product family approach is the systematic reuse of core artifacts for building related software products. A software product family typically consists of a product family architecture, a set of components and a set of products. Each product derives its architecture from the product family architecture, instantiates and configures a subset of the product family components and usually contains some product specific code [3].

When developing a software product family, software architects try to prepare the family architecture for different product contexts, i.e., they prepare the architecture to support product diversification. Product diversification is based on the concept of variability and appears in all family artifacts where the behavior of an artifact can be changed, adapted or extended. Examples of variability are a configuration wizard to select the language or a license key entered by the customer to enable (or disable) particular functionality. Variability is implemented by delaying design decisions to a specific moment in the software development process, i.e., variant selection is delayed until a specific development or deployment phase such as implementation or run-time is reached. A typical example of a delayed design decision is a software system that adapts its run-time behavior to its environment by selecting alternatives on-the-fly according to criteria that have been defined in the development process.

```
#define EMBEDDED /* SIMULATION */

#ifdef SIMULATION
       #include <stdio.h>
#endif

#ifdef EMBEDDED
       #include <drivers.h>
#endif
```

Figure 1. Variation point for selecting embedded or simulation mode at compile-time.

1.1 Variation Points

So-called variation points have been introduced in [4] and are often used to express variability. A variation point refers to one or more delayed design decisions, has as an associated set of variants and indicates a specific moment in the development process. A typical example of a variation point is the preprocessor directive shown in Fig. 1. This variation point provides a choice in software operation mode and has two variants, namely embedded and simulation mode. The actual selection of a mode variant is delayed until source code compilation. In other words, the design decision 'operation mode' is not taken prior to compilation, meaning that the consequences of this decision should not affect the development phases prior to the compilation phase.

As also discussed in [5], all design decisions are still open when developing a system from scratch, but they are not left open intentionally. In other words, they do not refer to a particular variation point and are therefore implicit. Design decisions become explicit in the development process when the corresponding variation point (or points) is identified, i.e., when it is taken into account that specific variants are required. Once a variation point is identified, it is an unbound state, meaning that a variant has not been selected

yet from the set of variants associated with this point. The variation point is bound as soon as a variant is selected and incorporated into the system. This implies that a variation point is in one of the following states:

- Implicit: the design decision is not identified, but is accidentally left open.
- Explicit: the design decision is identified and intentionally left open:
  - Unbound: no variant has been selected from the set of variants associated with the variation point, i.e., the design decision is open until it binds a variant.
  - Bound: a particular variant is selected from the set of variants associated with the variation point and incorporated into the system, i.e., the design decision is final.

1.2 Domain and Application Engineering

There are two relatively independent development cycles in software product family engineering, namely domain and application engineering. Domain engineering is responsible for the design, development and evolution of the reusable artifacts, i.e., the product family architecture and shared components. Application engineering, on the other hand, is about adapting the product family architecture to the architecture of the required product. In other words, domain engineering *prepares* for product diversification and application engineering is the *act* of product diversification itself.

1.3 Variability Dependencies

Product families need to embed increasing amounts of variability, i.e., system functionality and system properties such as safety, security, reliability and usability move away from mechanics and electronics to software and become an integral part of the variability infrastructure. In addition, the number of products in a product family tends to grow, meaning that the variability infrastructure becomes more fine-grained and therefore more complex. The increasing amount of variability leads to a situation where variability engineering becomes of primary concern in software development. The number of variation points for industrial product families may range in the thousands, which is already a challenging number to manage, but the number of variability dependencies typically has an exponential relationship to the number of variation points, meaning that it is impossible to manage variability without systematic approaches. Systematic approaches require a formalization of variability dependencies, i.e. a variability notation.

We detail a case study in representing variability in a family of Magnetic Resonance Imaging (MRI) scanners developed by Philips Medical Systems in [6]. The MRI case illustrates how variability can be made an integral part of system development at different levels of abstraction and identifies several research questions. These questions have a common focus, namely: *How to formalize variability dependencies in product family engineering?*

The remainder of this paper is organized as follows. The next section suggests a formalization of variability dependencies in a product family context. The formalization is used in a case study in developing a run-time reconfigurable signal processing board in section 3. Section 4 summarizes and concludes the paper.

2 Formalizing Variability Dependencies

This section suggests a formalization of variability dependencies in a product family context. The formalization assumes that software variability originates from both soft- and hardware functionality.

