INFORMATICA, 2002, Vol. 13, No. 3, 255–274

© 2002 *Institute of Mathematics and Informatics, Vilnius*

Separation of Concerns in Multi-language Specifications

Robertas DAMAŠEVIČIUS, Vytautas ŠTUIKYS
Software Engineering Department, Kaunas University of Technology
Studentų 50, 3031 Kaunas, Lithuania
e-mail: damarobe@soften.ktu.lt, vytautas.stuikys@if.ktu.lt

Received: April 2002

Abstract. We present an analysis of the separation of concerns in multi-language design and multi-language specifications. The basis for our analysis is the paradigm of the multi-dimensional separation of concerns, which claims that multiple dimensions of concerns in a design should be implemented independently. Multi-language specifications are specifications where different concerns of a design are implemented using separate languages as follows. (1) Target language(s) implement domain functionality. (2) External (or scripting, meta-) language(s) implement generalisation of the repetitive design features, introduce variations, and integrate components into a design. We present case studies and experimental results for the application of the multi-language specifications in hardware design.

Key words: multi-language design, separation of concerns, meta-programming, scripting, hardware design.

1. Introduction

Today the high-level design of a hardware (HW) system is usually based either on the usage of the pure HW description language (HDL) (e.g., VHDL, Verilog), or the algorithmic one (e.g., C++, SystemC), or both. The emerging problems are similar for both, the HW and software (SW) domains. For example, the increasing complexity of SW and HW designs urges the designers and researchers to seek for the new design methodologies. The main emphasis in the methodologies is given to raising the level of abstraction, and applying the separation of concerns in design process. Designers are mainly focusing on the language-based solutions to these problems as follows. (1) Designing within the realms of a particular language (e.g., introducing HW design concepts into C++ with class libraries (Swan, 2001)). (2) Extending the syntax of an existing language (e.g., C++ with abstractions for high-level parameterization (Singhal *et al.,* 1993), or VHDL with customizable interfaces (Siegmund *et al.,* 2000)). (3) Designing a new (usually domain-specific) language, which represents the domain content more suitably (e.g., parameterised HW design (Luk *et al.,* 1998)).

These approaches, however, have many weaknesses as follows. (1) To design a system, all designers are required to use the same language, which is working at the same

level of abstraction, i.e., the implementation of domain functionality. (2) The concerns are usually separated only at the component level of a particular programming language; and (3) designers are focused at a specific solution of a problem rather than designing for reuse. This 'one-dimensional' view to system design is undermining some important problems such as generalization and customization, which do not have an adequate solution within the one-language design paradigm.

The usual way to managing complex problems in system design is to apply at a higher extent the principle of the *separation of concerns* also known as 'divide-and-conquer' strategy. In its most general form, it refers to the ability to identify, encapsulate, and implement at a time only those parts of a design that are relevant to a particular concept, goal, or purpose. Although, the *separation of concerns* has been known for decades in SW engineering, up to recently it was used mostly at the component level, i.e., to reduce the complexity of algorithms by dividing the problem into the several smaller ones, and implementing them separately. The most evident result of employing this '*multi-component design*' strategy is the usage of component libraries. However, since the complexity of designs has grown significantly, the separation of concerns has to be considered at a higher level of abstraction, too.

Ossher, Tarr and their colleagues (Ossher *et al.,* 2000) introduced the concept of the *multi-dimensional separation of concerns* (MDSoC), which is a new direction in SW engineering research and practice. The MDSoC aims at (1) identifying and encapsulating *multiple* dimensions of concern simultaneously, (2) identifying and encapsulating new concerns during SW lifetime, and (3) representing and managing the overlapping and interacting concerns.

Concurrently, there is a great deal of research (Multi-language design, Aspect-Oriented Programming, Subject-Oriented Programming, Generative Programming, Intentional Programming, etc.), which emphasise (directly or indirectly) the separation of concerns and the higher levels of abstraction (including the usage of multiple languages) in SW and HW design.

The contribution of this paper is as follows: (1) the classification of the multi-language design paradigms, specifications, languages and their roles in these specifications; (2) the application of the explicit separation of concerns for the multi-language specifications in HW design.

The structure of the paper is as follows. We review the related works in Section 2. We consider the multi-language design systems and specifications, and analyze the roles of languages in Section 3. We discuss the multi-dimensional separation of concerns in HW design in Section 4. Case studies and experimental results are presented in Section 5. Finally, we provide the conclusions in Section 6.

2. Related Works

The MDSoC paradigm is discussed in a number of papers. Ossher *et al*. (2000) hypothesised that major difficulties associated with the improvement of SW reuse, comprehensibility, component integration, system composition and decomposition, and high impact

of modifications in SW systems are due to a deficiency of the separation of concerns. Authors emphasise the need to consider the separation of overlapping concerns, separation in various dimensions of concerns at a time or simultaneously, to deal with the interaction and integration of concerns.

Tarr *et al.* (2000) claim that concerns are the primary motivation for organising and decomposing SW into manageable and comprehensible parts. Many different kinds, or *dimension*s, of concerns may be relevant to different developers in different roles, or at different stages of the SW lifecycle. Silva (1999) formulates the requirements for the MDSoC approaches as follows. (1) Provide flexible abstractions for SW concerns. (2) Support the abstraction composition. (3) Describe abstractions and abstraction composition independently of integration mechanisms. (4) Balance the abstraction flexibility and usage simplicity.

Despite of the fact that this direction is still in the initial stage and many problems are yet to be solved, the benefits of the MDSoC have already been reported in several works of other authors. For example, Kandè *et al*. (2000) apply the MDSoC in the description of SW architecture, which allows considering the system from multiple perspectives simultaneously. Batory (2000) emphasises the role of encapsulation in the MDSoC. Murphy *et al.* (2001) study the MDSoC in object-oriented (OO) systems.

The multi-language design approach proposed by Jerraya *et al.* (1999), implements the MDSoC at the level of programming languages. They argue that different languages that are more suitable and efficient for the specification of subsystems of a large system should be used. Kleinjohann (1998) investigates the semantic problems in multi-language design with regard to the composition of system parts specified in several languages, and considers language integration and coupling models.

Meta-programming (MPG) has been known for a long time, especially in formal logic programming. Sheard (2001) gives an excellent overview of the MPG systems, and taxonomy of meta-programs. MPG in high-level system design languages is not so much the covered topic. Veldhuizen (1995) considers the template meta-programs in C++. Meiyappan *et al.* (1999) discuss the MPG capabilities of VHDL.

Czarnecki *et al.* (2000) introduce a program development approach for generating customised components and systems called *Generative Programming* (GP). The focus is on automating the mapping between the *problem* and *solution* domains given by an established *architecture*. GP uses the principle of the separation of concerns to separate each domain problem into a distinct generic component or sets of components, which are used to generate a target program.

Harrison *et al.* (1993) propose *Subject-Oriented Programming* (SOP), which is a program composition technology that supports building OO systems as compositions of *subjects*. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. SOP allows customising and integrating systems and reusable components.

