SYSTEM LEVEL DESIGN MODEL WITH RE-USE OF SYSTEM IP

SYSTEM LEVEL DESIGN MODEL WITH RE-USE OF SYSTEM IP

Edited by

Patrizia Cavalloro
Italtel SpA, Milan, Italy

and

Christophe Gendarme
Alcatel Bell, Antwerp, Belgium

and

Klaus Kronlöf
Nokia Research Center, Helsinki, Finland

and

Jean Mermet
ECSI, Grenoble, France

and

Jos van Sas
Alcatel Bell, Antwerp, Belgium

and

Kari Tiensyrjä
VTT Electronics, Oulu, Finland

and

Nikolaos S. Voros
Technological Educational Institute of Western Greece, Computer and Informatics Engineering Department, Greece

Kluwer Academic Publishers
Boston/Dordrecht/London

Contents

Preface

This book presents the perspective of the SYDIC-Telecom project on system design and reuse as perceived in the course of the research during 1999 - 2003.

The initial problem statement of the research was formulated as follows:

"The current situation regarding system design in general is, that the methods are insufficient, informally practiced, and weakly supported by formal techniques and tools. Regarding system reuse the methods and tools for exchanging system design data and know-how within companies are ad hoc and insufficient. The means available inside companies being already insufficient, there are actually no ways of exchanging between companies. Therefore, there hardly exists any system IP (Intellectual Property) industry. Although system design know-how is one of companies' main assets, it cannot be reused and capitalised effectively enough today. There is a lack of rational design flows supporting a design methodology based on reuse of IP, and few design tools to support it. Even guidelines on how to use existing tools in the design flow for this purpose often do not exist."

The problem was known to be hard and the scope broad. The plan of attack was first to analyse the state-of-the-art and the state-of-the-practice, then to identify potential improvements, and finally to synthesise a formalised proposal for implementation. The approach was applied to different system-level issues, e.g. design flows, terminology, languages, reuse, design process and object of design.

The eight chapters and five annexes describing the main results obtained in the research are ahead of you, for learning and taking benefit of.

Acknowledgements

The research work that provided the material for this book was carried out during 1999 - 2003 mainly in the SYDIC-Telecom project supported partially by the European Commission under the contract IST-1999-11376. Guidance and comments of Marc Engels, Michael Hohenbichler, Uros Janko, and Serge Vernalde on research direction are highly appreciated.

In addition to the editors, the contributions of the following project members and partners' personnel are gratefully acknowledged: Konstantina Barkoula, Spyros Batistatos, Gabriela Bilinkova, Alex Birbas, Franca Brivio, Rene du Cloux, Lars-Olov Eriksson, Giulio Gorla, Bruno Gualdi, Gjalt de Jong, Brian Lashley, Roel Marichal, Adam Morawiec, Jean-Pierre Moreau, Steffen Müller, Alex Niemegeers, Jonas Plantin, Florence Pourchelle, Annette Reutter, Sophia Tsatsakou, Patrick Valdenaire, and Dyson Wilkes.

Of them, the editors express their special thanks to Gjalt de Jong, Giulio Gorla, Brian Lashley, Konstantina Barkoula, and Dyson Wilkes for their substantial contributions to Chapter 2, Chapter 3, Chapter 4, Chapter 7, and Annex A5, respectively.

Valuable feedback to working documents was received from: Brian Bailey, Markus Brenner, Eliane Fourgeau, Philippe Launay, Grant Martin, Roberto Passerone, and Ian Phillips. The consortium designer review of the System Design Conceptual Model was performed by: Anselmo Biffi, Bruno Gualdi, Richard O'Connor, Miika Rummukainen, and Maurizio Vitale. External experts in the language analysis were: Axel Braun, Eike Grimpe, Nicolas Halbwachs, Stefan Hallerstade, Odo Knut, Antti Pelkonen, Alain Vachoux, and Avi Ziv.

Chapter 1

INTRODUCTION

Klaus Kronlöf
Nokia Research Center, Helsinki, Finland

Abstract: This chapter explains the motivations behind the work that provided the material for the book. The chapter includes a guide that suggests focus areas and reading flows for different types of readers.

Key words: System design, system architecting, intellectual property, reuse.

1. MOTIVATION

This book is an outcome of a joint European project, SYDIC-Telecom (SYstem Design Industry Council of European Telecom Industries). The motivation of the participating companies to join their efforts was the agreed observation that system design know-how, being one of their main assets, cannot be reused and capitalized effectively enough today.

Challenges and problems are caused by the following development in the industry:

1. Fast evolution of product properties, e. g. more functionality and diversity, requiring managing of product families, more effort needed to design the new, complex functionality, more designers involved in design projects, etc.
2. Fast evolution of technologies, requiring frequent adaptations of the methodology.
3. New application domains appearing, requiring novel use of designer expertise.
4. Decreasing time-to-market (TTM) despite increased functionality, diversity and complexity.

5. Demand for decreasing cost in the market encountering price erosion.

Adding the fact that it today is very difficult to reuse existing knowledge, the industry is faced with an increasing and serious problem: lack of experienced system designers.

The results of the project presented in this book are meant to provide the foundation to improve the product development capabilities of companies to cope with the above challenges. For this we need a system level design methodology with reuse of know-how. This can also open new business opportunities for providers of such know-how, i.e. a market for system design Intellectual Property (in the following called 'IP').

2. GUIDE FOR THE READER

The authors of this book are well aware that there is quite a lot of material in this book and that different types of readers are likely to be interested in specific sections of the book. Consequently we recognize need to give some practical advise where to put the focus based on the background of the reader. In the following we identify typical classes of readers and suggest them a reading flow.

The first type is a reader who wants to get an overall picture of the topics of the book and who is not necessarily very familiar with the domain. Such a reader is advised to follow the course of presentation. When reading Chapter 4, it is useful to frequently refer to Annex A5.

The second type is an industrial product development practitioner, such as a system engineer or a system architect. In this case Chapters 2 and 3 do not probably bring much new information and can be skipped or browsed lightly. Reading Chapter 4 is necessary for understanding the rest of the book, so it should not be skipped although it appears rather theoretical. Annex A5 helps digesting Chapter 4. Chapter 5 may or may not be of interest, depending on how much the reader is using design languages. However, we believe that Chapters 6, 7 and 8 are the most interesting ones for an industrial system developer.

The third type is a person responsible of methodology development and/or product development process improvement in a company. Similarly as above, Chapters 2 and 3 can be browsed lightly. Chapter 4 is the most important for a methodology developer. We believe that the System Design Process Model (SDPM) can be directly applied as a metamodel for process definition purposes. Likewise, the model of System Under Design (SUDM) can be used as a metamodel for methodology development, e.g. for the definition of UML profiles. Chapter 5 and especially Annex A3 give a useful framework for design language selection. Finally, Chapter 7 contains practical advice for the development of reuse methodology.

The fourth type is a tool developer, for example in an EDA company. In our opinion the System Design Conceptual Model (SDCM) of Chapter 4 forms the basis of any tool that is supposed to support system design as we see it. The action semantics of Annex A2 provides the core concepts for behavior-oriented tools, such as simulators and formal verifiers. Since any tool needs some kind of representation of the design, Chapter 5 is of interest to tool developers. Annex A3 can be of practical use for language selection. Depending on the nature of the tool, Chapter 6 may or may not be of interest.

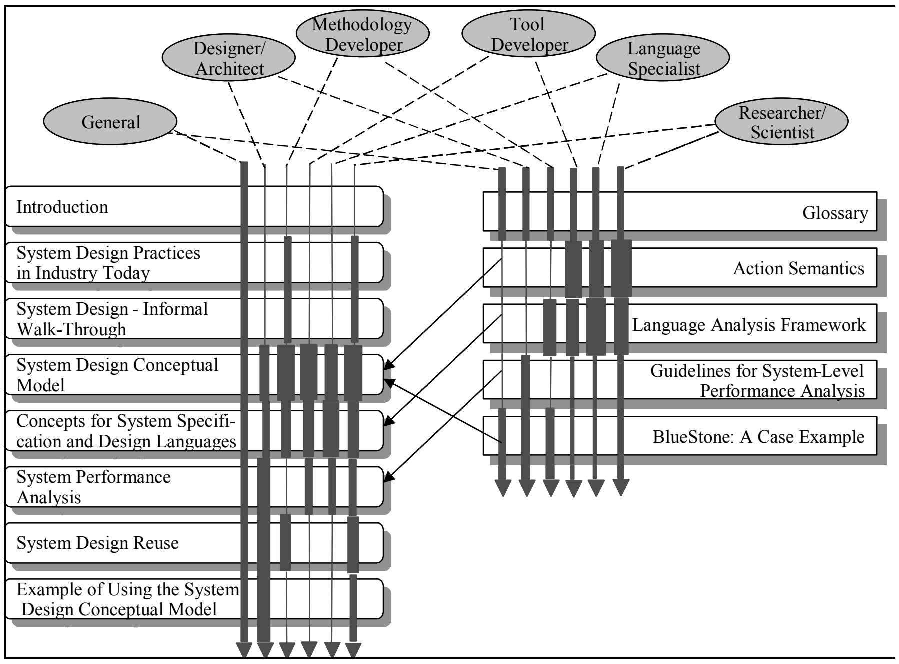

Methodology Developer
Tool Developer
Designer/ Architect
Language Specialist
General
Researcher/ Scientist
Introduction
System Design Practices in Industry Today
System Design - Informal Walk-Through
System Design Conceptual Model
Concepts for System Specification and Design Languages
System Performance Analysis
System Design Reuse
Example of Using the System Design Conceptual Model
Glossary
Action Semantics
Language Analysis Framework
Guidelines for System-Level Performance Analysis
BlueStone: A Case Example

Figure 1-1. Focus areas and reading flows for different types of readers.

The fifth type is a language specialist in industry, academia or standardization bodies. Obviously Chapter 5 and Annex A3 are most interesting sections for such a reader. However, the SDCM of Chapter 4 should not be skipped, since Chapter 5 is written with the assumption that the reader is familiar with the SDCM.

The sixth type is a researcher or scientist. From the scientific point of view, the SDCM of Chapter 4 is the key contribution of this book. However, all the chapters of the book provide ideas for further research. Especially Chapter 7 is of interest in this respect.

Figure 1-1 presents the suggested focus areas and reading flows in a graphical form. The chapters are on the left side of the picture and the annexes are on the right side. The arrows show how the annexes are related to the chapters. Notice that the Glossary of Annex A1 is used in all the chapters, although this is not shown explicitly in the picture.

Chapter 2

SYSTEM DESIGN PRACTICES IN INDUSTRY TODAY

Klaus Kronlöf[1],
Nikolaos S. Voros[2]
[1] Nokia Research Center, Helsinki, Finland
[2] INTRACOM S.A., Patra, Greece

Abstract: This chapter introduces the domain of the book and describes the basic steps of current industrial design process. The results of the analysis of real-life design flows in the participating companies are presented.

Key words: System design, system architecting, design flow, design process, design reuse, intellectual property reuse, tacit know-how.

1. BASIC STEPS OF AN INDUSTRIAL DESIGN FLOW

The companies involved in embedded system design have usually in-house design methods and practices and use specific languages for system design. In most cases, industrial design practices usually involve design flows reflecting the background experience of each company. In the next paragraphs, we present the basic design steps currently used for the design of real world applications.

1.1 Informal System Specification

An embedded system design that encompasses both hardware and software, starts with an informal system specification which is usually written as a document that describes high-level aspects of the system. These informal system descriptions are formalized during the next steps of the design process.

1.2 Formal System Specification

The formal system specification is carried out using appropriate formal specification languages based on the requirements posed by the informal specification. It is also a common practice to employ multiple formalisms for describing different parts of the same application. This is either due to the suitability of a specific formalism to accommodate the efficient description of certain parts, or due to the necessity to reuse existing components. The use of more than one formalisms is a bottleneck in the design process since there is usually no direct connection among the different formalisms used.

1.3 Architecture Exploration and System Partitioning

The architecture exploration and system partitioning phases are mainly based on the engineering experience of the designer, and employ informal system specification (e.g. block diagrams) of the system as input. The final system architecture emerges as the result of several successive trial and error iterations of this step. The use of informal specifications usually implies either lack of formal specifications, or bad abstraction level for the latter (not purely functional but implementation specific). The problem remains even in the case of having formal specifications available; if more than one independent specification language is involved for describing system models, they are usually independent among each other.

1.4 Concurrent Hardware and Software Development

Concurrent (and almost independent) hardware and software development is the next design-step. The interaction required between the hardware and software design teams is achieved through informal system specifications. Nothing ensures the lack of inconsistencies and misunderstandings between the hardware and software engineers, as there is not a unified functional system representation and a systematic way for solving concurrent engineering problems. The development of hardware modules takes place involving simulation using HDLs (Hardware Description Languages) and synthesis using commercial tools. Regarding the software components, they are described in a formal description language and then translated to C (or other implementation) language. The next step is software development using algorithmic simulation of the software parts using compilers, assemblers etc. Driver development and verification of their functionality takes also place through the use of instruction set simulators, extended with the required dummy hardware models [1].

The ultimate goal of this hardware-software co-development stage is to produce a design where both software running on a specific microprocessor, and system's dedicated hardware are offered as a system on a chip. CoWare [2] and Seamless [3] are typical co-design tools used in industry for hardware/software co-simulation and co-verification. The system description is given in VHDL or Verilog languages for hardware and C language for software. Both of them allow co-simulation between hardware and software at the same abstraction level.

2. OBSERVATIONS FROM DESIGN FLOW ANALYSIS

In order to set the baseline and to concretize the objectives, the project started by conducting an analysis of selected real-life design flows of the participating companies. The aim was also to derive generic characteristics of system design and to find areas for improvement. The method used for design flow analysis explicitly identified activities, actors, roles, formal and informal information flows as well as the reuse of artifacts and other Intellectual Property. In addition, we tried to analyze where the innovation really occurs and what kind of explicit and tacit know-how is involved. This chapter summarizes the main observations and conclusions.

2.1 System Architect Is the Key Actor in System Design

The system architect has the global system know-how, functionally as well as architecturally. The system architect also is well aware of the non-functional properties (performance, cost, etc.) and their impact on design decisions.

A lot of previous design-know and other experiences are also typically present. As the owner of the system architecture process, the system architect maintains the integrity of the system development activities, while taking care of the consistency and balance of requirements, design, implementation and verification.

Architecture design is a multi-level and multi-faceted activity. Besides architecture definition, it also consists of feasibility analyses and assessments. The architect addresses the system (under design) as well as the product development (process). Besides the technical content, a good system architect also takes into account: the organizational constraints (which type of skills available and needed); product cost issues; and trade-offs between short term needs and long term interests.

There is no systematic method and tool support available for this multi-level and multifaceted activity, but an architect should be capable to do it all in parallel.

2.2 Design Know-How Capture

The know-how to produce (good quality) designs is consolidated in the design flows, design processes and organizations. Examples of consolidated know-how are scripts, tools, design activities, checklists and guidelines, and involvement of same people (with different roles) in different activities.

2.3 Reuse of Intellectual Property

Intellectual property (IP) reuse happens but it is not guaranteed. Reuse awareness exists by designers but reuse is typically not institutionalized and consequently the opportunities for reuse cannot be qualitatively and quantitatively determined and assessed. The same applies to the creation of IP in the process of system design and development.

Among the partners of the project Philips was an exception in this area (which may be more typical for a semi-conductor company). A corporate IP repository had been installed and associated policies and procedures had been defined. The repository contains pre-defined IP (microcontrollers, DSPs, interfaces, etc.), and own application-specific IP. Dedicated support and maintenance groups for this IP have been set-up.

However, also in this case, IP in the repository are the (end-) results from normal business development projects, i.e. without explicit design-for-reuse intentions. Policies to determine whether offered IP should be added/included to the repository, and/or to solicit specific IP are missing. The lack of such IP characterizations and criteria means that the quality of the repository cannot be assessed. It requires business incentives to move up the reuse process maturity hierarchy. Separate development activities for identified IP and reusable components have to be recognized as having business value, and need to be established.

Typical examples of Intellectual Property (IP) are:

1. DSP algorithms, or more general, specifications of functional system modules

2. Subsystems with well-defined "interfaces" (i.e. in OSI (Open Systems Interconnect) protocol stack terms, e.g. physical layer). These subsystems can be used in later product derivatives.
3. Microcontrollers, DSPs, memories etc. These are typically external IP. And although high-valuable components, not considered to be of the system companies core business; they are a necessary need/use and can be "easily" exchanged in a design (at least: does not constitute and affect our own system IP).

2.4 Interfaces

By the nature of telecom systems, and by the nature of structured-analysis of system and architecture design, interfaces are key objects. Typically, many interface errors are caught during system integration. The introduction of hardware/software co-verification partially remedies this, and reduces the cost of these errors. But interfaces deserve to be a first-class citizen in the design flow.

2.5 System Validation

It is advantageous to validate systems early in the design and development flow. Errors found during system integration should be minimized. Conceptual errors should also be detected early. High-level executable models may enable earlier system validation. Issues to be considered are the consistency between models used in the different design stages, and the level of abstraction at which the models have to be written. These models should be easy and fast to develop. Executable models are not necessarily full-functional models. Possible alternatives include abstract token models, interface and bus-functional models and architectural models.

2.6 Concurrent Engineering

In a sense, the design flow and process is top-down, but with a large degree of concurrent engineering. The typical example is the concurrent development of hardware and software design. However, also the system and architecture phases run concurrently with the hardware and software design phases. Hardware design can even run fully ahead, for example in the case of a platform development.

A process is therefore a particular ordered view that not necessarily reflects the actual time ordering of a set of activities. Promoted new design flows and tool-supported design flows should take such concurrent engineering into account. In our view, component-based design and concurrent engineering are non-conflicting design approaches.

2.7 Hardware/Software Partitioning

In the design flow analysis, hardware and software design are separate design and development activities. These activities are performed in different organizational units. The applied processes fit a generic scheme. However, during the first stages of system and architecture design, hardware/software partitioning is not a clearly identified activity. This is counterintuitive from many top-down design flow descriptions as presented and promoted by research communities.

Defining the functional and physical architecture are both key (but separate) activities. Hardware/software partitioning is therefore a more informal and intuitive process and activity that are maybe more culture and application domain based. In that sense, hardware/software partitioning is more apparent in the design as defining the hardware platform onto which the application will be run. The partitioning itself is not performed and detailed, only its characteristics are considered to define the processing/processor platform. The definition of the exact hardware/software interface can be identified as a design step, which can be considered as a limited hardware/software partitioning.

2.8 Test and Debug

Design-for-test and design-for-debug should be considered during all phases of system design and its implementation. Test and debug are orthogonal to the design views. These views may currently be underestimated, but will become more important. Awareness may be sufficient, and with the appropriate measures and strategies defined, as it probably does not deserve to be the main paradigm of the design process.

3. AREAS FOR IMPROVEMENT

Architecture in all its facets is the main area for improvement. An architect has to compare and trade-off alternative solutions and implementations. Better means are needed to assess and support design decision taking, mainly at the architectural level. The system architect’s activities and roles have to be more formalized in order to achieve better reuse of design models, components, IP and know-how. We also need a paradigm shift from function-oriented design to component/ communication/interface-oriented design.

Systems should be validated for their correctness and their performances earlier than it happens today. Executable specifications are marketed as the solution. The caveat is that such an early modeling activity should not consume the same amount of resources as the current formal and simulatable specifications that are later in the design process. The reuse of testbenches and the ability to run mixed-level simulations are probably a pre-requisite for the success of such an approach.

Reuse should be made a well-defined and institutionalized activity. This typically requires changes in the organizational structure of companies as well as infrastructure investments. It is also necessary to be able to express and assess the quality of the designs to be reused.

The range of reusable artifacts should be expanded. Currently only implementation-oriented components can be reused. A big productivity gain is possible if also the design information of the earlier system design phases could be reused. This requires a more formal design approach that produces unambiguous models that leave no room for misinterpretations. It is also crucial to guarantee consistency of different models. Currently system design information too often becomes obsolete or at least inaccurate due to late bug fixes.

Current state-of-the art system design languages are quite expressive, imposing few restrictions of what can be described. Guidelines, style conventions, language subsets should be defined for the different stages/phases of the design flow, and which support a particular (system and architecture) design activity.

The languages are also geared towards representation of the end-result of the design activity. They do not lend themselves well to the description of the design intent, i.e. what it means and/or why it is being developed (semantics, basic concept, etc.). Having this information available allows for a higher level of reasoning. Other realizations, e.g. in another context or setting, can then be derived more easily.

REFERENCES

[1] Tsasakou S., Voros N., Birbas A., Koziotis M., Papadopoulos G., High-level co-simulation based on the extension of processor simulators. Journal of Systems Architecture 2001; 47:1-13.

[2] CoWare Inc, CoWare N2C Methodology Manual. 2001.

[3] Klein R. A hardware-software co-simulation environment. Proceedings of 7th IEE International Workshop on Rapid System Prototyping; 1996 June; Thessaloniki, Greece.

Chapter 3

SYSTEM DESIGN - INFORMAL WALK-THROUGH

Kari Tiensyrjä
VTT Electronics, Oulu, Finland

Abstract: This chapter introduces the reader to main system design concepts and gives a generic outline of phases and steps performed in system design. First, the scope of system design is discussed. Next, the phases called System Design - Refinement and System Design - Partitioning are addressed. Finally, considerations about reuse of system design know-how are expressed.

Key words: System design, functionality, System Design - Refinement, functional architecture, System Design - Partitioning, architecture template, specification, system design process, system under design, pattern, idiom, reuse, platform, system design know-how.

1. SCOPE OF SYSTEM DESIGN

A number of needs and inputs usually generate the idea for a new system. These could be market opportunities, brilliant intuitions, realisation of existing needs, technological progress, etc. These inputs construct the initial specification of the system, as **a set of user and domain requirements**, usually built upon, and prejudiced by, the pre-existing professional knowledge and habits of the originators. The first insight generates some brainstorming as the very beginning of the design of a new system.

From this stage, a sequence of progressive refinements will result in **a set of technical requirements specification** of the system to be designed as depicted in Figure 3-1. We call 'System Design' this Refinement activity – labelled briefly SD-R - as well as the succeeding one – 'System Design' to Partitioning, briefly SD-P - going from that specification of system functionality to the specification of the building blocks functions. These shall then be synthesised into real working parts in the activity labelled as SD-S, which however is not specifically addressed in this work.

In general, system design and the role of system designer can be considered from the gathering of customer needs until the volume reproduction. The perspective taken here is more limited, the SD-R and SD-P could together be also called system architecture creation.

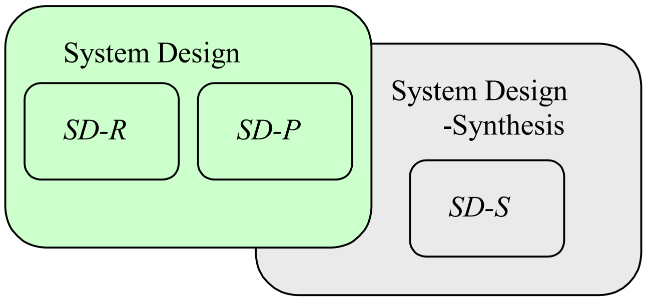


Figure 3-1. Scope of system design.

2. SYSTEM DESIGN - REFINEMENT

The SD-R elaborates the functionality requirements of the system until the functional architecture as outlined in Figure 3-2.

At SD-R, the background knowledge is entered by means of some pre-conception:

- The semantic concepts familiar to the participating designers, firstly reshuffled to fit a common understanding, are taken as building blocks of a specification.
- The experience on known system organisations and their working is the layer of architectural ideas that set a “frame” to the progressive identification of the functionality. This knowledge contributes to give consistency to the ideas by assigning some functional task to 'transformation' activities, and some other task to 'communication' between them.
- The refinement based on common language generates constructs that are often understood as underpinning to the corresponding roles between parts: fixed versus programmable, hardware versus software, control versus processing. This identification is the seed for the definition of system architecture.

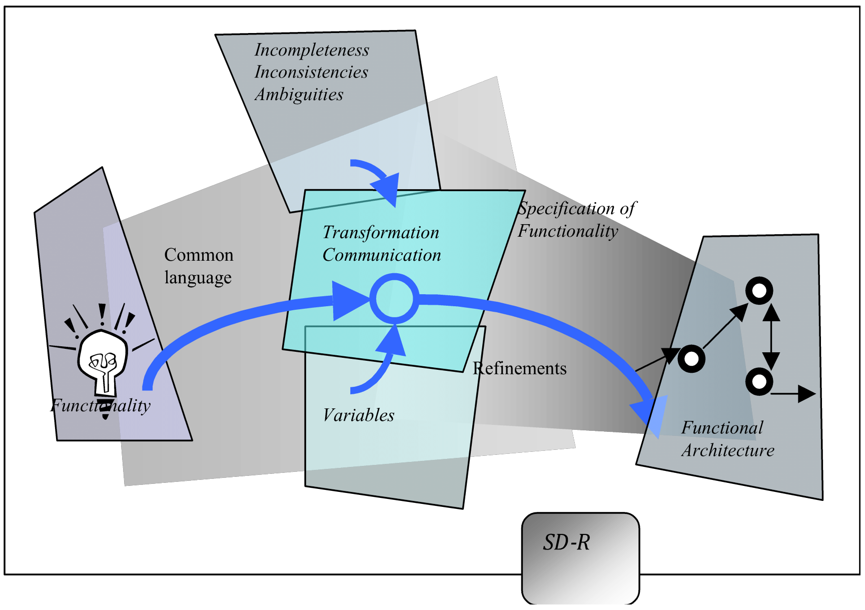


Figure 3-2. System Design - Refinement.

- Some of the constraints of the initial specification become satisfied, while refinement introduces new derived constraints due to e.g. communication, power, cost etc. concerns. The personal sensibility of every designer assigns some 'variables' either to system parameters or to system constraints, depending on whether they are considered to be presented as delivered characteristics of the system or to be managed as adverse inputs to be accepted or design ranges to be restrained.

Incompleteness, inconsistencies and ambiguities are the richness of design in this refinement stage; their removal by specification and choice will define the particular architecture and performance of the final specification.

The study of the way of proceeding concerning this phase of system design is related to the way of making refinements. This study is what is properly referred to as 'methodology', as the study of the methods that could be adopted. We see here that the common language used to share the ideas between engineers is simply dictated by the need to adopt shared semantic classes, and the issue is "understanding each other applying the minimum possible bias from one discipline to another".

The specific methods to select what task, or part of it, is to be treated as a transformation on data; what other is considered as communication between stages; how their connection is organised, as well as their access and share of a set of information repositories, represent methodologically a "what-if" procedure. After these choices, successive refinements result in **a set of system/architecture specification** consisting of design artefacts produced in the course of the SD-R.

The amount of variables - constraints or parameters - and the amount of inconsistencies - ambiguities and incompleteness - assigned and solved or left in the specification, depends on the degree of freedom that is planned for the subsequent stage of generation of synthesisable parts, somewhat approximately called 'partitioning'.

3. SYSTEM DESIGN - PARTITIONING

As the conclusion of the SD-R phase, a specification is available as a description of a functional system, made of transformation and of communication functions.

While formal refinements can continue to be applied to the specification, they must now be associated with another method to transform the description into one that can enter a deterministic synthesis process. At the SD-P phase, depicted in Figure 3-3, the toolkit to play with is no more just a basket of semantic primitives, but is based on modelled or descriptive portraits of reference instances.

According to models of computation, the specification is mapped on different architectural templates, or architectures. Concurrently, the architectural functions, concerning computing and communication, can be built using "objects", or functional blocks, that can in turn be mapped to their software or hardware implementations resulting in **a set of system/architecture description**, which contains design artefacts produced at SD-P.

At present there is more heuristics than theory about the optimal choice of the methods for the design phase from functionality to functions. Very often the procedures follow specific flows according to the chosen design tools and design quality standards - we designate these flows as **idioms** - but every one choice can be traced to belong to a family of **patterns**. And in turn these can be traced to a couple of fundamental archetypes: one in which the designers impose the adoption of functional blocks and arrange the architecture around them, and the opposite, in which the most performing architectural template is chosen and is then implemented by means of existing building block objects. Both ways then describe the function by means of synthesis languages with associated tools, which offer both verification and synthesis support.

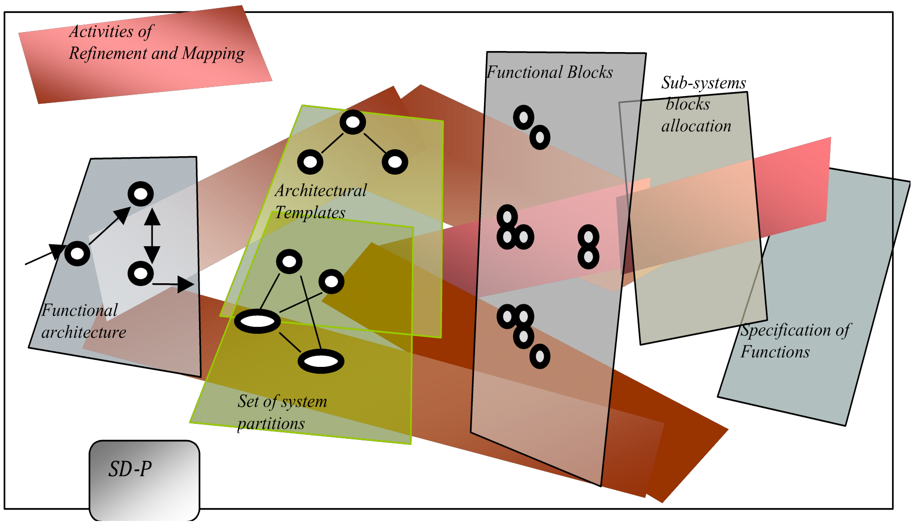

Activities of Refinement and Mapping
Functional Blocks
Sub-systems blocks allocation
Architectural Templates
Functional architecture
Specification of Functions
Set of system partitions
SD-P

Figure 3-3. System Design - Partitioning.

This second set of steps is largely based on the **reuse of instances of knowledge** more detailed than just semantic concepts, as it was predominantly in the first phase. This reuse, often demanded within a design strategy of design-for-reuse, can apply bottom-up (a design is forced to build up on specific pre-existing subsystems, inheriting parts, this is often called design-to-reuse), or top-down (a design is oriented to take advantage of pre-existing parts within a set to choose from, made of hard or flexible items. This can be quoted as design-with-reuse).

Despite the simplification given by the reuse of known functions, as IP instances, the process of moving from the functionality to the specified function generates a large amount of detail information, and its verification becomes cumbersome. That is why often instances of architectures, or **platforms** are adopted, to build different systems by simple variations of the sort of functions they make – like with different software procedures.

A platform is a generic architecture and a reference design where to store know-how that is important both for the different instances of system under design within the problem domain, and for the organisation of the development work, i.e. the design process. A platform is also a means of know-how and technology transfer.

The role of a platform is also that of "concentrating" a class of systems under design to collapse on it, thus reducing the multiplicity of alternatives and the design complexity. It must also, as much as possible, be able to target as many implementation alternatives as possible. A platform is thus a sort of compromise between the aim of a legacy IPI and that of a widely applicable IP.

A coarse discrimination can be made between two types of platforms based on degrees of flexibility versus performance and cost:

- Product platforms, which aim at sharing parts of systems in order to capitalise on the commonality between them. It can be considered as a layered architectural environment for a system that facilitates the development of an architectural instance.
- Integration platforms, that are defined as an underlying enabling technology on which the object of reference is rendered functional. It can be considered as a set of interoperable subsystems with a set of rules, which enable third party subsystems to be included. Integration platforms are often identified with their technological soul, of being respectively either portable between technologies or user programmable.

The whole set of steps and procedures that we have mentioned is called design process, and can be methodologically studied by means of meta-models of both design items and design procedures. The system design process (SDP) describes how the development of system under design (SUD) in a specific organisation is arranged, i.e. it determines the type and order of stages that are involved in system development and establishes the transition criteria between adjacent phases.

4. REUSE OF SYSTEM DESIGN KNOW-HOW

The development and deployment of electronic systems have for decades relied on reusable components that are based on standards, and have been put forth in catalogues, libraries and design rules of how to assemble electronic systems out of these components. Starting from transistors and primitive logic circuits in the sixties the principles have evolved so that one can now buy sophisticated processors, memories, I/O sub-systems etc. as commercial off-the-self components. In software engineering similar historical threads, although in much more diversified ways, can be recognised.

Design reuse became a commonly accepted panacea in nineties. Two main threads are related to the virtual component paradigm represented by the VSIA [1] and the object modelling paradigm represented by the OMG [2]. Further proposals based on these are e.g. the platform-based design [3] and the model-driven architecture [4], respectively. Although building foundations, none of the above has been able to address explicitly how to reuse design expertise. That is why, the software engineering community welcomed in the nineties the technique called patterns that originated from the work of building architect Christopher Alexander:

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. As an element in the world, each pattern is a relationship between a certain context, a certain system of forces, which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant. The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing ".

In the software engineering, motivations are expressed as follows [5]:

"*One of the first things that any science or engineering discipline must have is a vocabulary for expressing its concepts and a language for relating them together. Patterns help create a shared language for communicating insight and experience about common problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge, which comprises our understanding of good architectures that meet the needs of their users. Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and desig*n".

The identified needs of the SYDIC-Telecom are in principle similar [6], but the domain is now system design and the patterns should be found and adapted accordingly. Previously, patterns have been applied to e.g. organisation structures, analysis of business applications, micro-architectures of object-oriented software design and software engineering processes. The hypothesis of the SYDIC-Telecom project is that patterns are a viable mechanism for obtaining reuse of system design expertise.

5. SUMMARY

This chapter has given a short introduction to the system design and elaborated related key concepts. The scope of system design was defined to consist of refinement and partitioning phases. Methodology and means of making system design expertise sharable and reusable were also outlined in order to facilitate dealing with rapidly increasing complexity in all aspects of system design.

REFERENCES

[1] On-Chip Bus (OCB) Attributes Specification Version 2.0, VSI Alliance (VSIA), 2001.

[2] Unified Modeling Language Specification Version 1.3. Object Management Group Inc., 1999.

[3] Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A., Todd, L. Surviving the SOC Revolution. Kluwer Academic Publishers, 1999.

[4] Model Driven Architecture. Object Management Group Inc., 2001.

[5] Appleton, B. (2000). Patterns and Software: Essential Concepts and Terminology. Retrieved January 2003 from: www.enteract.com/~bradapp/

[6] De Jong, G. (ed.). SYDIC-Telecom Perspective on IP & Reuse, Proceedings of the Forum on Design Languages, FDL'2000, 4 - 8 September 2000.

Chapter 4

SYSTEM DESIGN CONCEPTUAL MODEL

Kari Tiensyrjä[1],
Jean Mermet[2]
[1] VTT Electronics, Oulu, Finland
[2] ECSI, Grenoble, France

Abstract: This chapter presents the foundations of the System Design Conceptual Model (SDCM). The SDCM is a meta-model that serves as a reference model of, and gives a global view and perspective on system design. The SDCM is used to describe system design from the viewpoints of the System Design Process (SDP) and the System Under Design (SUD). The SDP and the SUD are related to each other via design artefacts produced and consumed during the design process. The SDCM can be used in various ways in enhancing system design: understanding system design; analysing and assessing existing design flows; instantiating design flows for new design paradigms; eliciting requirements for methods and tools; organising teams; educating employees, partners and customers.

Key words: System design, System Design Conceptual Model, meta-model, reference model, System Design Process, System Under Design.

1. OVERALL CONTEXT OF SDCM

The overall context of the SDCM is depicted in Figure 4-1. The SDCM considers system design phases of system development that constitute the core technical part of the product creation process of an enterprise [1].

As to system design reuse, the SDCM considers both the design process know-how and the design artefacts.

System IP related to design processes can be:

- Components of the process
- Know-how of methods (analysis, synthesis, etc.) encapsulated in rules and guidelines (possibly implemented by tools and design patterns)

- Know-how of design styles (modelling, verification, etc.) encapsulated in checkers
- Know-how of use of tools encapsulated in scripts.