2.1 System Variability

Software product family engineering is characterized by two main types of development, i.e., development with an embedded and development with a non-embedded perspective. Embedded software is used to control electronic products not normally identified as computers, meaning that it usually executes on an internal microcontroller or a Digital Signal Processor (DSP) to control other product components. Typically, such software must be extremely reliable, very efficient, compact, and precise in its handling of the rapid and unpredictable timing of inputs and outputs. Non-embedded software, on the other hand, is intended to run on a separate computer, often a personal computer or work station, and may be used to enhance the operation of another device or devices. One of the main differences between embedded and non-embedded software is the hardware, i.e., as opposed to non-embedded software, embedded software is typically (very) hardware dependent.

It is sometimes argued that there is no actual distinction between embedded and non-embedded software. From a theoretical viewpoint, two alternative programs and the hardware they run on can easily be equivalent, even if one appears to be embedded and the other does not. This may be true in theory, but it generally does not hold in practice. For example, the display size of mobile phones in pixels depends on the manufacturer and sometimes even varies within a particular product family. The different display sizes are hardware variants and each variants requires a specific display driver. In other words, the software is either dedicated to a specific display size or incorporates a variation point that supports different display sizes in the form of variants, i.e., these two alternatives are not equivalent.

Product families are often embedded systems that strongly depend on hardware, but this is hardly addressed in presentations of ideal applications. The concept of software reuse is typically overestimated, i.e., the flexibility of software is less than expected due to dependencies between soft- and hardware. Supporting the differences in hardware is called the hardware challenge in [7].

In academia, system development is often considered as being independent from the hardware, i.e., the software does not have to take the hardware into account. However, organizations that develop complex systems tend to use a more generalized approach, i.e., system development comprises both soft- and hardware aspects. The reuse infrastructure in a product family may consist of only software or a combination of soft- and hardware. This is also the approach taken by, e.g., the CAFÉ project (from Concept to Application in system-Family Engineering), i.e., the focus

is on software, but in a soft- and hardware context. See [8] for more information on CAFÉ and similar projects.

As detailed in [6], we have identified (at least) two types of *software* variability in relation to the *hardware* configuration of a system:

- Hardware neutral variability: software variability independent from the hardware configuration, e.g., multiple language support in a mobile phone.
- Hardware enforced variability:
  - Software variability that depends on the hardware configuration, e.g., text output formatting depending on the display size.
  - Software variability that is required to enable the hardware configuration, e.g., display drivers.

Literature on variability most often refers to hardware neutral variability, i.e., it generally assumes that variability originates from software and not from the combination of soft- and hardware. We refer to software variability as the *combination of hardware neutral and hardware enforced variability, i.e., system variability*.

2.2 Describing Variability

Binding a variation point involves establishing a *relationship* between the variation point and the selected variant. This relationship may imply certain *dependencies* (constraints), e.g., a system generally requires that specific variation points are bound to have a working, minimal system. There can be many different types of dependencies and pinpointing them requires a more formal way to describe variability. We use a notation that has many characteristics of a constraint specification language. Constraint specification languages have been developed outside the immediate software engineering research community such as the configuration management community and have been used in practice for several decades now. See, e.g., [9]. Please note that we are trying to pinpoint the different types of variability dependencies and that we use a constraint specification language as a tool to prevent ambiguities. In other words, it is a means and not a goal in itself. The following nomenclature aims for describing variability in system-independent terms, i.e., independent from a particular system, method or organization:

- The set of all variation points: $VP = \{vp_a, vp_b, vp_c, \ldots\}$
- The set of variants for vp_x: $vp_x = \{v_{x1}, v_{x2}, v_{x3}, \ldots\}$
- The power set (the set of subsets) of all variants: $V = \{\{v_{a1}, v_{a2}, v_{a3}, \ldots\}, \{v_{b1}, v_{b2}, v_{b3}, \ldots\}, \{v_{c1}, v_{c2}, v_{c3}, \ldots\}, \ldots\}$
- A relationship between vp_x and v_{xn} (vp_x binds v_{xn}): (vp_x, v_{xn})

Dependencies between variation points and variants can be expressed in the form of conditional expressions:

- if vp_x is bound then vp_y should be bound: **if** vp_x **then** vp_y
- if vp_x is bound then vp_y should bind v_{yn}: **if** vp_x **then** (vp_y, v_{yn})
- if vp_x binds v_{xn} then vp_y should be bound: **if** (vp_x, v_{xn}) **then** vp_y
- if vp_x binds v_{xn} then vp_y should bind v_{ym}: **if** (vp_x, v_{xn}) **then** (vp_y, v_{ym})