Simonyi (1995) proposes *Intentional Programming*, a language-independent programming environment, in which all source code is represented as an *abstract syntax tree (AST)*. Nodes of an AST are called *intentions* and correspond to the semantic constructs of a language. Examples of intentions include if-statements, type declarations,

assignment statements, etc. The Intentional Programming system manipulates with the intentions at a higher-level of abstraction.

Kiczales *et al.* (1997) propose a novel programming paradigm called *Aspect-Oriented Programming* (AOP). The AOP-based specification of an application consists of *component* and *aspect* specifications. A component is a generalised procedure, which is implemented using the *component language*. An aspect is a property of the component that affects its performance or semantics in a systematic way, and is implemented using the *aspect language(s)*.

Nuseibeh *et al.* (1994) consider multiple *viewpoints* that hold partial requirement specifications described and developed using different representation schemes and development strategies. Leite *et al.* (1991) makes a further distinction between *views, perspectives* and *viewpoints.* By explicitly deploying *views* that encapsulate partial specifications together with the development techniques by which they are produced, the problems of integration may be addressed.

VanHilst *et al.* (1996) address the *collaboration-based* (or *role-based*) designs, which decompose an OO application into a set of classes and a set of *collaborations*. Each application class encapsulates several *roles*, where each role embodies a separate aspect of the class behaviour. A co-operating suite of roles is *collaboration*, which expresses the distinct aspects of an application. Turner *et al.* (1998) consider *Feature Engineering*. *Features* are a grouping or modularization of particular requirements and their implementation within a specification.

Ousterhout (1998) drew attention to the scripting technology. He distinguishes the roles of the scripting language (ScL) (such as Perl, TCL) and the system programming language (such as C++, Java). A system programming language serves for the design of domain algorithms from scratch. A ScL serves for gluing (integrating) components into a system. According to (Schneider *et al.*, 1999), scripting can be considered as a *higher-level binding technology* for component-based systems, which denotes abstractions for connecting components. Achermann *et al.* (2000) claim that scripts are *high-level specifications* that make the co-ordination of components explicit. Nierstarsz *et al.* (2000) examine the language support for the separation of concerns as follows: (1) component algebras, (2) higher-order wrappers, (3) glue abstractions, (4) mixins and other composition mechanisms.

We can summarise the overview of the related works as follows. (1) The authors solve the complexity problem of designs by using the principle of the separation of concerns, and extracting multiple concerns, which are called *dimensions, aspects, features, views, viewpoints, roles, collaborations,* or *intentions* by different authors. (2) The authors use multiple languages (implicitly or explicitly) for implementing the concerns. (3) A great emphasis is given to the integration and composition of concerns. In the following section we consider the usage of the multi-language specifications in SW and HW design in detail.

3. Multi-Language Specifications and Design Systems

3.1. *Motivation for Multi-Language Specifications*

Most existing system specification languages are based on a *single language* paradigm. Each of these languages was developed for a given application domain, e.g., SW design (C++, Java), HW design (VHDL, Verilog), etc. However, the languages which provide general solutions for all domain problems, are not as optimal and efficient for problem solution as domain-specific languages (DSLs). For example, VHDL has to deal with HW design issues as follows. (1) *Behavioural description* of the functionality of the HW models with signals (signal is a domain-specific object in VHDL). (2) *Generic description* of the similar HW models (*generic map*, *conditional* and *repetitive generate* statements). (3) *Structural description* of the particular compositions of the lower-level HW models (*port map*, *configuration* statements, *packages*, etc.).

Only the behavioural description of domain objects by signals and their events is a HW design-specific problem in VHDL. The other issues are well known in other domains as well, and are solved by a number of languages (i.e., meta-, scripting, shell languages, etc.). It would be reasonable, according to the MDSoC paradigm, to leave the solution of these problems for the *separate* languages. For example, the first language (*algorithmic*) deals only with the solution of the specific domain problems. The second language (*meta-language*) deals only with the generalisation and parameterisation of the specific 'look-alike' components. The third language (*scripting*) deals only with the establishment of connections between the component instances. Programs written in different languages cooperate together in a *multi-language specification* of a design. A particular number of languages used may be left to the designer's free choice, and is topic worth of the separate consideration.

The usage of the external languages and multi-language specifications are well known in the domains of meta-programming and scripting. Further we consider these programming technologies in detail.

3.2. *Preprocessors, Macro Languages and Meta-Programming*

The multi-language specifications have been known for a long time in SW engineering. The examples are the various macro languages and pre-processors (e.g., CPP for C, M4 for FORTRAN and C), which allow modification of source code before actual compilation. Macro programming systems were first created in the 1960s as a way to ease the development of assembly language programs, but the idea survived to our days and led to the invention of the automatic configuration systems, such as *autoconf, metaconfig*. These systems help to make a target code configurable and portable to various platforms and operating systems by using a sort of an external language to control the compilation process. The concerns "compilation" and "execution" are clearly separated.

One of the most popular pre-processors, CPP, allows to define a new syntax, abbreviate repetitive or complicated constructs, inline functions, propagate symbolic constants,

eliminate dead code, etc. CPP also permits the system dependencies to be made explicit, which results in a clearer separation of concerns.

Formal specification languages often use several languages at a time. For example, BNF (Backus–Naur Form) developed in 1960, is used for the description of the syntax of a programming language. Extended BNF, which is used in compiler generators, such as Lex and Yacc (Terry, 1997), integrates both BNF grammar rules and TL code, where the concerns "syntax" (described in BNF) and "semantics" (described in a TL) are clearly separated.

The term 'meta-program' has been introduced in formal logic programming, and initially meant the higher-order logic program. As the scope of application broadened, the term has acquired new meanings. According to Batory *et al.* (1992), a meta-program is "*a program that generates the source of the application ... by composing pre-written code fragments*". In a meta-programming (MPG) system meta-programs manipulate target programs (TPs): they may construct TPs, combine TP fragments into larger TPs, analyse the structure and other properties of TPs, and execute (instantiate) TPs to obtain their values (instances), etc.

We can define MPG as a programming technique that enables manipulation with other program structures. Another definition can be found in (Ryman, 1990): "*meta-programming is the process of specifying generic software source templates from which classes of software components, or parts thereof, can be automatically instantiated to produce new software components*". An example of such implicit MPG is C++ templates. This case of MPG, however, is implemented within the realms of a single language, therefore, the separation of concerns is not fully implemented.

Another case of MPG involve the usage of the explicit meta-language, i.e., *"any language or symbolic system used to discuss, describe, or analyse another language or symbolic system"* (Czarnecki *et al.,* 2000). To distinguish between different languages, a lower-level language is called *target language* (TL), and a higher-level one – a *meta-language* (ML). These different levels of abstraction can be described using (1) different subsets of the same language, (2) language extensions, or (3) actually different languages.

A common usage of MPG is to provide mechanisms for writing *generic code*. Usually a TL implements commonalties in a domain, while a ML allows developers to specify variations to be implemented in the target system and to synthesise customised implementations by composing TL code fragments. The genericity can be achieved by the *parameterisation of differences* in different *program representations,* which allows representing components with many commonalties in a compact way. This simple feature allows improving reusability substantially by providing parameterised components, which can be *instantiated* into target programs for different choices of parameters.