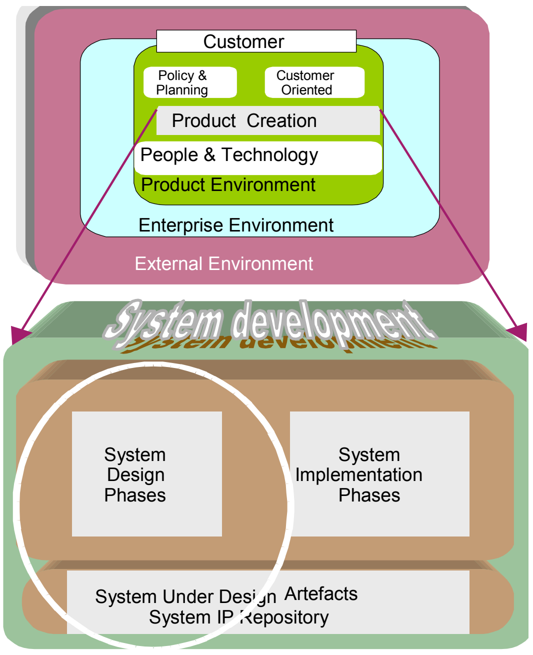


Figure 4-1. Overall context of SDCM.

System IP related to design artefacts, can be:

- Algorithmic knowledge
- Application components
- System (HW and/or SW) architecture
- System components that are in the stable core area of the domain, i.e. probability for reuse is high
- Out-source IPs that are developed and maintained by 3rd party on behalf of system house
- Pre-defined star IP.

In general, it is assumed that design-for-reuse methodologies are applied to system IP. System IP shall be documented, packaged and stored in a kind of repository. In addition to repositories, design patterns and process patterns are concepts that are promoted to encapsulate design and process know-how.

2. SYSTEM CONCEPTUALISING AND MODELLING

This section introduces basic concepts related to conceptual modelling and ontology. The main constructs of a conceptual model are first described, and then the role of ontology in meta-modelling is outlined.

2.1 Conceptual Model Constructs

The conceptual model captures the meaning of an application domain as perceived by someone [2], i.e. knowledge about a real-world domain. Real-world is perceived as **things**, often referred to as **entities**, and **associations**, often referred to as **relationships**.

The world is made of things that possess **properties**. The properties of a thing exist, whether or not we are aware of them, and properties are always attached to things. The notion of **concrete thing** applies to anything perceived as a specific object by someone. We conceive of things, however, in terms of models of things. Such models are **conceptual things**. The properties of conceptual things are termed **attributes**. Attributes are characteristics assigned to models of things according to our perceptions.

A **class** is a representation of a set of things having common properties in a conceptual model. An attribute in a conceptual model is a representation of an **intrinsic** (i.e. dependent only on the thing itself) property of a thing in the real world. A relationship in a conceptual model is a representation of a **mutual** (i.e. dependent on two or more things) property of a thing in the real world. Depending upon circumstances, we may use different models of the same thing, and therefore assign different sets of attributes to the same thing. An attribute, however, may or may not reflect a substantial property (i.e. a property of a concrete thing). Moreover, in a given model, not every property will be represented as an attribute.

The UML-like notation is used in this document to visualise graphically the conceptual model [3]. For those who are not familiar with the UML, an introductory book [4] is recommended.

2.2 Ontology

An ontology is a description of **concepts** and **relations** that exists in a particular domain such as an application area [5]. The advantage of an ontology is getting rid of several problems usually linked to natural language vocabularies. There are several levels and kinds of ontologies [6]. Usually there are three kinds of information (or levels) inside a given ontology [5]:

1. **Terminology** level. This is the basic set of concepts and relations constituting the ontology. It is sometimes called the definition layer of the ontology.
2. **Assertion** level. This is a set of assertions applying to the basic concepts and relations. It is sometimes called the axioms layer of the ontology.
3. **Pragmatic level**. This is the so-called toolbox layer. It contains a lot of pragmatic information that could not fit in the terminology or assertion levels.

The main properties of an ontology are **sharing** and **filtering**. Sharing means that an agreement may exist, based on the acceptance of common ontology, about the same understanding of a given concept. Filtering is linked to abstraction of models that take into account only a part of the reality. Usefulness of models is based on the ability to filter out undesirable characteristics. Ontology defines what should be extracted from a system in order to build a given model of this system.

The basic use of ontology is that it facilitates the separation of concerns. When dealing with a given system, we can observe and work with different models of this same system, each one characterised by a given ontology.

Let us call the notion of context as space, i.e. a model is a space, a meta-model is a space, and a meta-meta-model is a space. Most recent meta-meta-model proposals consist of the three basic notions: {concept, relation, space}. In Figure 4-2 there are two spaces presented, a model X and a meta-model MX.

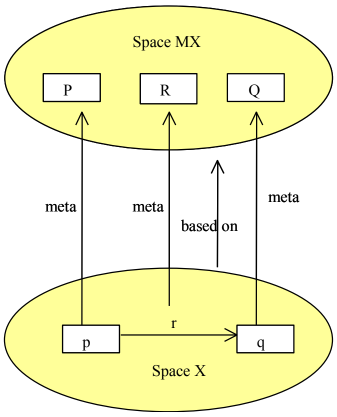


Figure 4-2. Nature of meta-model.

For each entity present in X, there is a corresponding meta-entity present in MX. The relation r(p, q) is defined in X, but the concepts P and Q and the relation R are defined in MX. The relations meta(p, P) and meta(q, Q) hold. MX is the ontology of X and this corresponds to the relation BasedOn(X, MX).

A **model** is always built for a given **purpose**, usually of understanding some aspects of the system. This purpose should be clearly defined and associated to the ontology. Ontology considers the triad {**system, ontology, model**} as depicted in Figure 4-3, but the ontology-based extraction task will have to be performed by an actor. Ontology corresponds to the classical definition of a meta-model as it is used e.g. in the UML. Ontology contains the concepts and the relations that are relevant to a given modelling task.

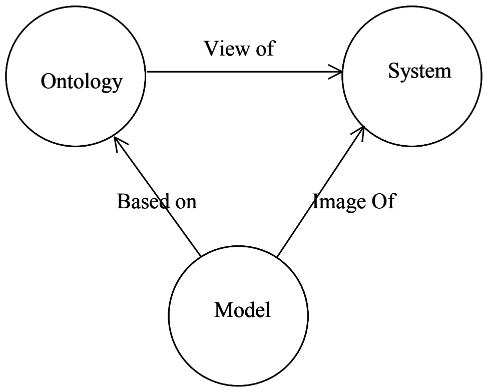


Figure 4-3. Triad {system, ontology, model}.

3. MODEL OF DESIGN PROCESS

This section presents the foundations of the System Design Process Model (SDPM). Firstly, the context of the SDPM is outlined, and then the basic concepts and their relationships are described using four main views.

3.1 Context of System Design Process Model

The context of system design and implementation is a partially ordered set of facets from an idea to an implementation. In this work the capture of user and domain requirements is not specifically addressed, neither the synthesis to implementation. Figure 4-4 depicts facets of a generic design process, output artefacts of the facets, various languages, and global views to the subject of design.

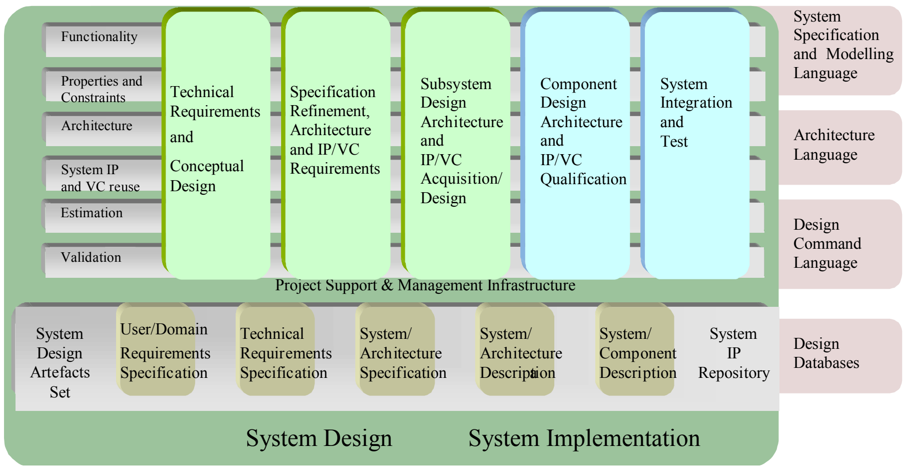

Functionality
Properties and Constraints
Architecture
System IP and VC reuse
Estimation
Validation
Technical Requirements and Conceptual Design
Specification Refinement, Architecture and IP/VC Requirements
Subsystem Design Architecture and IP/VC Acquisition/ Design
Component Design Architecture and IP/VC Qualification
System Integration and Test
System Specification and Modelling Language
Architecture Language
Design Command Language
Design Databases
Project Support & Management Infrastructure
System Design Artefacts Set
User/Domain Requirements Specification
Technical Requirements Specification
System/ Architecture Specification
System/ Architecture Description
System/ Component Description
System IP Repository
System Design
System Implementation

Figure 4-4. Overall design process.

The generic facets of system design and respective artefacts considered in the SDPM are:

- Technical requirements and conceptual design resulting in the technical requirements specification. This corresponds to the layer L1 of the concepts for the System Under Design (SUD), which will be described in more detail later in this chapter.
- Specification refinement and architecture and IP/VC (Intellectual Property/Virtual Component) requirements resulting in system/architecture specification. This corresponds to the layers L2 and L3 of the concepts for the SUD.
- Sub-system design and architecture and IP/VC acquisition and/or design resulting in system/architecture description. This corresponds to the layer L4 of the concepts for the SUD.

The following global views to the system under design (SUD) are considered:

- Functionality, which is refined and relates later to the behaviour of the system.
- Properties and constraints that are either invariant or later refined and relate to the system and its environment.
- Architecture, which encompasses different kinds of architectures during refinement.
- System IP and VC reuse, which denote using of pre-existing know-how or reusable components during refinement and design.

- Estimation is a technique that aims at providing estimates of the consequences of design decision, and traverses all the facets of system design.
- Validation is a technique that aims at proving that what is designed is the right thing and that the result of design is correct. Also validation traverses all the facets of system design.

As presented in the next chapters of the book, appropriate means to represent the views considered are:

- System specification and modelling language that describes the functionality and properties of the SUD all along the design process.
- Architecture language that describes the different structures of the facets of the system and relations between them.
- Design command language, which allows exercising the model in the frame of different applications.
- Design databases that store in retrievable way the design artefacts produced and consumed.

A lot of concurrency is involved between and within design facets. Especially, validation of results and estimation of outcome are globally applied to qualify the design contents of facets and to predict impacts of decisions made.

3.2 System Design Process Model

This section formalises the constituents of a System Design Process by presenting a System Design Process Model (SDPM). The model is represented using UML [3, 4]. The SDPM uses as a background the Software Process Engineering Metamodel (SPEM) [7].

As the Software Process Engineering Metamodel (SPEM) is on the way to becoming a standard, it is beneficial if it is also applied to system design. This is however not a one to one mapping, given the differences in scope of Software Engineering and System Design Processes. The SPEM was studied as a prerequisite to defining the SDPM and the SDPM tries to reuse SPEM definitions and concepts where possible. Of course, this was not always possible and some concepts are redefined in the SDPM.

In order to achieve manageability, the presentation of the SDPM is divided into four views:

- The Core View depicts the conceptual model and relates the key concepts: Artefact, Activity, Role and Actor

- The ProcessDefinitionElement View identifies the major components of the SDPM: Resource, WorkDefinition, Actor, Role, Artefact and Tool
- The WorkDefinition View relates work elements: ProcessLifeCycle, Phase, Iteration, WorkFlow and Activity
- The ProcessComposition View shows how to compose Process from ProcessComponents and ProcessDefinition Elements.

3.2.1 Core View

Activity is at the centre of the Core View of the SDPM as shown in Figure 4-5.

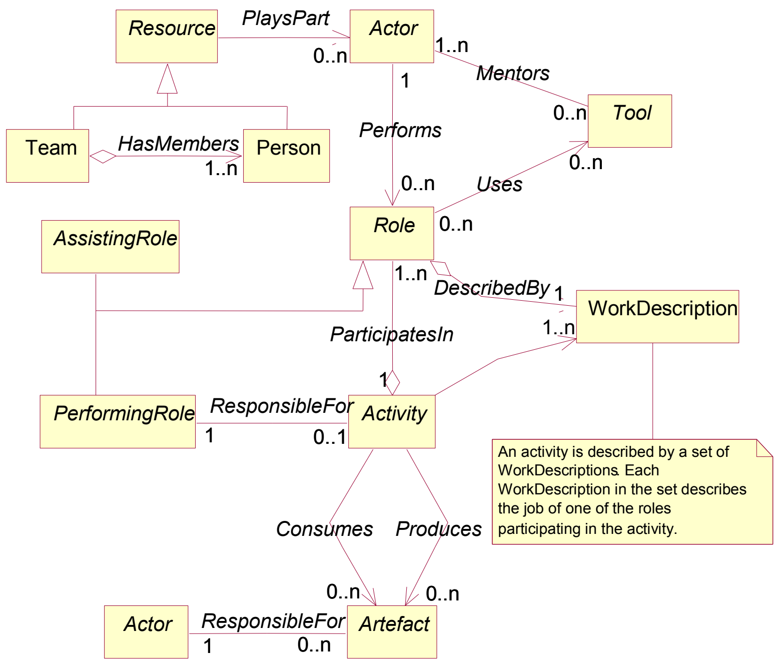


Figure 4-5. Core View.

Activity denotes work performed by Roles on Artefacts. Activity has a number of Roles, each of which is filled by exactly one Actor who performs the Role. The same Actor may perform Roles in different Activities. This makes visible what would normally be considered as an informal dependency between Activities. Our experience is that this type of dependency is crucial for the process to work. It therefore deserves to be a formal part of the process description, which is how we have treated it.

There are two basic types of Roles to be played in an Activity. They are AssisitingRole and PerformingRole. Exactly one of an Activity's PerformingRoles should be designated ResponsibleFor the Activity. A Role optionally uses one or more Tools to perform its task.

Each Tool should have at least one Actor specified as Mentor to provide support in its use. The Tool Mentor is someone with specific competence in the Tool. This applies even outside of the design process and is applicable to a Person. Mentor could be seen as a Role but it is separated out to show that this special Role is not part of the process of design itself but it is obligatory with respect to Tools.

The work to be done and responsibilities of each Role in an Activity is captured in the Role’s WorkDescription. The set of all Role WorkDescriptions completes the Activity description.

An Activity consumes and produces Artefacts. For each Artefact, a single Actor must be assigned as ResponsibleFor.

Actor is a logical entity and does not represent a physical Person. Resource planning is a specialisation of Activity whereby Resources are allocated to be Actor’s within the Process. A Resource can be a physical Person or a Team.

3.2.2 ProcessDefinitionElement View

ProcessDefinitionElement, shown in Figure 4-6, is the superclass for all major components in the SDPM. Actor, Role, Artefact, Tool, Resource and all WorkDefinition classes are ProcessDefinitionElements.

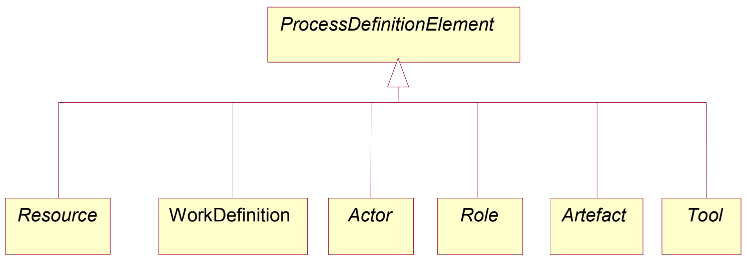


Figure 4-6. ProcessDefinitionElement View.

Activity is the main subclass of WorkDefinition. Other subclasses are ProcessLifeCycle, Phase, Iteration and WorkFlow.

3.2.3 Work Definition View

The following are definitions of the SDPM work elements as shown in Figure 4-7. WorkDefinition is the superclass of all work elements. A WorkDefinition HasEntryCriteria and MeetsExitCriteria. Entry criteria are captured as WorkPrecondition's, while exit criteria are captured as WorkGoal's. A WorkGoal can be a Milestone, in which case it represents a crucial goal that serves as a Go-No Go decision point.

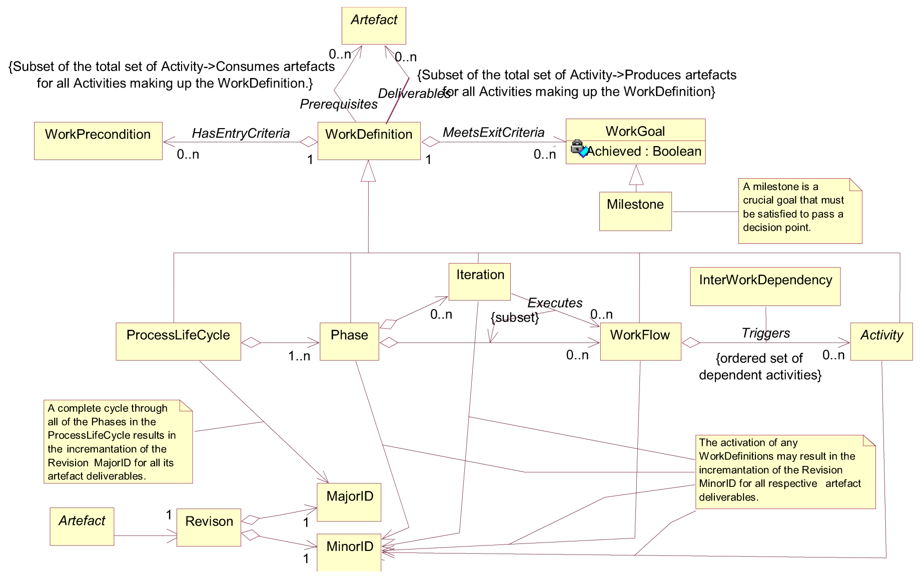

Artefact
0..n
0..n
{Subset of the total set of Activity->Consumes artefacts for all Activities making up the WorkDefinition.}
{Subset of the total set of Activity->Produces artefacts for all Activities making up the WorkDefinition}
Deliverables
Prerequisites
WorkPrecondition
HasEntryCriteria
0..n
1
WorkDefinition
MeetsExitCriteria
1
0..n
WorkGoal
Achieved : Boolean
Milestone
A milestone is a crucial goal that must be satisfied to pass a decision point.
Iteration
InterWorkDependency
Executes
0..n
{subset}
0..n
ProcessLifeCycle
Phase
1..n
WorkFlow
0..n
Triggers
Activity
{ordered set of dependent activities}
0..n
A complete cycle through all of the Phases in the ProcessLifeCycle results in the incremantation of the Revision MajorID for all its artefact deliverables.
The activation of any WorkDefinitions may result in the incremantation of the Revision MinorID for all respective artefact deliverables.
MajorID
Artefact
1
Revison
1
MinorID
1

Figure 4-7. WorkDefinition View.

A WorkDefinition can specify lists of Artefacts, as shown in Figure 4-8, that are Prerequisites and Deliverable (in given states if relevant).

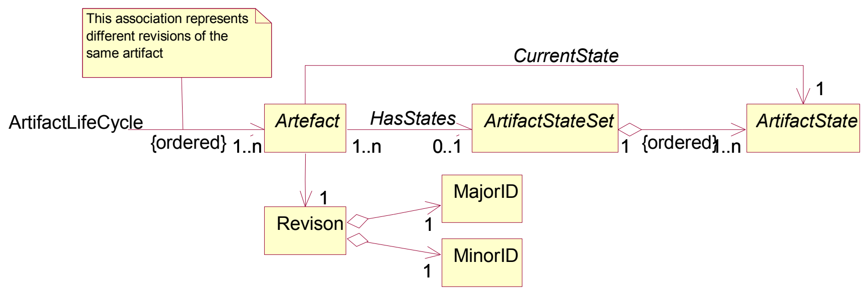


Figure 4-8. Artefacts.

A Phase is a specialisation of WorkDefinition bounded by two conditions: a precondition that defines the entry criteria and a goal, Milestone in this case, that defines the exit criteria. Phases are defined with the added constraint of sequentiality; that is they are executed with a series of Milestone related dates spread over time and often assume minimal (or no overlap) of their activities in time.

A Process Lifecycle is defined as a sequence of Phases that achieve a specific goal. It defines the complete process to be enacted in a given project or program. It is a Process.

An Iteration represents the activation of a set of Workflow's. This set being a subset of the Workflow's making up a Phase.

A Workflow represents a collection of interdependent Activity's (as per InterWorkDependency) with their corresponding Actor's, Artefact's and Tool's. The definition of InterWorkDependency does not restrict interdependency between Workflow's. Such interdependencies, as depicted in Figure 4-9, however should be restricted to the Prerequisites and Deliverables associations to Artefacts that are inherited from the superclass to Workflow.

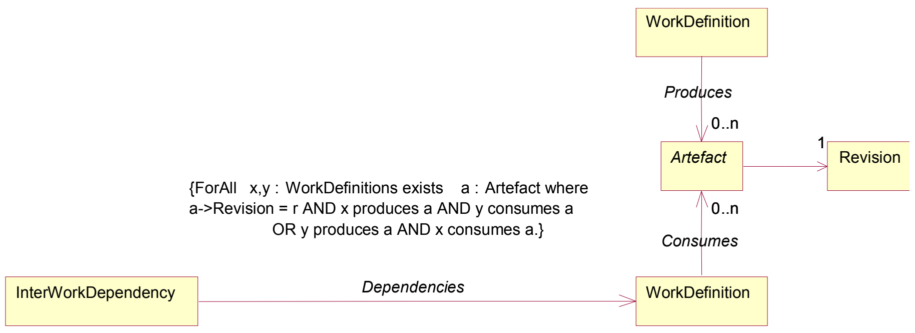

WorkDefinition
Produces
0..n
Artefact
1
Revision
{ForAll x,y : WorkDefinitions exists a : Artefact where a->Revision = r AND x produces a AND y consumes a OR y produces a AND x consumes a.}
0..n
Consumes
InterWorkDependency
Dependencies
WorkDefinition

Figure 4-9. Interdependencies.

3.2.4 Process Composition View

A Process is composed of one or more ProcessComponent's as shown in Figure 4-10. A Process is in itself a ProcessComponent thus allowing ProcessComponents to be hierarchically combined. A ProcessComponent consists of a self-contained set of ProcessDefinitionElement's. This set can include specialisations of ProcessLifeCycle, Phase, Iteration, Workflow and their constituents.

ProcessComponents are the building blocks that can be composed to create a complete process. ProcessLibrary represents a repository for ProcessComponent's.

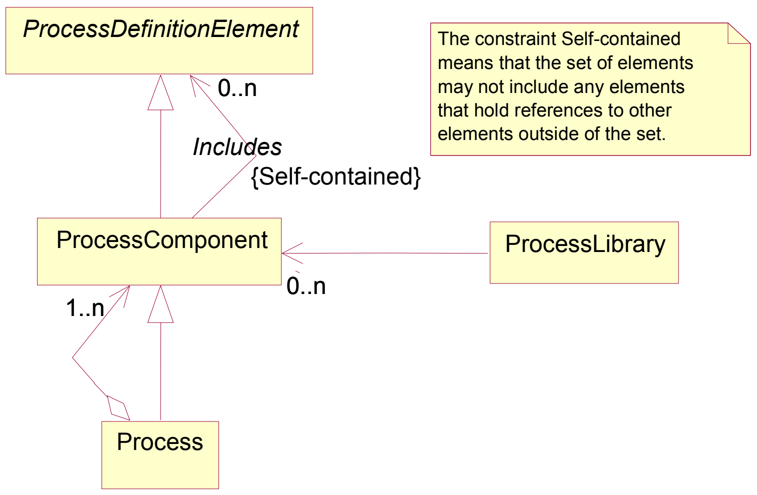

ProcessDefinitionElement
0..n
Includes
{Self-contained}
The constraint Self-contained means that the set of elements may not include any elements that hold references to other elements outside of the set.
ProcessComponent
ProcessLibrary
0..n
1..n
Process

Figure 4-10. Process Composition View.

4. MODEL OF SYSTEM

This section presents the foundations of the System Under Design Model (SUDM). The purpose of the SUDM is to define the concepts, and relationships between them that are needed for representing a subject of design, i.e. design contents, during the various phases of system design and according to various views of stakeholders.

In this document, the set of models representing the conceptual entity of system that is being designed is called as System Under Design (SUD), and the corresponding set of meta-models is called as System Under Design Model (SUDM).

4.1 External and Internal Viewpoints to SUD

The SUD can be observed either from external, i.e. environment-centric or internal, i.e. system-centric viewpoints.

Use or design of any artefact is triggered by a **need** or a desire by some human in some context. The ontology of needs and goals is the same as that of functional descriptions, both are represented as desired behavioural constraints in some universe. Needs often undergo a sequence of **transformations** before they become the **specifications** for an artefact. Need has to be recognised as something to be satisfied, and some human has to set up a goal or have a purpose to satisfy the need. Problem solving produces a sequence of transformations of the need such that **objects** or object configurations, and **means of interacting** with them, can be identified so that the need can be satisfied.

Design exists in order to deliver artefacts that have desired **functionalities**. The designer's job is to design artefacts that are intended to have certain **functions** (as services). When designing, designers look for **components** that can achieve certain functions (as services). In **compositional design**, the designer uses components from a component **library** to specify a **set of components** and **relations** between the components as a design. As the designer creates candidate designs by composing components, he needs to verify that the system in fact has the **properties** or the **behaviours** that can satisfy the **functionality requirements**. When choosing components from a component library, a designer might come up with a design in which only the function (as service) of a certain component is identified, but not yet the component itself.

4.2 Concepts for SUD

System is a **thing** that exists in its **environment**, together they are called as the **universe of discourse**. System is something of interest as a whole or as comprised of parts. Therefore a system may be referred to as an entity. A component of a system may itself be a system, in which case it may be called a **subsystem**.

The systems we consider are technological entities, i.e. they are artefacts with properties that agents in the environment, e.g. users and other objects, interact with and expect to cause desired effects.

In the context of system design, we are interested in **design artefacts**, which can be anything produced and/or consumed in the course of system design. For a system designer, artefacts are **views** to the SUD according to viewpoints that define various **concerns** (interests) of the system designer.

Abstraction is the main categorisation of design artefacts. In general, there are various ways and criteria of how to organise **abstraction layers** so that they are useful in our understanding of system design. The SUD layers and related concepts are depicted in Figure 4-11. The correspondence of layers and the main sets of **design artefacts** is roughly as follows:

- Above Layer 1: User and domain requirements specification
- Layer 1 (L1): Technical requirements specification
- Layer 2 (L2) and Layer 3 (L3): System/architecture specification
- Layer 4 (L4): System/architecture description.

The subsequent discussion is structured according to the abstraction layers.

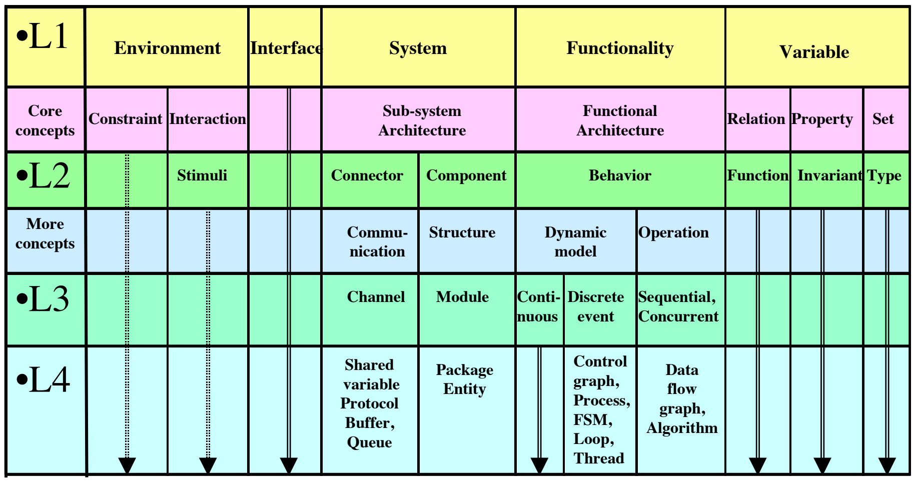


Figure 4-11. SUD layers and concepts.

4.2.1 User and domain requirements

The system under design (SUD) appears first in the form of **user and domain** requirements, because we consider technological systems that have users or other agents in the environment. The **interface** of the system to its environment is another source of requirements. The environment can conceptually be considered as another system.

The **use model** typically concentrates on **functionality** and **usability** issues, but may also contain **non-functional requirements**. Functionality requirements are expressed as a set of **services** users or other external agents expect from the system.

Other important set of requirements are those of other **stakeholders** that have interests in or work on the SUD. These requirements often are more oriented towards non-functional issues, like those related to development and performance, but may provide additional functional requirements.

4.2.2 Layer L1 abstraction

At the highest level of model abstraction of the SUD we find the notions of **system** itself, defined by **functionality**. We identify main **states** of the SUD, either by the **sets** they belong to or by **variables**. There exist **relations** between variables. We also discover **properties**. Some constants of the informal specification appear like parameters of the system in the technical requirement specification.

4.2.3 Layer L2 abstraction

Refinement of the technical requirement specification will introduce new concepts and new derived requirements. Functionality is translated into **mathematical functions**, or **functions as services**, which are the first form of **behaviour**.

The characteristic feature is the appearance of **components** (abstract machines) and **connectors** that the functionality and communication of system components is mapped onto. This first partition of the SUD results in a system **architecture** decision.

The properties are refined into static (logic) properties, which constitute the **invariant** of the SUD, dynamic properties, which will be integrated into the behavioural description through the refinement process, and **constraints,** which create new boundaries of the design space. Dependencies appear among refined variables through functions. The model of computation is based on causality. Timing requirements may apply as constraints.

4.2.4 Layer L3 abstraction

At the layer L3 components are refined into **modules**. The communication links are refined from connectors to **channels**. The model becomes **structured** as a **hierarchy**. The upper level of the hierarchy is a bipartite graph with two kinds of nodes: modules and channels. A module is only connected through interfaces to channels, and vice-versa.

Variables, functions and operations can be refined into new **abstract data types**. Constraints are derived to apply to new variables and data types. Logical (static) properties are refined and proven to maintain the invariant of the SUD.

4.2.5 Layer L4 abstraction

At the layer L4 and downwards from it, there are an arbitrary number of levels of refinement. In the case of channels, while applying the VSIA standard process we meet the notions of **protocol, shared variable**, **buffer** and **queue**. **Entities** encapsulate modules, and may be organised into **packages** as appropriate in the domain.

As far as behaviour is concerned, operations can be structured as **data-flow graphs** and **algorithms**. The overall model control is of the kind of a **control graph**. The control can further be refined into a hierarchy of **FSM**s.

It can also take the form of a network of **processes**. The notions of **loop** and **sequencing** may appear related to algorithms. The model of control can be refined into **threads**, and a mix of asynchronous FSMs and instruction cycle.

4.2.6 Architecture

The architecture includes the structuring concepts. These are visualised in Figure 4-12 using UML-like notation.

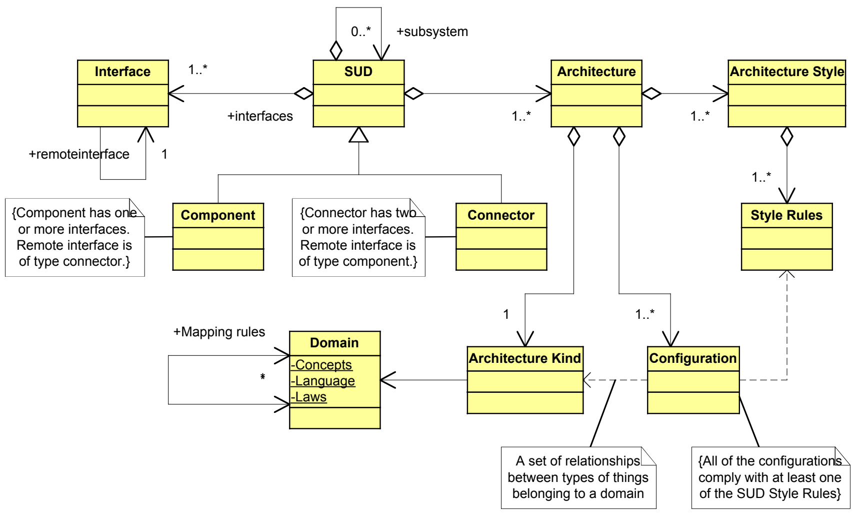

0..*
+subsystem
Interface
1..*
SUD
Architecture
Architecture Style
+interfaces
1..*
1..*
+remoteinterface
1
1..*
{Component has one or more interfaces. Remote interface is of type connector.}
Component
{Connector has two or more interfaces. Remote interface is of type component.}
Connector
Style Rules
1
1..*
+Mapping rules
Domain
-Concepts
-Language
-Laws
*
Architecture Kind
Configuration
A set of relationships between types of things belonging to a domain
{All of the configurations comply with at least one of the SUD Style Rules}

Figure 4-12. Architecture.

The concept of System Under Design (SUD) is related to **architecture** by virtue of the fact that every designed system has an architecture whether expressed explicitly or not. As soon as one conceives of a system one starts to think about its architecture. The SUD concept implies the existence of the concept of architecture in an explicit form.

The SUD has one or more **interfaces** to its environment. The interfaces designate the points and types of interaction between the SUD and the environment. The SUD is composed of zero or more **subsystems**, i.e. it is hierarchical. A system can be a subsystem of another system, and a subsystem can itself be a system.

There are different **kinds of architecture** each of which exist in a **domain**. The concept of domain embodies the **concepts, language and laws** used to conceive a given kind of architecture. Examples of architecture kind are software, hardware, structural etc.

According to the ontology, any domain can be described in terms of things and linkages that exist among them. In the SUD architecture schema, **component** is the concept used to represent a thing, and **connector** is the concept used to represent a relationship. As the SUD is a model, both component and connector are conceptual constructs.

The concept of **architectural style** is such that an architecture will follow a style which is embodied in a set of **style rules**. Architecture can be described as a **configuration** of types of conceptual things. A configuration is a set of relationships between types of conceptual things.

The conceptual things in a given architecture will be taken from the set of concepts belonging to the domain for the given architecture kind. The architecture will be described using the language of the domain. The things and configuration will also obey the laws of the domain.

4.2.7 Behaviour

The behaviour includes the concepts for representing functionality. Some of them are visualised in Figure 4-13 using UML-like notation. See Annex A2 for more detailed and formal definitions of the related concepts.

The concept of System Under Design (SUD) is related to **behaviour** by virtue of the fact that every technical system interacts with its environment through its interfaces. Behaviour represents the evolution (operation) of the system and its responses to external stimuli.

The behaviour is characterised by its collection of **events** and by its **state**. The behaviour of a system can also be defined as the **set of its operations**. Still another way is to define it as the set of threads grouped in a **behaviour process**.

An event is defined as the occurrence of the change of the value of a variable. Sometimes events are represented as sets of pairs {**tag**, **value**}. Then an event can be associated to more than one value or tag, respectively.

A behaviour **interface** is an abstraction of the behaviour that consists of a subset of the **external events**. Each external event of the behaviour belongs to a unique behaviour interface. Every external event is an instance of communication. **Internal events** occur inside the behaviour and have impacts on the internal evolution of the system, but do not imply communication outside.

A **maplet** is an ordered couple of 2 different variables belonging to the specification of the system. It denotes the influence exercised by the first variable on the other variable. A **causal chain** is a set of maplets connecting a **totally ordered** subset of variables. In a causal chain all variables but 2 appear exactly twice, once as fist element and once as second element of a maplet.

An **operation** is defined as the association of a set of tags to the **variables** involved in a **function**. A set of totally ordered operations linked by at least one causal chain forms a **sequence**. A **thread** is a set of sequences with common events. **State** is the set of all values of variables after a **transition** has occurred, i.e. a transition is associated to two states, one before and one after the transition.