Dependencies may involve negation, meaning that the condition and expression of the **if-then** statement can exclude variation points or variants from binding. For example, if vp_x binds v_{xn} then vp_y should not bind v_{ym} is expressed as **if** (vp_x, v_{xn}) **then** $\neg(vp_y, v_{ym})$. A relationship between a variation point and a variant may impose dependencies on other variation points and variants. For example, if vp_x is bound then both vp_y and vp_z should also be bound is expressed as **if** vp_x **then** $(vp_y \wedge vp_z)$. Similarly, if vp_x is bound then vp_y or vp_z or both should be bound is expressed as **if** vp_x **then** $(vp_y \vee vp_z)$, i.e., it is an inclusive OR operation. To complete the nomenclature, the exclusive OR operation, meaning that either vp_y or vp_z is bound, is written as **if** vp_x **then** $(vp_y \veebar vp_z)$.

3 Case Study: RASP

This section discusses a case study in developing a Real-Time Adaptive Signal Processing (RASP) system. RASP is a run-time reconfigurable signal processing board based on Field Programmable Gate Array (FPGA) technology. The formalization of variability dependencies has been used to design, implement and document RASP.

3.1 Dynamically Reconfigurable Systems

The term 'reconfigurable system' applies to a broad range of systems. It originally referred to systems in which hardware is reorganized to adapt to a changing context, but is now also used for systems that can be reconfigured during any of the development or deployment phases such as design or execution. Configurability is introduced in the development process by delaying design decisions, i.e., configurability is based on the concept of variability.

A system is dynamically reconfigurable if its configuration can be changed during deployment. A typical example of dynamic reconfiguration is an embedded system that adapts its run-time behavior to its environment by selecting alternatives on-the-fly according to criteria that have been defined in the development process. The software architecture of such a system is dynamic, meaning that it is not static during development nor at deployment. Dynamic software architectures [10] are an extreme form of variability, i.e., it is not possible to delay design decisions beyond run-time. Even a conditional expression is sometimes considered as a form of variability, meaning that virtually all software architectures incorporate a certain amount of dynamic behavior. The counterpart of dynamic *software* architectures is dynamic *hardware* architectures, i.e., deployment-time reconfigurable hardware. Whether the soft- or hardware architecture is static or dynamic, reconfigurability is based on system variability, i.e., reconfigurability relates to both hardware neutral and hardware enforced variability:

- Static architectures:

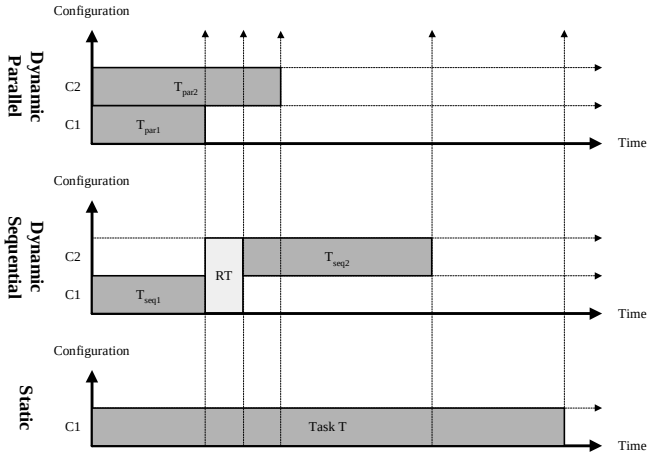

Figure 2. Static versus dynamic processing. RT: Reconfiguration Time.

  - Not adaptable: The configuration is the same throughout development and deployment.
  - Development adaptable: The configuration can be changed during development, but not during deployment.
- Dynamic architectures: The configuration is deployment adaptable, most notably at run-time.

An application area of dynamically reconfigurable systems is speeding up computation. As shown in Figure 2, if a task T can be partitioned into two *sequential* subtasks T_{seq1} and T_{seq2}, each subtask can be processed with a run-time configuration that is optimized for a part of the total computation. In addition, if task T can be partitioned into two *parallel* subtasks T_{par1} and T_{par2}, both subtasks can be processed simultaneously due to the intrinsic parallelism available in reconfigurable hardware such as FPGAs.