Examples of the MPG usage, with different behaviour models, include (1) macro languages like M4 or CPP, where strings are transformed to generate other strings. (2) HTML/XML generators that transform inputs into forms interpretable by web clients. (3) Lisp language, where an input list can be transformed into another list before being compiled. (4) MetaML (Sheard, 2001), where programs are transformed into an annotated syntax tree.

3.3. *Program Generation and Transformation Systems*

Another example of the multi-language systems are SW generators, which usually generate a target code from a high-level specification, and often use an intermediate language or abstract syntax tree as a convenient representation to perform the optimization and transformation of a target code before actual generation. A related area is program transformation systems. Successful program generation and transformation systems include Draco (Neighbours, 1989), FermaT (Ward, 1989), TXL (Cordy *et al.,* 2001), and many others.

Draco factors the domain of application into *modeling domains* (e.g., a network domain or a database domain). A modeling domain is a pure abstraction of the knowledge about the domain and does not specify how abstractions in that domain will actually be implemented. Abstractions and their operations in each domain are defined by a DSL. The programmer writes his programs in the languages of these various domains, which are compiled into lower-level domain languages until executable code in a language like C, C++, or Java. Therefore, different system concerns are isolated in different domains, and are implemented using different DSLs.

FermaT is a program transformation system based on two layers of languages. These languages implement different concerns as follows. (1) A high-level, general purpose language WSL is used for recording, analyzing and manipulating programs and fragments of programs. (2) A very high-level DSL, called METAWSL (an extension of WSL) is used for representing programs as tree structures, and has constructs for pattern matching and iterating over components of a program structure.

TXL is a programming language and rapid prototyping system specifically designed to support the structural transformation of programs. Each TXL program has two components specified in different languages as follows. (1) *Description of the Structures to be transformed* is specified using an EBNF grammar. (2*) Set of Structural Transformation Rules* is specified by an example, from which an application strategy is automatically inferred. Therefore, the concerns "program" and "transformation rule" are clearly separated.

3.4. *Scripting*

The main idea of the *scripting technology* is that an application developer only has to write a small amount of *wiring code* in order to establish a connection between components. It can take various forms, depending on the nature and granularity of the components, the nature and framework of the problem domain, and the composition model. Scripting denotes abstractions for connecting components. *Components* implement the provided functionality behind a standard interface and generally represent the *stable* parts of applications, whereas s*cripts* plug components together, and represent the *flexible* parts of applications.

A scripting language (ScL) is a higher-level language, used to assemble components into the pre-specified software architecture. Well known examples of ScL are shell languages for UNIX command composition, PERL for generating HTML pages and other tasks, and Visual Basic for rapid prototyping of GUI.

Shell languages (such as *Bourne Shell, c shell, k shell*) are dedicated for issuing commands to the operation system or other applications. They offer a simple component model (usually called *pipe and filter* model) based on *commands* and byte streams, which can be connected using the *pipe* operator ‘ | ’ and file/stream *redirectors* (‘<’, ‘>’, etc.). Commands can be implemented in any programming language, provided they support the ability to read from the standard input stream and produce output onto the standard output and/or error streams. Shell languages represent the pure scripting paradigm, concentrating on controlling data flows and gluing components (scripts) rather than performing computations.

In the next subsection, we look at the separation of concerns in distributed and internet-based computing.

3.5. *Distributed and Internet-Based Computing*

With the arrival of the internet-based technologies, the multi-language design gained the popularity. One can mention COM and CORBA for distributed computing, JavaScript embedded in HTML documents, Java applets and servlets, CGI programming, dynamic web page generation using PHP, JSP (*Java Server Pages*) or ASP (*Active Server Pages*), mobile code systems, etc.

CORBA is a framework that defines how distributed objects can inter-operate. CORBA objects can be written in any programming language supported by a CORBA SW manufacturer such as C, C++, Java, Ada, or Smalltalk. The language independence is made possible via the construction of interfaces to objects using the Interface Description Language (IDL). IDL allows all CORBA objects to be described in the same manner; the only requirement is a ”bridge” between the target language and IDL.

CGI (*Common Gateway Interface*) programming is about extracting information from HTML forms and generating new HTML pages using PERL scripting language. PERL encapsulates part of the domain of text file processing (e.g., pattern searching and replacement, arbitrarily sized strings, arrays, “associative arrays”, etc.), and also provides a uniform access mechanism to various operating system functions. In CGI, the concerns are clearly separated between “interface” and “information processing”.

In Java servlets, different languages represent different levels of abstraction. Java is used for processing user requests and generating HTML code. HTML is used for GUI interfacing and reading user requests. Therefore, we have a clear separation of concerns “interface” and “request processing”.

3.6. *HW Design and HW/SW Co-Design*

We, however, are mostly interested in application of the multi-language specifications in HW design. For a long time, HDLs such as VHDL, Verilog, existed nearly independently of each other. However, the complexity of HW designs is increasingly growing, and designers start to concern more about designing system-on-chips (SoCs) from pre-designed IPs (*Intellectual Property* components), which can be implemented using different HDLs,

rather than designing from scratch. The integration of such IPs is an issue of HW/HW co-design.

Furthermore, more and more popular are embedded systems (ES), which are single-purpose computers that are often a part of consumer electronics such as mobile phones. Their design requires a tight integration of the application-specific HW and SW, and often has strong constraints on power, cost, and speed.

HW/SW co-design is a new research area growing out of HW design. Briefly, the purpose of co-design is to facilitate the design of ES, i.e., systems consisting of (1) *hardware* (a microprocessor (such as DSP), plus one or more semi-custom ASICs (*Application-Specific Integrated Circuits*) or FPGAs (*Field Programmable Gate Arrays*)), for which (2) *software* is required. Therefore, the issue of the separation of concerns is a hot topic in HW design. Various dimensions of concerns at various abstraction levels should be separated and later integrated. Here we consider several HW/SW co-design systems as follows.

In POLIS (Balarin *et al.,* 1997), the high-level application specification is written in ESTEREL ("system-level" concern), and reusable modules are written in VHDL, C or Assembler ("component-level" concern). The application is later synthesized into VHDL ("hardware" concern) and C (Assembler) ("software" concern) specifications. The MCI (Hessel *et al.,* 1999) co-simulation tool allows the usage of four different languages (C for algorithmic descriptions, VHDL for hardware, SDL for state-based specifications, and MatLab for continuos computation) for the modeling of different concerns of complex systems. The COSYMA (Ernst *et al.,* 1996) co-synthesis system uses C^X for system description at a high level. After partitioning, the description of a system is translated into C ("software") and HardwareC ("hardware"). Agliada *et al*. (2001) developed a system, which uses SystemC for high-level architecture description ("structural" concern), and VHDL for implementing components ("behavioral" concern).