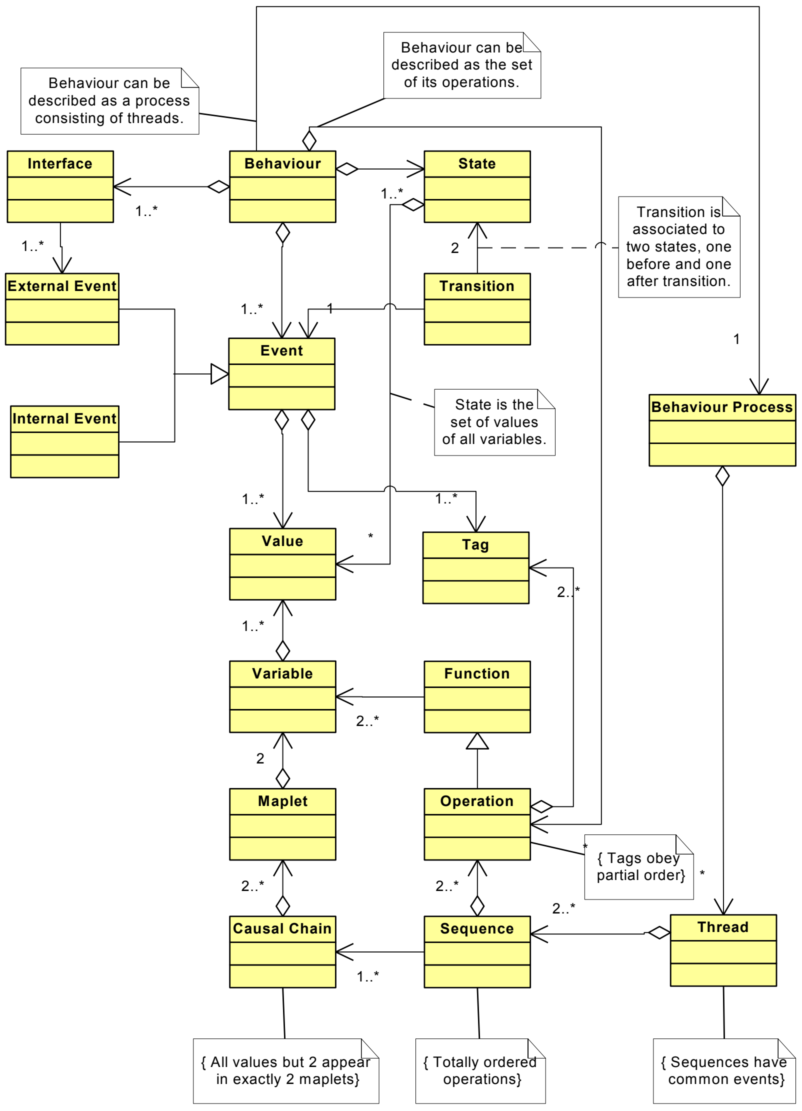

Behaviour can be described as a process consisting of threads.
Behaviour can be described as the set of its operations.
Interface
Behaviour
State
Transition is associated to two states, one before and one after transition.
External Event
Transition
Event
Internal Event
State is the set of values of all variables.
Behaviour Process
Value
Tag
Variable
Function
Maplet
Operation
{ Tags obey partial order}
Causal Chain
Sequence
Thread
{ All values but 2 appear in exactly 2 maplets}
{ Totally ordered operations}
{ Sequences have common events}
1..*
2
1
2..*
*

Figure 4-13. Behaviour.

4.2.8 Domain Mapping

The concept of a design task is something that uses information about the SUD. Tasks that are useful also produce information about the SUD. This information exists in the domains mentioned above.

If a given task will use information from more than one domain, it will need to relate concepts from different domains. The architecture conceptual model therefore includes a relationship between domains called mapping rules. These provide ways to map between concepts in different domains.

For example a software concept is a function (as in C language) that is implemented as a sequence of instructions coded as instruction words. The instructions execute on a processor and code resides in a read-only memory. These last two sentences are domain mapping rules. As they relate the software concepts of instructions and code to the hardware concepts of processor and read-only memory.

5. LINKAGE OF SUDM AND SDP

The SDP cannot exist in isolation from the SUDM. The SDP creates and uses the SUDM, which is the principal purpose of the SUDM. Other purposes are e.g. documentation and archive.

From the viewpoint of the SUD, the SUDM consists of a set of overlapping models of the SUD. Each model covers a subset of the concepts of concern to the stakeholders, and we can identify sets of types of model, for example a performance model. We have a list of concepts that are of possible concern to the stakeholders.

5.1 Users/Usage of Models

The link between the SUDM and the SDP is the set of usage (uses) by designers. A model may have more than one use, for example model checking and synthesis. The concept of usage/use requires the concept of a user. This is a concept that lives in the SDP. Let us call this a role which is played by at least one actor. A usage/use of the SUDM defines a view of the model. A view is a mapping between a model set in the SUDM and a role in the SDP. This is visualised in Figure 4-14, where the SDP can be seen as consisting of a set of activities *(*shown as boxes, marked as Ax) connected by dependencies (shown as arrows).

Activities are carried out by actors playing roles. The actors are not shown to simplify the diagram. The endpoint of a dependency is the role that the actor(s) of that activity play in the dependency.

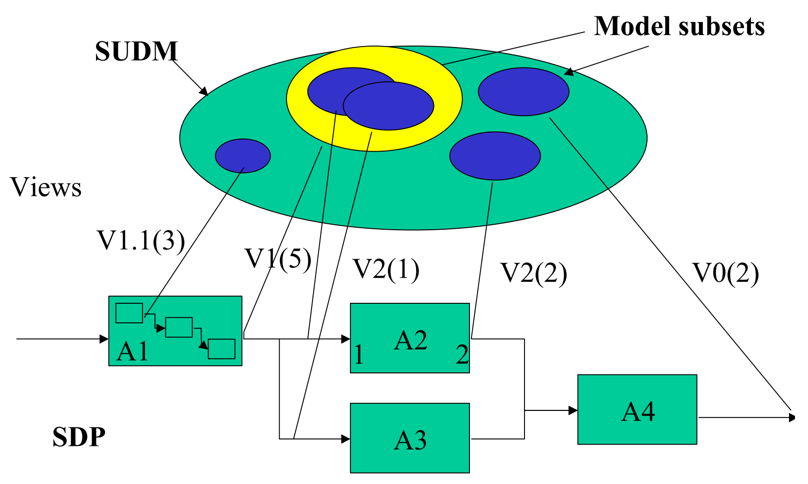


Figure 4-14. Views between SDP and model subsets of SUDM.

5.2 Views

The views in Figure 4-14 are labelled to identify the role in the SDP that uses it. The view is related to an activity. An activity uses and produces subsets of the SUDM. A view is defined by the subset and the use it is put to. To illustrate this the views are labelled according to the activity number and the dependency endpoint (role) in brackets ().

A branch in a dependency arrow indicates that a subset of the original SUDM model set is used by the role at the far end. A joining of arrows shows that the using role needs the union of the model sets represented by the incoming arrows.

5.3 Uses/Purpose/Kind of Model Subset

The kinds of use of model subsets is as follows:

- Description (descriptive model) consisting of structure, configuration and relations.
- Prediction used for predicting something about the SUD.
- Prescription used to make, use and install usually presented as a list of instructions to a machine or human.

6. SUMMARY

The SDCM is a meta-model that differentiates the models of system design process (SDP) and system under design (SUD), which however are related in practice. The SDP creates and uses the SUDM. The SDPM and the SUDM as meta-models define the concepts and their relationships that are needed in the creation of corresponding instances of models for specific purposes. The models are generic enough so that they can be applied in various organisations and for various kinds of systems.

The system design process model (SDPM) is an adaptation of the Software Process Engineering Metamodel (SPEM) that is in the process of the OMG to become a standard. The SDPM consists of four views: core, process definition element, work definition and process composition.

The system under design model (SUDM) presents concepts and their relationships for describing the system under design (SUD). The concepts are structured according to abstraction layers. Architecture and behaviour as main views to the SUD are described and visualised using UML-like diagrams. Concepts for representing behaviour are defined in more detail in Annex A2: Action Semantics.

The user is expected to instantiate both the design process, modelling methods, languages and the specific artefacts according to the needs of her/his organisation. This requires effort from the user, but the payback will come from improved reuse capability of the organisation.

REFERENCES

[1] G. Muller (2002). System Architecting. Philips Research. Retrieved January 2003 from: http://www.extra.research.philips.com/natlab/sysarch/SystemArchitecting.html

[2] Wand, Y., Storey, V. C., Weber, R., An Ontological Analysis of the Relationship Construct in Conceptual Modeling. ACM Transactions on Database Systems. 1999; 24:494-528.

[3] Unified Modeling Language Specification Version 1.3. Object Management Group Inc., 1999.

[4] Fowler, M., Scott, K., UML Distilled: A Brief Guide to the Standard Object Modeling Language, 2nd Edition. Addison Wesley, 1999.

[5] Bezevin, J., Who's Afraid of Ontologies, OOPSLA98 - CDIF Workshop. 1998.

[6] Vanwelkenhuysen, J., Mizoguchi, R., Ontologies and Guidelines for Modeling Digital Systems, Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Canada, March 1995.

[7] Software Process Engineering Metamodel Specification (SPEM), Object Management Group Inc., 2001.

Chapter 5

CONCEPTS FOR SYSTEM SPECIFICATION AND DESIGN LANGUAGES

Patrizia Cavalloro
Italtel SpA, Milan, Italy

Abstract: This chapter concerns the definition of System Specification and Design Languages characteristics in order to allow the description of a System under Design. First, a general introduction to languages is presented. Then, a classification of languages is proposed. Finally, concepts related to languages are identified and classified.

Key words: System-Level Design, System Specification and Design Languages (SSDL), SSDL classification, SSDL concepts.

1. INTRODUCTION TO SYSTEM SPECIFICATION AND DESIGN LANGUAGES

1.1 General

The System Design Conceptual Model (SDCM) described in Chapter 4 addresses the early phases of system development, but no technology-specific design. It considers as "meta-models" the System Under Design (SUD) and the System Design Process (SDP).

A System Specification and Design Language is a language to describe a SUD at required levels of abstraction providing required views to the SUD in order to allow actors to perform transformation, validation and analysis tasks that are specific to the level of abstraction and to the design process applied. Specifically, the System Specification and Design Language should allow the description of system in terms of external and internal views to the modelling domains of structure, connectivity and behaviour.

The choice of the appropriate System Specification and Design Language (SSDL) for the design of a system is based on a set of criteria, such as the expressiveness of the language, the automation implied by the computational model it is based on, as well as the tool support and the associated development methodologies. Even though there are various SSDLs, the computational models upon they are based are either *data oriented* or *control oriented*. In both cases they can be synchronous or asynchronous.

1.1.1 Basic Terms for System Description

Apart from the conventional programming aspects, system description with the use of SSDLs is based on four basic terms [1]:

Parallelism and concurrency: although both refer to the distribution of operations among resources, concurrency is a way of implementing parallelism and can be achieved by interleaving or simultaneously executing two or more threads. Both parallelism and concurrency can be at the bit level (i.e. n-bits adder), operational level (i.e. multiple operational units), procedure level (i.e. multi-process specification) or processor level (i.e. distributed systems).

As far as parallelism is concerned, it can be expressed using either control flow or data flow. In the first case we deal with models where, during the design of the system, the execution sequence of the system parts is determined. CSPs (Communicating Sequential Processes) and FSMs (Finite State Machines) are classical approaches of control oriented parallelism. In the second case of data flow the command execution flow is determined by the data dependency, which is expressed in Data flow graphs.

Hierarchical development: it allows the hierarchical development of complex systems according to which the designers partition the system functionality into subsystems, which are easier to be designed. There exist two categories of hierarchical development: *the behavioural hierarchy* and the *structural hierarchy*.

Communication: this allows the subsystems to exchange data and control information. There are two basic models: message passing and shared memory.

Synchronisation: it defines the primitives governing the communication among the various subsystems. There exist two techniques: message queues and rendezvous.

1.1.2 A Taxonomy of SSDLs Based on the Computational Model

The expressiveness of an SSDL is drawn from the computation model it uses. The differences between the SSDLs concern the ways they allow for the design of the particular sub-systems, the interconnection techniques, the communication between them as well how the various subsystems compose the final system. According to D.Gajski [1] there are five categories of SSDLs: *a) State oriented b) Activity oriented c) Structure oriented d) Data-oriented, and e) Heterogeneous.* SSDLs, which belong to categories a) and b) allow system description through state machines and transformations, Systems which are described by SSDLs which belong to category c) give emphasis to the system structure while those in d) give emphasis to system description which process information.

An objective taxonomy of SSDLs would be based on the computation model, which each SSDL supports. The system description used by every language reflects the syntax structure of the language and not the computation model used by the language. The computation model is related to the theoretical background on which the execution model of the language is based. The computation model can be considered as an orthogonal combination of the *communication model* and the *control model*. An SSDL can support one of the following communication models*: synchronous (or single threaded)* and *distributed* where the communication model between threads is well defined. The control models, which an SSDL supports, could be either *control flow* oriented or *data flow* oriented.

Most co-design tools are also using internal language representations, which ease the model refinement. Usually they are taking input expressed in an SSDL (SDL (Specification and Description Language), C, VHDL, JAVA etc.). There are two categories of intermediate representations one *language oriented* and one *architectural oriented.* Both can be used for system representation and in the necessary transformations during system refinement. The representations oriented to languages are based on graphs use (Data flow graphs - DFG or Control Flow Graphs - CFG) while the representations oriented to architecture are based on FSMs. The representations oriented to architecture refer more to the system architecture and not to the initial system description. Those are FSM for Data (FSMD) or FSM with Coprocessors (FSMC). Co-design tools of the latter category, found in bibliography, include COSYMA, VULCAN, and LYCOS, which use FSMCs for system description.

1.2 Classification of Languages from Literature

SSDLs are originally drawn from software engineering where the ever-increasing development and maintenance cost for software led people to put emphasis at the *specification and requirement analysis*. This is followed now in the system design in order to handle the ever-increasing system functionality and complexity. Result of this is the higher quality of the final system development through the "gradual refinement" and verification at the early stages of the design flow.

Existing languages have been classified in several different ways, depending on language characteristics. One possible classification is the following:

Architecture Description Languages: Architecture Description Languages (ADLs) are formal languages for representing the architecture of a system [2]. Architecture means the components that comprise a system, the behavioural specifications for those components, and the patterns and mechanisms for interactions among them. An ADL must explicitly model components, connectors, and configurations. PMS (Processor, Memory, Switch from Bell & Newell) can be considered as the first architecture description language. More recently ACME, Aesop, C2, Darwin, MetaH, Rapide, SADL, UniCon, Weaves, and Wright, are some of the languages addressing Architecture specification.

Hardware Description languages: Software engineering specification languages were followed by the hardware specification languages such as CASSANDRE, DDL and CONLAN of the '60s and '70s up to the most updated ones such as HardwareC, SpecCharts, SpecC and the Hardware Description Languages, such as VHDL and Verilog. Languages focusing in more specialised tools (as it is DSP) also appeared, like SPW and COSSAP.

Protocol Design languages: In the telecom domain many languages have appeared which are specialised for the requirements capturing and description of telecom protocols. Those are based in the so-called *formal system description* and include the FDT (Formal Description Technique), the LOTOS (Logical Temporal Ordering Specification- an OSI standard), SDL and ESTELLE. LOTOS consists of two parts: in the description of the *behavioural pa*rt of the system based in *process algebra* and the data description based on *abstract data typ*es. Specification and Description Language (SDL - ITU standard Z.100) uses Finite State Machines (FSMs) for system and operations description. System description includes apart the dynamic behaviour of the individual parts, the structure definition of the whole system, as well as, the communication between the various parts. SDL supports both graphical and textual representation. ESTELLE is an ISO (International Standardization Organization) standard with procedural features like PASCAL, and it is more a programming language than a specification one.

Reactive Design Languages: A big category of SSDLs called reactive is the one, which describes systems interacting with their environment in real time. It includes ESTEREL, LUSTRE, SIGNAL, Statecharts and even Petri Nets. ESTEREL supports parallelism and has the ability to describe systems in a formal way from the initial design stages up to the final implementation.

Its basic concept is the synchronous "event". LUSTRE is a development and programming language for automata. SIGNAL is differentiated from LUSTRE in the fact that it uses multiple clocks in the same program, which can be combined through temporal operators. Statecharts is a visual formalism for the description of reactive systems, based on the concept of *state*. This formalism extends the traditional FSMs by adding hierarchy, parallelism and communication. Communication is based on broadcasting and the execution model is synchronous. Petri Nets allow the representation of systems, which support discrete events.

Programming Languages: The majority of the programming languages have been used for system description and design. C, C++ and Java are included. Their basic drawback is the lack of ability for the description of time and parallelism. Extensions of the above-mentioned languages in that direction include HardwareC, SystemC and SpecC. SystemC is oriented in using C++ in all stages of the development of a system. The fundamental building blocks in a SystemC description are processes. A process is similar to a C or C++ function that implements behaviour. A complete system description consists of multiple concurrent processes. Due to the extension of C++ for parallel processing and hardware description, SystemC is not usually explicitly included in one category.

Parallel Programming Languages: They have been used for hardware description due to the same requirements imposed by hardware and parallel programming. Their major problem is the lack of timing concept. Between them, languages such as OCCAM and Unity have been used for system specification and design.

Languages for Functional Programming: Languages based on functional programming and algebraic notations have been used for system hardware design. Paradigms of such languages include Haskell, VDM, Z and B. B is a language, which is formal and models globally the system and its environment. It supports proven system refinement down to implementation as a mix of hardware and software.

Structural Languages: They have been drawn from the software development domain. They are systematic languages based on the fragmentation of the system in smaller subsystems easier to develop and manage. Most of them are object oriented. Between them we mention the HOOD, OMT and lately UML (Unified Modeling Language). The last one has emerged as a system analysis and development language. Their disadvantage is the fact that they are based on non-executable models so they cannot be used for simulation and synthesis.

Languages for Continuous Systems: They are based on the usage of differential equations in order to describe continuous systems. Most popular ones are: VHDL-AMS, Matlab, MatrixX, and Mathematica. They have great expressional power and employ floating point calculations, which makes problematic their usage for synthesis.

1.3 Requirements for System Specification and Design Languages

Experience shows that there is not just a globally accepted language for system design. The choice is based usually on the application and the designer's experience. The system design requirements impose complex criteria leading thus to new language concepts, which include multi-formalism and feature combination taken from various existing languages. Therefore, the definition of an effective Specification and Design Language should be based on the ultimate criteria of a language. Those are the expressiveness, the analytical power (ability for simulation and verification at various design stages) and the usage cost (something related to vendors, standards, and simplicity of the models developed).

The SYDIC-Telecom project has identified a set of terms relevant for System Design aspects, providing a definition for all of them, in order to overcome wording misunderstanding. The terms and their definitions are collected in the SYDIC-Telecom Glossary, described in Annex A1.

For the term "Specification and Design Language" the following definition has been provided:

"A System Specification and Design Language (SSDL) is a language to describe a system under design (SUD) at required levels of abstraction providing required views to the SUD in order to allow actors to perform transformation, validation and analysis tasks that are specific to the level of abstraction and to the design process applied. Specifically, the SSDL should allow the description of system in terms of external and internal views to the modelling domains of structure, connectivity and behaviour".

When considering languages that should be able to support design at system level, some more needs have to be satisfied:

- *A well-defined set of concepts:* terms related to system design should be identified and defined, and a system design language should be able to support them
- *Unambiguous, clear, precise, and concise specifications:* ambiguous, unclear and imprecise specifications do not allow to describe a system in a unique way, and this lead to the impossibility to proceed in the design process *A basis for determining the consistency of specifications:* the same comment as in the previous point is valid here: inconsistent specifications don't allow to proceed in the design process

- *A thorough and accurate basis for analysing specifications:* the possibility of analysing specification allows a better understanding of the system and a better quality of the design process
- *A basis for determining whether or not an implementation conforms to the specifications:* verification must be possible at all design steps
- *Computer support for generating applications:* a language without a computer support cannot be used in real-life applications.

The activities performed in the SYDIC-Telecom project concentrated on the first of the above-mentioned points, that is the identification, definition and classification of System Level Design concepts applied to System Specification and Design Languages. Next chapter will describe the proposed classification.

2. CLASSES OF LANGUAGES

The basic idea of the proposed work consists in the identification of concepts which are relevant at system level in relation with the SUD and that any System Specification and Design Language must be able to express.

The approach proposed by the SYDIC-Telecom project consists in the classification of concepts in three classes that have been related to languages able to specify systems from different views. It should be noted that the concept "classes of languages" does not in this context necessarily mean separate sets of languages for every class, but rather the needs of various stakeholders in system design, and how their concerns should be supported. An optimum would be one simple and understandable language that could provide all the support. Unfortunately, this seems not possible, and our current baseline assumption is accordingly that several languages would be needed.

The three identified classes of languages are defined as follows:

System Specification and Modelling Languages: It describes the functionality and properties of the System under Design all along the design process.

Architecture Language: It describes the different structures of the facets of the system and relations between them. It should allow architects to use necessary operations to change the design to establish a good mapping between the different facets. It should allow this work to be controlled by a set of architectural rules.

Design Command Languages: It allows exercising the model in the frame of different applications; it verifies fulfilling of rules and handle of criteria.

Figure 5-1 shows the context of the System Design Process (SDP) in the chain from system idea to implemented system.

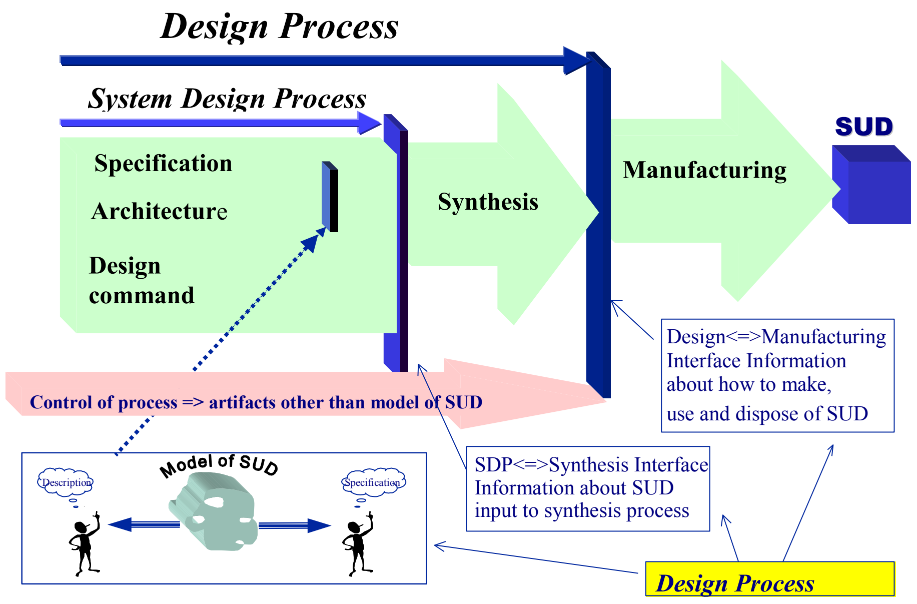

Design Process
System Design Process
Specification
Architecture
Design command
Synthesis
Manufacturing
SUD
Control of process => artifacts other than model of SUD
Design<=>Manufacturing Interface Information about how to make, use and dispose of SUD
SDP<=>Synthesis Interface Information about SUD input to synthesis process
Model of SUD
Description
Specification
Design Process

Figure 5-1. Context of System Design Process (SDP).

It is the first step in the overall Design Process and precedes the Synthesis step. The synthesis or manufacturing phases are not considered in this context. So we will mainly consider the classes of languages listed above. Figure 5-1 also shows a relationship between the System Under Design (SUD), the design process and model(s) of SUD. The box under the SDP arrow shows a view of the process in terms of Design Process Interfaces (DPIs). Language classes that have been identified are listed in the leftmost box: Specification, Architecture, Property/Constraint and Validation/Evaluation/Verification. There are two major DPIs that help us define the context of the SDP: Design<=>Manufacturing Interface and SDP<=>Synthesis Interface.
The figure is biased to data flow in the SDP so the "control process arrow" has been included to show the control aspect. Artifacts such as test reports are produced, and they are not part of the Model of the SUD. These are part of the control flow of the process.
The SDP consists of a set of sub-processes connected by internal DPIs. A generic picture is shown as an icon in the dotted box in Figure 5-1 and in full in Figure 5-2. The interface is defined as two viewpoints [3] of the Model of the SUD, here shown as a 3D structure with holes.

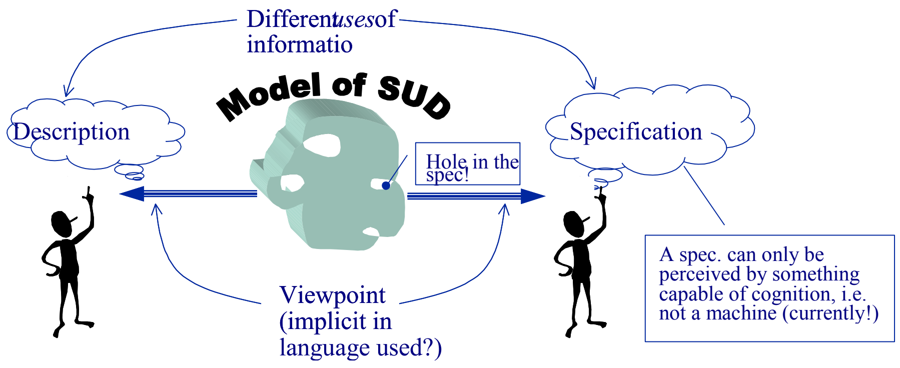

Different *uses* of informatio
Model of SUD
Description
Specification
Hole in the spec!
A spec. can only be perceived by something capable of cognition, i.e. not a machine (currently!)
Viewpoint (implicit in language used?)

Figure 5-2. Different views to the same System under Design (SuD) model.

In general the supplier to the interface has a viewpoint that characterizes the model as a description whereas the receiver has a viewpoint, which sees the model as a specification. It is to be highlighted that only a human is capable of adding something new to a design so only a human can perceive the model as a specification.

What we see from this simple picture is that both sides of the interface have different uses of the model. The implications on any SSDL are wide reaching as it implies the need for support for viewpoints in SSDLs.

3. CONCEPTS TO BE SUPPORTED

One need that must be satisfied when considering languages that should be able to support design at system level is that terms related to system design should be identified and defined and a system design language should be able to support them.

In the proposed approach, approximately one hundred and twenty specific language concepts have been identified. These terms have been defined precisely, so to prevent misunderstanding in meanings. A System Specification and Design Language must be able to express those concepts.

Once identified the final list of concepts, each concept has been assigned to at least one class of language. Some concepts, for instance *interface*, appear in more than one class.

3.1 Architecture Language

In literature, Architecture Description Languages (ADLs) are formal languages for representing the architecture of a system. Architecture means the components that comprise a system, the behavioural specifications for those components, and the patterns and mechanisms for interactions among them. A formal architecture representation is done with an ADL. An ADL must explicitly model:

- Components
- Connectors
- Configurations

Furthermore, to be truly usable and useful, it must provide tool support for architecture-based development. Having realised the importance of an architecture language, the dedicated class Architecture language has been identified and further subdivided in five sub-classes. Table 5-1 gives the list of concepts belonging to each sub-group of the class Architecture Language.

Table 5-1. Architecture Language concepts in sub-groups.

Architect discipline	*Architect primitive operations*
Design space	Compose
Specification of Requirements	Encapsulate
Refinement	Decompose
Architecture Rule	Bind
Architecture Pattern	Projection
Constraints	Instantiate
Non-functional Property	Connect
	Viewpoint
Primitive architecture elements	*Complex architecture elements*
Component	Configuration
Connector	View
Interface	Facet
Function (service)	
Modelling capabilities	
Compositionality	Heterogeneity
Scalability	Evolvability
Dynamic restructuring	

The role of each sub-group is briefly outlined in the sequel:

- *Architect discipline*: it groups concepts related to the activities of the design architect.
- *Architect primitive operation*: it indicates possible operation on SuD.
- *Primitive architecture elements*: it groups basic architecture (description) concepts.

- *Complex architecture elements*: it groups more complex architecture (description) concepts.
- *Modelling capabilities*: it groups properties of the architecture model.

3.2 System Specification and Modelling Languages

Six sub-classes have been identified in this class. For each sub-group of the class System Specification and Modelling Languages the list of concepts belonging to it is given in Table 5-2.

Table 5-2. System Specification and Modelling Language concepts in sub-groups.

Scope	*Communication-related concepts*
Requirement (user, domain, e.g. technology)	Channel
Use case	Interface
Specification	Protocol
Functionality	Synchronous
	Asynchronous
	Queue
	Buffer
	Message
	Shared variable
Basic constructs	*Order- and Time-related concepts*
Abstract Data Type	Causality
Abstract Machine	Finite State Machine
User Defined Data Type	Control Graph
Generics	Event
Parameter	Concurrency
Assertion	Sequence
Predicate/formal property	Parallelism
Invariant	Data Flow
Module	Algorithm
Object	Clock
Component/entity	Process
Operation/service	Cycle-based
	Instruction
	Thread
	Continuous
Modelling capabilities	*Qualifiers*
Reasoning about the design	Semantics (formal, operational, informal)
Abstraction process	Applications (simulation) independent
Refinement process	Automatically translatable to application formalism (performance evaluation, synthesis, model checking)
Decomposition/partitioning	Declarative/imperative
Composition process	
Indeterminism	
Incompleteness	
Encapsulation	
Hierarchy	
Inheritance	

The role of each sub-group is briefly outlined in the sequel:

- *Scope*: general concepts related to the class.
- *Basic constructs:* basic concepts for language aspect.
- *Communication-related concepts:* concepts related to communication aspects.
- *Order- and Time-related concepts:* concepts related to order and time.
- *Modelling capabilities:* Languages capabilities on modelling.
- *Qualifiers:* Attributes of the languages.

3.3 Design Command Languages

The idea behind the basic notation presented in this class is that a design process is a co-operation of skilled individuals and/or tools, or teams thereof, in specified roles to perform specified activities on artifacts.

In this class, five sub-classes have been identified. For each sub-group of the class Design Command Languages the list of concepts belonging to it is given in Table 5-3.

Table 5-3. Design Command Languages concepts in sub-groups.

Design elaboration	*Design verification*
Objective function	Check
Pragma	Collecting
Tool interface	Coverage
Assessment of tool result	Response analysis
Design Constraint	Simulation
Implementation generation	Simulation scenario
	Stimuli generation
	Model/equivalence checking
Design validation	*Design management*
Requirement	Design pattern
Evaluation (performance, power, etc.)	Design history
Proof	Traceability
Analysis	Documentation
IP Reuse and retrieve	
IP repository management	
Intelligent access to SIP	
Formal/informal set of reusability properties	
Reuse rules	
Design data generality	

The role of each sub-group is briefly outlined in the sequel:

- *Design elaboration*: concepts related to use of tools which elaborate the design.

- *Design management*: concepts related to the management of designs.
- *Design validation*: concepts related to validation of the design.
- *Verification*: concepts related with verification of the design.
- *IP Reuse and retrieve*: concepts related to reuse and IP (Intellectual Property).

3.4 IP Reuse and Retrieve

One important aspect related to system level languages concerns IP and reuse. It has been noticed that concept of reuse (and related concepts) is not peculiar of a single class of languages, but it is related to all of them.

Therefore the proposed classification of languages and concepts does not identify a specific class for IP aspects, but introduces a subcategory in Design Command Languages class: "IP reuse and retrieve", that collects some concepts that are mainly related to IP management.

It has to be noticed that two different groups of terms for reuse have been identified: those facilitators of the reuse, and those that can be really reused.

Examples of concepts for facilitators of reuse are:

- Refinement, Abstract data type, Abstract Machine, Interface belonging to the System Specification and Modelling language class.
- Interface, Compositionality, Architecture pattern, Process belonging to the Architecture language class.

Examples of concepts really reused are:

- Algorithm, Applications (simulation) independent, Component belonging to the System Specification and Modelling language class.
- Component, Connector belonging to the Architecture language class.

4. SUMMARY

In this chapter, after a general introduction to System Specification and Design Languages, a new approach of classification of System Specification and Design Languages has been proposed. Concepts related to languages have been identified and classified accordingly.

5. REFERENCES

[1] Gajski D., Vahid F., Narayan S. and Gong J. "Specification and Design of Embedded Systems", Prentice Hall, 1994.

[2] Medvidovic, N., Taylor, R.N., A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, Vol. 26, No. 1, January 2000, pp. 70 - 92.

[3] Introducing P1471 Recommended Practice for Architectural Description, IEEE Computer Society Architecture Working Group, 30 March 1999.

Chapter 6

SYSTEM PERFORMANCE ANALYSIS

Christophe Gendarme,
Jos van Sas
Alcatel Bell, Antwerp, Belgium

Abstract: This chapter gives a general overview of the performance analysis techniques, used during architecture exploration on different levels of abstraction, of a networking environment. The basic techniques for constructing, validating and verifying models are discussed. A set of generic guidelines and warnings for interpretation of simulation results and for architecture exploration is given. We state that performance analysis can significantly support architecture validation and exploration for complex systems through learning about the system, detect unforeseen bottlenecks or shortcomings early in design flow, quantitative assessment of impact of design decisions, algorithm exploration, tuning functional algorithm to practical design, aid in determining optimum dimensions, settings (thresholds, ...) and sensitivity. We will focus on a methodology enabling the assessment of system level modelling from a performance modelling in the context of system-level IP Reuse.

Key words: System design, Performance Analysis, Property, Abstraction, Switch Core, Flow Control.

1. INTRODUCTION

This chapter gives a general overview of the performance analysis techniques, used during architecture exploration on different levels of abstraction, of a networking environment. The basic techniques for constructing, validating and verifying models are discussed. A set of generic guidelines and warnings for interpretation of simulation results and for architecture exploration is given.

System design encompasses the definition of a functional architecture (behaviour) and the selection of appropriate resources (logical architecture) given a number of specified requirements and constraints, as depicted in Figure 6-1. Often, the constraints impose the use of a given functional algorithm and the use of specific hardware resources. Performance analysis provides feedback between the logical and the functional architecture decisions: it answers the question whether a given functional behaviour can be realized with the selected resources in terms of throughput, latency and the required resources. Vice versa, it helps in designing appropriate functional algorithms for achieving the required performance with the selected resources (optimum resource utilization). Hence performance analysis and architecture/algorithm design and exploration are closely related.

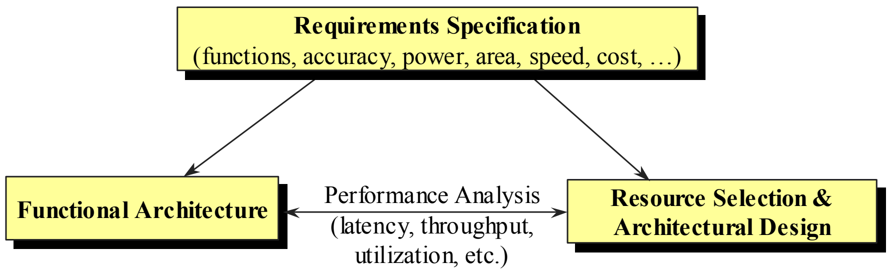

Requirements Specification
(functions, accuracy, power, area, speed, cost, …)
Functional Architecture
Performance Analysis
(latency, throughput, utilization, etc.)
Resource Selection & Architectural Design

Figure 6-1. System design space.

In order to be successful, the methodology and environment used for performance analysis must satisfy at least the following prerequisites:

- High simulation speed: high simulation speed is mainly achieved by making the right abstractions.
- Scalability: the modeling environment used must provide the right modelling constructs and objects to enable the design of scalable models.
- Small effort: the modelling effort is determined by the abstractions made, and the primitives offered by the environment.

In this document we present two modelling frameworks satisfying these requirements. We will illustrate them by a number of examples at different levels of abstraction. The eventual goal is to demonstrate the system-level IP reusability (assessment criteria) from a performance point of view.

2. PROPERTY FORMALIZATION

2.1 Modelling Paradigm

We will use three levels of hierarchy to define a system and its environment:

- At the **network level,** the system and its environment is described as a set of interconnected *nodes*. The nodes are interconnected with *links*, which are mainly characterized by their bandwidth and (transmission) delay.
- At the **node level**, an individual node is described as a set of communicating *processors*, exchanging *messages*. The node interfaces with the environment are specified using standard library blocks (transmitters and receivers).
- Finally, the **processor level** defines a queueing system and the process of serving the queues (see section 2.2 standard template for a detailed description)

The model hierarchy is depicted in Figure 6-2.