3.1.1 Field Programmable Gate Arrays

The field of reconfigurable hardware has become an active area of research after the introduction of FPGAs by Xilinx in the mid-1980s [11]. An FPGA provides a matrix-like structure of logic blocks and interconnects that can be altered in the field to create any combinatorial and sequential operations that fit into the chip. Figure 3 shows the Configurable Logic Blocks (CLBs) and interconnection lines in an FPGA. CLBs are based on Field Effect Transistors (FETs), which use an electric field to switch to and maintain the logic on state or, depending on the type of FET, the off state. The function configuration for a CLB and its connection to other blocks are typically loaded from distributed Static Random Access Memory (SRAM) into the FPGA in the form of precompiled reconfiguration bit streams. The configurable logic in a CLB is often a so-called Lookup Table (LUT). A LUT with n input lines and one output line implements any function of n variables by storing the value of the function for each input combination of the LUT.

3.2 Real-Time Adaptive Signal Processing

The formalization of variability dependencies has been used to design, implement and document a Real-Time Adaptive

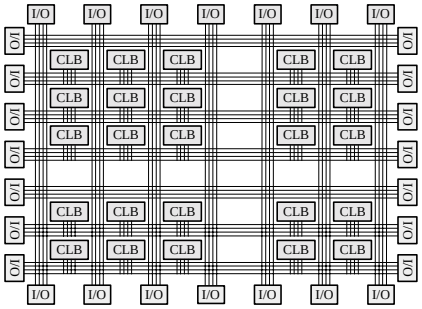

Figure 3. Field Programmable Gate Array. CLB: Configurable Logic Block. I/O: Input/Output.

Signal Processing (RASP) system. RASP is under development by Sophalgo Data Systems and targets hard real-time applications such as monitoring a dedicated software component that synchronizes the processes that run on a multitude of PowerPCs and on-the-fly data monitoring of data streams to trace anomalies such as a control unit that suddenly operates outside its bounds. System stability and change management are key issues in the development of RASP and depend strongly on run-time variability dependencies, meaning that variability has been made an integral part of the soft- and hardware architecture. All in all, the idea behind RASP is to have a generic soft- and hardware platform with the characteristics of a hard-wired solution that can be reconfigured at run-time into the most optimal signal processing variant depending on the changes in the input signal.

3.2.1 Hardware Architecture

The RASP hardware architecture consists of an FPGA that is tightly coupled to a microcontroller and is designed for applications requiring high throughput, hard real-time signal processing in industrial settings. Microcontrollers are integrated computer systems on chip consisting of a processor and support functions such as memory, analog/digital converters, counters/timers, communication ports, etc. The Peripheral Component Interconnect (PCI) local bus specification with a bus width of 64 bit and a bus speed of 66 MHz is used for interacting with the environment, which is fast, architecture independent and standardized. Figure 4 shows the main components of the hardware architecture:

- FPGA: Gate array implementation of the signal processing algorithm. The FPGA can be reconfigured while processing the data stream, i.e., in real-time.
- Microcontroller: Evaluates the signal processing results. Depending on the evaluation, the microcontroller selects the most suitable signal processing configuration and writes it to the configuration controller.
- Data memory: Location for storing volatile microcontroller and FPGA data during execution.
- Program memory: Write-protected location to store the microcontroller program and the precompiled FPGA reconfiguration bit streams.
- Memory controller: Two-way, independent bus connections for data and program memory, respectively.

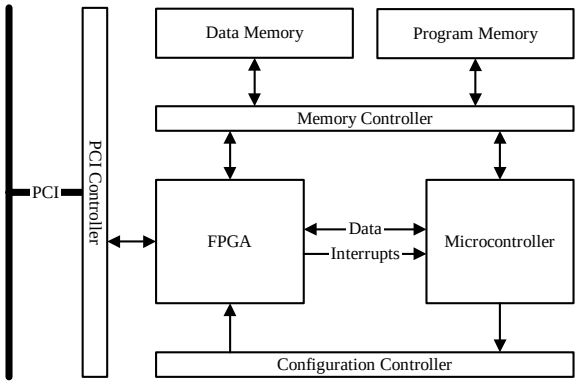

Figure 4. RASP hardware architecture.

- Configuration controller: One-way bus connection and configuration protocol implementation for reconfiguring the FPGA.
- PCI controller: Local I/O bus for interfacing with the environment. This bus carries the data stream as input and signal processing results as output.