Scripting plays an important role in the HW design, too. It has been used to automate a design flow, integrate a variety of EDA tools, extract required information from a design, and prepare a configuration file (Chen *et al*., 2001).

3.7. *Summary*

After analysis of the multi-language systems, we classify the *design paradigms* (DP) as follows: (1) *one-language paradigm* (1LP), which employs only one language, and (2) *multi-language paradigm* (MLP), which employs multiple languages for describing different concerns of a design. We further categorise MLP into (1) *co-design* (CoD), where components are implemented using different DSLs at the same abstraction level; (2) *scripting paradigm* (ScP), which emphasises component integration, and (3) *meta-programming* (MPG) paradigm, which emphasises component generalisation and composition issues. We distinguish the *internal* (or *homogeneous*) MPG (IMPG), when MPG is applied implicitly using the MPG constructs of a single language, and the *external* (or *heterogeneous*) MPG (EMPG), when the languages are separated explicitly, and ML performs manipulations of a TL code.

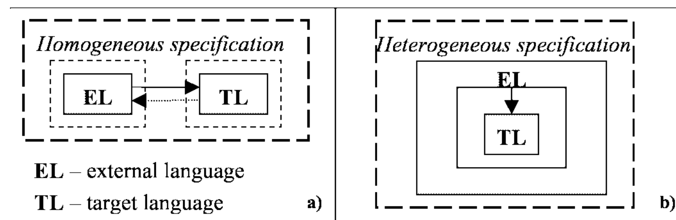


Fig. 1. Multi-language specifications: (a) homogeneous, and (b) heterogeneous.

After analysis, we categorise the multi-language specifications (Fig. 1) as follows: (1) *homogeneous* specifications (HMS), and (2) *heterogeneous* specifications (HTRS). In HMS, the languages are clearly separated, i.e., we actually have the separate specifications for each language. The concerns are not clearly separated or separated only at the same level of abstraction (e.g., Java native interfaces to C++). In HTRS, the languages are not clearly separated, i.e., the programs of the languages are intermingled and we actually have only one specification written in two (or possibly more) languages. However, the concerns are clearly separated and usually expressed via the generic parameters. Different languages represent different levels of abstraction (e.g., Java and HTML in Java servlets).

We categorise the roles of the languages in multi-language specifications as follows: (1) *target languages* (TLs) and *domain-specific languages* (DSLs) are for expressing domain functionality; and (2) *external languages* (ELs) are for modification, composition and generation of TL code. The latter can be further categorised into (1*) scripting languages* (ScLs), which are usually used in HMS, and (2) *meta-languages* (MLs), which are usually used in HTRS. These languages, in turn, can be (1) the different subsets of the same language in the IMPG paradigm, or (2) actually different languages in the EMPG paradigm. Of course, a role of a language depends upon the context of its usage. Furthermore, a particular language can have multiple roles. The roles of the ELs in HMS and HTRS are rather different. In HMS, the EL is used for delegating and gluing the implementation of domain functionality at the *same level of abstraction*. In HTRS, the EL is used for customising and generating TL code at the *higher level of abstraction*.

Additionally, high-level system programming languages such as C++ can be used as (1) a TL in the one-language design paradigm, (2) a TL in the multi-language design paradigm, (3) a ML in the MPG paradigm, or (4) one of many DSLs in the co-design paradigm. We summarise the roles of languages as follows: (1) TL (or DSL) is for expressing domain functionality (F), (2) ScL is for composition (C), (3) ML is for generation (G), (4) ML and ScL are for parameterisation (P). Additionally, there can be other roles for specific domains and languages.

We summarise our analysis in Fig. 2. We have identified four dimensions of the concern “system design” as follows: design paradigms, specifications, languages, and roles of languages. We performed the decomposition with respect to the design paradigms, and present the kinds of specifications, languages, and roles of languages used for each paradigm.

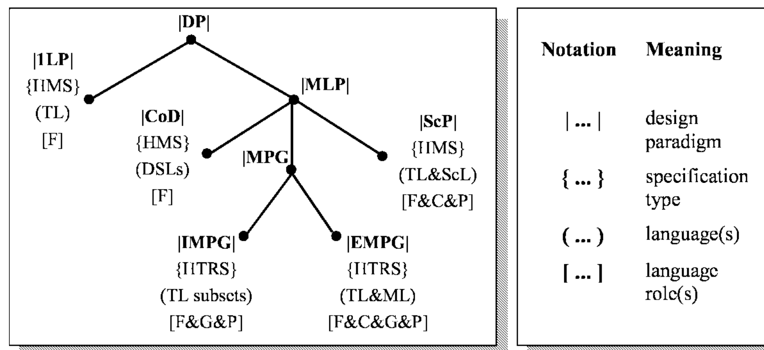


Fig. 2. Classification of design paradigms, specifications, languages, and roles of languages.

4. Multi-Dimensional Separation of Concerns in HW Design

Tarr and Ossher use the term *multi-dimensional separation of concerns* (MDSoC) to denote the separation of concerns involving: (1) Multiple, arbitrary dimensions of concern. (2) Separation along these dimensions *simultaneousl*y. (3) The ability to handle new concerns or their dimensions *dynamically* as they arise throughout the SW lifecycle. (4) Overlapping and interacting concerns. (5) Concern-based integration.

The MDSoC is based on the idea that independent concerns should be represented independently, and programs should be developed by the composition of separate concerns according to the systematic rules. The MDSoC approaches usually deal with abstractions and integration mechanisms. The abstractions describe the solutions for domain-specific aspects of the design. The integration mechanisms are responsible for integrating abstractions among themselves and gluing with the domain object. Further in this paper, we apply the ideas of the MDSoC for HW design.

A key objective in designing reusable HW modules (aka *soft IPs*) is to encapsulate within each module a single aspect of a design. There are many aspects or *concerns* when dealing with HW design (Fig. 3). Every concern relates to the different activity in the do-

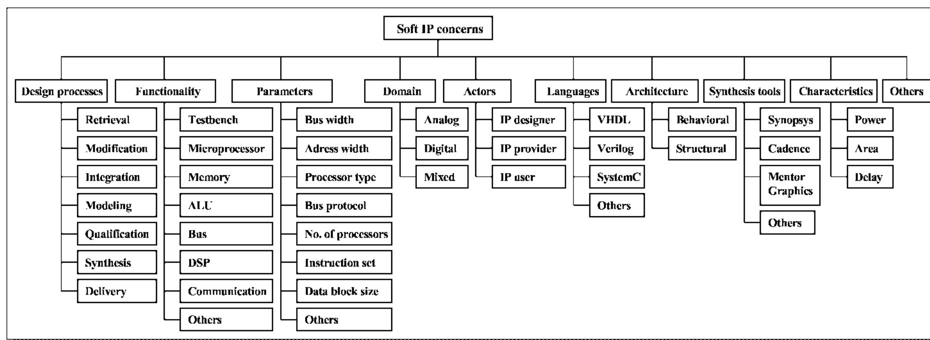


Fig. 3. Different concerns in soft IP design.

main. Concerns and their dimensions can be (1) *orthogonal* (undependable), e.g., “data width” and “address width”, and (2) *overlapping* (dependable), e.g., “data width” and “processor type”. It is comparatively easy to identify, encapsulate and integrate concerns for the orthogonal separation. The orthogonal separation can be introduced intuitively, presented implicitly and does not require the development of an explicit model. On the contrary, it is not an easy task to identify, decompose, encapsulate and integrate the overlapping concerns. In this case, we usually need to build a relationship model in order to understand the concerns and extract the benefits of the relationship. Related concerns can be grouped into kinds or *dimensions*, which emphasise a particular aspect of a problem. A model to deal with a design problem is obtained when dimensions are re-integrated together.