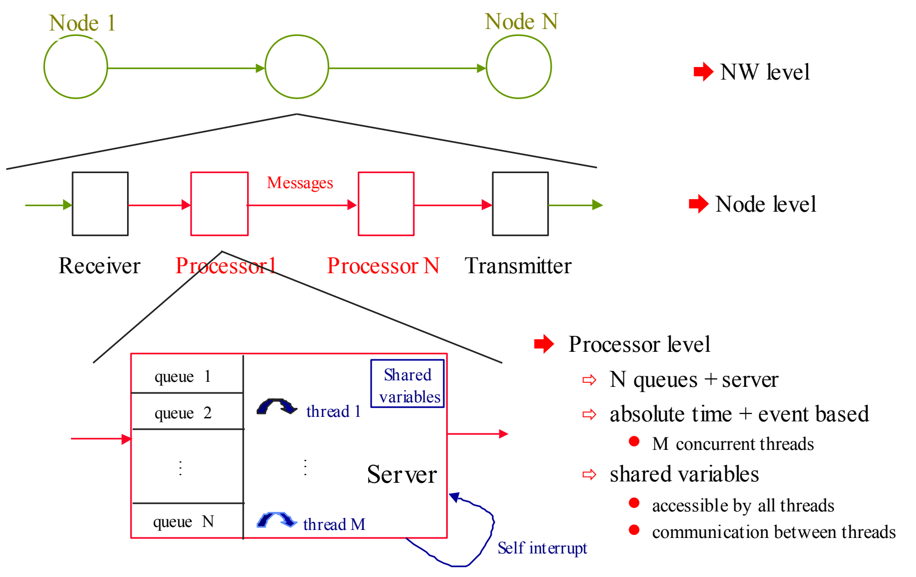


Figure 6-2. Hierarchy of the model.

The simulator is event-based: interrupts are scheduled at an absolute moment in time. Interrupts originate from other processors of the same node (*messages*), or originate from the same processor (*self-interrupt*). In either case, the interrupt causes the execution of behaviour (execution of functions and/or scheduling of new interrupts) but involves no progress of real time. Progress of time is managed by a central event-scheduling mechanism. A processor is specified by means of states and functions. The mechanism of self-interrupts enables the modelling of concurrent threads (see section 2.2). All threads of one processor have access to the shared state variables.

2.2 Standard Template

By means of the standard processor model of Figure 6-3, we will demonstrate how the environment supports modelling parallelism and the construction of scalable models.

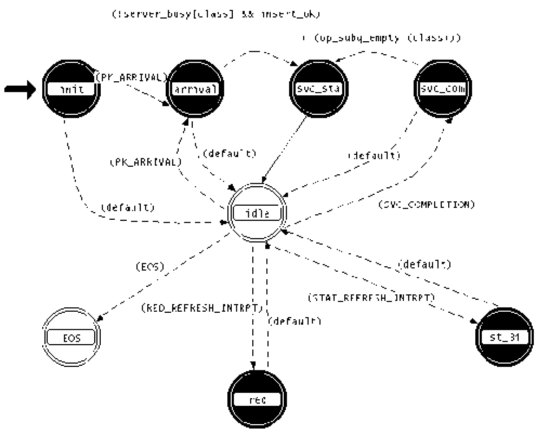


Figure 6-3. Standard processor model.

The simulator uses the concept of stable and temporary states. Between successive events, the system is in a stable state (the idle state or the end of simulation (EOS) state of Figure 6-3 - white circles). The temporary states represent execution of functions (black circles). Upon reception of an interrupt, the system leaves the stable state and enters a temporary state. The temporary state is selected on the basis of the interrupt type.

The model depicted above is the standard template used for modelling a queueing system consisting of an arbitrary number of subqueues, served by an arbitrary number of concurrent servers. The template works as follows: the arrival of an information entity (data packet) causes an interrupt (type packet arrival). The processor goes from the 'idle'-state into the 'arrival'-state. In this state, the packet is conditionally (e.g. buffer acceptance mechanism) stored in the appropriate subqueue (say subqueue L). The simulator offers the required primitives for queue - management (packet insertion, removal etc…). Depending on a condition, the processor will either enter a second temporary state or return to the stable idle state. If the packet was accepted, and if the server of the respective subqueue L is idle, the service of the packet is started (system enters service start state). In this state, the completion of the packet processing is scheduled at a time, depending on the service rate (which is a state variable) of this particular subqueue and the packet length (and/or other packet attributes). From this state the system returns to the idle state. The self-interrupt, scheduled in the 'service-start' state will bring the system in the service completion-state where the packet will be removed from the respective subqueue. The self-interrupt has an argument, identifying the queue it was stored in. The service of the next packet is started if any packets are waiting in the same subqueue. As the thread ID can be passed as an argument with the self-interrupt, an arbitrary number of **concurrent (parallel)** threads can be modelled in a **scalable** way (number of subqueues and concurrent servers are simulation parameters).

3. EVALUATION OF PERFORMANCE MODELLING THROUGH EXAMPLES

We will focus on a methodology enabling the assessment of system level modelling from a performance point of view. As this study is part of an example, it will allow at least a qualitative benchmarking of the methodology, and perform an in depth analysis of the design and decision process for one type of system requirement (performance).

The decision to design a model in a new product allowing IP Reuse, imposes some extra constraints on the other components that make up the system. One aspect covered in this case study is the overall system performance. Given the performance limitation of the existing IP, the overall system performance requirements must be translated to the performance requirements for the components to be designed. The final decision to effectively reuse the existing IP will depend on the feasibility of the entire system in terms of extra resources in the new components required to meet the overall system's performance. A founded IP reuse decision requires a high level optimization for the resource management algorithms for the new components in order to create a thorough understanding of the minimal amount of resources needed to meet the overall performance requirements. Hence it is inevitable to do a minimal design of algorithms and architecture in this stage of the design.

The following process applies:

1. System requirements.
2. Selection of existing IP (= SubSystem A, where SubSystem A + SubSystem B = Entire System under design).
3. Impact of existing IP performance limitations on the performance requirements for the rest of the system (SubSystem B).
4. Resource selection for SubSystem B.
5. Resource management algorithm design for optimal resource utilization.
6. System modeling and performance evaluation of entire system: achievable performance = f(resources).
7. Final assessment: minimum required resources vs. technological limitations and other system requirements (maximum total area, power etc…).

The main difficulties in the system level modeling used for performance analysis are addressed in this chapter:

- System level IP representation: performance evaluation at the system level implies a description of the behavior of the IP at the correct abstraction level. More specifically, a formal or semi-formal description of the interface behavior is required, covering all aspects relevant to the performance characteristics of the IP.
- Abstraction level definition of the system under design model: the efficiency of the methodology resides in the omission of the irrelevant design aspects.

Below follows a brief technical description of the system under design and an explanation of the modeling methodology applied.

The system under design is an Internet Protocol router, terminating N bi-directional OC-48 links (2.488 Gbps). Internet Protocol traffic from N input termination modules (ingress line cards) needs to be routed and switched to N output termination modules (egress line cards). For the switching, an existing proprietary scalable switch will be reused. The system under design is depicted in Figure 6-4. Each of the N ingress line cards may send traffic to each of the N egress line cards. The bandwidth from each line card towards the switch is limited, as well as the bandwidth from the switch towards each egress line card.

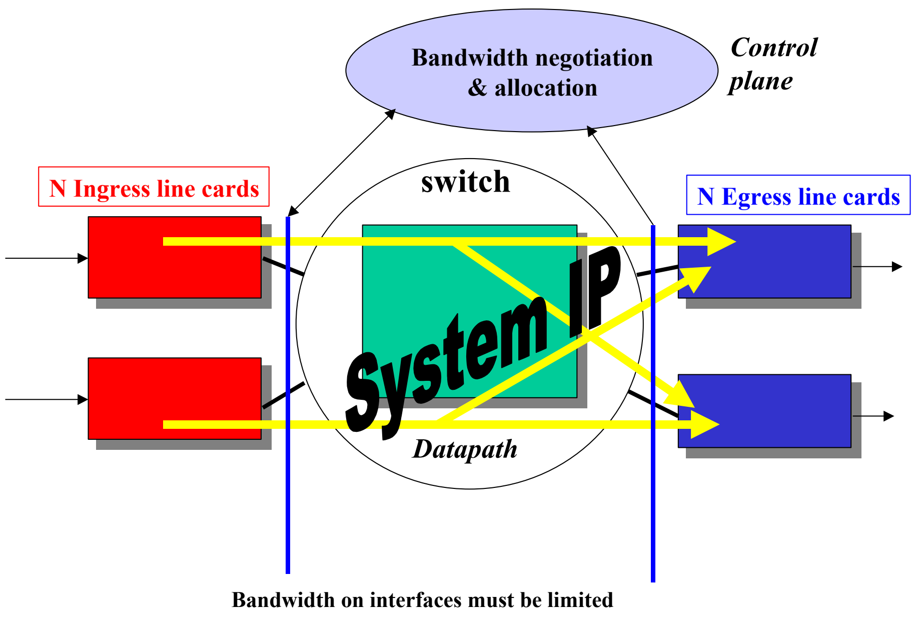


Figure 6-4. Internet Protocol core router overview.

The switch (system A, = system level IP (Intellectual Property)) has the following performance characteristics:

- If the input traffic is constant bit rate (CBR), the output traffic has a known delay vs load distribution.
- The switch can be assumed to be lossless if the total load stays below a given upper bound.

The required system performance is expressed in terms of e.g.:

- Average and minimum latency.
- Minimum throughput.
- Maximum drop probability.

In order to achieve the required performance, a bandwidth negotiation and allocation mechanism must be optimized. This mechanism is required in order to:

- Limit the total input traffic towards the switch (to ensure lossless behavior).
- Limit the traffic between the switch and each individual line card.

The resources available on the line cards (which make up SubSystem B) are:

- Bandwidth towards the switch.
- The available buffer space on each line card.

Because of the limitation of the total load on the switch, the bandwidth of the line cards to and from the switch is resource shared between all line cards. In order to evaluate the maximum achievable performance as a function of available buffer space and bandwidth, the buffer and bandwidth management algorithms need to be incorporated in the model, and the algorithms must be explored and optimized.

3.1 Example 1: Buffer Dimensioning for TCP Traffic

This example illustrates the highest abstraction level considered during network elements system design. This level of abstraction can be considered the intermediate level between the research phase (where network architectures are investigated) and the hardware implementation. From research on TCP congestion control, the advantages of the RED mechanism (Random Early Discard - buffer management algorithm for TCP traffic) are well known. However, it is assumed that the buffer management is done on a per flow basis. A flow is defined as the traffic between a given source and destination. In this example it is defined as all the traffic towards a given output queue (one per output port per service class). The Internet Protocol Core router will handle thousands of flows. Because of the limited hardware resources, it will not be possible to apply RED on a per flow basis (the RED mechanism has to keep state information for each flow). In stead, RED will be applied on an aggregate of flows (RED on the aggregate shared buffer of a linecard). The example below investigates the impact of this reduced RED on the capability of the system to provide QoS (quality of service), and the required buffer space. Effects of the network topology as well as the impact of the hardware resources is considered at this level of abstraction.

Short description of investigated system

The routing elements use the RED buffer management mechanism to control queueing delay and to minimize synchronization of TCP sources due to tail drop [1]. A DiffServ class is an aggregate of TCP-connections, receiving an aggregate guaranteed bandwidth, maximum delay etc. With each DiffServ class corresponds a flow queue in the Internet Protocol Core router. In this example we assume there can be up to 2000 flow queues. Ideally, RED is applied on individual subqueues. However, due to limited resources, RED will be applied on the aggregate queue. With such a system, a number of questions arise:

- If RED is applied on the aggregate queue, is service differentiation still possible and how?
- If so, what are the optimum RED parameters and buffer dimensions with respect to throughput, delay and buffer utilization?

Abstractions made

Very simple models have been used to gain insight in the issues related to RED, TCP-congestion control and service differentiation. An example is given below. Figure 6-5 represents a queueing system of 10 subqueues, each receiving a fixed service rate. RED is applied on the aggregate queue. For each subqueue, an equal number of connections is active. The model has also been used to investigate the behaviour of asymmetric systems, where different number of connections per subqueue and different service rates apply.

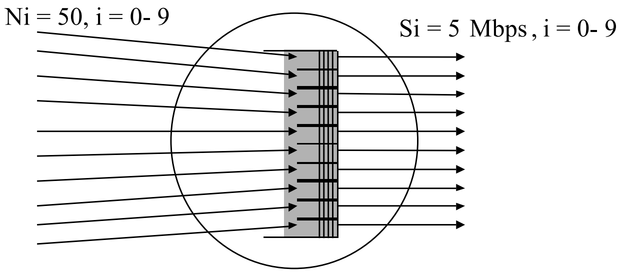


Figure 6-5. Queuing system example for RED.

Scaling of simulation results

Important problems arise here with respect to:

- Simulation duration (i.e. model execution time): as TCP responds on a time-scale of the round trip time (typically 20 to 100 ms), the simulated real time must be a multiple of the round trip time: about 20 seconds real time. Given line speeds of 10 Gbps, a huge number of events (Internet Protocol packets) corresponds to this real time. Simulating the complete system would take ages.

- Number of TCP-connections: the model for the TCP sources can handle up to a maximum number of 500 active connections. The number of active TCP connections in the real system with 2000 flow queues will by far exceed this limitation of the model.

The solution is to find an appropriate way to scale the simulation results and to find appropriate invariants to do so. Our analysis of TCP-behaviour revealed such an invariant: the flow pressure (number of connections divided by service rate of subqueue). The minimum number of subqueues, required for obtaining scalable results is determined by simulations.

The relevant simulation results are shown in Figure 6-6. They show the average queue filling level per subqueue and the standard deviation of the queue filling level per subqueue becomes independent on the number of subqueues (horizontal axis) for more than 10 subqueues.

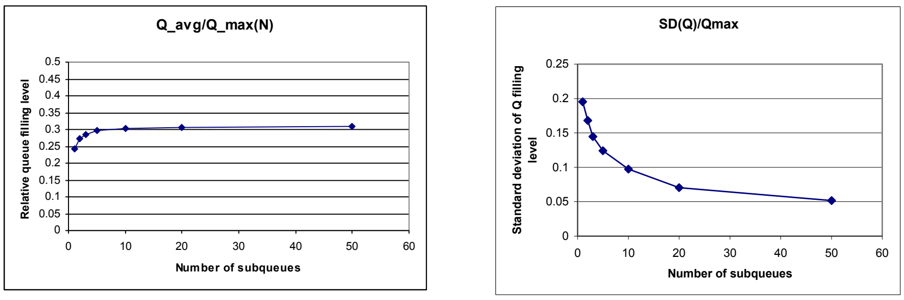


Figure 6-6. RED simulation results (Q = queue filling level, average or maximum).

Scaling of simulation results now works as follows:

1. The system simulated contains N subqueues, where N is the minimum required to account for the effect of statistical multiplexing.
2. The real system consists of S.N subqueues, where S is the scaling factor.
3. The average queue occupation of the real system equals S times the average queue occupation of the simulated system (same for standard deviation), *provided* that the flow pressure (i.e. the invariant) distribution of the subqueues is the same in both the real and the simulated system.

The performance characteristics (such as queueing delay, link utilization, TCP-throughput) of a queueing system with 2000 flow queues and with a total buffer size of 200, will be identical to the results for a queueing system with 10 flow queues and a total buffer size of 1, if for both systems the same flow pressures apply.

Typical outcome

- An estimate on the required total buffer size to provide service differentiation for 2000 flow queues. Figure 6-7 shows the TCP-throughput as a function of the total buffer size (scaled to a system with 10 subqueues). An optimum buffer size can be deduced from Figure 6-7: from a given total buffer size, the gain in throughput is only marginal.
- Optimum RED parameters and sensitivity analysis
- Better insight in impact of TCP-dynamics on this kind of queueing systems (introducing the concept of flow pressure)

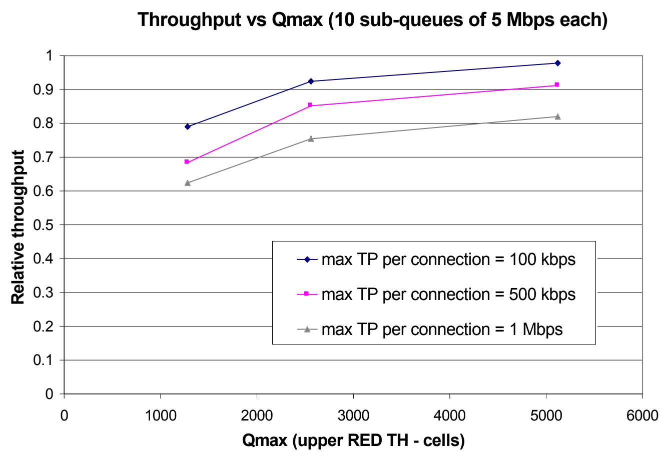


Figure 6-7. Throughput versus Qmax (TP = throughput).

3.2 Example 2: Exploration of Flow Control Mechanisms

At a lower abstraction level, a flow control mechanism at the level of the complete Internet Protocol Core router is investigated. Network aspects such as network topology are not considered here. Also TCP-traffic is not considered here as the relevant timescale for this flow control mechanism is one or two orders of magnitude shorter than the TCP-round trip time. In stead, more implementation aspects are taken into account on this level. The modelling paradigm is inspired on earlier work, described in [2] and is only briefly explained here.

Short description of investigated system

The environment consists of N source blades (linecards), the Switch Core and N target blades (linecards), as depicted in Figure 6-8. With each (source blade, target blade)-pair corresponds a VIEP (Virtual Ingress Egress Pipe). In total there are NxN VIEPS. In each source blade, we have a subqueue for each of the N VIEPS (virtual output queueing). Only the total occupation of the source blade is limited: in principle, one VIEP can occupy the total available buffer space. The Flow Control Mechanism allocates the available bandwidth to the VIEPS according to their need for bandwidth and the available bandwidth.

Purpose of performance analysis

The purpose of the performance analysis was an assessment of the system performance (drop rate, queue filling level), given the limited buffer sizes and bandwidth expansion factors (resources). Modifications and enhancements to Flow Control Mechanism have been investigated, as well as the optimum configuration and sensitivity of the system.

Choice of abstraction level

The traffic was modelled at the level of individual packets. In the model, abstraction was made of the Switch Core. Flow Control Mechanism is assumed to limit the load of the switch core to 0.85 Erlang (Flow Control Mechanism protects the switch). With this load, the Switch Core can be considered loss-less.

Figure 6-8 gives an overview of the modelled system, and the mapping on the node level.

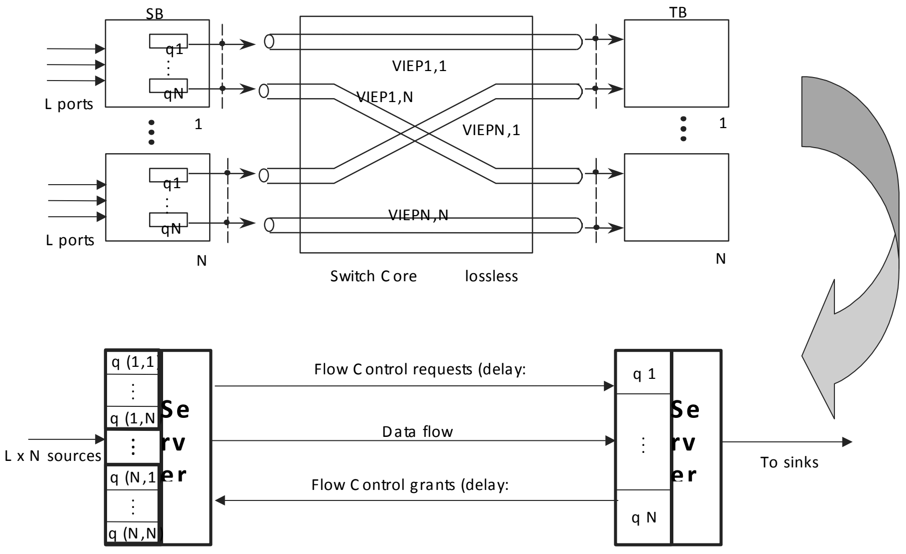


Figure 6-8. Model of the switch core used.

Typical outcome

A number of enhancements to Flow Control Mechanism algorithms has been investigated and introduced:

- Alternative need for bandwidth calculation: the calculation method of the bandwidth needed by a single VIEP affects the buffer filling level and the drop rate. An optimal calculation method (depending on the configuration) resulted. The final need for bandwidth formula takes the delay between need for bandwidth request and application of the new bandwidth into account.
- Another enhancement to Flow Control Mechanism was the introduction of a prioritization mechanism for the bandwidth requests (VIEPs originating from source blades on the verge of overflow have a higher priority). An overflow notification is given if a threshold is exceeded (so-called bypass threshold). An optimum value for the threshold was selected by means of simulations.
- A final enhancement was to limit the total amount of buffer space occupied by a single VIEP in the Traffic Manager. This limitation is called the per-VIEP threshold. The optimum per-VIEP threshold was determined by simulations.

Simulations showed that combining the latter two flow control mechanisms resulted in other values for the optimum thresholds. The simulations were used to determine the sensitivity of the performance characteristics to these settings.

Introducing bias

To quantify the performance of the system under the unfavourable condition where more traffic targets the same destination, intentional bias can be introduced in the (otherwise uniform) distribution of the destination addresses.

The result is presented in Figure 6-9.

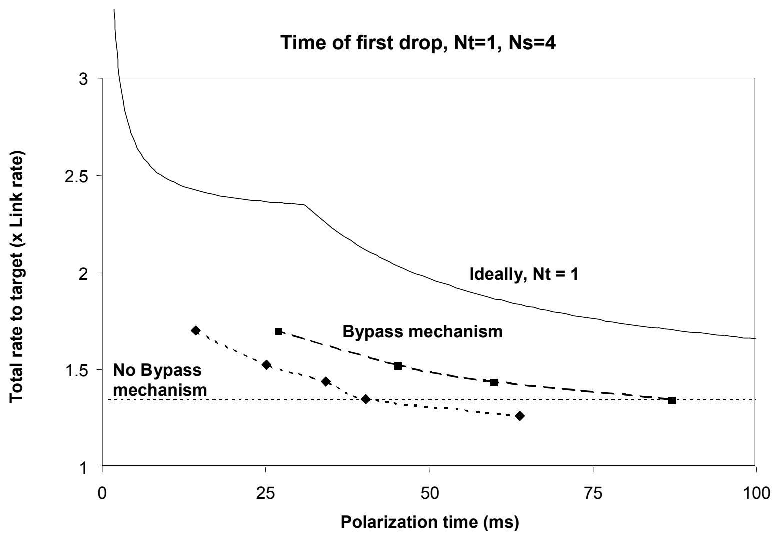


Figure 6-9. Effect of bias introduction.

The ordinate quantifies the polarization (or bias). Under normal conditions, the average rate towards one target blade is the link rate. Under polarized conditions, the distribution of the destination addresses is altered such that the total traffic rate towards a given destination (or a number of destinations) exceeds the average rate. The abscise represents the duration of the bias. The simulation scenario is such that normal traffic is applied during long enough time to let the system reach its average queue occupation ('equilibrium' condition). After this, polarization of the destination addresses holds on for a given time, and afterwards the polarization is switched off again. The simulation is not immediately stopped because even after the switch off, drop can occur. Figure 6-9 shows the time when the first packet is dropped. It shows the effect of the flow control enhancements (No Bypass mechanism, i.e. Flow Control Mechanism without enhancements vs. bypass mechanism active: Flow Control Mechanism combined with optimum bypass threshold) on the ability of the system to sustain bursts of traffic towards the same destination.

Potential problems and lessons learned

- Special attention should be paid to situations where different threads access the same state. In general, the combination of shared variables and event-based simulation can result in inconsistent state information. The methodology does not provide any built in check.
- An alternative, more detailed model could be used (cell-based in stead of packet based) for more accurate results. However, with such a model, the simulation duration would increase with an order of magnitude for the same simulated real time. A packet-based model was used for all simulations. Special care must be taken to avoid possible round-off errors. In some specific cases, such round-off errors may accumulate instead of averaging out.

To avoid problems of either kind, extensive validation is mandatory and is done by tracing as much information as possible, such as queue filling levels, arrival rates, allocated bandwidth and bandwidth utilization.

3.3 Model Validation, Verification and Reliability of Results

3.3.1 Validation

Validation concerns checking whether a model shows the intended behaviour. This is the most difficult and crucial step in building a model, because at this point in time, there exists no formal model to compare with and because it requires a thorough understanding of the system. Below is given a non-exhaustive checklist, which may assist the validation process.

3.3.1.1 Component Level

Component level validation encompasses:

- Visualizing all context variables and stepping through execution of member functions (all branches of the execution tree)

- Generation and check of all corner cases, such as wrap around of counters, overflow of buffers etc…

3.3.1.2 System Level

The complexity of the total system may be too large to allow a detailed validation like for the individual components. Therefore it is easier to *gradually add functionality and validate*.

Below follows a list of guidelines for validating a given system model.

Consistency of context information

Special attention must be paid to the situations where more than one thread accesses the same context variable of an object.

Output sequence of system

For a number of deterministic simulation scenarios the output can be exactly predicted and checked. It is important to do this for a maximum number of different relevant cases. A maximum number of aspects can be verified in this step by an appropriate selection of the token annotation, such as the origin of the token, the token identity/sequence number, the control information used to process the token, etc.

Note: in our model we abstract from real data. A frame is represented by a data structure, containing information such as origin, sequence number, length etc. We make abstraction of the frame content.

Walk through system

A walk through the system traces the path of an individual data-token.

Conservation of data-tokens

Conservation of data-tokens involves accounting the number of tokens passing every interface and the number of tokens stored in queues and discarded.

Symmetry

If a number of identical or equivalent resources are operating in parallel, and the system is well designed, the utilization of every resource should be the same. Any asymmetry should be well understood.

Analytical approach

For most systems, a number of corner cases can be defined for which the performance characteristics (average bandwidth, average queue occupation, average latency) can be analytically calculated, mostly using the assumption of steady state conditions.

Build in specific checks for validation and debugging

Building in some well-chosen model specific checks can facilitate the validation and debugging process to a large extent. If for example a model is built to check whether a (shared) resource constitutes a bottleneck, it is crucial to prevent over-usage of the resource by construction of the model itself, and not to rely on the correct behaviour of the model for the resource allocation process.

Unexpected behaviour

Any counter-intuitive behaviour of the model should be well understood.

3.3.2 Verification

Verification compares an implementation of behaviour to a higher level description/model of the same behaviour. Verification is easier than validation as now we can compare two executable models. In some cases the implementation behaves differently than the higher level model because limited resources are modelled: mostly the implementation can be configured such that it behaves identically e.g. by taking large buffer sizes. If the same seed is used, equivalent models should produce exactly the same output. Note that this will not work if extra random numbers are generated in the implementation, as this modifies the random sequence.

3.3.3 Reliability of simulation results

The simulation results mainly concern the metrics of throughput (drop rate), queue occupation and latency. Simulations provide an average, a maximum and a histogram (frequency of a given value – distribution function). Reliable results show the following characteristics:

- They do not depend on the seed of the random number generator.
- For long enough simulated time, the simulated distribution function should not change with simulated time.

If these conditions are not met, the following may be the cause:

- A bad random number generator is used (in the model we use random number generators when non-deterministic behaviour is represented). If simulations depend on the seed of the random number generator, either a bad random number generator is used, or the simulations are not reliable. We refer to [3] for a discussion on random number generators.
- Simulated time is too short. The simulation time needed to produce stable results depends on the system under investigation. At least, the simulation results should converge as simulated time increases. Also the seed dependence should decrease with simulation time.
- Under certain conditions, simulation results do not converge for reasonable simulation times. An example is a steadily increasing queue occupation (in the initial stage of the flow, mostly semi-infinite queues are used). In most cases, unstable simulation results occur if the system is fully loaded so that any inefficiency cannot be worked away. Another possibility is that one deals with rare events. A very unlikely sequence of events (e.g. all traffic temporarily going to one destination) may cause the system to discard data, even if it is not fully loaded. This may result in non-zero drop for one seed, and zero drop for another seed. A profound analysis is required to determine whether a given instability is inherent to the system. If so, it must be possible to define a simulation experiment that confirms this (decreasing load, increasing resources, introducing more randomization, introducing bias). The performance impact of rare events can be assessed by intentionally introducing bias.
- If the instability can not be explained, there is probably an error in the model, and one should repeat a number of validation checks (e.g. check symmetry of the simulation results to detect unintentionally introduced bias etc…).

The result of the analysis is a reliable assessment of the performance limitations of the system, serving as the input for the architecture validation. Note that an incorrect model can produce stable results.

4. SUMMARY

Architecture validation confronts the simulated performance characteristics to the requirements and hence relies on the validation and verification steps of the previous section.

Abstraction level

The choice of abstraction level determines what can be validated, and is imposed by the characteristics of the system one wants to investigate. Abstraction level is a multi-dimensional concept where the most important axes are the timing precision, the data-abstraction, functional abstraction and structural abstraction. The level of abstraction modelled can be different for different aspects of the system. In the example given, design aspects related to the control flow are modelled with a high level of detail, while other aspects, such as the exact content of the data are abstracted from (token based performance model). The abstraction level is the key factor in the simulation speed.

Pitfalls

A potential pitfall here is that a model shows correct behaviour and produces stable results, but does not represent reality because of the abstractions/simplifications made or the assumed initial conditions. Consider for example a shared resource. With a badly constructed model, a situation may arise where all competitors for the resource never come in conflict (we call this effect synchronization). In a performance model, abstraction is made of parts of the system. Part of the system may be represented by e.g. a simple FIFO. With such abstractions one must take care that the aspects, relevant for the performance, are not abstracted out. In the example given, the fact that the latency of frames is not fixed but randomly distributed, is crucial for the performance and hence the abstract representation of the behaviour must also mimic this random latency. Otherwise, performance might be over-estimated. Unintended synchronization effects can be avoided by introducing enough randomization in the model.

Design decisions

If, on the basis of the simulation results, it is concluded that the requirements are not met and the architecture needs to be changed, we recommend to find an analytical approximation for the simulation results on which the change will be based, in order to maximize the confidence in the model. Also, the alternative architecture should be evaluated for all possible configurations and simulation scenarios (regression test).

Table 6-1 briefly summarizes the abstractions made (system & environment representation, data granularity and indication of relevant time unit) and the respective shared resource where the performance analysis is all about.

Table 6-1. Abstractions made.

Modelling example	*Limited resources*	*System representation*	*System environment*	*Data granularity*	*Relevant time unit*	*Simulated real time*
1: RED	Shared total buffer, RED on total buffer	Queuing system with L sub-queues	Network, TCP-connections (responsive)	Internet Protocol packets	TCP round trip time 20 – 100 ms	20 – 50 s
2: Flow Control Mechanism	Limited total buffer size, limited bandwidth expansion	Input-output buffered switch with NxN virtual ingress egress pipes	Ethernet sources (non responsive)	Frames	Flow Control Mechanism refresh time 250 us	2 s

Simulation speed

For the simulation times mentioned in the above table, the simulation duration was about a few minutes. For checking the long run stability of the simulated distribution functions, much longer simulations were performed for a number of critical cases (sometimes taking several hours). The simulation speed is mainly determined by the abstractions made (determining the number of events to be simulated).

Simulation of large systems requires a huge number of events to be simulated. In some specific cases, the number of events can be reduced, by applying scaling of simulation results. This scaling must be justified by simulations and requires the identification of appropriate invariants.

Another possible reason for performing long simulations has to do with rare events, such as low drop rates. Consider for instance a system with a drop rate of 10^{-8}. Checking whether an algorithm can reduce this drop rate requires the simulation of at least 10^{8} events. The solution here is to introduce bias to intentionally create the rare conditions where drop occurs. Evaluation of the performance for a range of polarization rates (different strengths of bias) allows a good estimation of the performance impact of the algorithm.

General Conclusion

As a general conclusion, we state that performance analysis can significantly support architecture validation and exploration for complex systems through:

- Learning about the system.
- Detect unforeseen bottlenecks or shortcomings early in design flow.
- Quantitative assessment of impact of design decisions, algorithm exploration.
- Tuning functional algorithm to practical design.
- Aid in determining optimum dimensions, settings (thresholds, ...) and sensitivity.
- Quick answers to questions about entire system, little effort.

REFERENCES

[1] Floyd S., Fall K. Promoting the Use of End-to-End Congestion Control in the Internet. IEEE/ACM Transactions on Networking, August 1999.

[2] Niemegeers A., De Jong, G. An Incremental Specification Flow for Real Time Embedded Systems', Proceedings of the first IEEE/HLDVT workshop, 1999.

[3] Park S. K., Miller, K. W. Random Number Generators: good ones are hard to find. Communications of the ACM, Volume 31, Number 31, October 1998.

Chapter 7

SYSTEM DESIGN REUSE

Nikolaos S. Voros
INTRACOM S.A., Patra, Greece

Abstract: System Intellectual Property reusability is becoming a subject of great emergence for research aiming to extend the concept of reuse much further from ad-hoc reuse, to out-of-the engineering group bounds, including know-how reuse. Such a reuse practice should be founded on unambiguous definitions of System Intellectual Property and Reuse, on a systematic reuse methodology and on consensus and standardization of the form of Intellectual Property exchange. In this chapter, we explore the definition of System Intellectual Property and Reuse, focus on the current practices of reuse in organizations, industry, standardization bodies and academia and present a set of reuse criteria that can form the basis for effective system IP reuse.

Key words: System Intellectual Property, System IP, System level IP, IP Reuse, criteria for IP reuse, reuse automation

1. INTRODUCTION

System Intellectual Property (IP) reuse is a newly introduced research area, inspired by the more and more increasing productivity gap in current industrial processes. Current trends of research and industrial practices on System Intellectual Property and Reuse range from the conceptual identification of the relevant elements in the system design field to the formalization of practices of reuse, not to mention the legal and business issues.

In this chapter, we present the advancements on System IP Reuse, in order to provide a common basis of reference and stimulus for all the ongoing work on this area. Initially, in section 2, the basic definitions of the terms and concepts concerning system IP reuse are presented, while in section 3, the most representative organizations and the main work in industry and standardization bodies are described. Section 4 presents the relevant work regarding system IP identification, and in section 5 the current system IP reuse practices are explored. In sections 6 and 7, criteria for IP selection and reuse automation are described respectively. Finally, section 8 provides a summary of the main concepts introduced in this chapter.

2. SYSTEM IP REUSE: BASIC TERMS AND CONCEPTS

The definition of "System", on which the identification of System IP is based, is as follows:

A System is any composition of parts that performs a function or set of functions. The boundaries of a system usually follow the structural implementation, but may also cross physical boundaries. For instance, "The memory system xyz shares boards p, q, r, and s with other systems". Systems are typically hierarchical in that a system may be composed of sub-system components. A system is characterized by the interrelations and behaviors of its components [1].

Intellectual property is any product of the human intellect that is unique, novel and non obvious or any intangible asset that consists of human knowledge and ideas. System IP refers to system IP instances, such as algorithmic models, predefined IPs, components and to system and design know-how; the latter is the know-how of designers, architects and integrators and is mainly captured in the design process by checklists, guidelines, tool scripts and in proven architectures. In this context, the definition of System Intellectual Property Reuse as proposed by [2] is as follows:

System Intellectual Property Reuse is a methodology associated with discipline and means to facilitate use again of design artifacts and design knowledge at system-level, i.e. during early phases of product development (e.g. requirements definition, specification, architecture design, mapping). The reuse methodology requires establishing design-for-reuse and design-with-reuse processes.

3. IP RELATED WORK

One of the major representatives, aiming to provide means to IP reuse standardization is Virtual Socket Interface Alliance [3] that specifies open interface standards, which would allow Virtual Components (VCs) to fit quickly into virtual sockets, both at the functional level and the physical level. Object Management Group [4] promotes also reuse, having as primary goals the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. The fundamental mission of the Gigascale Silicon Research Center [5] is to empower designers to realize the potential of gigascale silicon by enabling scalable, heterogeneous, component-based design with a single-pass route to efficient implementation from a microarchitecture. While the previously mentioned organizations are dealing mainly with the technical aspects of system design and reuse, Reusable Application Specific Intellectual Property Developer Initiative [6] and International Council On Systems Engineering [7] are focusing on the creation of the appropriate culture and means to accelerate reuse practices. In the same context, in Europe, an analogous consortium, known as System Design Industrial Council-Telecom (SYDIC-Telecom) [8], focuses on the analysis of system level design flows, the definition of a system level conceptual model, system IP reusability assessment, and specification languages and formalisms analysis.

Main research results on system IP reuse can be found in the guidelines of design-for-reuse and design-to-reuse proposed by Motorola's Semiconductor Reuse Standards (SRS) [9] and Mentor Graphics Reuse Methodology Manual for System-On-a-Chip Design (RMM) [10]. These guidelines can also be used in methods for evaluating and qualifying IP for project use. OpenMORE [11] is a reference-scoring program for assessing the reusability of hard and soft IP cores for SoC design, based on the RMM and VSIA deliverables. Finally, several companies provide global collaboration networks for sharing design resources in the electronic SoC industry.