One of the key features of RASP is the ability to adapt to changes in the signal, i.e., the signal processing results are evaluated on-the-fly to determine if the FPGA is optimally configured with the available variants. The evaluation algorithm is implemented in the form of a mapping function and is executed in a dedicated fashion by the microcontroller. The mapping function requires a fixed number of CPU cycles for each evaluation and is executed at least 100 times per second, which is a guaranteed lower bound given the microcontroller specification. For each 250 evaluations, the configuration with the highest vote count is written to the configuration controller, assuming it is different from the current FPGA configuration. The configuration controller then reconfigures (parts of) the FPGA accordingly.

3.2.2 Software Architecture

The software architecture of RASP is designed, implemented and documented according to the variability nomenclature and independent from the hardware architecture. Due to the conditional expressions of the nomenclature, the run-time variability dependencies can be stored (program memory), processed (microcontroller) and written (configuration controller) in a one-to-one fashion, i.e., 'as is'. To prevent ambiguities in the product configuration process, a valid RASP configuration should bind all variation points. The Empty variant is available for binding variation points that are excluded from providing functionality. The RASP product family consists of five products, i.e., RASP identifies five process modes that are marketed as independent products. See also Table 1. Each mode requires a combination of the family variants shown in Table 2. The RASP product family is defined as follows:

- $RASP \equiv VP = \{Mode, SPA, Analysis, Compression, Indexing, Monitoring, Sampling\}$
- $Mode = \{Analysis, Compression, Indexing, Monitoring, Sampling, Empty\}$
- $Analysis = \{SPA, RM, AF, Empty\}$
- $Compression = \{SPA, RM, CF, Empty\}$

Table 1. RASP product family.

Process mode	Data stream operation
Analysis	Artifact recognition
Compression	Removing redundancy
Indexing	Artifact addressing
Monitoring	Anomaly tracing
Sampling	Sample set selection

Table 2. RASP product family variants.

Family variants	Description
SPA	Signal Processing Algorithm
RM	Reference Matrix
AF	Analysis Filter
CF	Compression Filter
IF	Indexing Filter
MF	Monitoring Filter
SF	Sampling Filter
Empty	*void functionality*

- $Indexing = \{SPA, RM, IF, Empty\}$
- $Monitoring = \{SPA, RM, MF, Empty\}$
- $Sampling = \{SPA, RM, SF, Empty\}$

Selecting a particular process mode means that the other modes can not be selected, which is expressed by binding the Empty variant:

- **if** $\neg(Analysis, Empty)$ **then** $((Compression, Empty) \wedge (Indexing, Empty) \wedge (Monitoring, Empty) \wedge (Sampling, Empty))$
- **if** $\neg(Compression, Empty)$ **then** $((Analysis, Empty) \wedge (Indexing, Empty) \wedge (Monitoring, Empty) \wedge (Sampling, Empty))$
- **if** $\neg(Indexing, Empty)$ **then** $((Analysis, Empty) \wedge (Compression, Empty) \wedge (Monitoring, Empty) \wedge (Sampling, Empty))$
- **if** $\neg(Monitoring, Empty)$ **then** $((Analysis, Empty) \wedge (Compression, Empty) \wedge (Indexing, Empty) \wedge (Sampling, Empty))$
- **if** $\neg(Sampling, Empty)$ **then** $((Analysis, Empty) \wedge (Compression, Empty) \wedge (Indexing, Empty) \wedge (Monitoring, Empty))$
- **if** $(Mode, Empty)$ **then** $((Analysis, Empty) \wedge (Compression, Empty) \wedge (Indexing, Empty) \wedge (Monitoring, Empty) \wedge (Sampling, Empty))$

Except for the Empty variant, the variants of the variation point Mode are also variation points and have a one-to-many relationship to the product family variants:

- **if** $\neg(Analysis, Empty)$ **then** $(Analysis, (SPA \wedge RM \wedge AF))$
- **if** $\neg(Compression, Empty)$ **then** $(Compression, (SPA, \wedge RM \wedge CF))$
- **if** $\neg(Indexing, Empty)$ **then** $(Indexing, (SPA \wedge RM \wedge IF))$
- **if** $\neg(Monitoring, Empty)$ **then** $(Monitoring, (SPA \wedge RM \wedge MF))$

Table 3. RASP signal processing variants.