Each concern used in a design can be represented as a slice or dimension in the design space. Here we understand the design space as a set of all feasible implementations of the domain model. The design space and the concerns are the result of domain analysis, which we do not consider here. The concept of the MDSoC is especially useful in the domain of HW design where a great variety of requirements exist at the different levels of abstraction. These requirements restrict the design space of the generic component. The design space is further detailed by the generic parameters to meet the specific requirements.

Soft IP may be parameterised in terms of its functionality (when a specific functionality is selected from a family of related functionality), architecture (different implementations), size (data path width, address width, etc.), control (e.g., pipelining), performance characteristics (power, area, delay), synthesis tools and target technology. The designer uses the specified parameters to instantiate a component design customised to his requirements. Common and variable features of domain components must be captured by suitable abstractions. Commonalties reflect the shared context that is invariant across the set of similar components (such as multipliers, ALUs, RAMs, etc.), whereas the variations, which capture the distinguishing properties of the components, have to be specified at a higher level of abstraction and represented via the generic parameters.

We can represent the separated concerns explicitly with the generic parameters, and the integrated concerns implicitly with a generic specification, which implements the entire family of the related components (Fig. 4). A generic specification is the multi-

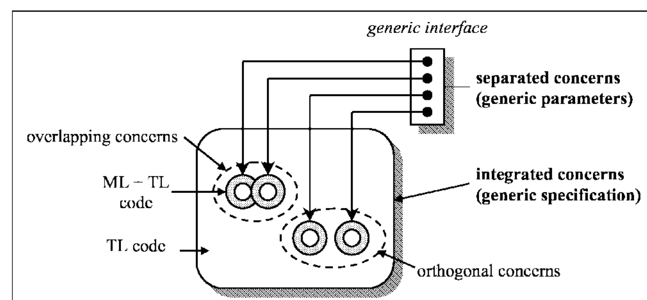


Fig. 4. Separation and integration of concerns in generic specifications.

language one, which is a particular composition of the TL and ML programs. The TL code encapsulates features common for the entire component family. The ML code encapsulates the concerns represented by the generic parameters.

In the following section we demonstrate the usage of the multi-language specifications as a particular application of the MDSoC paradigm in HW design by our case studies.

5. Case Studies

5.1. *Control Signal Insertion into VHDL Models Using Java*

In this case study, we demonstrate the explicit separation of concerns in VHDL models. We separate the concerns "control" and "functionality". By "control" we mean the control signals (clock, enable, reset) of a model. We use Java as an external language in order (1) to analyse the concern "functionality", and (2) generate VHDL code for the concern "control".

In this case study, Java constructs act as ML constructs with respect to TL (VHDL) constructs, which are at a lower level of abstraction. We use Java class and method declarations as a generic interface, Java method calls for the instantiation of TL components, loop statements for repetitive generation, conditional statements for conditional generation, and standard I/O statements for TL code generation.

We have implemented a mini-generator in Java, which uses a VHDL parser to analyse a user-supplied VHDL component, and generates a wrapper component using the external MPG techniques. Here we demonstrate the automatic insertion of the *enable* signal (Fig. 5). Note that the generated TL statements are shown in bold. The *enable* signal al-

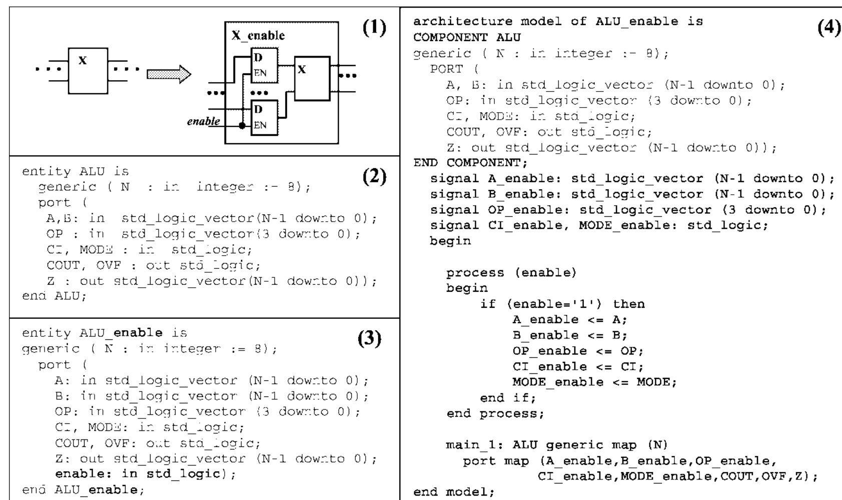
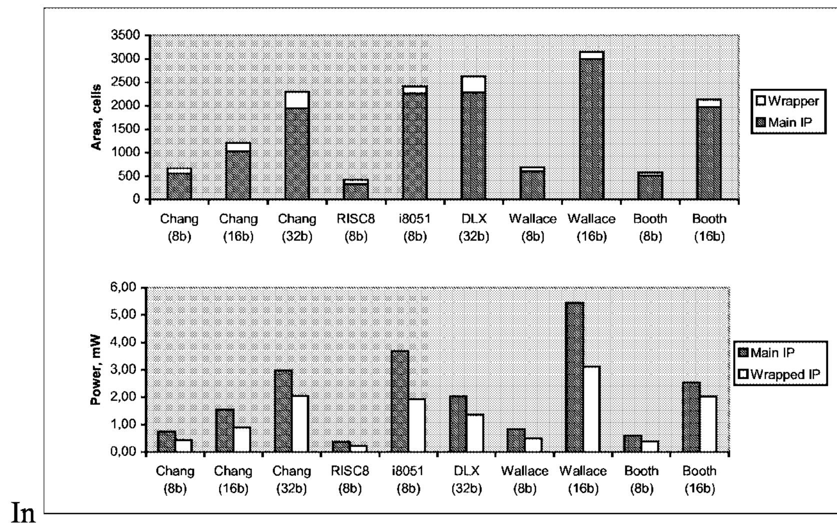


Fig. 5. Insertion of the enable signal into a VHDL model: (1) the modification scheme, (2) entity of a given soft IP, (3) a modified entity, and (4) the generated architecture.



In

Fig. 6. Synthesis results of the third-party IPs with respect to the insertion of enable signal.

lows saving power when designing low power components. The component can be turned off when it is not used.