4. SYSTEM INTELLECTUAL PROPERTY IDENTIFICATION

There is lack of a consensus on a complete, systematic and comprehensive interpretation of system IP in current practices. In many cases System-level IP, System IP and IP Instances (IPIs) are usually used interchangeably in a confusing way. The following paragraphs attempt to clarify the aforementioned terms by providing details of their exact meaning and the context in which they can be used.

4.1 System-level IP, System IP and IP Instances

System-level IP, System IP and IPIs are usually used as identical, ignoring the IP conceptual aspects as being hard to be formally captured.

System-level design refers mainly to the initial phase of the design of a complex system, where the functional specification and requirements of the components of the system are determined, while the attainment of the objectives of the system based on the properties of the system components is explored. *System-level IP components encapsulate design knowledge, experience etc.*

System IP (also known as IP), as defined in the section 2, includes the knowledge and experience obtained during the design process, the implementation and integration of the system, the feedback derived from the usage of the system as a product, the maintenance, updates, upgrades, compliance to standards and even the knowledge obtained during the withdrawal of the system from the market and its replacement by more advanced systems. Design know-how is captured by means of actors: the system architects, designers and integrators, having their own concerns e.g. functionality, performance etc. Thus, *system IP is the knowledge captured as experience in the designers' minds*. The know-how is also captured in the design process and includes the methodologies, techniques, tools, styles and the reasoning about the selected solutions. Means of capturing know-how in the design process and reusing it are scripts, guidelines, patterns and documentation especially in the case of reasoning.

Each implementation of an IP, in the context of the design of a system, is an instance of that IP (IPI). IP instances, also referred as IP components or macros, can be defined through algorithmic models, microprocessors, custom components etc. IP instances are the design artifacts e.g. the design descriptions or the system architecture, as well as the artifacts related to validation and verification, like parameterized testbenches and scripts e.g. scripts for mapping to different design flows, and documentation.

Since system-level IP is used for the stage of specification capturing, system IP refers to system design knowledge as a whole and IP instances are the artifacts that realize the concepts related to System IP, the terms system-level IP, system IP and IP instance refer to different phases of the product development cycle, and should not be used interchangeably.

4.2 Classification of IPIs

The organization of an IP infrastructure, including an IP repository and automation in IP selection has provided systematic means to IPI classification. Taxonomy can be defined in order to classify IPIs in families and subfamilies with respect to functionality [12]; it may also be based on the design space explored [13]. A mechanism that claims to allow transparent design reuse is the classification of IPIs by keywords, properties, levels of reuse, taxonomy, dependency and similarity [14].

5. SYSTEM IP REUSE PRACTICES

5.1 Reuse Strategies and Practices

Design reuse during system design can take place through alternative reuse strategies [15]. For example, in *design knowledge encapsulation reuse startegy*, the key idea is to encapsulate design data, specification and implementation in a standard and secured manner; in *design exchange strategy*, design teams exchange designs through standard formats, while in *design evolution strategy* changes are made to previous designs to achieve new functionality to a reused module; finally, the *field programmable design reuse strategy* is a software-like reuse where parameterization and programming are used to differentiate products based on standard structures and reused implementations.

This classification of reuse strategies has been proven as theoretical, since none of these approaches is applied in a stand-alone manner. Rather, combinations of them are used to exploit the reusability in system-design, leading to vague borders among them. The main practices currently used to apply the aforementioned reuse strategies in practice include: *core based design*, *interface based design*, *platform based design*, *parameterization* and *derivative design*. The relationship/classification schema of the strategies and practices for reuse is depicted in Figure 7-1, where the strategies for reuse are illustrated in orthogons and the reuse practices in oval. The covering of the relative areas reveals the relationship among the various strategies and practices.

Design knowledge encapsulation comprises the automatic implementation of the design knowledge captured in specifications and is based on the tool technology supporting system design. Moving to system IP reuse, the design exchange strategy and the design evolution strategy require the design knowledge encapsulation reuse.

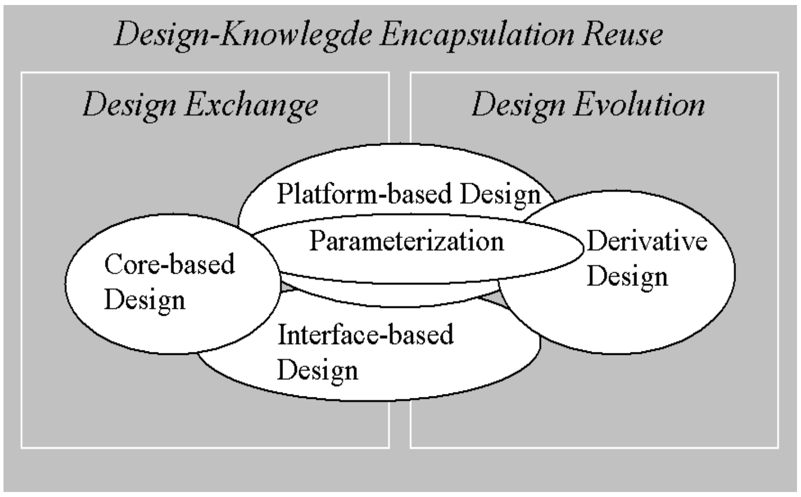


Figure 7-1. Relationship/Classification schema of strategies and practices for reuse

In design exchange strategy, fixed design exchange could be instantiated in core-based design and platform-based design, and need as prerequisite the interface-based design to efficiently apply in design reuse. For example, platform creation requires the designer to concentrate on the interfaces between architectural components and the functional blocks.

In design evolution strategy, the separation of behavior/functionality and communication in design is required in order to have the ability for the evolution of the functionality of the design. From a coarser-grained viewpoint, system design with derivative design could be also a practice classified in this category.

Parameterized system design accompanies always the design exchange strategy, since cores and platforms are parameterized or programmable, in order to provide the required flexibility. Platforms must be programmable at a variety of levels of granularity. Moreover, parameterized components must obey in standard interface implementation rules in order to be mixed and matched in systems.

In the following paragraphs, the most important practices of reuse are detailed, while the use of object oriented techniques for reuse and design patterns, as a newly introduced way for recording know-how, are presented as well.

5.2 Core Based Design

As defined in [16] *a core is a pre-designed and pre-verified block that can be used in building a larger or more complex application on a* *semiconductor chip.* Cores are often called macros and blocks. A core, apart from the delivered design, implies the know-how to use it. Cores are classified in soft, firm, and hard cores. Hard IP components are those fully implemented in a specific process technology. Soft IP components are in the form of synthesizable RTL/HDL code. They can be parameterized and they are user-configurable. Firm IPIs are an intermediate class of hard and soft IPIs. They have been optimized in structure and in topology for performance and area through floor planning/placement. The advantage of soft cores is the flexibility they provide compared to hard and firm cores. In contrast, soft cores suffer from performance evaluation inability, while firm and hard cores provide the required performance characteristics e.g. timing information (usually static). A core's reusability mainly depends on the design process, the technological process details and the data format used to deliver the core design.

The complete description of a core is comprised of a great number of models for design, verification, evaluation, timing characterization, testing, physical implementation etc. These are executable models, design descriptions at many levels of abstraction, datasheets and timing diagrams. The high-level models include architectural, instruction set and bus-functional models. The RTL, gate and transistor models provide increasingly more timing and functionality details. A hierarchy of models would provide functional information with increasingly more specific timing details. Moreover, a basic delay model and a peripheral interconnect models should additionally be considered [17].

One of the main technical challenges in core-based design is the verification and validation of cores. Test issues in core designs are described in detail in [16]. Generally, tests cannot be created by the user of the core, due to his/her lack of awareness of the internal implementation details. On the other hand, the core provider determines the test requirements of the core without knowing the target process and application. Thus, the core builder will not know which test method to adopt, the type of faults (static, dynamic, parametric), or the desired fault coverage. The core builder should prepare an internal test that is adequately described, ported, ready for plug and play and in a standard format. The IEEE P1500 working group [18] has as scope the development of a standard test method for integrated circuits containing embedded cores. Finally, it would be considered as optional that the test requirements could be entered at the system level and evolve through the design hierarchy, with a large degree of automation.

5.3 Interface Based Design

Reusing IPs developed in different design flows, using different computational models for their description, requires their interfacing in a unified model. Interface-based design is the design flow that moves design from an interconnected set of communicating processes with clearly defined and separately captured interface protocols (usually intended to test conceptual behaviour) to interconnected realized components in the final subsystem. A virtual component interface is defined as the information-transfer boundary between a VC internal behaviour and any communication channel connecting VC implementations or VC models.

Interface based design is the prerequisite technique to orthogonalize behaviour and communication, thus allowing the deployment of all the practices for reuse. Core-based design, platform-based design and parameterization could efficiently be realized through standardized interface design. VSIA is the major representative in the domain of interface-based design and the released standards seem to be commonly adopted. VSIA's System-Level Interface Behavioral Documentation Standard (SLIF) [19] describes a systematic documentation technique for system-level virtual component interfaces. At the higher levels of abstraction, the set of operations or tasks required to perform an application are initially linked by “ideal” channels; the information is sent and received as needed, without concern for conflicting resource requests or synchronization. At this stage, the architectural design may be concerned only with functionality or with communication protocols. As this design is refined, common communication resources are specified, control protocols are administered, and sharing of functional units is identified. The common issues associated with system design become visible, and the design moves from ideal to real. The separate specification of the interfaces allows the design process to proceed fully and concurrently with the minimum of design interference between teams working on separate components.

The current trend in design reuse is the usage of standard interfaces to connect third party IP to systems, while interface synthesis is the alternative solution. Standardized buses and protocols would allow for fast integration of compliant IP components [20]. During refinement, these buses could be still optimized according to the actual communication specifications. Interface synthesis is the automated synthesis of interfaces based on high-level communication and synchronization description [21]. Relevant research on interface synthesis can be found in the Chinook framework [22], Coware [23], Polaris [24] and [25, 26]. In the context of interface-based design several approaches have been proposed. In [27], a mechanism is proposed that adopts the token passing methodology from dataflow and discrete event system level while providing a method to refine communication mechanisms incrementally and hierarchically.

In the COSY model [28] higher abstraction level interfaces for VCs, similar to VSIA's interfaces, are proposed as depicted in Figure 7-2. Application-level transactions are used for programming a network of functions that specifies what the system is supposed to do. They are refined into system transactions when choosing implementation of functions to software and hardware components. Finally, system transactions that operate on abstract data-types and high-level I/O semantics are unraveled into more detailed interfaces. For hardware, VC Interface is used as a generic interface to "any" physical bus specific protocol.

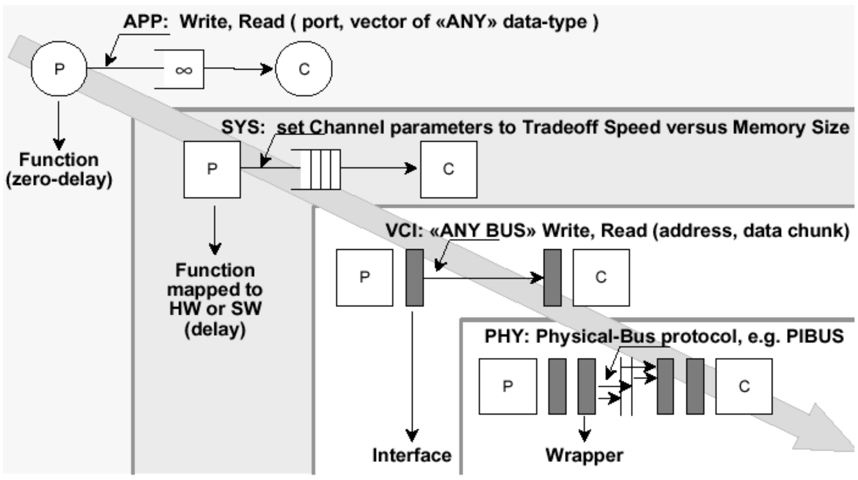

APP: Write, Read (port, vector of «ANY» data-type)
P
∞
C
Function (zero-delay)
SYS: set Channel parameters to Tradeoff Speed versus Memory Size
P
C
Function mapped to HW or SW (delay)
VCI: «ANY BUS» Write, Read (address, data chunk)
P
C
PHY: Physical-Bus protocol, e.g. PIBUS
P
C
Interface
Wrapper

Figure 7-2. COSY interface levels

In the SpecC model [29], the computation is encapsulated in behaviours and the communication is contained in channels. The communication part can be clearly identified and a different channel, which provides the same interfaces, can easily replace the channel. The SpecC model allows the hierarchical composition of both behaviours and channels providing interface adaptation and protocol conversion.

In [30], a formal verification approach around interface-based design with a component based system-level design methodology is provided. This approach is based on a timed-Petri Net notation. Once the model corresponding to the interface logic has been produced, the correctness of the system is formally verified based on the interface properties of the interconnected components and on abstract models of their functionality, without assuming any knowledge regarding their implementation.

Language requirements that would support interface-based design are addressed in [31]. The basic functionality that an interface language construct could provide is the visibility of the interface code to the IP consumer, while the interface construct is instantiated within the IP.

5.4 Parameterized System Design

Parameterization as a key strategy of reuse is presented in [32]. It can be applied in various aspects and granularities of system design, such as architectural or component design. The availability of parametric architectures widens the application domain where they can be reused. Parameterization of modules allows the exploration of different algorithms and architectures from which the most efficient solution can be chosen.

As far as parameterization is concerned, there are several kinds of parameters that can be used in practice [32]. A *static parameter* is a parameter whose value must be set before the fabrication of an instance of the System-on-Chip (SoC) and typically appears in the HDL source used to eventually create an instance of the SoC. A *dynamic parameter*, in contrast, is one whose value may be set in an already fabricated SoC that will contain extra on-chip structure to support various parameter settings. Parameters are also classified according to their level of abstraction: *circuit level*, *architecture level* and *application level*. Circuit-level parameters make small modifications to the way bits are stored or transferred e.g. parity, buffer sizes etc. Architecture-level parameters can reconfigure the system to very different logical architectures, changing for example the system's bus hierarchy, memory hierarchy or physical communication link. Finally, application-level parameters change the system's functionality.

Parameters are not usually orthogonal to each other. In fact, they are highly correlated lowering the parameter set alternatives. Parameter interdependencies either affect the overall reuse effort, or can even lead to invalid configurations. For this purpose, all the IP deliverables should be parameterized for the same parameter spaces and for synthesis, not only basic elements or complete synthesizable circuit specifications are necessary, but also archived design flows, according to the parameters of the IP.

The verification of parametric modules is a complex task and comprises a bottleneck in parameterization strategy. The complexity of verification grows as the number of parameters increases, because the number of parameters is the number of dimensions of the verification space. The interdependencies between parameters might reduce the complexity of the verification task. Parameterization is usually suitable for systems of low complexity, because the exclusive use of the parameter concept for designing complex systems implies large and hardly manageable parameter lists and unreadable component descriptions.

5.5 Platform Based System Design

Platform based design is based on constraining the design space through the use of families of architectures targeting a specific class of problems, by modifying or extending them [33, 34]. Integration platforms include both an architectural platform and the appropriate prequalified IPIs to assemble it. Thus, system design is based on selection of the appropriate platform conforming to the functional specification by probably iteratively refining or modifying it, and on selection or design of the relevant IP blocks, rather than making partitioning decisions or assembling/designing independent blocks. In [36], the key elements of an integration platform are defined, including IP, ranging from pre-verified blocks to knowledge, guidelines etc. A typical integration platform includes a specification for fitting the IP, prequalified relevant IPs, documented methods for IP development and integration, and a verification strategy.

In [36], the concepts of hardware platform, software platform and system platform are defined. A *hardware platform* is a family of micro-architectures that allows substantial reuse of software, while a *software platform* is the layer that performs the abstraction of the hardware platform at a level where the application software sees a high-level interface to the hardware. The combination of the hardware and software platforms constitutes the *system platform*. Hardware platform constraints are usually expressed in terms of performance and area. They emerge as a result of a trade-off between the size of the application domain, that reflects the space of support, and the size of the micro-architecture space, that reflects the degree of accepted over-design. The software platform wraps the essential parts of the hardware platform: the programmable cores and the memory subsystem via a RTOS, the I/O subsystem via the device drivers and the network connection via the network communication subsystem.

A system designer maps its application into the abstract representation that includes a family of micro-architectures that can be chosen to optimize cost, efficiency, energy consumption and flexibility. The mapping of the application into the actual architecture can be carried out, at least in part, automatically if a set of appropriate software tools (e.g., software synthesis, RTOS synthesis, device-driver synthesis) is available. In [34] four types of platforms are distinguished: *full application platforms*, *processor-centric platforms*, *communication-centric platforms* and *fully programmable platforms.* Full-application platforms let derivative-product designers create complete applications on top of hardware-software architectures and include a library of hardware modules in a variety of configurations. Processor-centric platforms focus on software access to a processor and allow also addition of specific hardware elements and selection of the appropriate operating system. Communication-centric platforms give consumers a communication fabric optimized from a specific application domain. Finally, fully programmable platforms allow customers to customize them by adding programmable logic.

5.6 Object Oriented System Design

Object-oriented methodologies focus on the organization of discrete objects, characterized by their structure and behaviour and the way they interact [37]. Object-oriented techniques were mainly used for software development, but nowadays there is an increasing trend to apply object-orientation to integrated systems and generally for the design of complex systems, especially at the stage of specification capturing and conceptual design. The object oriented aspects that are most relevant to system design are object/instance, class, attributes and aggregation. The object-oriented methodology allows intellectual property reuse through object libraries, design patterns, and automatic documentation generation/

Object-oriented design has main attributes that support complex system design.

- *Abstraction* provides means to focus on the essential aspects of a system for a given design goal, avoiding details, enabling an abstract and compact specification, but in the same time preserving the freedom to make decisions from the most powerful early abstraction levels.
- *Encapsulation* supports compact specification and promotes reuse. In the general sense, object-oriented technology utilizes objects and classes to define a system.

In [38], an object-oriented system engineering methodology is presented where class interfaces are used to provide structural representation through all abstraction levels. In [39] an approach for incorporating cores into a system-level specification is described, where an object-oriented language is used for specification, representing each core as an object. Three specification levels are defined and the appropriateness of existing inter-object communication methods for cores is evaluated. Another object-oriented language, called HDLC++, is presented in [40]. It is based on an object-oriented hierarchy of classes that describe the structure of the components. AMICAL behavioral synthesis tools [41] and OCAPI C++ Class Library [42], use object oriented behavioral descriptions for behavioral component-level reuse.

Object-oriented methods are also used to group IPIs into libraries, databases or clusters and to form the appropriate infrastructure for IP reuse [12, 43, 44].

5.7 Pattern Based Design

The concept of pattern has its origin on the work of Christopher Alexander [45]. According to Alexander:

Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice.

Patterns have been extensively used in object-oriented software development, and help the designers to create a way of communicating and exchanging experience about problems and their solutions, forming a powerful way of recording know-how [46]. The main types of patterns encountered in software systems consist of *architectural patterns*, *design patterns* and *idioms* (idioms are sometimes called coding patterns), referring to structural, refinement and programming strategies respectively [47].

In system design world, the design issues encountered are far more complicated compared to these of software world. Although the use of patterns is not adequately mature yet, there are several concepts that can be adopted from software patterns and can be used to describe system design concepts. For example, the *singleton pattern* described in [46] which is used in object oriented system design to ensure that only one instance of a class is created; other objects need not know that they are accessing a singleton or one of multiple instances of the class; this pattern serves as a useful access point to physical resources. An example of possible use of the singleton pattern in SoC design is access control to a shared resource. Within the singleton class, a semaphore can be used to guard against multiple consecutive attempts to access the resource.

Patterns can also be used in component-based design. Figure 7-3 describes the structure of a pattern named *CompositeComponent* [48], which can be used to describe the constituent components of the system under design. Grouping of related components to represent part-whole hierarchies decreases complexity, while allowing composites to be treated similarly since individual components hide details. A CompositeComponent consists of leaf components (that do not have hierarchy) and other composite components. Components in a composite interact through connectors that connect components, which are components as well.

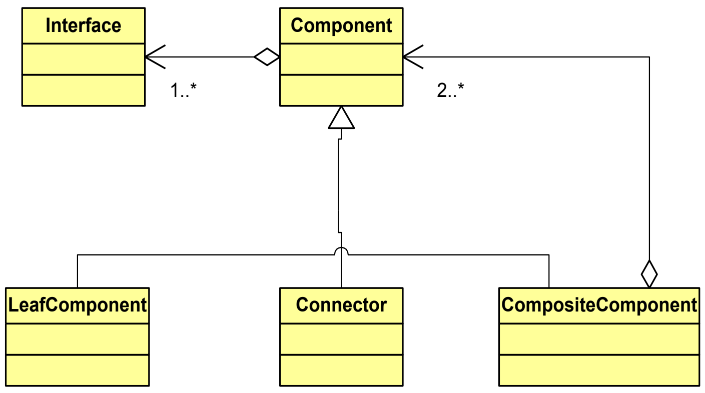


Figure 7-3. The CompositeComponent pattern

6. CRITERIA FOR IP SELECTION

In most companies, IP reusability is a challenging issue which has not been fully exploited yet. Reusability usually takes place through reuse of IP components. The two main categories of IP components are:

- **Internally developed components** which represent the know how of a specific company and encapsulate the experience of the design teams. They represent knowledge that is available among different design teams within the same company. The component development and on going support all over components' life cycle is maintained within the company.
- **Components acquired by third party suppliers**, also known as external components, are components implemented by a third party company. They are usually available in a form that does not reveal the internal details of the component e.g. net lists. They are black boxes for the design teams and they are maintained by the company producing them.

In the next paragraphs a set of criteria for evaluating IP components in practice is described. Not all criteria are always applicable, and this is mainly due to restrictions imposed by the nature of the component. For example, for external components the actual component description is not available to the end users. Table 7-1 presents an overview of the criteria that will be analyzed in the sequel, and the cases in which they are applicable.

Table 7-1. An overview of the criteria for IP evaluation/selection

Criterion	*Type of IP component*
Compliance with existing know how	External
Adequate documentation	Internal/external
Support of different technologies	Internal/external
Availability at different levels of abstraction	Internal (this is difficult for external components due to confidentiality restrictions)
Parameterization	Internal/external
Verification and testing	Internal/external
Compliance with existing standards	Internal/external
Availability of support tools	Internal/external
Component on going support	Internal/external

6.1 Compliance with Existing Know-How

Most companies have at their disposal repositories of internally developed IPs that reflect the company's know-how and imply a reuse plan through which future products will rely on existing IP components. As a result, before adopting a third party component the designers must make sure that it complies with company's know-how and reuse strategy. For example it is important for the component to support interfaces compliant with the ones of the IPs in the repository.

One additional parameter that is also important for the selection of an IP is whether it supports various technologies or not and in particular if the technology used for its implementation is compliant with the one used for the system in which it will participate.

6.2 Adequate Documentation

IP documentation is another important issue that must also be taken into account. No matter how efficient an IP component is, it is useless if the designer is not aware of its actual functionality, the operations supported, its configuration parameters etc. What is usually expected is a *user manual* and a *designer's manual* accompanying the IP. The user manual focuses on the functionality of the IP and how it can be used as part of a more complex design. Apart from the documentation of the core specification of the IP, documentation on its optimum integration is required as well. The designer's manual on the other hand, provides an in depth description of the IP focusing on implementation issues and is usually available for internally developed IPs. Its actual goal is to provide any details necessary for the designers that will produce the next updates of the component and thus facilitate the design know-how transfer among the designers.

Additionally, the IP documentation should allow the designer to communicate critical constraints from one team to another. Natural language specifications coming from long source documents are often incompatible, and make it difficult for the design teams to communicate on important design aspects e.g. constraints. The same problem appears in the communication between vendors and customers.

6.3 Availability at Different Levels of Abstraction

The design of complex systems usually relies on design process models like spiral or V-model [49], where the final product is produced through successive refinements of system models. Thus, it is necessary to have the IP component available at different levels of abstraction so as to make it part of the system as early as possible. What is usually available is either C models of the IP component, to experiment with the actual behavior of the IP early enough in the development cycle, or dummy models of the IP with well-defined interfaces to test the I/O timing of the IP at the level of integration. As soon as the IP integration is achieved, the dummy models can be gradually replaced by fully functional models of the IP component.

6.4 Parameterization

The use of parameters is not always indicative of the IP flexibility, since the increase in the number of parameters makes it difficult to use. Additionally, IP parameterization is cumbersome at the level of synthesis where parameters that are not fixed lead both to degradation of the performance/speed of the final implementation and increase of the number of gates used. For that purpose, some IPs require the definition of a critical performance factor, and how much RAM or how many I/O ports will be needed. In this way, the system designer adjusts the IP according to the performance requirements of the system under design.

6.5 Verification and Testing

IP verification must also be taken into account before selecting an IP. What the IP user has usually at his disposal is a "black box", with well-defined interfaces, that fulfils the functionality required. It is significant, in terms of quality, whether the IP is pre-verified or not. Additionally, means for verifying the IP as part of a larger system are necessary.

As far as IP testing is concerned, the ideal approach is the construction of a test bench that is offered along with the IP component. In this way, the potential user is able to instantiate the IP as part of the test bench provided and test if its actual behavior is in accordance with the IP documentation. Alternatively, models for testing the IP behavior through co-simulation could be provided. In both cases, the IP testing must be irrelevant of the environment in which the IP is used.

In terms of quality, silicon proven IPs are usually the most preferable ones; the existence of the test bench is definitely an advantage, while in some cases success stories of the specific IP are indicative of its quality. In cases where testbenches are not available, guidelines on testbench construction could be valuable.

6.6 Compliance with Existing Standards

During the last years, standardization bodies like the ones mentioned in section 3 promote the standardization of the IP interfaces. IPs compliant with existing standards make their integration in the system under design easier.

6.7 Availability of Support Tools

Tools for supporting the IP integration facilitate the use of IPIs, and make it easier for the designers to become familiar with the IP functionality. Such tools allow the designers to tailor the IP and integrate and verify it within the system under design. This means that the designers need a way to access the IP and to understand how it is implemented and how it will work in their design. Flexible, transparent tools that don't require the engineers to become an IP expert are needed.

6.8 Component On-Going Support

On going component support is a crucial parameter from the designer’s perspective. In the case of IP components supplied by third party companies, it is important to have on going support in order to handle the various problems arising during system design, since the components are used as black boxes by the design teams and the designers have no control over the component’s behaviour. The same holds for internally developed components, even though in this case component support is easier since the component intrinsic details are at the disposal of system engineers.

7. REUSE AUTOMATION

In [50] the work in the field of reuse automation is presented, and it is classified into four categories: (a) The *description oriented systems*, (b) the *content oriented systems*, (c) the *interface oriented systems* and (d) the *wrapper oriented systems*. The description oriented systems, such as READEE [51], RMS [52] and Bosch [53], are libraries of components together with a supporting searching mechanism. Content oriented systems, such as Polynomial Description systems [54] and Synopsys DesignWare [55] system, attempt to characterize the components functionally, in a formal way, setting the basis for automating the search in a component library. Interface oriented systems e.g. PIG [56], MODIS (Module Interface Synthesizer) [57], SpecC [29], Polaris [24] and Polis [58], automatically generate the communication logic between IPs. Finally, wrapper oriented systems, main representatives of which are OOCL [39] and SpecC, create a wrapper around the IP in order to be able to include the IPs into the specification, and emphasize on an executable system specification and on the refinement methodology that will lead to the final implementation.

8. SUMMARY

System IP reuse is becoming an emerging research field in the area of modern embedded system design. In the previous sections we presented an overview of the main reuse issues encountered in system design and attempted to clarify the concepts related to system IP reuse. The research activity in this area reveals that although system IP reuse is not an easy task to accomplish, it is considered as a promising alternative to handle the ever increasing complexity of the systems. The diversity of the approaches presented is not always an advantage for the potential designers; thus, the path to effective system IP reuse must pass through the stabilization of the proposed techniques and their integration in a common framework under globally accepted standards.

REFERENCES

[1] VSIA SYSTEM LEVEL DESIGN DWG (2001). VSIA: System Level Design Model Taxonomy Document Version 2. Retrieved May 2003 from: http://www.vsi.org/library/datasheets/ sld220ds.pdf

[2] SYDIC-Telecom WG1 (2002). SYDIC-Telecom: Glossary and Taxonomy Deliverable ND1.2 Release 2 version 1.0, Retrieved May 2003 from: http://www3.cti.ac.at/ecsi/ecsi/projects/sydic/store/welcome.asp?dir=WP1%20Glossary

[3] VSI, Virtual Socket Interface Alliance, 1996- . At http://www.vsi.org

[4] OMG, Object Management Group, 1998- .At: http://www.omg.org

[5] GSRC, Gigascale Silicon Research Center, 1998- . At: http://gigascale.org

[6] RAPID, Reusable Application Specific Intellectual Property Developer Initiative, At: http://www.rapid.org

[7] INCOSE, International Council On Systems Engineering, 1990-. At: http://www.incose.org

[8] SYDIC-Telecom, System Design Industrial Council of European Telecom Industries, 2000- . At: http://sydic.vitamib.com

[9] SRS,Semiconductor Reuse Standards, At:http://www.motorola.com/webapp/sps/site/prod summary.jsp?code=SRSSTANDARDS&nodeId=01Bfq62638Kcmw

[10] Keating, M., and Bricaud, P, Reuse Methodology Manual for System-on-a-Chip Design, Kluwer Academic Publishers, Boston, 2002.

[11] OpenMORE program, At: http://www.openmore.com/ openmore/about.html

[12] Behnam B, Babba K, Saucier G. IP Taxonomy, IP Searching in a Catalog. Proceedings of Conference on Design, Automation and Test in Europe; 1998 February; Paris. France.

[13] Peixoto H, Jacome M, Royo A, Lopez J. The Design Space Layer: Supporting Early Design Space Exploration for Core-Based. Proceedings of Conference on Design, Automation and Test in Europe; 1999 March; Munich. Germany.

[14] Reutter A, Rosentiel W. An Efficient Reuse System for Digital Circuit Design. Proceedings of Conference on Design, Automation and Test in Europe; 1999 March; Munich. Germany.

[15] Girczyc E, Carlson S. Increasing design quality and engineering productivity through design reuse. Proceedings of 30th Design Automation Conference; 1993 June; Dallas. USA.

[16] Gupta R.K., Zorian Y. Introduction to Core-based System Design. IEEE Design & Test of Computers 1997; 14(4):15-25.

[17] Hunt M., Rowson J. Blocking in a system on a chip. IEEE Spectrum 1996; 33(11): 35-41.

[18] IEEE P1500 Working Group, 1995- . At: http://grouper.ieee.org/groups/1500/

[19] VSIA SYSTEM LEVEL DESIGN DWG (2000). System-Level Interface Behavioral Documentation Standard Version 1. Retrieved May 2003 from: http://www.vsi.org/library/ specs/summary.htm

[20] Flynn D. AMBA: Enabling Reusable On-Chip Designs. IEEE Micro 1997; 17(4):20-27.

[21] Vermeulen F, *Reuse of System-Level Design Components in Data-Dominated Digital Systems*. Katholike Universiteit Leuven: PhD Dissertation, 2002.

[22] Chou P, Ortega P, Hines R, Partidge K, Borriello G. IPChinook: An Integrated IP-based Design Framework for Distributed Embedded Systems. Proceedings of Proceedings of the 36th ACM/IEEE Design Automation Conference; 1999 June; LA. USA.

[23] Bolsens I., De Man H., Lin B., Van Rompaey K., Vercauteren S., Verkest D. Hardware/Software Co-Design of Digital Telecommunication Systems. IEEE Special issue on HW-SW Co-Design 1997; 85(3): 391-418.

[24] Smith J, De Micheli G. Automated Composition of Hardware Components. Proceedings of the 35th ACM/IEEE Design Automation Conference; 1998 June; San Francisco. USA.

[25] Madsen J, Hald B. An Approach to Interface Synthesis. Proceedings of the 8th International Symposium on System Synthesis; 1995 September; Cannes. France.

[26] Madisetti V., Shen L. Interface Design for Core Based Systems. IEEE Design and Test of Computers 1997; 14(4):42-51.

[27] Rowson J, Sangiovanni-Vincentelli A. Interface-based design. Proceedings of Proceedings of 34th ACM/IEEE Design Automation Conference; 1998 June; California. USA.

[28] Brunel J, Kruijtzer W, Kneter H, Petrot F, Pasquier L, De Kock E, Smits W. COSY Communication IPs. Proceedings of Proceedings of 37th ACM/IEEE Design Automation Conference; 2000 June; California. USA.

[29] Domer R, Gajski D. Reuse and Protection of Intellectual Property in the SpecC System. Proceedings of the 2000 conference on Asia and South Pacific design automation; 2000 January; Yokohama. Japan.

[30] Karlsoon D, Eles P, Peng Z. Formal Verification in a Component-based Reuse Methodology. Proceedings of the 15th international Symposium on System Synthesis; 2002 October. Kyoto. Japan. New York: ACM Press, 2002.

[31] Flake P, Davidmann S, Kelf D, Burish C. The IP Reuse Requirements for System Level Design Languages. Proceedings of International Property Conference; 2000 April; California. USA.

[32] Givargis T, Vahid F, Henkel J. System-level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip, Proceedings of IEEE/ACM International Conference on Computer Aided Design; 2001 November; San Jose. USA.

[33] Chang, H, Cooke, L, Hunt, M, Martin, G, McNelly, A, Todd, L, *Surviving the SoC Revolution: A Guide to Platform-Based Design*, Boston: Kluwer Academic Publishers, 1999.

[34] Martin G., Schirrmeister F. A Design Chain for Embedded Systems. IEEE Computer 2003; 35(3):100-103.

[35] Filippi E, Licciardi L, Montanaro A, Paolini M, Turolla M, Taliercio M. The Virtual chip set: a parametric IP library for system in a chip design; Proceedings of IEEE Custom Integrated Circuits Conference; 1998 June; California. United States.

[36] Kreutser K., Mlik S., Newton R., Rabaey J., Sangiovanni-Vincentelli A. System Level Design: Orthogonalization of Concerns and Platform-Based Design. IEEE Transactions on Computer-Aided Design of Circuits and Systems 2002, 19(12):1523-1543.

[37] Lidsky, D, *The Conceptual-Level Design Approach to Complex Systems*, University Of California Berkeley: PhD Dissertation, 1998.

[38] F. Doucet and R. Gupta (2000). Microelectronic System-on-Chip Modeling using Objects and their Relationships. *1st Online Symposium for Electrical Engineers*, Retrieved may 2003 from: http://www.ics.uci.edu/~iesag/yaml/docs/osee.doc

[39] Givargis T, Vahid F. Incorporating Cores into System-Level Specification. Proceedings of 11th International Symposium on System Synthesis; 1998 December; Hsinchu. Taiwan.

[40] Heuser O, Fiedler H. New Method for Reuse-Driven Design of Digital Cirtuits. Proceedings of IEEE Custom Integrated Circuits Conference; 1999 May; San Diego. USA.