SPA variants	Description
2DP	2-Dimensional Processing
4DP	4-Dimensional Processing
8DP	8-Dimensional Processing
...	...
64DP	64-Dimensional Processing
128DP	128-Dimensional Processing
256DP	256-Dimensional Processing
Empty	*void functionality*

- **if** $\neg(Sampling, Empty)$ **then** $(Sampling, (SPA \wedge RM \wedge SF))$

As said, one of the key features of RASP is the ability to adapt to changes in the signal it is processing. Table 3 shows the variants of the Signal Processing Algorithm (SPA) variation point that are available at run-time. As opposed to process mode selection, selecting the signal processing algorithm is a one-to-one relationship between the variation point SPA and its variants. In other words, SPA variant selection is an exclusive OR operation:

- $SPA = \{2DP, 4DP, 8DP, 16DP, 32DP, 64DP, 128DP, 256DP, Empty\}$
- **if** $(Mode, Empty)$ **then** $(SPA, Empty)$
- **if** $\neg(Mode, Empty)$ **then** $(SPA, (2DP \veebar 4DP \veebar 8DP \veebar 16DP \veebar 32DP \veebar 64DP \veebar 128DP \veebar 256DP))$

The software architecture of RASP is the formal definition of the variability infrastructure as described above. This definition is implemented by the configuration controller in the form of programmable logic gates. In addition to writing the configuration bit streams to the FPGA, the configuration controller also provides an integrated soft- and hardware lock, i.e., logic gates (hardware) implement the variability infrastructure in the form of conditional expressions (software). An encrypted 32 bit string is programmed into the on-chip Electrically Erasable Programmable Read Only Memory (EEPROM) of the microcontroller and defines the default settings for RASP, i.e., the product (process mode) the customer has purchased and the default variant for the variation point SPA, which is usually 32DP. After reading this string from EEPROM at system start-up, the FPGA is programmed with the appropriate combination of reconfiguration bit streams.

Please note that the variation point SPA is used to reconfigure the FPGA to optimize the signal processing algorithm at run-time, whereas the other variation points are used to configure the FPGA as a member of the product family at system start-up.

4 Conclusions

In a product family context, system variability is the combination of hardware neutral and hardware enforced variability, i.e., variation points and variants appear in both soft- and hardware and may depend on each other regardless of their origin. Binding a variation point means establishing a relationship between the variation point and a variant. This relationship may impose dependencies (constraints) on other variation points and variants. The variability relationships and dependencies have been formalized in the form of a variability nomenclature. The nomenclature is used in a case study in developing a Real-Time Adaptive Signal Processing (RASP) system. RASP is a run-time reconfigurable, FPGA-based signal processing board.

The study shows that the nomenclature can be used to design, implement and document a dynamically reconfigurable product family architecture such as RASP. Both soft- and hardware variability dependencies are described in a relatively easy to understand notation, i.e., in constraint specification language style. Describing variability in a constraint specification language is particularly useful for automating the product configuration process and for binding variation points at run-time according to criteria that have been defined in the development process.

References:

[1] M. D. McIlroy: *Mass Produced Software Components*. NATO Software Engineering Conference, Garmisch, Germany, 1968, pp. 138-155.

[2] D. L. Parnas: *On the Design and Development of Product Families*. IEEE Transactions on Software Engineering, Vol. 2, No. 1, 1976, pp. 1-9.

[3] J. Bosch: *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[4] I. Jacobson, M. Griss, P. Jonsson: *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[5] J. van Gurp, J. Bosch, M. Svahnberg: *On the Notion of Variability in Software Product Lines*. Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, 2001, pp. 45-54

[6] M. Jaring, R. L. Krikhaar, J. Bosch: *Representing Variability in a Family of MRI Scanners*. Software: Practice and Experience, Vol. 34, No.1, 2004, pp. 69-100.

[7] A. Maccari, A. Heie: *Managing Infinite Variability*. Workshop on Software Variability Management, Groningen, The Netherlands, 2003.

[8] F. J. Linden: *Software Product Families in Europe: The ESAPS & CAFÉ Projects*. IEEE Software, Vol. 19, No. 4, 2002, pp. 41-49.

[9] F. J. Buckley: *Implementing Configuration Management: Hardware, Software and Firmware*. IEEE Press, 1996.

[10] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf: *An Architecture-based Approach to Self-Adaptive Software*. IEEE Intelligent Systems, Vol. 14, No. 3, 1999, pp. 54-62.

[11] W. S. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo and S. L. Sze: *A User Programmable Reconfigurable Logic Array*. Proceedings of the IEEE Custom Integrated Circuits Conference, Rochester, USA, 1986, pp. 233-235.