Suppose, we have obtained a third-party component X. We accept that X is already validated and qualified. Our aim is to add the enable signal to the model, thus allowing turning on/off a model on demand. After modification, component X is wrapped with wrapper logic (registers).

In Fig. 6, we present the synthesis results[1] from the experiments performed with a variety of the third-party soft IPs as follows. (1) Chang's ALU (Chang, 1997), (2) ALU from "Ans RISC8 Core" (Kook *et al.,* 2000), (3) ALU from i8051 micro-controller (Givargis, 2000), (4) ALU from DLX processor (Gumm, 1995), (5) Wallace Tree multiplier (Hollreiser *et al.,* 1994), and (6) Booth multiplier (Booth, 2001). The results show a slight overhead in area, and a decrease in power consumption for the generated VHDL models.

5.2. *Generalisation of VHDL FIR Filter Using Open PROMOL*

FIR (Finite Impulse Response) filters are widely used in DSP. We obtained a third-party implementation of FIR filter in VHDL from (Iwata, 2001). It consists of ROM, SRAM, multiplier, accumulator, divider and memory controller (Fig. 7).

We have identified the generic parameters of a FIR filter as follows: data width, coefficient width, FIR filter coefficients, and tap count. These parameters represent the specific concerns in FIR filter design. The problem is that these concerns (1) are scattered across the design, and (2) are not expressed explicitly, therefore, require the error-prone work to modify a component, if needed. These features make the available FIR design poorly reusable in other applications.

---
[1] we use Synopsys tools

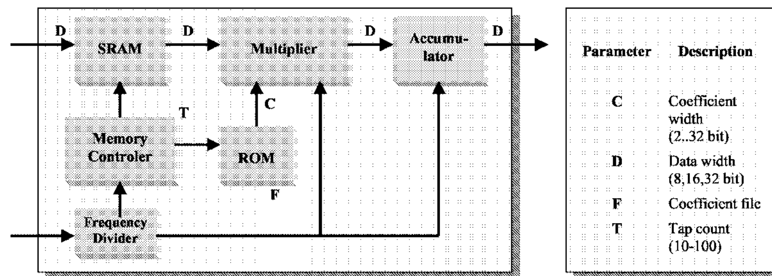


Fig. 7. FIR filter architecture and its parameters.

To make a FIR filter design more reusable, we generalise it as follows. (1) We separate the identified concerns from the common concerns related with the implementation of functionality (such as multiplier type), which we do not consider here. (2) We describe these concerns at a higher level of abstraction (using an EL). (3) We assemble the generic parameters and isolate them in a generic interface.

As an external language, we use Open PROMOL (Štuikys *et al.,* 2000; Štuikys *et al.,* 2002), which is a functional language and consists of an open set of external functions. All modifications in the specification are represented as a specific composition of the external functions with the TL code to be modified.

The implemented generic specification of a FIR filter (Fig. 8) is a two-language specification as follows. (1) Higher-level specification (in Open PROMOL; shown in bold) expresses the FIR filter design concerns explicitly via the generic parameters, glues lower-

```
$
@include[rom,sram,memctrl,divider,multiplier,accumulator]
$
"Enter data width:"              {4,8,12,16,32} d_width:=8;
"Enter width of coefficients:"   {2..32}        c_width:=16;
"Enter tap count:"               {10..100}      tap:=90;
"Enter the name of the data file for the coefficients:"
                                 { }            file:=data.txt;
$                                                              (1)
```

```
@move[a_width,1+log [tap-1]]
@macro[rom,d_width,c_width,a_width,file]
@macro[sram,d_width,a_width]
@macro[memctrl,a_width,tap]
@macro[divider]
@macro[multiplier,d_width,c_width]
@macro[accumulator,d_width]                                    (2)
```

```
entity fir_top_@sub[d_width]bits is
   Port (
-- reset
      RST : In std_logic;
-- Fs (a clock synchronized with input data)
      FSCLK : In std_logic;
-- input data is @sub[d_width] bits
      D_IN : In std_logic_vector(@sub[d_width-1] downto 0);
      MCLK : In std_logic; -- 384Fs (master clock)
-- filtered data output
      D_OUT : Out std_logic_vector(@sub[d_width-1] downto 0));
end fir_top_@sub[d_width]bits;
                                                               (3)
```

Fig. 8. Generic FIR filter specification: (1) PROMOL interface, (2) gluing of the external PROMOL specifications, (3) a composition of VHDL code and PROMOL functions (a fragment).

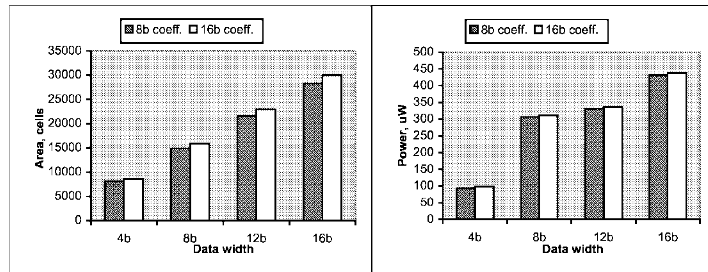


Fig. 9. Synthesis results of FIR filter.

level components of a FIR filter together, and performs the necessary TL code modifications to obtain a specific FIR filter design. (2) Lower-level specification (in VHDL) implements the functionality of the FIR filter.

In Fig. 9, we present the synthesis results for the generated FIR filter instances. We specified the values of the generic parameters as follows: the coefficient width is 8 and 16 bits, the tap count is 90, and the data width is 4, 8, 12, or 16 bits.

5.3. *Generalisation of SystemC Buffer Using C++ Templates and CPP*

To demonstrate the implementation of the separation of concerns in SystemC models, we decided to implement a generic buffer model. We use a FIFO buffer from (Swan, 2001) as a third-party IP. We identified the buffer concerns as follows: buffer size, data type of buffer elements, and buffer type (FIFO or LIFO). We have implemented these concerns at a higher-level of abstraction and expressed via generic parameters. We derived two implementations of a generic buffer using C++ templates (internal MPG) and CPP (external MPG) (Fig. 10, (1) & (2), respectively; note that constructs, which are used to separate

```
#define MAX 10
#define DATATYPE char
#define FIFO

class buffer : public sc_channel,
        public write_if, public read_if {
...
  void read(DATATYPE &c){
    if (num_elements==0) wait(write_event);
#ifdef FIFO
    c = data[first];
    first = (first + 1) % MAX;
#elif LIFO
    c = data[num_elements];
    first--;
#endif
    -- num_elements;
    read_event.notify();
  }
...
};                                          (1)
```

```
buffer<10,char,FIFO> *buf_inst;
buf_inst = new
buffer<10,char,FIFO>("Buffer1");            (2)
```

```
enum FIFO;
enum LIFO;

template <int max, class Datatype, class Buffertype>
class buffer :   public sc_channel,
                 public write_if<Datatype>,
                 public read_if<Datatype> {
...
  void read(Datatype &c){
    if (num_elements == 0) wait(write_event);
    Buffertype buf;
    do_read(buf,c);
    -- num_elements;
    read_event.notify();
  }
  template<class B> void do_read(B,Datatype &c) { }
  void do_read(FIFO,Datatype &c) {
    c = data[first];
    first = (first + 1) % max;
  }
  void do_read(LIFO,Datatype &c) {
    c = data[num_elements];
    first--;
  }
...
};                                          (3)
```

Fig. 10. Generalising a buffer in SystemC (a fragment): (1) using CPP; (2) using C++ templates, and (3) its instantiation (an example).