[41] Kission P., Jerraya A., Behavioral design allowing modularity and component reuse. Journal of Microelectronic Systems Integration 1997; 5(2): 67-83.
[42] Schaumont P, Cmar R, Vernalde S, Engels M, Bolsens I. Hardware Reuse at the Behavioral Level. Proceedings of 36th ACM/IEEE Design Automation Conference; 1999 June; New Orleans. USA.
[43] Bottger J, Agsteiner K, Monjau D, Schulze S. An Object-Oriented Model for Specification, Prototyping, Implementation and Reuse. Proceedings of Conference on Design, Automation and Test in Europe; 1998 February; Paris. France.
[44] Oberg J, Kumar A, Jantsch A. An Object-Oriented Concept for Intelligent Library Functions. Proceedings of 11th International Conference on VLSI Design; 1998 January; Chenai. India.
[45] Alexander, C, *The Timeless Way of Building*. New York: Oxford University Press, 1979.
[46] Gamma, E, Helm, R, Johnson, R, Vlissides, J, *Design Patterns: elements of reusable Object-oriented Software.* Addison-Wesley, 1995.
[47] Buschmann, F, Meunier, R, Rohnert, H, Sommerlad, P, Stal, M *Pattern-Oriented Software Architecture - A System of Pattern,* Wiley and Sons Ltd., 1996.
[48] SYDIC-Telecom WG2 (2003). SYDIC-Telecom: System Design Conceptual Model SDCM, ND2 Release 2 version 2.0, Retrieved May 2003 from: http://www3.cti.ac.at/ecsi/ecsi/projects/sydic/store/welcome.asp?dir=WP2%20SDCM
[49] Voros N., Sanchez L., Alonso A., Birbas A., Jerraya A. Hardware/Software Co-design of Complex Embedded Systems: An approach using efficient process models, multiple formalism specification and validation via co-simulation. Journal of Design Automation for Embedded Systems 2003; 8:5-49.
[50] Barna, C, *Reuse Automation*. FZI Forschungsbericht, 1999.
[51] Oehler P, Vollarath I, Conradi P, Bergmann R. Are you READEE for IPs? Proceedings of 2nd GI/ITG/GMM Workshop of Reuse Techniques for VLSI Design; 1998 September; Karlsruhe. Germany.
[52] Seepold, R, *A Hardware Design Methodology with Special Emphasis on Reuse and Synthesis*, University of Tubingen: PhD Thesis, 1997.
[53] Reutter, A, *Rechnergestutzte Wiederwendung Digitaler Schltungsmodule*, University of Tubingen: PhD Thesis, 1999.
[54] Smith J, De Michelli G. Polynomial Methods for Component Matching and Verification; Proceedings of International Conference on Computer Aided Design; 1998 November; San Jose. USA.
[55] SYNOPSIS DESIGNWARE (2003), Retrieved May 2003 from: http://europe.synopsys.com/dialog/ euro_compiler/issue 17/synopsys1.html
[56] Passerone R, Rowson J, Sangiovanni-Vincentelli A. Automatic Synthesis of Interfaces between Incompatible Protocols. Proceedings of the 35th ACM/IEEE Design Automation Conference; 1998 June; San Francisco. USA.
[57] Siegmund R, Mueller D. An Approach to Specification and Synthesis of adaptive Interfaces of reusable Hardware Modules. Proceedings of Forum on Design Languages; 1999 August; Lyon. France.
[58] POLIS Berkeley Co-design Environment (2003), Retrieved May 2003 from: http://www-cad.eecs.berkeley.edu/Respep /Research/hsc/abstract.html

Chapter 8

EXAMPLE OF USING THE SYSTEM DESIGN CONCEPTUAL MODEL

Nikolaos S. Voros
INTRACOM S.A., Patra, Greece

Abstract: The main goal of this chapter is to demonstrate how System Design Conceptual Model (SDCM) metamodel can be instantiated in the context of real world applications. For that purpose, a case study borrowed from the telecommunication domain is used in order to exhibit the relationship between SDCM concepts and the design phases followed for the design of complex systems. The application employed is part of a MAC layer protocol for wireless ATM networks.

Key words: Telecom system design, System Design Conceptual Model instantiation, System Design Process instantiation

1. INTRODUCTION

This chapter presents how the use of SDCM metamodel introduced in Chapter 4 can be instantiated in practice. As already explained, the main purpose of the SDCM is to provide concepts for the description of the models of the design process and the system under design. During the various design steps, the system under design is described through different views: functionality, architecture and estimation/validation.

The system under design is part of a MAC layer protocol, called MASCARA (Access Scheme based on Contention And Reservation for ATM), that offers the quality of a standard ATM network connection over wireless links [1, 2].

In the context of MASCARA, the System Under Design (SUD) is a set of models that evolve through successive refinements. The formalisms used for describing it at different levels of abstraction vary from natural language at early design stages to formal languages during the latest development stages. The SUD design artefacts pertaining to the first three layers of abstraction (L1, L2 and L3) are described using Specification and Description Language SDL [3], while for the artefacts of the last abstraction layer (L4) SOLAR [4] is employed; each layer encompasses several refinement cycles of the design artifacts [5].

The development process starts with an initial set of user requirements. Based on these, design artefacts of the MASCARA protocol are being developed at different layers of abstraction (L1 – L4). The aspects that are gradually refined at the four abstraction layers are the constituent parts of the SUD architecture and their behaviour respectively. More specifically:

- At layers L1 and L2 the components, the connectors and the external interfaces of the SUD are defined. The outcome is an incomplete functional architecture of the SUD.
- At L3 the complete functional architecture along with its configuration is available; both are described in SDL.
- At L4 the functional architecture is mapped on hardware and software architectures, while the appropriate configurations are defined as well. The outcome of L4 is a virtual prototype of the SUD physical architecture.

The design phases of the MASCARA protocol are detailed in section 2, while section 3 outlines how System Under Design Model (SUDM) and System Design Process Model (SDPM) are interrelated; finally, section 4 summarizes the main concepts presented in this chapter.

2. INSTANTIATION OF SDCM FOR THE DESIGN OF MASCARA PROTOCOL

2.1 User Requirements

MASCARA is a MAC layer protocol that allows mobile terminals moving indoors to connect to a core ATM network through standard QoS over wireless connections. MASCARA takes advantage of the wireless technology at the physical layer, and extends the traditional ATM protocol stack so as to be able to offer ATM QoS over air.

As presented in Figure 8-1, from the mobile user point of view what is required is to be able to:

- Connect to the ATM network.

- Negotiate the QoS for the required service e.g. video on demand.
- Participate in the network while allocated the required bandwidth.
- Move around without restrictions.

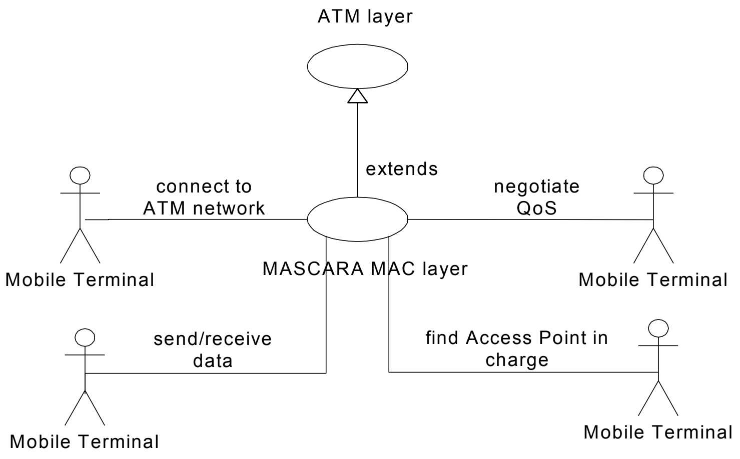


Figure 8-1. User requirements for a wireless ATM network.

The aforementioned set of end user requirements generates requirements that pertain to the physical infrastructure of an ATM network that supports ATM services over wireless connections. Figure 8-2 outlines the physical architecture of such networks.

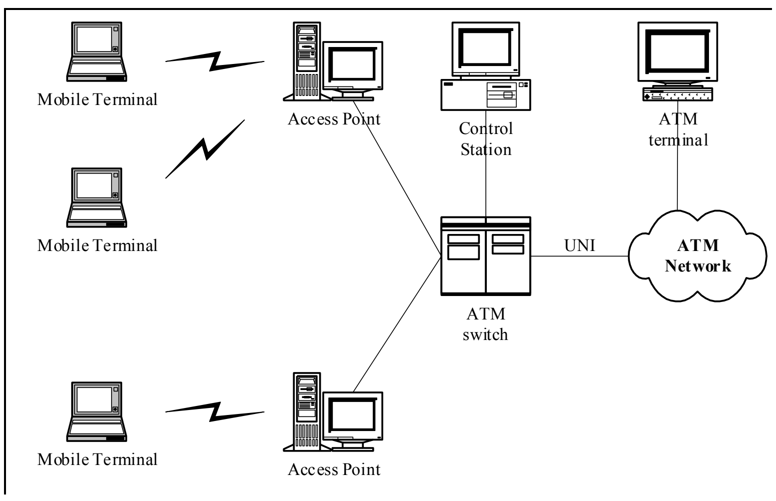


Figure 8-2. Overview of a wireless ATM system.

2.2 MASCARA at L1 Layer of Abstraction

The network architecture presented in Figure 8-2 reveals the real world entities that must be modeled:

- *ATM switch* is a standard customer premises access node, containing also mobility specific software and minimum hardware modifications.
- *Access Point (AP)* is the network element connected to the ATM switches with standard ATM connections.
- *Mobile Terminal (MT)* is the end-user equipment that contains the wireless ATM radio adapter card, interfacing the air-interface.

The MASCARA protocol as part of the ATM protocol stack is implemented both at MT and AP. At the MT side, MASCARA extends the MAC layer of the protocol stack by adding services through which the MT:

- Identifies the AP that is responsible for the area in which the mobile user is moving
- Issues requests for joining to the ATM network
- Negotiates the QoS of the connection
- Transmits data through the air interface during the uplink period (the period during which MTs send data to the AP)
- Maintains the QoS when the user moves in the area of another AP; the latter involves negotiation with the AP that is in charge of the new area.

The MASCARA implementation at the AP side is significantly different since the Access Point must:

- Inform the MTs in its area that it is alive
- Guarantee that the QoS required by the MTs moving around in its area is consistent with the one initially negotiated
- Schedule data transmission to the MTs during the downlink period (the period during which AP transmit data to the MTs).

Figure 8-3 depicts the role of the MASCARA protocol as part of the wireless ATM protocol stack. The environment of the system modeling the MASCARA MAC layer is the ATM layer and the Physical RF layer.

At this level of abstraction, the SUD artefacts are described in SDL. They are mainly focusing on initial system decomposition and textual descriptions of the MASCARA subsystems. The design artefacts at L1 are described using the basic SDL constructs like SDL blocks and channels in order to identify the main functional units of the SUD.

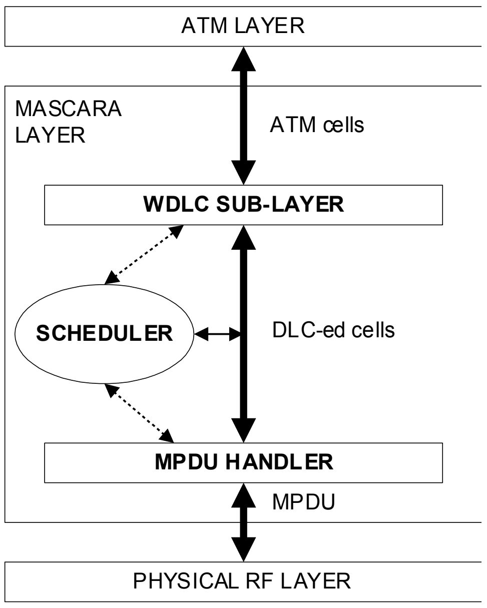


Figure 8-3. Data path for the MASCARA protocol.

2.3 MASCARA at L2 Layer of Abstraction

Based on the SUD artefacts developed in L1, the next step is to refine them and produce the artefacts of L2 layer of abstraction. The language used for artefact description at L2 is SDL and the SDL constructs used are the same as the language constructs used in L1.

Through successive refinements, the subsystems constituting the SUD at L1 are refined; the textual description of their functionality is replaced with components that communicate with ideal channels. If there are reusable components already available as IPs, they are included at this level of abstraction. In the case of MASCARA, the components implementing segmentation and reassembly in the MPDU subsystem are dummy models of IPs already available though a company internal IP library.

The functionality of the processes constituting the components is not available yet. The same holds for the communication among components, which is abstract as well. The outcome of L2 level of abstraction is an initial *functional architecture* of the SUD. Figure 8-4 presents the initial system decomposition in functional blocks that communicate among each other with ideal communication channels. The communication with the environment is achieved through channels as well (channels C6, C7 and C8).

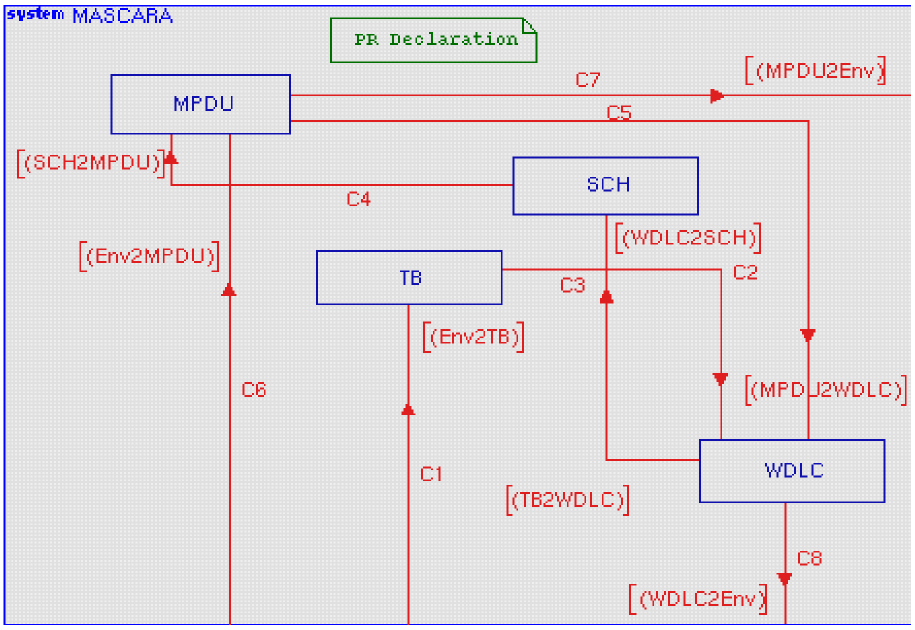


Figure 8-4. Functional architecture of the MASCARA protocol in SDL.

The exact role of the MASCARA protocol subsystems depicted in Figure 8-4 is described as follows:

- Wireless DLC (WDLC) is decomposed into the processes depicted in Figure 8-5, and is responsible for recovering from the low quality of the air-interface. WDLC_Xmit_Data process is in charge of building the cell trains, and WDLC_Rcv_Data is in charge of sending to the MASCARA-ATM interface the cells.
- MAC Data Pump Unit (MPDU) manages the slot map according to the connection profile (AAL) and the Quality of Service (QoS) parameters. Furthermore, it sends and receives MPDU to/from the physical layer.
- Scheduler (SCH) block includes the Scheduler and differs between the AP (master side) and the MT (slave side).

Figure 8-5 outlines the part of the functional architecture that pertains to the WDLC block of Figure 8-4. The leaf nodes refer to functions/procedures used by the processes implementing the WDLC block behaviour.

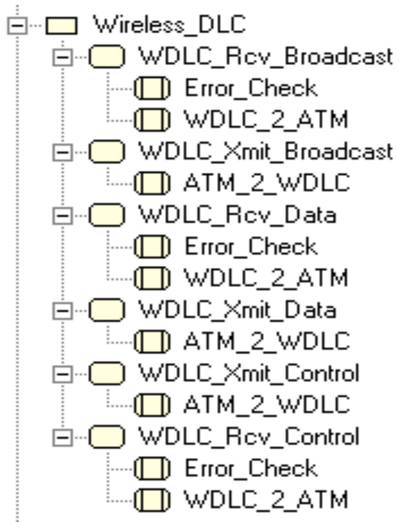


Figure 8-5. Functional decomposition of the WDLC block of the MASCARA protocol.

2.4 MASCARA at L3 Layer of Abstraction

Having described the functional structure of the MASCARA protocol, the next step is to complete the functional architecture by adding functions and procedures that implement the actual behaviour of each process. The detailed definitions of SDL states, SDL transitions and SDL signals that activate the transitions between the states of each SDL process are defined at this level of abstraction as well.

Since each process has a well defined behaviour, the SDL signal lists (e.g. [WDLC2Env] in Figure 8-4) of the signals exchanged between the MASCARA system blocks become more detailed too. The signal exchange between each process and its environment is achieved through zero delay channels called *signal routes* in SDL, which connect the process to a specific called *gate*. Gates act either as interfaces between the process and its surrounding block, or as interfaces between the blocks constituting the SUD. The exact signals exchanged between the process, and between the blocks, if we consider the system at L1, are defined at this level of abstraction.

With respect to the constituent parts of the SUDM, at this level of abstraction the SUD subsystems are defined through components (SDL blocks and SDL processes) and connectors (SDL ports and SDL gates).

```
/*********************************************************************
OPERATOR: PUT_WDLC_HEADER_FIELD
PARAMETERS: WDLC_Cell_ptr, Integer, Integer
RETURN VALUE: Integer
DESCRIPTION: Writes WDLC header field. Coding as in
GET_WDLC_HEADER_FIELD.
*********************************************************************/

#ifndef NOANSI
SDL_integer PUT_WDLC_HEADER_FIELD(#(WDLC_Cell_ptr) yParam1,SDL_integer
yParam2, SDL_integer yParam3)
#else
PUT_WDLC_HEADER_FIELD(yParam1,yParam2,yParam3)
WDLC_Cell_ptr yParam1;
SDL_integer yParam2;
SDL_integer yParam3;
#endif
{
switch (yParam2)
{
case 0: (yParam1->#(wdlc_header).mvc_id)=yParam3; return yParam3;
case 1: (yParam1->#(wdlc_header).request)=yParam3; return yParam3;
case 2: (yParam1->#(wdlc_header).sn)=yParam3; return yParam3;
case 3: (yParam1->#(wdlc_header).rn)=yParam3; return yParam3;
case 4: (yParam1->#(wdlc_header).reserved)=yParam3; return yParam3;
case 5: (yParam1->#(wdlc_header).pt)=yParam3; return yParam3;
case 6: (yParam1->#(wdlc_header).clp)=yParam3; return yParam3;
default:
return -1;
}
return 0;
}
```

Figure 8-6. C function for WDLC_Cell_ptr.

One of the main problems encountered during the design of MASCARA protocol was the inefficiency of the existing SDL data types. As illustrated in Figure 8-3, MASCARA accepts ATM cells, transforms them appropriately (DLC-ed cells) and sends them to the physical layer. An initial approach for the MASCARA functionality would be to simply copy cell trains from ATM layer, enrich them with the information required by the MASCARA sub-layer, and copy them to the physical layer for transmission. The simulation revealed that this copy of ATM cells is computationally intensive and reduces the performance of the overall system. The approach adopted was, instead of copying cell trains, to simply pass between the adjacent layers a pointer to the cell train. The inefficiency of SDL lies in the fact that the language does not have structures to describe the concept of pointer. Consequently, the designers had to define new abstract types for describing pointers and operations related to them. The approach followed was to define the abstract data types (ADT) as external C functions that implement the required functionality. The C function depicted on Figure 8-6 is an example of the operations defined; it updates the header field of a WDLC cell through the WDLC_Cell_ptr pointer.

As a result of the detailed behaviour definition of the various SUD parts either through FSMs describing the behaviour of SDL processes, or through external C functions, the complete functional architecture of the SUD is available at this point.

The transition from L3 to L4 level of abstraction relies on the mapping of the functional architecture to the hardware and to the software architectures. The use of SDL abstract channels at L1 and L2 layers of abstraction is an advantage since it enables the designer to focus on the functional decomposition and behavioural description of the sub-systems. At levels L3 and L4, the designer has to focus on communication among subsystems and decide for the used protocols, shared variables, buffers, queues etc. For that purpose, the SDL description of the MASCARA protocol was transformed to an intermediate formalism called SOLAR [4] that supports refinement of subsystem communication and mapping to hardware and software architectures through hardware/software partitioning of the functional architecture. Figure 8-7 outlines the functional architecture of MASCARA described in SOLAR.

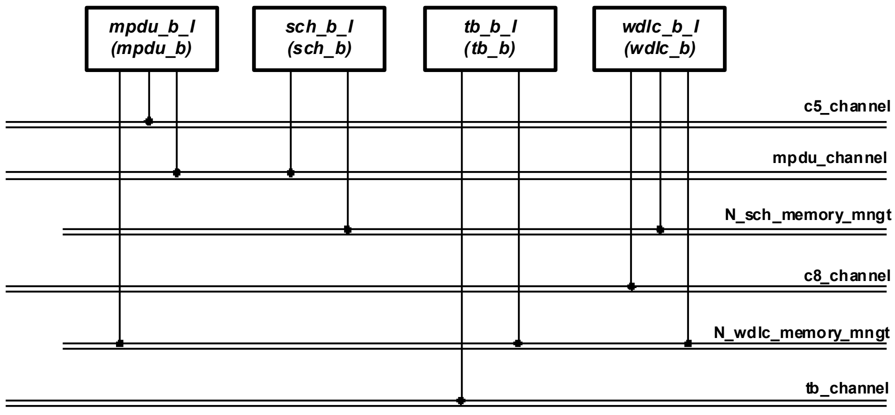


Figure 8-7. Functional architecture of MASCARA protocol described in SOLAR.

During the conversion of SDL specification to SOLAR, SDL processes are converted to design units, SDL processes' behaviour is converted to state stables (the equivalent SOLAR concept for FSMs used in SDL processes) and SDL abstract channels are converted to SOLAR abstract channels. The latter enable communication refinement from abstract communication to communication through specific protocols or shared variables.

2.5 MASCARA at L4 Layer of Abstraction

At this level of abstraction the behaviour of the MASCARA subsystem is described directly through state tables, and the next step is to refine the abstract communication between the subsystems. The refinement of the abstract communication is achieved through the selection of the exact communication mechanisms through a library that contains alternative communication protocols like FIFO, rendezvous and shared variables.

As soon as the designer has decided in favor of a specific protocol the next step is to replace the abstract channels depicted in Figure 8-7 with well defined signal interfaces. In contrast to the functional architecture described in SDL where the connectors (SDL ports) between the SUD components support FIFO queues with no practical limit on the number of signals waiting for transmission, the refined communication must commit to finite queue lengths before mapping the functional architecture on hardware and software. Figure 8-8 outlines the refined MASCARA functional architecture in SOLAR.

The mapping from abstract to concrete communication implies the insertion of extra logic for controlling the communication though the refined channels e.g. N_wdlc_memory_mngt_FIFO. The state table at the lower part of the Figure 8-8 describes in SOLAR part of the behaviour of the WDLC block presented in Figure 8-4; at the upper part of the state table the communication ports and the signals, through which the specific state table communicates with the rest of the system, are defined.

Hardware/software assignment is the next step in the design of the MASCARA protocol. By exploring different hardware/software allocations, the designer is able to experiment with alternative mapping relations of the MASCARA functional architecture to hardware and software architectures, and decide in favor of the most appropriate i.e. the one that meets the constraints defined by the architecture rules of the SUD. In the case of MASCARA, the architectural rules were mostly related to performance constraints that had to be fulfilled in the final product. Consequently, the mapping of the functional architecture on hardware and software architectures had to take into account whether the architectural rules of the SUD were fulfilled or not.

In Figure 8-8, every block is characterized either as hardware block ([H]) or as software block ([S]). More specifically, the state tables describing *WDLC*, *SCH* and the *N_WDLC_memory_mngt_FIFO* controller are set to be implemented as software; *TB*, *MPDU* and the other *FIFO controllers* were set to be implemented as hardware.

The generation of a virtual prototype of the physical architecture relied on the translation of the SOLAR description of Figure 8-8 into executable code (C and VHDL). It is composed of a set of distributed modules, represented in VHDL for hardware elements and in C for software elements, communicating through communication modules from a library of components.

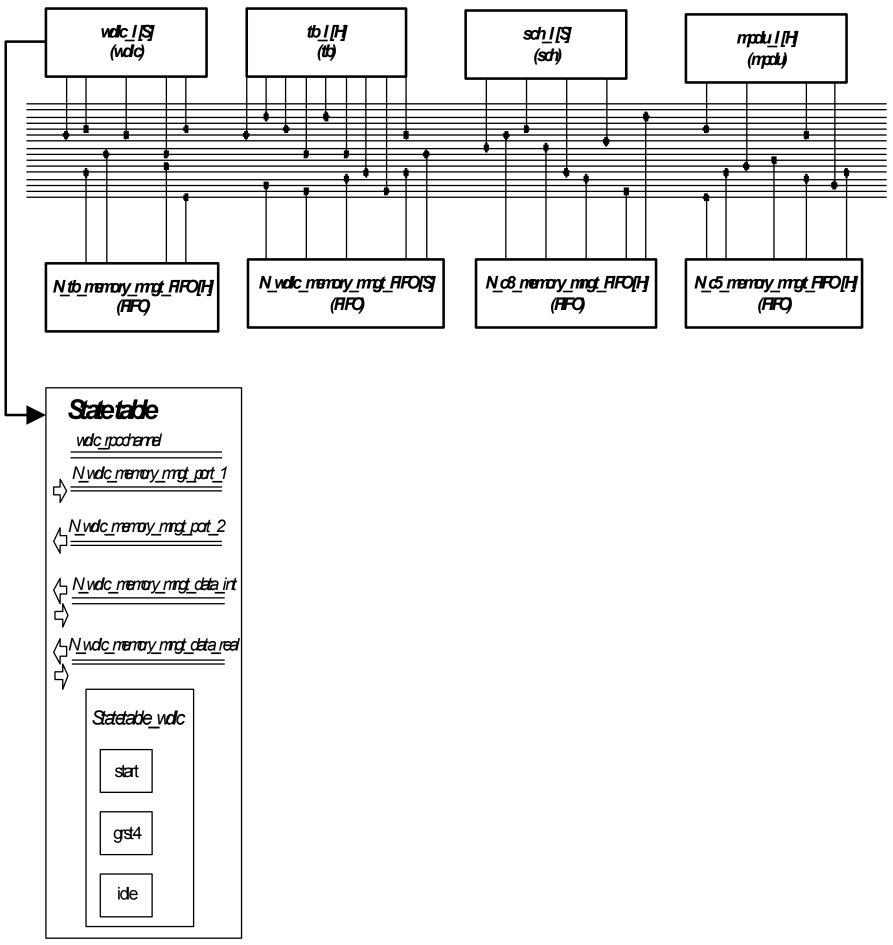


Figure 8-8. Refined MASCARA functional architecture in SOLAR.

As a result of the virtual prototype generation, extra blocks for communication between hardware and software, e.g. the SCH block in the VHDL simulator box in Figure 8-9, were required. The latter are produced automatically from SOLAR during the mapping on the physical architecture.

The virtual prototype presented in Figure 8-9 is a simulatable model of the SUD. It is the result of the mapping of the hardware and the software architectures on a physical architecture that represents the physical implementation of the SUD.

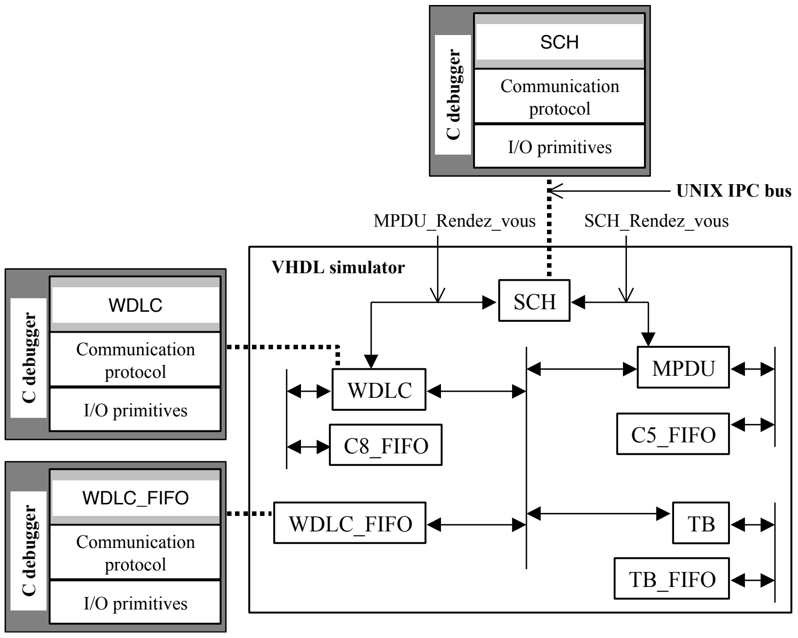


Figure 8-9. Refined virtual prototype of MASCARA protocol Physical architecture.

3. LINKAGE OF SUDM AND SDPM FOR MASCARA

There is an inherent relation between SDPM and SUDM. In the case of the MASCARA protocol, SUDM consists of set of models that describe the SUD at different levels of abstraction. SDP for MASCARA on the other hand, consists of set of steps that lead to a virtual prototype of the system. More specifically, SDP for MASCARA consists of the following stages:

- System specification,
- Functional decomposition,
- Functional description of subsystems,
- Communication refinement,
- Hardware/software partitioning and
- Virtual prototype of the final system.

As already explained, during each step of SDP the SUDM is represented through appropriate models of the SUD. Figure 8-10 describes the set of models composing the SUDM for MASCARA protocol and their correspondence to the SDP employed, while Table 8-1 outlines the relationship between SDP stages and SUD models for the MASCARA case study.

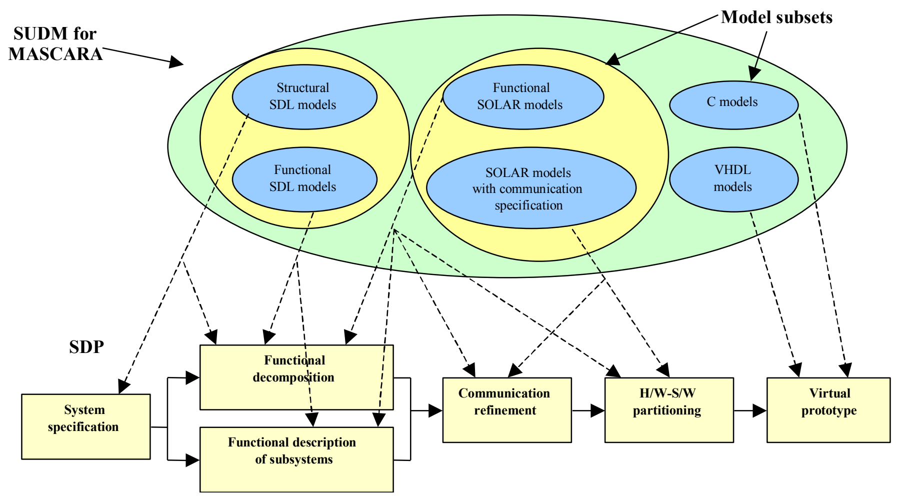


Figure 8-10. Linkage of SUD and SDP for MASCARA protocol.

Table 8-1. Correspondence between SDP stages, abstraction layers and SUD models for the development of the MASCARA protocol.

SDP stages	*Abstraction layer*	*SUD models involved*
Functional specification	L1	Functional model
Functional decomposition	L1, L2	Functional model
Functional description of subsystems	L3	Functional model
Communication refinement	L4	Functional model
HW/SW partitioning	L4	Functional model Hardware model Software model
Virtual prototype	L4	Physical model

4. SUMMARY

In the previous sections, the main concepts of SDCM metamodel were exemplified through its instantiation for the design of a real world application. The example presented relies on MASCARA MAC layer protocol. The goal of the previous sections was to provide a proof of concept that a generic conceptual metamodel like SDCM can be mapped on an existing system designs. Most of the concepts conveyed by SDCM and presented through MASCARA case study, can be instantiated in different context for the design of different kinds of electronic systems. As a complement to the current chapter, Annex A5 provides additional details on system design through the instantiation of SDCM in the context of a second example.

REFERENCES

[1] Bauchot F, Decrauzat S, Marmigere G, Merakos L, Passas N. MASCARA, a MAC Protocol for Wire-less ATM. Proceedings of ACTS Mobile Summit; 1996 November; Granada. Spain.

[2] Mikkonen J, Kruys J. The Magic WAND: A Wireless ATM Access System. Proceedings of ACTS Mobile Summit; 1996 November; Granada. Spain.

[3] Mitschele-Thiel, A, *Systems Engineering with SDL: Developing Performance-Critical Communication Systems*. John Wiley & Sons, 2001.

[4] Jerraya, A., O' Brien, K. SOLAR: an intermediate format for system-level modeling and synthesis. In Computer Aided Software/Hardware Engineering, ed. J. Rozenblit and K. Buchenrieder, IEEE Press, 1994.

[5] Voros N., Sanchez L., Alonso A., Birbas A., Jerraya A. Hardware/Software Co-design of Complex Embedded Systems: An approach using efficient process models, multiple formalism specification and validation via co-simulation. Journal of Design Automation for Embedded Systems 2003; 8:5-49

Annex A1

GLOSSARY

Klaus Kronlöf
Nokia Research Center, Helsinki, Finland

Abstract: This glossary defines the key terms used in this book.

Key words: System design, system design process, system under design, system IP, system architecture, platform.

A

Abstract data type

An object defined by a set of values and available operations on those values. It can be seen as a data type that is not precisely defined in implementation terms.

Abstract machine

A procedure for executing a set of instructions in some formal language, possibly also taking in input data and producing output. Such abstract machines are not intended to be constructed as hardware but are used in thought experiments about computability. Examples: Finite State Machine, Turing Machine [1].

In B, an abstract machine, is a formal specification of system or sub-system, encompassing variables, operations, variant and invariant properties. A B specification is a hierarchy of abstract machines and their refinements [2].

Abstraction level

A set of self-sufficient and consistent concepts used to specify a system.

A system specification can be described at different abstraction levels that are characterised by their specific data types and time model. Higher abstraction levels have compact descriptions that hide the details typical to lower levels [3].

Abstraction
A means to hide details (known or unknown) in order to describe an object. A higher-level of abstraction indicates that fewer details are included and vice versa. Can be divided into levels and between points of view. An unlimited number of abstraction levels and views can exist. A given type of abstraction – level or view – can be defined by a limited set of basic concepts providing that it is self-sufficient, consistent and has enough semantic power. It must be noted that abstraction level does not indicate accuracy, and abstraction point of view does not indicate a part of.

Activity
A single step in a phase of the design flow, where actors do transformations of the model of the system under design (SUD) typically from a higher level of abstraction to a lower one that is closer to implementation.

Actor
Actors perform activities of the design process. An actor has one or more roles, each of which defining a specific responsibility in the design process.

Allocation
The process (or results of) of distributing requirements, resources, or other entities among the components of a system or program [4].

Analysis
The part of the development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses on what to do, design focuses on how to do it.

Application (simulation) independent language
A formal language that describes design artefacts in the same way independently of the way they are used inside different applications (simulation is a particular application).

Architect
A person who is responsible of defining architectures and using them to construct things. The term is used in multiple domains and should be therefore qualified. In the domain of systems engineering, "system

architect", "software architect", etc. are particular roles played in the system design process.

Architectural configuration

Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure.

Architecture

In general, an architecture is a concept that defines the principles and rules used when constructing a thing. The term is used in all domains of design and engineering, and it should be therefore qualified to give it a more precise meaning. Relevant qualifiers in the domain of systems engineering include for example "functional architecture", "implementation architecture", "software architecture", and "hardware architecture".

Architecture language (design or description or definition)

In general, a language for describing architectures, architectural configurations and relations between different structural elements. It should allow architects to use necessary operations to change the design to establish a good mapping between the different facets, such as hardware and software. It shoud allow this work to be controlled by a set of architectural rules.

Architecture model

A set of descriptions (that may be written in an architecture language) that define an architecture or a configuration or a combination of an architecture and a compatible configuration (that obeys the rules defined by the architecture).

Architecture pattern

A proven generic solution to a class of system construction problems that can be used to derive a specific architectural configuration to a specific problem.

Architecture rule

Part of the architecture definition. An architecture rule constrains the available design choices and thereby eases the task of the designer.

Assertion

A statement in a behavioural description that expresses a condition that must be true at that point of execution. The purpose of inserting assertions to the description is to ease the verification task.

Asynchronous
A system that cannot be said synchronous is said asynchronous.

B

Behaviour
Influence of the input and internal variables of a system on its output variables. In the context of action semantics, the behaviour of a system can be defined as the set of its operations.

Behavioural model
A model that describes the dynamic internal evolution (operation) of the object of reference (system, subsystem, component) and its response to external stimuli.

C

Causality
The relation between a cause and its effect or between regularly correlated events or phenomena [5].
This concept is needed to explain the action semantics of behaviour. In a system model causality is expressed as an asymmetric relation between variables in a system, leading to a partial order (i.e., a set of maplets).

Causal chain
This concept is needed to explain the action semantics of behaviour. It is defined by a set of maplets connecting a totally ordered subset of variables. In a chain all variables but two appear exactly twice, once as fist element once as second element of a maplet.

Channel
A subsystem of the functional configuration that defines a connectivity mechanism between two or more functional entities (e.g., functions, services, operations).

Clock
A concept used in a behavioural model to represent time. In the action semantics it is defined as an application of the set of natural numbers into the time domain with the usual metric: the number of clock ticks (events), between two events is the difference of the associated naturals.

Communication

In the context of system design, transfer of data among functional units according to sets of rules governing data transmission and the coordination of the exchange [6].

Component

A component is any part of a design that may be instantiated one or more times and combined with other components to form a system or higher level component [7]. It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces [8]. In architectural modelling, a component is a subsystem capable of performing operations (services). See: Connector.

Component based design

A design paradigm where the system under design is composed of discrete components such that the design of one component depends only on the interface to other components, not on their internal design. Often standard-based interfaces are used to enable system development from pre-designed components.