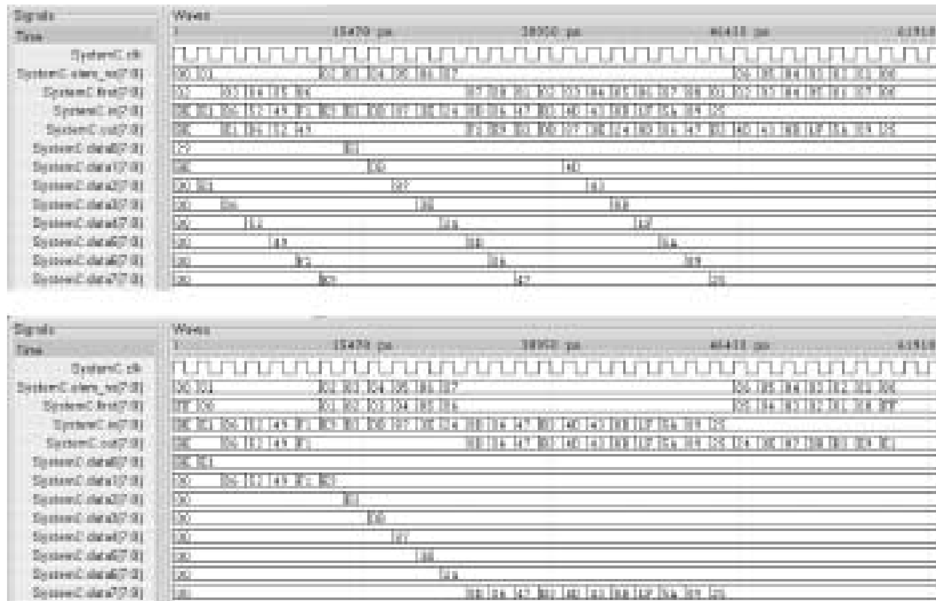


Fig. 11. Modelling results of the generic buffer instances: FIFO (above) and LIFO (below).

different concerns, are shown in bold). Both implementations allow the explicit separation of concerns, however the template-based one is not yet synthesizable by available synthesis tools. On the other hand, in the CPP-based implementation only one instance of the buffer can be used in a design.

In Fig. 11 we present the modelling results of the generic buffer instances. We specified values of the generic parameters as follows: buffer type is FIFO or LIFO, data type is *bv<8>* (i.e., 8-bit vector), and buffer size is 8.

6. Conclusions

The separation of concerns is a usual way to deal with complex problems in a system design. The multi-dimensional separation of concerns (MDSoC) paradigm handles multiple dimensions of concern simultaneously, as well as overlapping and interacting concerns. The MDSoC can be used as a general framework to better understand the well-known programming technologies such as meta-programming (MPG) and scripting, as well as adopting other new technologies such as generative, intentional, aspect-oriented programming, etc. The evolution of the MPG paradigm is the multi-language design.

The multi-language design introduces a clear separation of concerns in a design by using different languages for different purposes. When generalisation and generation is concerned, MPG (especially, the external one) allows the explicit separation of domain functionality from the generalisation and composition issues.

We have demonstrated the application of the MDSoC and MPG techniques for developing the multi-language specifications of HW models, thus achieving higher flexibility, customisability and reusability.

Acknowledgements

Authors thank to the anonymous reviewer whose comments served for improving the paper.

References

Achermann, F., S. Kneubuehl, O. Nierstrasz (2000). Scripting coordination styles. In *Proceedings of the Coordination'2000, Lecture Notes in Computer Science*, Vol. 1906. Springer–Verlag, 19–35.

Agliada, N., A. Fin, F. Fummi, M. Martignano, G. Pravdelli (2001). On the reuse of VHDL modules into systemC designs. In *Forum on Design Languages FDL'2001*, Lyon, France.

Balarin, F., Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Passerone, C., Sangiovanno–Vincentelli, A., Sentovich, E., Suzuki, K., Tabbara, B. (1997). *Hardware–Software Co-Design of Embedded Systems – The POLIS Approach*. Kluwer Academic Publishers.

Batory, D. (2000). Refinements and separation of concerns. In *Second Workshop on Multi-Dimensional Separation of Concerns, International Conference on Software Engineering*, Limerick, Ireland.

Batory, D., S. O'Malley (1992). The Design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Enginnering and Methodology*, **1**(4), 355–398.

Booth multiplier (2001). `http://www.cs.umbc.edu/help/VHDL/samples/`

Chang, K.C. (1997). *Digital Design and Modeling with VHDL and Synthesis*, IEEE Computer Society Press, The Institute of Electrical and Electronic Engineers, Inc., Los Alamitos.

Chen, P., K. Keutzer (2001). Fast integration of EDA tools and scripting language. In *8th IEEE/DATC Electronic Design Processes Workshop*, Monterey, California, USA.

Cordy, J.R., T.R. Dean, A.J. Malton, K.A. Schneider (2001). Software engineering by source transformation – experience with TXL. In *Proc. IEEE 1st International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, Florence, Italy, IEEE CS Press, 168–178.

Czarnecki, K., U. Eisenecker (2000). *Generative Programming: Methods, Tools and Applications*. Addison–Wesley.

Ernst, R., J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny (1996). The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, **20**(3), 159–166.

Givargis, T. (2000). *Intel 8051 Microcontroller*, `http://www.cs.ucr.edu/~dalton/i8051/i8051syn/`

Gumm, M. (1995). *DLX Processor*. `ftp://ftp.informatik.uni-stuttgart.de/pub/vhdl/vlsi_course/vhdl_src/source_files.tar.Z`

Harrison, W., H. Ossher (1993). Subject-oriented programming – a critique of pure objects. In Paepcke, A. (Ed.) *Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, USA. SIGPLAN Notices, **28**(10). ACM Press. pp. 411–428.

Hessel, F., P. LeMarrec, C.A. Valderrama, M. Romdhani, A.A. Jerraya (1999). MCI – multilanguage distributed co-simulation tool. In F. Rammig (Ed.), *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers. pp. 191–200.

Hollreiser, M., T. Bissuel (1994). *Wallace Tree Multiplier*. `http://mikro.e-technik.uni-ulm.de/vhdl/vhdl_models.html`

Iwata, T. (2001). *FIR Filter*. `http://www.digitalfilter.com/`

Jerraya, A.A., M. Romdhani, Ph. Le Marrec, F. Hessel, P. Coste, C. Valderrama, G.F. Marchioro, J.M. Daveau, N.-E. Zergainoh (1999). Multi-language specification for system design and co-design. In A.A. Jerraya, J. Mermet (Eds.), *System Level Synthesis*. Kluwer Academic Publishers.

Kandé, M.M., A. Strohmeier (2000). On the role of multi-dimensional sdeparation of concerns in software architecture. In *Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA'2000*, Minneapolis, Minnesota, USA.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin (1997). Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science*, Vol. 1241. Springer–Verlag. pp. 220–242.

Kleinjohann, B. (1998). Invited talk: Multilanguage design. In *Proc. of International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES'98)*. Paderborn, Germany.

Kook, I., T. Park, W. Park (2000). *Ans_RISC8_Core*. `http://www.anslab.co.kr/goodkook/vhdl/Document/RISC8_Core/`

Leite, J.C.S.P., P.A. Freeman (1991). Requirements validation through viewpoint resolution. *Transactions on Software Engineering*, **12**(12), 1253–1269.

Luk, W., S. McKeever (1998). Pebble: a language for parameterised and reconfigurable hardware design. In Hartenstein, R.W., A. Keevallih (Eds.), *Field Programmable Logic and Applications. Lecture Notes in Computer Science*, Vol. 1482. Springer. pp. 9–18.

Meiyappan, S., K. Jaramillo, P. Chambers (1999). 10 tips for generating reusable VHDL. *EDN Magazine*, August 19, 49–62.

Murphy, G.C., A. Lai, R.J. Walker, M.P. Robillard (2001). Separating features in source code: an exploratory study. In *Proceedings of the 23rd International Conference on Software Engneering (ICSE 2001)*, Toronto, Ontario, Canada, IEEE CS Press, 275–284.

Neighbors, J.M. (1989). Draco: a method for engineering reusable software systems. In Biggerstaff, T.J., A. Perlis (Eds.), *Software Reusablility, Vol. I: Concepts and Models*. ACM Press. pp. 295–319.

Nierstrasz, O., F. Achermann (2000). Separation of concerns through unification of concepts. In *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*.

Nuseibeh, B., J. Kramer, A. Finkelstein (1994). A framework for expressing the relationships between multiple views in requirements specifications. *IEEE Transactions on Software Engineering*, **20**(10), 760–773.

Ossher, H., P. Tarr (2000). Multi-dimensional separation of concerns and the hyperspace approach. In M. Aksit (Ed.), *Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer Academic Publishers.

Ousterhout, J.K. (1998). Scripting: higher level programming for the 21st century. *IEEE Computer*, **31**(3), 23–30.

Ryman, A. (1990). Requirements for a metaprogramming language. *Presentation at the 24th meeting of IFIP Working Group 2.4*. Kingston, Canada.

Schneider, J.G., O. Nierstrasz (1999). Components, scripts and glue. In Barroca, L., J. Hall, P. Hall (Eds.), *Software Architectures – Advances and Applications*. Springer. pp. 13–25.

Sheard, T. (2001). Accomplishments and research challenges in meta-programming. In *2nd International Workshop on Semantics, Application, and Implementation of Program Generation (SAIG'2001)*, Florence, Italy. *Lecture Notes in Computer Science*, Vol. 2196. Springer. pp. 2–44.

Siegmund, R., D. Mueller (2000). A method for interface customization of soft IP cores. In R. Seepold, M. Navidad (Eds.), *Virtual Component Design and Reuse*. Kluwer Academic Publishers.

Silva, A.R. (1999). Separation and composition of overlapping and interacting concerns. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems* (at OOPSLA '99).

Simonyi, C. (1995). The death of computer languages, the birth of intentional programming. *NATO Science Committee Conference*.

Singhal, V., D. Batory (1993). P++: a language for large-scale reusable software components. In *Proc. of the 6th Annual Workshop on Software Reuse*, Owego, New York.

Swan, S. (2001). *An Introduction to System Level Modeling in SystemC 2.0*, White paper, OSCI.

Štuikys, V., R. Damaševičius (2000). Scripting language open PROMOL and its processor. *INFORMATICA*, **11**(1), 71–86.

Štuikys, V., R. Damaševičius, G. Ziberkas (2002). Open PROMOL: an experimental language for target program modification. In A. Mignotte, E. Villar, L.S. Spruiell (Eds.), *System-on-Chip Design Languages*. Kluwer Academic Publishers.

Tarr, P., M. D'Hondt, L. Bergmans, C.V. Lopes (2000). Workshop on aspects and dimensions of concern: requirements on, and challenge problems for, advanced separation of concerns. In *ECOOP Workshop Reader*. pp. 203–240.

Terry, P.D. (1997). *Compilers and Compiler Generators: An Introduction With C++*. International Thomson Publishing Inc., UK, Oxford.

Turner, C.R., A. Fuggetta, L. Lavazza, A.L. Wolf (1998). Feature engineering. In *Proceedings of the 9th International Workshop on Software Specification and Design (IWSSD'98)*. Kyoto, Japan, IEEE CS Press. pp. 162–164.

VanHilst, M., D. Notkin (1996). Using role components to implement collaboration-based designs. In *Proceedings of OOPSLA'1996*. ACM Press. 359–369.

Veldhuizen, T.L. (1995). Using C++ template meta-programs. *C++ Report*, **7**(4), 36–43.

Ward, M. (1989). *Proving Program Refinements and Transformations*. PhD. Thesis, Oxford University.

R. Damaševičius received MSc degree in informatics from Kaunas University of Technology, Lithuania in 2001. Currently he is Ph.D. student at Informatics Faculty, Kaunas University of Technology. His research interests include software reuse, multi-language design, software generation and program transformation, and hardware design with VHDL and SystemC.

V. Štuikys received Ph.D. degree from Kaunas Politechnic Institute in 1970. Currently he is in the position of a Professor at Software Egineering Department, Kaunas University of Technology, Lithuania. His research interests include domain-specific reuse, high level domain-specific languages, expert systems and CAD systems, including VLSIC design based on high-level hardware description languages and soft IP design.

Koncepcijų atskyrimas daugiakalbėse specifikacijose

Robertas DAMAŠEVIČIUS, Vytautas ŠTUIKYS

Šiame straipsnyje mes analizuojame koncepcijų atskyrimą projektuojant daugiakalbes sistemas ir specifikacijas. Analizės pagrindas yra daugiadimensinio koncepcijų atskyrimo paradigma, kuri teigia, jog atskiros koncepcijų dimensijos turi būti realizuojamos nepriklausomai viena nuo kitos. Daugiakalbės specifikacijos yra specifikacijos, kuriose skirtingi sistemos aspektai yra realizuojami naudojant skirtingas kalbas. (1) Tikslo kalbos aprašo srities funkcionalumą. (2) Išorinės (arba scenarijų, meta-) kalbos aprašo pasikartojančių sistemos bruožų apibendrinimą, įveda variantiškumą per parametrizavimą ir atlieka komponentų integravimą į sistemą. Mes pateikiame daugiakalbių specifikacijų taikymo aparatūrinės įrangos projektavimui tyrimus ir eksperimentinius rezultatus.