Composition

The aggregation of two or more entities to form a new entity. Composition does not in general preserve proven properties.

Conceptual design

A system design activity concerned with specifying the logical aspects of the system organization, its processes, and the flow of information through the system. [9]

Conceptual model

The Conceptual Model is a meta-model that serves as a reference and gives a global view and perspective. Any design model should be a specialisation of the conceptual model.

Concern

Those stakeholders' interests which pertain to the development, operation, or other key characteristics of the system (e.g. performance, reliability, security, evolvability, distribution, …).

Concurrency

The occurrence of two or more non-ordered activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. In the case of action semantics, non-ordered events (i.e., associated to non-comparable tags) are said concurrent.

Concurrent

In general, two or more activities of a behavioural model are said concurrent if their execution order is not defined. Within the context of action semantics, two non-ordered events in the same interval are said concurrent.

Configuration

The actual construction of the system, typically hierarchical decomposition to subsystems. The configuration must conform to the chosen architecture of the system. In the simplest case a configuration can be represented by a fixed graph that defines how the interfaces of subsystems are connected.

Connector

In architectural modeling, a connector is a subsystem of the configuration whose purpose is communication. A connector has two or more interfaces. Connectors are architectural entities used to specify interactions between any components and properties attached to those interactions. See: Component.

Constraint

A limitation or implied requirement which constrains the design solution or implementation of the systems engineering process, is not changeable by the performing activity, and is generally non allocable [10]. It is a boundary condition within which the developer must remain while allocating performance requirements and synthesizing system elements [11].

Constraint language

Specific, usually formal, language to express constraints. The Object Constraint Language (OCL) of the UML is an example [8].

Continuous time

Marked by uninterrupted extension in space, time, or sequence [5]. In the context of action semantics, continuous time is modelled as a mapping of the set of reals into the time domain.

Contemporary events

Two ordered sets of events are contemporary if there exists one event in each set that is lower (higher) than one event of the other set.

Control

A means or device to regulate a process or sequence of events [12].

Control graph

Graphical notation used to describe how the control flows between objects of a system. It is a graph whose nodes are control primitives. Contol nodes have inputs coming from state variables and from external events. Control primitives generate outputs that trigger system's operations.

Coverage (of verification)

The degree to which a given verification procedure examines the state space of the system under design.

Cycle

In the context of the action semantics of behaviour, a cycle is a chain in which a maplet joining the last variable to the first one is added.

Cycle-based

Cycle-based simulators are trigerred by a general clock that associates new events to each variable at each new clock cycle. Although they have to consider non-significant events they are generally faster than event driven simulators because they have no event queue to maintain.

D

Data flow (model)

Computational model which can execute completely on the basis of the availability of data to its operations [3].

Data flow graph

A graphical notation used to describe a data flow model. A system is specified by a directed graph in which nodes perform computation and edges carry totally ordered sequences of events (represented by tokens).

Decomposition

The principle of breaking a problem to smaller pieces with potentially simpler solutions or at least better understanding. In system design decomposition means breaking the system to subsystems. Subsystems can be (hierarchically) decomposed further as needed.

Derivative design

Modifications, changes, replacements, enhancements etc. of a product or an integration platform in order to obtain a new instance, usually for a different target product [13].

Derived requirements

Requirements that are not explicitly stated in the customer requirements, but are inferred (1) from contextual requirements (e.g., applicable standards, laws, policies, common practices, and management decisions), or (2) from requirements needed to specify a product component. Derived requirements can also arise during analysis and design of components of the product or system [14].

Design

(1) The process of defining, selecting, and describing solutions to requirements in terms of products and processes, or (2) the product of the process of designing that describes the solution (either conceptual, preliminary, or detailed) of the system, system elements or system end-items [12].

Design artefact

A physical piece of information that is used or produced by the system design process [8].

Design flow

Decomposition of the design process into a number of phases, e.g. requirements definition, specification, architecture design, mapping, software design, hardware design, and integration. Phases are composed of activities, where actors do transformations of the system model of the system under design (SUD) from a higher level of abstraction to a lower one.

Design flow model

A model of the design flow considered as a system. A design flow model can be represented by patterns or activity diagrams.

Design pattern

A proven generalised solution to a generalised problem that can be used to derive a specific solution to a specific problem.

Design rule

Piece of knowledge applicable to a class of designs that defines a constraint on how design shall be performed or on design content.

Design space

The set of all possible implementations of a system (whether realistic or not). The design space is bounded in practice by the capabilities of the technology, the cost of a solution and the imagination of the system architect.

Design task

Activity of architect or designer which adds information and specialisation to the system under design model (SUDM).

Design-for-functionality (paradigm)

Design paradigm emphasising detailed functions and their relationships as first descriptions of a system.

Design-for-reuse (paradigm)

Design paradigm emphasising component design with reusability as one of the main concerns.

Design-with-reuse (paradigm)

Design paradigm emphasising reuse of existing components reusability as one of the main concerns.

Determinism

Occurrences in nature that are causally determined by preceding events or natural laws [5].

Discrete event

The events triggering the system obey a discrete timing metric. If the events have a common time base, then they are totally ordered. Note that different subsets of events can have different discrete time bases, leading to a partial order despite the system is discrete.

Discrete time

Property of a time metric in which between any two events there are a finite number of events. A necessary condition is that the set of discrete events is order-isomorphic to a subset of natural numbers (which implies that it is totally ordered). This condition may be sufficient if the time domain is infinite but not when it is finite. A sufficient condition for a set of events to be discrete is that there exists a least lower bound of the distance between two events.

Domain

An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area [8].

Domain understanding

Knowledge and experience related to the (problem) domain.

Duality (principle of architecture)

An interface of a connector can only connect to an interface of a component and vice versa. That is, an interface of a connector (component) cannot connect to an interface of another connector (component). See: Component, Connector, Architecture, Configuration.

E

Encapsulation

A development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module [4].

Environment

Another system which constitutes a closed system when composed with the system under design.

Estimation

Estimation of performance: Getting data on the physical behaviour of a system or device based on timing, power consumption, heat dissipation, signal propagation, etc. [3].

Evaluation

The process of determining whether an item or activity meets specified criteria [15].

Event

An event is a change in the state of the system. In certain models of computation it can be represented as a pair consisting of a tag and a value.

Event driven

Quality of a system whose behavior is triggered by events.

Evolution

Modification of system properties to meet the changing needs/requirements of stakeholders. Since evolution (i.e., maintenance) is the single costliest development activity, system evolvability becomes a key aspect of architecture-based development.

Executable model

Model that can be translated to a form executable on a computer.

F

Facet

The subset of system under design (SUD) defined by a domain, for example by an engineering discipline. In a N-dimensional design space, a facet of the SUD is its intersection with any surface of dimension less than N that represents a characterisation along a set of criteria.

Finite state machine

An abstract machine consisting of a set of states (including the initial state), a set of input events, a set of output events, and a state transition function. The function takes the current state and an input event and returns the new set of output events and the next state. Some states may be designated as "terminal states". The state machine can also be viewed as a function which maps an ordered sequence of input events into a corresponding sequence of (sets of) output events [1].

FSM can be non-deterministic either because the environment is non-deterministic (inputs order is unknown) or because they can transit randomly to several states (behaviour is incompletely specified). Making a non-deterministic FSM deterministic often results in an exponential growth of the set of states.

FSMs can be organized hierachically and can communicate (e.g., State-charts).

Function (mathematical)

A relation that defines an unique mapping from one set onto another. That is, $F \subseteq A \times B$ is a function, if there is one and only one element in F for each $a \in A$. A function is denoted by $F: A \rightarrow B$. If $(a, b) \in F: A \rightarrow B$ then we write $F(a) = b$, meaning the application of F to a yields b.

Function (service)

A task, action or activity that must be accomplished to achieve a desired outcome, or to provide a desired capability [10].

Functional

Designed or developed chiefly from the point of view of use and/or performing or able to perform a regular function or service [5].

Functional analysis

Examination of a defined function to identify all the sub-functions necessary to the accomplishment of that function; identification of functional relationships and interfaces (internal and external) and capturing these in a functional architecture; and flow down of upper-level performance requirements and assignment of these requirements to lower-level sub-functions [14]. (See also "functional architecture.")

Functional architecture

The architecture that guides the (functional) decomposition of the functionality of the system under design (SUD). See: Functional configuration.

Functional configuration

The hierarchical arrangement of functions, their internal and external (external to the aggregation itself) functional interfaces and external physical interfaces, their respective functional and performance requirements, and design constraints [14]. See: Functional architecture.

Functional model

A functional model describes a system or component by means of functions. These functions define a mapping of a subset of the set of interface variables (IV) of the system (or component) onto another sub-set. If the union of these two subsets is not equal to the set of IV, the functional model is partial. If the two subsets are always disjoint, then the first can be qualified as inputs and the second as outputs. A behavioural model can be functional, if time appears as annotations (clocks, delays...) to a functional model. A functional model can be behavioural, if time is added to the variables used by the functions.

Functionality
The functionality of a system is its customers use model or its marketing product requirements. The functionality is part of the specification document.

G

Generics
The set of parameters that make an entity to be a family of less generic entities.

Global time
The situation which occurs when all events of a system behaviour are given a time stamp in the same totally ordered time domain.

Global variable
A variable that is accessible to two or more processes of a concurrent behavioural description.

H

Hardware (architecture or design)
All or part of the physical components of an information processing system [6].

Hardware architecture
A special case of an architecture where all the subsystems are pure hardware.

Hardware configuration
The composition of hardware from its parts. This is guided by the hardware architecture.

Hardware/software co-design
The simultaneous and interactive hardware and software design of parts of a system. This implies the ability to modify the hardware-software partitioning at any time during the design.

Hardware/software partitioning

Step of design process that decides what parts of a system will be allocated to hardware implementation and which parts to software implementation, respectively.

Heterogeneity

Property of a system composed of sets of different kinds subsystems.

Hierarchy

A means to represent decomposition and composition. A system can be decomposed into subsystems presented at a lower level of a hierarchy. Several subsystems can be composed into one system presented on a higher level of a hierarchy. This also applies to different views of a system: any can be decomposed into hierarchies, and an open methodological issue is the correspondence between one hierarchy of, say, a model, and that of an architecture. The hierarchy is not corresponding to the level of detail, but only to the level of command assigned to pieces inside a composition.

I

Idiom

An idiom is an expression peculiar to certain description technique or design culture. In the particular case of patterns it designates a specific implementation of a pattern with a choice of tools and parameters.

Implementation

In general, 1) a definition of how something is constructed or computed or 2) the process of creating an implementation. In the domain of systems engineering, an implementation is the end product of a system design process. The final implementation is the status of the system as delivered to its users.

Implementation mapping

A mapping from the set of functional subsystems (services, functions, operations and channels of the functional configuration) to the set of subsystems (components and connectors). It defines where different parts of the system functionality are implemented. The mapping must obey consistency rules that among other things ensure that the necessary communications can be implemented.

Incompleteness (of specification)

A specification that leaves the details of some system features ambiguous is said incomplete.

Indeterminism

Possiblility for a system to go from one particular state to several successor states with no choice criteria.

Informal specification

Specification expressed by techniques, e.g. natural language document, that do not belong to the class of formal techniques, or specification that has not been approved.

Inheritance

The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior [8].

Instantiation

Instantation is the mechanism that brings to the provision of an instance (a concrete evidence of) in support of the description of a specific target. The target of electronic system design is typically a specification, which goes through a sequence of steps - a "procedure" – that can be partially or totally supported by a computer (while an instance can be provided also for a theory, concept, claim, or the like). A single specification can enter p different procedures, which in turn can provide i different instances. Thus, p times i instances can be derived from one single specification. As in spoken language, an instance signifies the case or occurrence of anything, thus expressing the event of tangible innovation.

Instruction accurate

A modelling level in which the granularity of all events correspond to processor instruction fetch

Intellectual property (e.g. out-source, pre-defined, hard/soft/firm)

(1) Law. property that results from original creative thought, as patents, copyright material, and trademarks. (2) Reusable (encapsulated, documented) information about a system and how to develop it. (3) The original idea from which springs an added value.

Intellectual property instance

An intellectual property instance (IPI) is an implementation of the intellectual property (IP) in the form of software code, or hardware device, or whatever mixture of them. The IP rights are put forth on such instances.

Interface

An element of a system that defines a communication capability with other systems. A system can communicate with other systems only through its interfaces. A system may have zero or more interfaces. A system with zero interfaces is defined as a closed system. An interface is defined by various characteristics pertaining to the services, physical interconnections, signal exchanges, and other characteristics, as appropriate. It may include not only the static types and sizes of ports, but also the definition of the entire protocol necessary to communicate.

Interface based design

Interface-based design is the design flow that moves design from an interconnected set of communicating processes with clearly defined and separately captured interface protocols (usually intended to test conceptual behavior) to interconnected realized components in the final system. At this design point, interactions conform to the interface specifications captured at all the levels of abstraction. At the higher levels of abstraction, the set of operations or tasks required to perform an application are initially linked by "ideal" channels through which information is sent and received as needed, without concern for conflicting resource requests or synchronization. At this stage, the architectural design may be concerned only with functionality or with communication protocols. As this design is refined, common communication resources are specified, control protocols administered, and sharing of functional units identified. The common issues associated with system design become visible, and the design moves from that of the ideal to the real. The separate specification of the interfaces allows the design process to proceed fully and concurrently with the minimum of design interference between teams working on separate components [7].

Interface model

A component model that describes the operation of a component with respect to its surrounding environment. The external port-structure, functional, and timing details of the interface are provided to show how the component exchanges information with its environment. An interface model contains no details about the internal structure, function, data values, or timing other than that necessary to accurately model the external interface behavior. External data values are usually not modeled unless they represent control information. An interface model may describe interface details of a component at any level of abstraction. The terms "bus functional" and "interface behavioral" have also been used to refer to an interface model and are considered synonyms. The more general interface model name is preferred to the anachronistic term "bus functional" [7].

Invariant
A subset of the properties of the system that will remain as permanent properties, to be satisfied forever. They will constitute the invariant.

L

Layer
The organization of classifiers or packages at the same level of abstraction. A layer represents a horizontal slice through an architecture, whereas a partition represents a vertical slice [8].

Loop
In the context of the action semantics of behaviour, a set of operations derived from a cycle is a loop

M

Maplet
This concept is needed to explain the action semantics of behaviour. It is defined by an ordered pair of two variables belonging to the specification of the system. It denotes the influence exercised by the first variable on the other variable.

Meta-model
A model that defines the language for expressing a model [8].

Method
A formal, well-documented approach for accomplishing a task, activity, or process step governed by decision rules to provide a description of the form or representation of the outputs [10]. Within an engineering discipline, a method describes a way to conduct a process. In the context of systems engineering, a method is defined as consisting of [16]:

1. An underlying model
2. A language

3. Defined steps and ordering of these steps
4. Guidance for Applying the method

Methodology
The term methodology generally means a study of methods. A methodology is a coherent set of theories, methods, techniques and/or principles used to analyse and/or develop methods for a particular domain, for example the domain of systems engineering [16].

Metric
A quantitative measure of the degree to which a system, component or process possesses a given attribute [4].

Model of communication
The mathematical instruments necessary to build a model of the communication between two or more systems.

Model of computation
The mathematical instruments necessary to build a model of what the system can compute. Examples are: Turing machines, TLA, CHOCS, Lambda calculus, Pi calculus (also a model of communication), CCS, Petri nets, Linear logic, etc.

Module
A unit of design description that is discrete and identifiable [4].

N

Non-functional property
Attribute of a system or component that does not contribute to functionality or distinguish itself through functionality.

O

Object
An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class [8].

Objective function

In the context of synthesis process, objective function measures the "goodness" of the result.

Operation

A service that can be requested from an object to affect behavior. An operation has a signature, which may restrict the actual parameters that are possible. An operation can be a hierachy of functions and other operations [8]. In the action semantics of behaviour an operation is defined by the association of an event to a function (or multi-function).

P

Parallel

In general, two or more activities of a behavioural description are said parallel if they occur at the same time. In the context of the action semantics of behaviour, two sequences that are not in the same thread are parallel if they have contemporary events. In some models, the events of parallel sequences can be interleaved.

Parallelism

In general, a property of a behavioural description that has parallel activities. In the context of the action semantics, a property of having two or more parallel sequences.

Parameter

A variable that is given a constant value for a specified application [4]. Design parameters are qualitative, quantitative, physical, and functional value characteristics that are inputs to the design process, for use in design tradeoffs, risk analyses, and development of a system that is responsive to system requirements [17].

Partitioning

The process of identifying parts in a whole, for example subsystems in a system.

Performance

An observable and measurable characteristic or attribute of a system. It represents, in general, a quality attribute that can be used to establish how well a functional or non-functional requirement is met. There are multiple performance characteristics that are of interest for a system or component.

Each characteristic could be called a "performance index" [18]. Note that a performance is in general a non-functional characteristic.

Performance evaluation

Performance evaluation is the process of checking the system properties with respect to given quality attributes [10]. See performance.

Performance model

A particular view of system under design (SUD). The SUD may include several performance models each of which are used to analyse a given quality attribute. See performance [7].

Physical system

The tangible target of system design. Manufacturing is responsible of transforming SUD to physical system and reproducing them.

Platform

A platform is a package of technology to deliver a predictable functionality, i.e. a platform can be relied upon to behave (physical) as predicted (modeled). The deployment of a platform does not require detailed knowledge of the process within (beneath) it and it is completely supported in its context of use. A platform is a special kind of sub-system, in that its functionality-in-context is a finite but large degree of flexibility, such that it may be used to implement a number of different things. Examples of platform technologies are ARM Instruction Set, Gate-Array Logic, Micro-Controler, C/C++. Platforms can be hierarchical (or layered) such that one platform technology uses another beneath it. Example: ARM CPU implemented on Gate-Array.

Depending of the owner and purpose, platforms can be classified as product platforms and integration platforms. Product platforms are used by system houses as an isolation layer between their product family design and the implementation technology. Integration platforms are used by implementation technology vendors and component vendors to facilitate their customers' product development and their own technology protection. In both cases the isolation provided by the platform facilitates technology refreshment, i.e. changing the underlying implementation technology with minimal impact to the product design [13].

Platform based design
A layered design methodology that utilizes platforms. The goal of platform based design is to achieve high productivity through large scale planned design reuse[13].

Pragma
An element in a design description that direct the operation of specific tools.

Process
A set of interrelated activities which transform inputs to outputs [19].

Proof
Logical process that establishes the truth of a statement.
Property (static, dynamic, other)
A predicate on variables of the system that can be true or false.

Protocol
A set of semantic and syntactic rules that must be followed to perform communication within a communication system (i.e. connector).

Q

Queue
An ordered set of entities (elements) where the removal and insertion of elements is restricted to the first and last element, respectively.

R

Refinement (of specification)
A refinement is the action performed by the system designer on a specification in order to introduce his knowledge and experience to produce a more detailed specification, closer to an implementation. The design choices of the designer, operated through the refinement process, reduce progressively the undeterminism of the specification.

Relation
An aspect or quality (as resemblance) that connects two or more things or parts as being or belonging or working together or as being of the same kind [5]. Mathematically a relation is defined by a subset of the product of two

sets. If (a, b) $\in R \subseteq A \times B$ then we write a R b, meaning a is related to b by R.

Relationship
A formalised statement of interdependence.

Repository (of system intellectual property)
Organised and maintained facility for storing and retrieving system intellectual property.

Requirement
Requirements are statements of fact or assumptions that define the expectations of the system in terms of mission or objectives, environment, constraints, and measures of effectiveness. Requirements should not prescribe or imply implementation details unless they are specifically features which are required.

Requirements analysis
The determination of product-specific performance and functional characteristics based on analyses of: customer needs, expectations, and constraints; operational concept; projected utilization environments for people, products, and processes; and measures of effectiveness [14].

Requirements formalization
A particular kind of analysis that transfoms all relevant requirements into logical properties suitable for reasoning or readable by various formal tools.

Resource
An entity that is utilized or consumed during the execution of a process. Resources may include diverse entities such as personnel, facilities, capital equipment, tools, and utilities such as power, water, fuel and communication infrastructures. Resources may be reusable, renewable or consumable [19].

Response (of simulation)
In the context of simulation, response is the result of aplying a specific stimulus in a simulation process.

Reusability
Reusability is the degree to which a module, component, or system may be used again in other instances for which it may or may not have been specifically intended. Reuse occurs across several dimensions, such as life-

cycle phases, at the packaging levels, and across model-years. Reuse occurs at various distinct levels, such as [7]:

1. reuse of components (hardware parts) or modules (software object-code), also called direct implementation
2. reuse of hardware logic or software source-code recast in new technology or integrated with other logic or code
3. reuse of architecture through re-implementation of functional block concept with new partitioning, integration, or technologies

Reuse (of system design know-how), system intellectual property reuse

The action of using again a part of a design developed previously. The levels of reuse are:

1. Using again to produce valuable results, like for a tool: the value doesn't come from the object itself but it is exploited in a field different from the one the object belongs to (a synthesis tool applied to different libraries).
2. Using again to produce valuable new objects: the value comes and it is exploited in the same field the object belongs to (different semi-custom ASICs built with the same library).
3. Using again to extend the reach of the object: the value comes from the extension of the exploitation field the object belongs to (same algorithm applied in a more general case).
4. Using again the metaphor to produce different objects: the value comes from a new field, not existing before (exploitation in new different markets).

Role

An actor plays a role in an activity. A role is a specific behavior of an actor participating in a particular context. In the context of system design, a role is defined by a set of actions/activities of the system design process.

S

Semantics

The relationships of symbols or groups of symbols to their meanings in a given language [4].

Sequence (of operations)

In the action semantics of behaviour a sequence is defined as a set of totally ordered operations derived from at least one causality chain.

Service

A task, action or activity that must be accomplished to achieve a desired outcome, or to provide a desired capability.

Simulation

A mathematical model that emulates a system, usually using a standard simulation procedure or computer language, to predict the value of a parameter or set of parameters for a given system.

Simulation mode

Discrete event simulators (Verilog, VHDL) use a global event queue in which events are chronologically sorted.

Continuous simulators are generally based on an equation solver. In case of linear approximation they solve a system of linear equations, more generally they solve a system of algebro-differential equations which is often stiff and requires implicit numerical methods.

Cycle-based simulators are trigerred by a general clock that associates new events to each variable at each new clock cycle. Although they have to consider non significant events they are generally faster than event driven simulators because they have no event queue to maintain.

Instruction accurate (or instruction based) simulators emulate the execution of a processor-under-design instruction code. They are a special case of event driven simulators in which the event list is the list of instructions (the program) to be simulated.

Mixed-mode simulators combine different simulation modes to simulate multi-level models.

Simulation scenario

A set of stimuli and expected responses for a simulation process.

Simultaneous

Several events treated as a single one while keeping their consequences distinct. For example the changes in the values of two variables are associated with the same tag, or several variable assignements take place in the same transition, etc.

Software (architecture or design)

All or part of the programs, procedures, rules, and associated documentation of an information processing system [6].

Software architecture

A special case of an architecture. It is defined with the terms of the software world and applies to the software parts of the system only.

Software configuration

The decomposition of software to its subsystems. This is guided by the software architecture. It contains all software routines and services for meeting a system's objective. Software application, operating system and communication protocols can describe layers of a software architecture

Specialisation

In object modelling techniques, specialisation means derivation: the derived type or class is the base type or class, but with additional (or modified) properties. The derived (sub-) type or class is a specialisation of (extends) the base (super) type or class.

Specification (formal, informal)

A specification is the set of the information related to a system to be designed (variables, functionality, properties) which is available at the beginning of a design process. A specification can be informal or formal. Extracting a formal specification from an informal one is the first task of the system designer to be done in co-operation with the informal specification owner. A specification can be complete, incomplete or redundant, but the formal specification must be consistent, non-ambiguous and machine process-able. It can be written using a single notation or multiple inter-linked formalism. The result of the design process can be either an implementation of the designed system or, in case of a multi-step design process, a more detailed specification to be used by the next design step.

Stakeholder

An interested party having a right, share or claim in the system or in its possession of characteristics that meet that party's needs and/or expectations [19]. It is a person or party who has a (financial) interest in the successful outcome of the identified activity. In the context of the design process model a stakeholder is a special case of a role.

State (of SUD)

The state of the system under design (SUD) is a condition that determines the set of all sequences of events that can occur in the course of execution from that point on. In essence, the state of the SUD encapsulates all that needs to be known of its execution history in order to reason about its possible future behaviour. The set of possible states of the SUD forms its state space. The state space can be continuous or discrete, finite or infinite. At given instant of time, the SUD is always in one of its states. Changes in the state of the SUD are triggered by events, either external or internal.

As a practical example, the state space of a SW object is the cartesian product of the sets of the possible values of its variables. This is finite and discrete, provided that all the variables are finite and discrete (which they are in any real computer program). Also the state space of any digital HW component is finite and discrete. In the case of functional components it is possible, that the state space is continuous and/or infinite.

Within the context of action semantics, the state of the SUD depends on the state variables and only on them. This can be elaborated as follows: Each variable is associated with a set of values. The the state of the SUD at given moment (of time) depends on the values of variables at that moment. Hence the state space is the cartesian product of all variables' value sets.

Structural model

A structural model represents a component or system in terms of interconnections of its constituent components. The components can in turn be described structurally then creating a hierarchy. The hierarchy can be related to, for example, the organisation of a set of software modules or to the physical organisation of a specific implementation. A structural model can represent either an abstract network or be isomorphic to the physical structure of a specific implementation. In any case it specifies the components interconnection topology.

Structure

Structure is something arranged in a definite pattern of organization, the organization of parts as dominated by the general character of the whole and the aggregate of elements of an entity in their relationships to each other [7]. It can be defined as the "Architecture as implemented".

Subsystem

A system that is part of a larger system.

Synchronicity

An axiom which allows two or more variables of the system to have changes in their values such that these changes are mapped on the same element in some associated partial order. When we use an ordered set of tags to represent time and when we associate each change in the value of any variable with a tag, then synchronicity allows several value changes to be associated with the same tag.

Synchronous

Property of operations triggered by simultaneous events.

System

A combination of interacting elements organised to achieve one or more stated purposes. A whole that cannot be divided into independent parts without losing its essential characteristics as a whole. It is complete in its context and can be used without reference to its internal processes.

System design

A process of defining the hardware and software architecture, components, modules, interfaces and data for a system to satisfy specified requirements [9].

System design conceptual model

A meta-model comprised of models of system under design and system design process.

System design description

A desricption of the system under design (SUD). It should include all the views of SUD needed by the development and reproduction organisation.

System design process

A description of how the product development of system under design (SUD) in a specific organization is arranged. Its core is a design flow that describes the decomposition of the design process into a number of phases. Design process is associated with activities for measuring the quality (in broad sense) of the SUD. If the quality criteria, i.e. properties of the SUD are not met at a phase, iteration may occur to one of the previous phases. A true process is regular and repeatable, and is traversed many times without change. The value of establishing a process is that it can be optimised by feedback and continuous improvement.

System designer

An engineer who analyzes requirements, performance and functions of the total system including hardware and software, partitions this system into elements showing requirements and functions allocated to these elements.

System (design) intellectual property

As to design processes, system intellectual property can be:

1. Components of the process
2. Know-how of methods (analysis, synthesis, etc.) encapsulated in rules and guidelines (possibly implemented by tools and design patterns)

3. Know-how of design styles (modelling, verification, etc.) encapsulated in checkers
4. Know-how of use of tools encapsulated in scripts.

   As to design artefacts, system intellectual property can be:
1. Application components
2. System architecture
3. SW architecture
4. HW architecture
5. Knowledge of an invariant set of properties
6. System components (= sub-systems) that are in the stable core area of the domain, i.e. there is a high probability for reuse
7. Out-source intellectual property that are developed and maintained by 3rd party on behalf of system house
8. Pre-defined star intellectual property.

System intellectual property reuse

A methodology associated with discipline and means to facilitate use again of design artefacts and design knowledge at system-level, i.e. during early phases of product development (e.g. requirements definition, specification, architecture design, mapping). The reuse methodology requires establishing design-for-reuse and design-with-reuse processes.

System specification language

An informal or formal language able to capture the available knowledge about properties and behavior of a system.

System Specification and Design Language

A language to describe a system under design (SUD) at required levels of abstraction providing required views to the SUD in order to allow actors to perform transformation, validation and analysis tasks that are specific to the level of abstraction and to the design process applied. Specifically, it should allow the description of system in terms of external and internal views to the modeling domains of structure, connectivity and behavior.

System under design (SUD)

The set of models representing the conceptual object of system design. It includes all intermediary models produced during the system design process.

T

Technical requirement

Properties (or attributes) of products or services to be acquired or developed [14].

Thread

A single path of execution through a program, a dynamic model, or some other representation of control flow. Also, a stereotype for the implementation of an active object as a lightweight process [8].

In the context of a multi-tasking operating system, a thread is a separable process within the current task that shares the same memory space as the other threads in the task (as opposed to tasks themselves that all have the full memory space available to them).

In the context of the action semantics of behaviour, a thread is a set of causal chains with common variables. It is a directed acyclic graph (DAG) and it represents the dependency graph of variables.

Time base

A set chosen to be order-isomorphic to the set of events . It can be an enumerated, a discrete, a countable or a continuous (real) set, with or without partial or total order, with or without metric.

Time metric

In the case of timed events, the time base may be a metric space. The events are said to have a time metric. Delays can be specified, one pass from chronology (total order) to chronometry (measure).

Tool interface

The facility offered by a tool for interaction by users or by other tools.

Traceability

The capability to track system requirements from a system function to all elements of the system which, collectively or individually, perform the function; an element of the system to all functions which it performs: a specific requirement of the source analysis or contractual constraint which originated the requirement.

Transformation

A design activity that transforms a design artefact to another related design artefact. Typically a design transformation refines a design artefact from one level of abstraction to a lower level that is closer to implementation.

U

Use case (or scenario, or model)

The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system and which demonstrate predictable response [8]. A set of Use-Cases is used to demonstrate gross functionality in a system under design (SUD).

User requirements

A set of requirements that describe services and related properties and constraints that user(s) want a system to deliver.

V

Validation (strategy)

Confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled. Validation in a system life cycle context is the set of activities ensuring and gaining confidence that the system is able to accomplish its intended use, goals and objectives [19].

Variable

A symbol representing a quantity that may assume any one of a set of values [5].

Verification

The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [4]. Verification can be by observation, by experimental use, by simulation of a system's model or by proof that a formal representation of the system under design (SUD) implies its formal specification.

View

A projection of a model which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective [8]. In the case of the system under design (SUD), a view is the result of looking at the SUD from the perspective of a stakeholder, in particular from the various perspectives of intending deployers. To be sucessfully used (deployed) a SUD must have all of the required views available. Formally a view is the result of applying a viewpoint to a specific SUD.

Viewpoint

A viewpoint is a way of looking at something, for example the system under design (SUD). A viewpoint defines how a view is obtained from a specific SUD. Formally it is a function that yields the models of interest (a view) when applied to SUD. In the design process a role may define several viewpoints.

W

Workflow

A workflow is a set of inter-related activities performed by a set of actors on a set of artefacts.

REFERENCES

[1] Howe D. (Editor), FOLDOC, The Free On-line Dictionary of Computing. http://www.foldoc.org/

[2] Abrial J.-R. The B Book: Assigning Programs to Meaning. Cambridge University Press, 1996.

[3] EDAA System Design Technology Roadmap. Available from http://www.edaa.com/

[4] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology

[5] Merriam-Webster's Collegiate® Dictionary, Tenth Edition. ISBN: 0-87779-709-9

[6] ISO/IEC 2382-1:1993, Information technology Vocabulary Part 1: Fundamental terms.

[7] VSI AllianceTM System Level Design Model Taxonomy, Version 2.1 (SLD 2 2.1), July 2001, Available from http://www.vsi.org.

[8] OMG Unified Modeling Language Specification, Version 1.4, September 2001, Available from http://www.omg.org/

[9] ISO/IEC 2382-20:1990, Information technology -- Vocabulary -- Part 20: System development

[10] IEEE Std. 1220-1998: Standard for Application and Management of the Systems Engineering Process

[11] MIL-STD-499B, Systems Engineering (Draft Military Standard)

[12] INCOSE Systems Engineering Handbook V2.0, Available from http://www.incose.org/

[13] Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A., Todd, L. Surviving the SOC revolution: A Guide to Platform-Based Design. Kluwer Academic Publishers, 1999, ISBN 0-7923-8679-5.

[14] CMMISM for Systems Engineering/Software Engineering/Integrated Product and Process Development, Version 1.02, CMU/SEI-2000-TR-031, ESC-TR-2000-096, November 2000

[15 DOD-STD-2167A

[16] Kronlöf K. (Editor), Method Integration: Concepts and Case Studies. John Wiley & Sons, 1993, ISBN 0-471-93555-7.

[17 MIL-STD-1388-1A,

[18] Thomé B. (Editor), Systems Engineering: Principles and Practice of Computer-Based Systems Engineering. John Wiley & Sons, 1993

[19] ISO/IEC CD 15288 CD3, System Life Cycle Processes.

Annex A2

ACTION SEMANTICS

Jean Mermet
ECSI, Grenoble, France

Abstract: This annex relates together a series of definitions of the main terms used in the SYDIC-Telecom glossary and the SDCM chapter, with regard to the behavior of a system. This is not yet another formal semantics, but mostly an attempt to introduce in a concise way more internal consistency and more precise meaning to several usual modelling concepts.

Keywords: System behavior, model of computation

1. INTRODUCTION

This annex defines and relates the basic concepts that occur in the different aspects of the behavior of a system. There is a link to the section dealing with abstraction layers in the SDCM chapter and the idea is that of a top down constructive approach. However the subject is difficult and addressed by an immense literature. There is no claim here to produce a new formal semantic representation of system behavioral concepts. The goal is to introduce concepts in the order which is required by their mutual dependencies and to give illustrations of their definitions through simple examples.

2. TOP-DOWN INTRODUCTION OF CONCEPTS

Binary relation:

At the beginning of the identification of a system there are variables (there can be different sets of variables defining different types) and some of them are linked by the assumption of mutual influence. This is a symmetric relation. When made explicit it can take the form of a property because what is described has properties not behavior. The functions, if any, appear in equations. It can be the static equilibrium of a system in its environment without stimuli (this corresponds to abstraction level 1).

Example 1: Let x_1, $x_{2,\dots}$, $x_n \in N_{10}$ be the set of 10 digit integers, representing for example phone numbers. Then $x_1 \longleftrightarrow x_2$ is the binary relation establishing that any two numbers could be connected.

Example 2: Let u_1, $u_2,\dots,u_n \in U$ be a set of phone users. If any user has a single phone number, there is a binary relation $u_i \longleftrightarrow x_i$ associating a 10 digit natural to each user (bijection in this case).

Property:

A property on some variables is either their belonging to some sets or their appearance in a predicate that must remain always true to guarantee the property.

Example 3: For any x_1, x_2, $x_3 \in N_{10}$ (Typing),
$\neg(((x1 \longleftrightarrow x2)\&(x2 \longleftrightarrow x3))V((x2 \longleftrightarrow x3)\&(x1 \longleftrightarrow x3))V((x1 \longleftrightarrow x2)\&(x1 \longleftrightarrow x3)))$
"The connect relation cannot connect any given number to 2 other numbers at the same time".

Equations:

An equation of n variables e_n is a n-uple of variables $e_n \in \{V, V, .., V\}$ tied together by a property

Example 4: $i_k \in V$ being the currents at an electric node:
$i_1 + i_2 + \dots + i_n = 0$ (Ohm law)

Example 5: u_1, $u_2,\dots,u_j \in Uj$, $Uj \subset U$ being the subset of users connected to a phone station Sj, we have property:
Card $\{ (x_l \longleftrightarrow x_k) = \text{true}\} \leq \mu$, μ being the capacity of the station.
But we can also have:
$\max(\mu) = 1/2$ Card $\{Uj\}$ (if the capacity allows all users to talk to each other at the same time, while satisfying the previous property).

Invariant:

A subset of the properties will remain as permanent properties of the system, to be satisfied forever. They will constitute the invariant.

Maplet:

A maplet is an ordered couple of 2 different variables belonging to the specification of the system. It denotes the influence exercised by the first variable on the other variable.

Example 6: u_i ——> u_j, user u_i wants to call user u_j
Example 7: st_l ——> st_k, station l connects itself to station k
Example 8: u_i ——> st_l, user u_i is identified by his station l
Example 9: st_k ——>u_j station k connects itself to its customer u_j

Causal Chain:

A chain is a set of maplets connecting a subset of variables in such a way that all variables but 2 appear exactly twice, once as fist element and once as second element of a maplet.

Example 10: c1 = (v1, v4), c2 = (v4, v5), c3 = (v5, v7), c = (v7, v8)

Example 11: u_i ——> st_l,, st_l ——>st_k, st_k ——> u_j, user u_i wants to call user u_j, this is done through his station and u_j's station

Cycle:

A cycle is a chain with a maplet added to join the last variable to the first.

Thread of maplets:

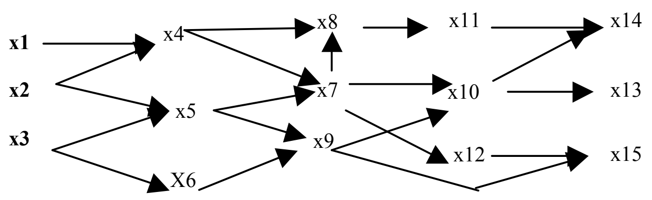
A thread of maplets is a set of causal chains with common variables. It is a directed acyclic graph (DAG) the variable dependency graph. It is also the skeleton of a multiple pre--functions.

Example 12: Let us consider the following 6 chains

x_1 —> x_4—> x_7 —> x_8 —> x_{11} —> x_{14}
x_2 —> x_4 —> x_8
x_7 —> x_{12} —> x_{15}

$x_2 \longrightarrow x_5 \longrightarrow x_7 \longrightarrow x_{10} \longrightarrow x_{14}$
$x_3 \longrightarrow x_5 \longrightarrow x_9 \longrightarrow x_{10} \longrightarrow x_{13}$
$x_3 \longrightarrow x_6 \longrightarrow x_9 \longrightarrow x_{15}$

The corresponding DAG is:



Causality:

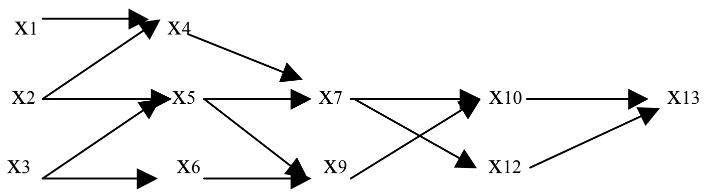
A set of maplets (sub-set of a Cartesian product {V, V}) establishes an asymmetric relation that pre-figures causality.

Pre-function:

Each maplet, because it has a single target variable, can be considered as a not yet defined function and can be refined into a function. Similarly, several maplets which have the same target, can be considered as yet-to-be-defined functions of the cartesian product of the sets of their origin variables on the set of the target variable. Let's call them pre-functions. Contrarily to functions, for which each argument has a single image, pre-functions may be defined with the same set of origin variables and different target variables. Each will be refined as *different* functions of the same variables.

Finally, in the same way as a function of functions is a function, a thread of maplets that has a single terminal variable is a pre-function.

Example 13:



The set of pre-functions x_4 (x_1, x_2), x_5 (x_2, x_3), etc. define x_{13} as a pre-function of x_1, x_2, x_3.

Example 14: Let's assume now that the users belonging to a set U are mobile and that there is a network of fixed switching stations ST supporting phone connections. There is a need for a pre-function "*attach* (u_i,l_i) —>st_k" able to attach user u_i, when he is at location l_i, to station st_k (we shall assume that some device can provide l_i). Then station st_k will connect itself to station st_i (where u_i is registered as a customer). Station st_i will connect itself to station st_j where user u_j is registered, thanks to the number x_j sent by u_i.. Then *attach* (u_j, l_j) —> st_l will allow the direct connection of st_k to st_l. As soon a the connection is set a pre- function "*charge bill*(l_i, l_j) —> au_i" starts measuring the duration of the connection to charge the account of u_i.

$$\begin{tikzcd}
u_i \arrow[r] \arrow[dr] & x_j \arrow[r] & st_i \arrow[r] & st_k \arrow[r] & st_j \arrow[r] & st_l \arrow[r] \arrow[ddr] & u_j \\
& x_i \arrow[ur] & & & & & \\
& & l_i \arrow[u, to=1-3] \arrow[rrrr, bend right] & & l_j \arrow[uur, to=1-6] \arrow[rr] & & au_i
\end{tikzcd}$$

Functions:

Functions, in the mathematical sense are applications between sets. Any element of the first set has a single image in the second set. Pre-functions are applications between "things" or "objects"("users" for example above). They can have multiple interpretations depending on the multiple attributes and properties of the considered things or objects. They will be refined progressively into functions between the sets of attributes of these things or objects. These refinements will also give implementable types.to the values of the attributes

System function:

The overall function of a system can be defined as the influence of its input variables on its output variables. The composition of all maplets from the inputs down to the outputs constitute an abstraction of this function.

Event:

An event can be defined as the occurence of the change of the value of a variable or a set of variables. In the former case it is practical to characterise events as it is proposed by Lee and Sangiovanni as a pair {tag,value} but, of course, other definitions are possible. In the case for example of a system described as a set of predicates and its changes described as substitutions operated on variables by a predicate transformer, the couple of tags {before,after} is enough to describe the system as a FSM in which substitutions make the system transit from the state of its variables *before* to the state of its variable *after* substitutions, and so on iteratively.

At this level causality only exist.

In the {tag,value} paradigm, each variable is associated an element of a set of tags. However the fact that two variables can have the same tag raises the basic question whether or not 2 events can be simultaneous. The positive answer is the *synchronicity axiom* and must be considered as a modelling approximation that can become legitimate already in early phases of refinement. But initially tags will be assumed different. The synchronicity hypothesis will be a particular interpretation of concurrency (see below).

If synchronicity is assumed, then concurrent events that are not tight by any causality, (they don't belong to the same ordered set), can be simultaneous.

The set of used tags must be given initially a partial order isomorphic to the causality of the maplets (variable dependencies).

Example 15:

$$\begin{tikzcd}
u_i \arrow[r, "t_1"] \arrow[dr, "t_1"'] & x_j \arrow[r, "t_2"] & st_i \arrow[r, "t_4"] & st_k \arrow[r, "t_5"] & st_j \arrow[r, "t_6"] & st_l \arrow[r, "t_8"] \arrow[ddr, "t_8"] & u_j \\
 & x_i \arrow[ur, "t_3"] & & & & & \\
 & & l_i \arrow[uu, "t_3"] \arrow[rrrr, bend right, "t_3"'] & & l_j \arrow[uur, "t_7"] \arrow[rr, "t_7"] & & au_i
\end{tikzcd}$$

Let us chose an ordered set of 10 tags {t1,t2,t3,t4,t5,t6,t7,t8, t9,t10}. In the graph the tags are associated to the variable on the left of the arrow. The same tag may appear on different edges, because a given event (for example new value for (u1,t1)) can be seen in different nodes (xi,xj). But it in the case of t3 for example, we are assuming that 2 events (changes of (xi,t3), (li,t3)) have the same tag assuming in this case the synchronicity.

The {tag,value} paradigm is naturally associated to a trace semantics. The way such semantics relate to others like, the functional (also called denotational) semantics, or an operational, semantics, or the weakest precondition predicate transformer semantics, is a difficult problem.

Event chain:

An event chain is the association of a totally ordered set of tags to a chain of maplets. We can define this chain of events as an event associated to the target by causality.

Example 16: From above

$$\begin{tikzcd}
u_i \arrow[r, "t_1"] & x_j \arrow[r, "t_2"] & st_i \arrow[r, "t_4"] & st_k \arrow[r, "t_5"] & st_j \arrow[r, "t_6"] & st_l \arrow[r, "t_8"] & u_j
\end{tikzcd}$$

In the transition system interpretation, an event chain is an ordered set of transitions between the states of the system (like in state-charts). Inside a single transition there can be several causal chains that are time-free(causality only). In the {tag,value} representation, either each variable of the chain should be given the same tag,which is contradictory to their ordering, or 2 kinds of topologies should be defined on 2 sets of tags. the time free chain can rely on partial order, but the sequence of transition chains contain many cycle due to the loops in the underlying automaton. Furthermore, the question of associating tags to the variables in the guards is also opened.

Operations:

An operation is the association of an event to a pre-function.
In the {tag,value} representation, an operation using functions of functions can be interpreted as a composite event.

Example 17:
Attach ($st_j(st_k(st_i(x_j(u_i))))$, l_i) —> st_l

$$\begin{tikzcd}
u_i \arrow[r, "t_1"] & x_j \arrow[r, "t_2"] & st_i \arrow[r, "t_4"] & st_k \arrow[r, "t_5"] & st_j \arrow[r, "t_6"] & st_l \\
 & & & & l_i \arrow[ur, "t_7"'] &
\end{tikzcd}$$

Concurrency:

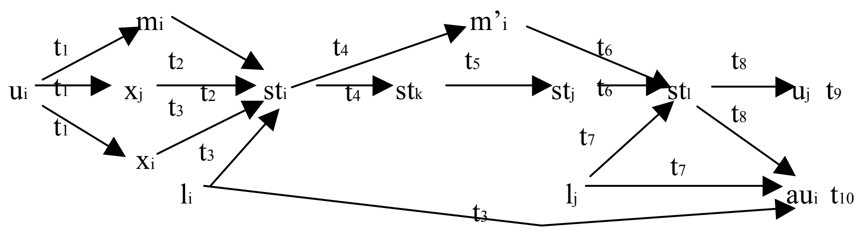
Non ordered events (i.e associated to non comparable tags) are concurrent (example above: {x1, x2, x3}, {x4, x5, x6}, but also {x1, x5, x6}).

Example 18: In the example of the telephone network, let us consider one subset **DU** of couples of users {ui,uj} ∈ DU ⊂ U*U such that any user appear in no more than one couple. In this case all operations *connect* (u_i,u_j) applied on elements of this subset, can be considered as concurrent events.

Ordered operations:

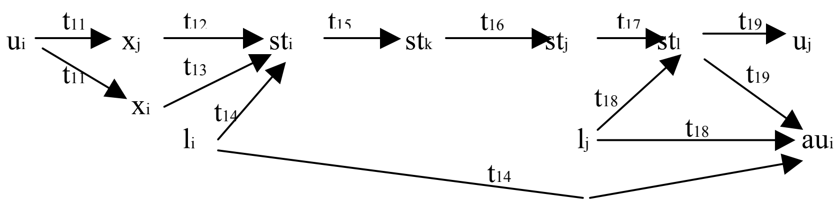
Two operations are ordered if they are associated to ordered events (if their operands have tags, all of tags of the first set of variables -the operands- of the first operation must be lower than at least one tag of the operands of the second operation).

Example 19: We can now refine the previous example by making the assumption that the system can have 3 ordered operations: *connect, communicate, disconnect*. We add to the specification that, after being connected, u_i can send a message m_i to u_j, and, that after receiving the message u_j will hang up and st_l will stop charging au_i

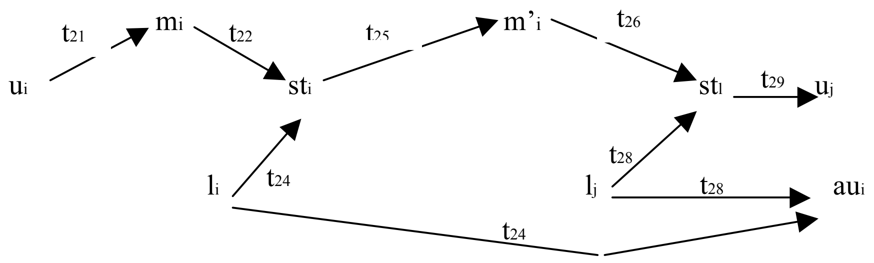


With the chosen ordered set of 8 tags $\{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8\}$ we would be obliged to create more simultaneous events in order to respect the causal order (see above (m_i,t_2), (m'_i,t_6)..), but the 3 operations (*events* in event-B) induce a new order and duplicate some state variables with the *before/after* paradigm.

Connect:



Communicate:



m’$_i$ can be the encoded state of m$_i$, the counter of duration of au$_i$ has been triggered, li and lj remain monitored by their *attach* function (because both u$_i$ and u$_j$ are moving)

Disconnect

$$\begin{tikzcd} sti & st_k \arrow[l, "t_{35}"'] & st_j \arrow[l, "t_{34}"'] & st_l \arrow[l, "t"'] \arrow[dr, "t_{32}"] & u_j \arrow[l, "t_{31}"'] \\ & & & & au_i \end{tikzcd}$$

We see that the causality chain (st$_i$, st$_k$, st$_j$, st$_l$, u$_j$) is inverted without creating loops (the *before/after* values of these variables creates memory).

Sequence:

A set of ordered operations linked by at least one causality chain is a sequence.

Example 20: It is easy to see in example above that each operation can be further refined into a sequence of operations, replacing the causal maplets.

Thread of operations:

A thread of operations is a set of sequences with common events

Contemporary events:

Two ordered sets of events are contemporary if there exist one event in each set that is lower (higher) than one event of the other set.

Parallel sequences:

Two sequences are parallel if they have contemporary events. In some models, the events of parallel sequences can be interleaved (There are many models of parallelism and the fairness problem results only from interleaving).

Consecutive sequences:

Two sequences, not linked by any causality, are consecutive if they are not parallel and if there is at least one event of the first comparable to one event of the other.

Remarks.

- 2 consecutive sequences between which a causality link is established become a single sequence;
- 2 concurrent events can always be assumed simultaneous if the axiom of synchronicity is accepted (i.e. this axiom is not accepted in event-B).

Loop:

A set of operations linked by a cycle is a loop

Behavior:

The behavior of a system can be defined as the set of its operations.

FSMs, control graphs, data-flow graphs:

As soon as events are defined in the specification of a system, the question of transition between two consecutive (comparable) events is raised.
Several mathematical concepts have been proposed to describe the reaction of the system to events and the generation of new events as a consequence of this reaction. Finite state machine was the first to appear, which introduced the notion of *internal state***.** Control graphs like data flow graphs represent the distributed production of events as firing of various primitive and propagation of tokens.

There is a variety of such graphs and they are not always easy to compare but they use the same concepts. The main difference between data-flow and control flow is that operations are performed at the nodes of the graph in the former and are triggered by the node and performed elsewhere in the latter. The control graph propagates the conditions enabling events and the data flow graph the change of variables producing events.

The plain deterministic FSM deals with a totally ordered sub-set of events. This is why combinations of FSMs have been imagined to deal with real concurrent systems. Hierarchical Communicating FSMs**,** represented by State-Charts is the most famous. It must be noticed that, when establishing communication between independent FSMs one decreases the non-determinism of the system by making independent subsets of ordered events comparable. The initial partial order moves progressively to a total order.

3. CONCLUSION

No notion of **time** has been necessary so far. All previous concepts can work with an asynchronous partially ordered set of events. However it is a natural support to thinking to decompose the behavior of the system into 2 alternative phases: during the 1st phase the system receives events from the environment, then during the second phase it responds by new events to the environment. This is an other avatar of the{before,after} paradigm.

The notion of an *abstract clock* follows. It must be noticed that this is enough to refine the system down to programs or to cycle based hardware models. But this clock will become a more concrete local counter that will count phases, most likely represented as rising or falling edges of the clock.

All concurrent events occurring during each clock phase are then assumed to occur on one of this edge, (remember that simultaneous is a particular case of concurrent). In the tag-value paradigm, this assumption defines an isomorphism between the set of tags and the set of integers.

The system at this point has become a timed system. Concepts like data flow or control flow graphs can in their turn become timed graphs. FSMs are synchronized by a clock.

One more level of refinement will consist in mapping the set of tags onto a metric space. Then time will be measured, delays, intervals will appear. The notion of discrete event will have to be defined: intuitively, it is a space where only a finite number of events can occur between any 2 events.

Mapping the set of tags onto a subset of the real numbers will allow to model continuous systems.

Annex A3

LANGUAGE ANALYSIS FRAMEWORK

Patrizia Cavalloro
Italtel SpA, Milan, Italy

Abstract: It is not always easy to provide appropriate support for expressing concepts at system design level. This annex proposes the use of a questionnaire in order to understand if a selected language is useful for system specification and design. Examples of the use of the questionnaire are provided.

Keywords: Language analysis, language assessment guidelines.

1. INTRODUCTION

The purpose of this annex is to propose guidelines for the analysis and selection of System Specification and Design Language(s) in order to provide appropriate support for expressing concepts needed in a given application area. The main idea is that in principle and in practice it is not so easy to understand if a chosen language will be able to express all the concepts needed at system level: this section gathers the important aspects of language analysis, and helps the system designer to focus on the real important points.

Of course, this is a proposal based on the results of the SYDIC-Telecom project. Other approaches could be envisaged, and also this approach could be improved, but nevertheless it is believed that this is a good starting point.

The goal is not to provide a semantic and/or a syntax definition for a new language, but to propose a way of analyzing existing languages to verify their usefulness for system level specification and design.

2. LANGUAGE ASSESSMENT GUIDELINES

One of the goals of the SYDIC-Telecom project was to suggest guidelines for the assessment of System Specification and Design Languages. In order to provide such guidelines, a questionnaire has been invented in which system design concepts were listed, and questions were asked for each identified concept.

The scope of the questionnaire is to understand if the language under analysis is able to express concepts that are considered important in system design. Language and concepts classifications are described in Chapter 5. The definitions of all concepts are available in the SYDIC-Telecom Glossary, in Annex A1 of this book.

2.1 The Questionnaire

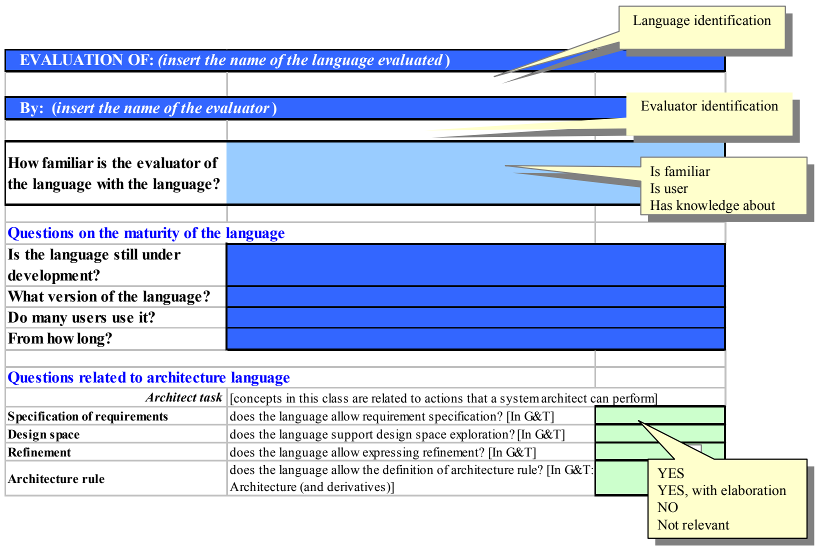
The questionnaire is divided in the following sections:

1. Identification: expert and language identification.
2. Questions on the familiarity of the evaluator with the language.
3. Questions on the maturity of the language.
4. Questions related to Architecture Language.
5. Questions related to System Specification and Modeling Languages.
6. Questions on Design Command Languages.
7. About reuse.
8. Examples.
9. Notes.

Sections 1 to 3 are used to identify the language under consideration, and the evaluator identity. Questions on the familiarity of the evaluator with the language can be used as a weight factor in the final evaluation of the language. Questions on the maturity of the language are important in order to understand if the language itself is stable or improvements and modifications are still to be provided. An excerpt of the questionnaire sections 1 to 3 is shown in Figure A3-1.

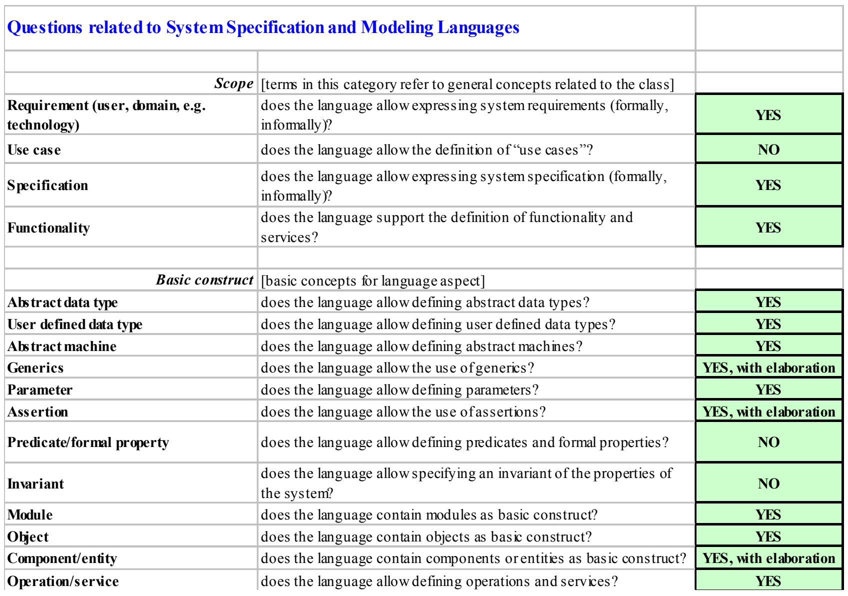
Sections 4 to 6 refer to the concept and language classification proposed in Chapter 5. For each concept, the evaluator is asked to choose among four available answers in order to show how good is the language in expressing it. Possible answers are:

- *YES*: the expression of the concept in the language is straightforward
- *YES, with elaboration*: the concept can be express by the language, but not with basic constructs
- *NO*: the language cannot express the concept
- *Not relevant*: the concept has no meaning for the language



EVALUATION OF: ***(insert the name of the language evaluated)***		
By: *(insert the name of the evaluator)*		
How familiar is the evaluator of the language with the language?		
Questions on the maturity of the language		
Is the language still under development?		
What version of the language?		
Do many users use it?		
From how long?		
Questions related to architecture language		
Architect task	[concepts in this class are related to actions that a system architect can perform]	
Specification of requirements	does the language allow requirement specification? [In G&T]	
Design space	does the language support design space exploration? [In G&T]	
Refinement	does the language allow expressing refinement? [In G&T]	
Architecture rule	does the language allow the definition of architecture rule? [In G&T: Architecture (and derivatives)]	

Figure A3-1. Sections 1 to 4 of the questionnaire.

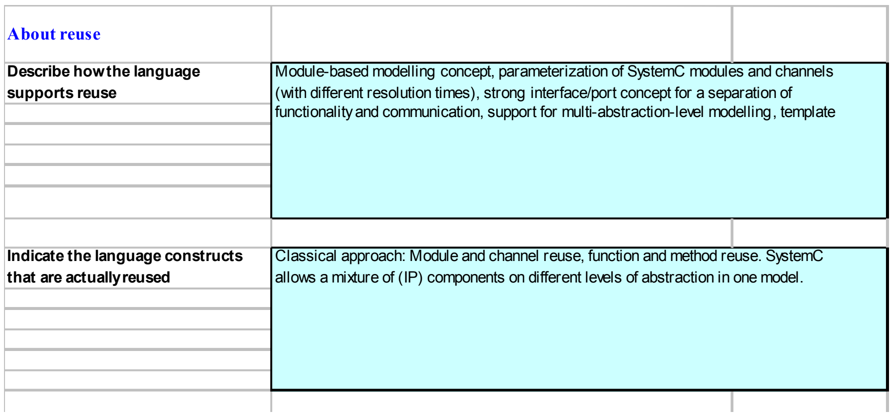


Questions related to System Specification and Modeling Languages		
Scope	[terms in this category refer to general concepts related to the class]	
Requirement (user, domain, e.g. technology)	does the language allow expressing system requirements (formally, informally)?	**YES**
Use case	does the language allow the definition of "use cases"?	**NO**
Specification	does the language allow expressing system specification (formally, informally)?	**YES**
Functionality	does the language support the definition of functionality and services?	**YES**
Basic construct	[basic concepts for language aspect]	
Abstract data type	does the language allow defining abstract data types?	**YES**
User defined data type	does the language allow defining user defined data types?	**YES**
Abstract machine	does the language allow defining abstract machines?	**YES**
Generics	does the language allow the use of generics?	**YES, with elaboration**
Parameter	does the language allow defining parameters?	**YES**
Assertion	does the language allow the use of assertions?	**YES, with elaboration**
Predicate/formal property	does the language allow defining predicates and formal properties?	**NO**
Invariant	does the language allow specifying an invariant of the properties of the system?	**NO**
Module	does the language contain modules as basic construct?	**YES**
Object	does the language contain objects as basic construct?	**YES**
Component/entity	does the language contain components or entities as basic construct?	**YES, with elaboration**
Operation/service	does the language allow defining operations and services?	**YES**

Figure A3-2. Example of assigning answers.

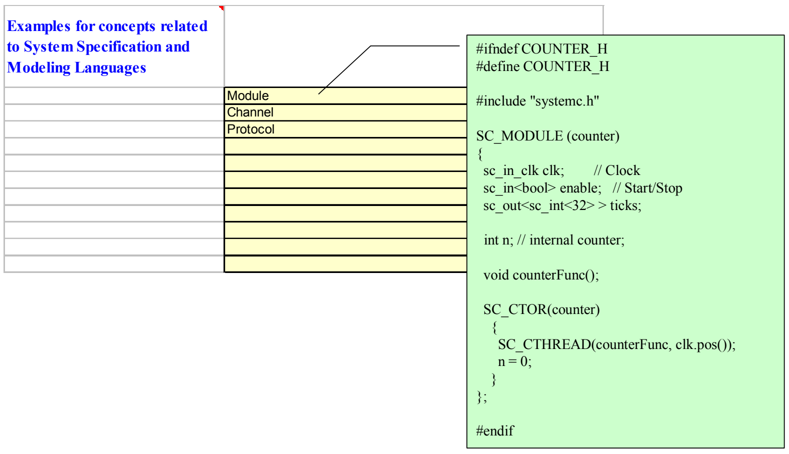
Figure A3-2 shows an example of the compilation of a subset of questions in the questionnaire, concerning the language SystemC.

Section 7 refers to the reuse aspects of the language (Figure A3-3). The evaluator is asked to describe how the language supports reuse and to indicate the language constructs that are actually reused.



About reuse	
Describe how the language supports reuse	Module-based modelling concept, parameterization of SystemC modules and channels (with different resolution times), strong interface/port concept for a separation of functionality and communication, support for multi-abstraction-level modelling, template
Indicate the language constructs that are actually reused	Classical approach: Module and channel reuse, function and method reuse. SystemC allows a mixture of (IP) components on different levels of abstraction in one model.

Figure A3-3. Section 7 of the questionnaire.



Examples for concepts related to System Specification and Modeling Languages	
	Module
	Channel
	Protocol

```
#ifndef COUNTER_H
#define COUNTER_H

#include "systemc.h"

SC_MODULE (counter)
{
  sc_in_clk clk;        // Clock
  sc_in<bool> enable;   // Start/Stop
  sc_out<sc_int<32> > ticks;

  int n; // internal counter;

  void counterFunc();

  SC_CTOR(counter)
    {
      SC_CTHREAD(counterFunc, clk.pos());
      n = 0;
    }
};

#endif
```

Figure A3-4. Section 8 of the questionnaire.

In Section 8. (Figure A3-4) the evaluator can explain through examples how the language can express some particular concepts.

Finally, the evaluator can use section 9 to comment particular aspects linked to the evaluation of the language.

2.2 How to Use It

The first thing to do is to understand what is the purpose of the analysis of the language that is going to be performed. The questionnaire could be tailored to the specific purposes e.g. adding questions related to important concepts in the domain of interest etc.

Nevertheless, the first step towards the analysis of a language is to fill the questionnaire. It could be filled completely, or just in parts, depending on the analysis the user is interested in, as we will see in next chapter.

The second step relates to the elaboration of the answers given in the questionnaire. It is convenient to convert answers to the questions on concepts into numbers, with the following correspondence:

YES → 3,
YES, with elaboration → 1
NO → 0
Not relevant → 0

If for some reason multiple evaluations of the same language are available, answers should been combined into one by taking average of scores and rounding it to the closest integer.

In the third step, the individual scores of sub-categories for each language are summed up and then normalized by dividing by the sum of the maximum possible scores for corresponding sub-categories.

The result will be a numerical table, and numbers will be taken into account in the further elaboration of the language analysis.

3. GUIDELINES

This section will provide some guidelines of potential ways of using the analysis results.

Once the questionnaire has been filled, several different uses of it can be envisaged, depending on the interest of the language evaluator.

We can envisage the following uses:

1. Analysis of a language in general

2. Analysis of a language with respect to a particular domain application
3. Comparison of a language with respect to available languages (library of existing language analyses)
4. Analysis of combination of languages with respect to a particular domain.

3.1 Analysis of a Language

When analysing a language, three things may be of interest: the general support of a subcategory, the support of single concepts and the support of a particular level of abstraction.

Following the steps indicated in section 2.2, it can be observed that in step three, when the result of the normalization is 1, the concepts in the sub-category are fully supported by language constructs. On the contrary, when the result is 0, there is no support for the concepts of the category.

Analysis in deeper details can be extended to individual concepts, in order to highlight one hand the concepts with good support and on the other hand the concepts with little or no support. In this case the analysis can be done just after step two.

If the analysis of the language is performed in order to understand if the language itself can be used at a particular level of abstraction, then the concepts appearing in Figure 4-11 (SUD layers and concepts) of Chapter 4 are to be considered. First, the level of abstraction has to be identified, then concepts in that level have to be selected in the questionnaire and then answers must be analyzed.

For example, in order to understand if a language can be used for the design description at abstraction level L2, the following concepts must have a good support: Constraint, Stimuli, Interface, Connector, Component, Behavior, Function, Invariant, Type: they obviously belong to different subcategories. And cover all the design aspects at that level.

3.2 Analysis of a Language with Respect to a Particular Domain Application

A method to understand if a language can be useful at system level could be that the user selects a subset of key concepts in its domain among the ones available in the questionnaire, looks at the evaluation made with the questionnaire and uses the language if also those concepts get good evaluations (in addition to basic concepts).

It should be noted that in addition to the domain concepts, the language evaluation might be complicated by other factors, external to the application itself, for example the availability of tool support, and the company policy.

3.3 Comparison of a Language with Respect to Available Languages

In some cases it might be of interest to analyse more than one language, and compare them in order to understand which of them better support e.g. the specification phase.

Evaluations of different languages can be performed through the questionnaire and stored in a database

It should be noted the fact that when making direct comparison, one should be careful as to the skills, expertise, background and history of the evaluators in order to get objective evaluations.

3.4 Analysis of Combination of Languages with Respect to a Particular Domain

Before analyzing combination of languages, some considerations should be done.

For large projects, it is very common to find a mixture of languages used in a system design. Usually, this is because it may happen that description to be reused (existing system libraries, organizational reuse libraries, IP) is in a language other than the primary language, or else a particular language is required to accomplish a particular function for some special reason. In this case, the primary language chosen will be the "glue" that will bind all the system together, and it should provide good support for this. Other code to be developed should probably also be in the primary language. However, if another language better suits the development needs, then both languages could be chosen, each for its specific purpose, provided that the languages can be readily interfaced.

For example, the combination of UML and SystemC shows to be a very good combination in covering a large number of the concepts appearing in our questionnaire. UML allows graphical specification, visualization, construction and documentation of systems, and can be useful in the first design phases. SystemC and standard C++ development tools can be then used to quickly simulate, to validate and optimise the design, to explore various algorithms, and to provide an executable specification of the system.

The formal aspect of the specification is still mixed in this language combination, and the use of B in the correct design phase could improve the overall design.

Note that interfacing two languages should consider two factors. If the calling language has a built-in ability to do language interfaces, the language mix will probably produce more reliable results. Also, if an IP product provides interfaces for the "glue" language, the interfacing is smoother and more straightforward than if such bindings must be developed.

The questionnaire we have created faces this aspect through some concepts appearing in the Design Command Languages class.

In particular, concepts in the *IP Reuse and Retrieve* subcategory are strictly related with the ability of the language to interface with IP repository, while the concept Tool interface in subcategory *Design Elaboration* concerns the interface aspect of a language towards other languages (and tools).

Mixing languages is not as straightforward as using just one language. While there is always good reason to reuse proven components, including IP, regardless of the primary language used, the use of two or more development languages together can often be more trouble than it is worth. However, a practice gaining in support is the use of an interface standard to facilitate the communication between heterogeneous systems. Using such a standard would make mixing languages much easier and more predictable.

4. EXAMPLE OF ASSESSMENT

4.1 Analysis of a Language: SDL

An exercise developed during the project concerned the analysis of some existing languages. This section contains some results related to the analysis of the Specification and Description Language (SDL) in order to understand if the language could be considered an Architecture language.

In this section, observations about language support for some of the sub-categories of the Architecture language class described in Chapter 5 of this book is summarised. The individual scores of sub-categories for SDL are summed up and then normalised by dividing by the sum of the maximum possible scores for corresponding sub-categories. When the result is 1, the concepts in the sub-category are fully supported by language constructs.

The result per sub-category is depicted in Figure A3-5. SDL shows reasonably good support for Architect primitive operations and for Modelling capabilities. The sub-category Architect discipline is well supported by SDL. The sub-category Primitive architecture element gets full scores SDL, while the sub-category Complex architecture element gets low scores.

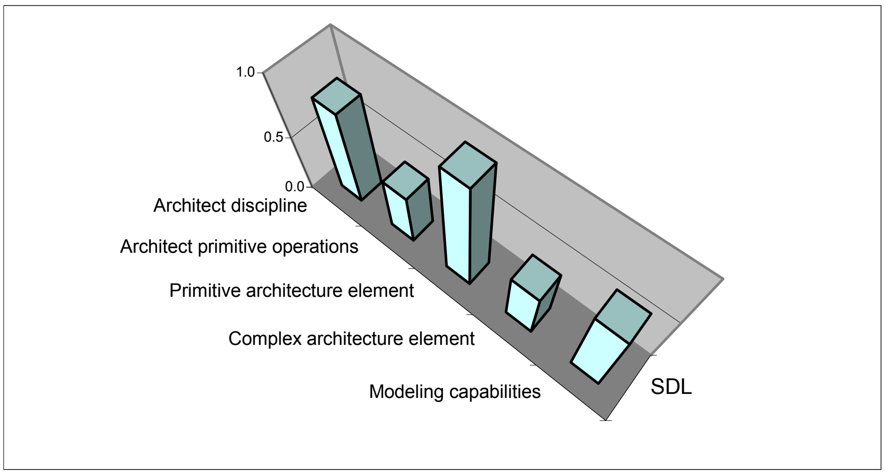


Figure A3-5. Summary of support for sub-categories of architecture concepts.

In the rest of this section, support for individual concepts of sub-categories are presented by highlighting on one hand the concepts with good support and on the other hand the concepts with little or no support.

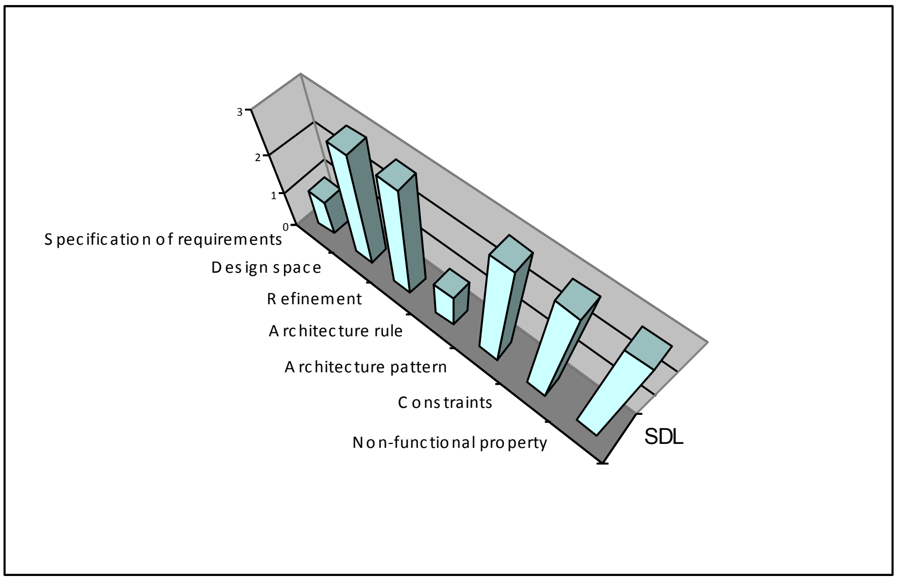


Figure A3-6. Support for concepts of Architect discipline.

The Architect discipline sub-category groups concepts that are related to the activities of the design architect. The results per concept are depicted in Figure A3-6. They show that for the subcategory Architect discipline SDL fully supports most of the concepts except for only little Specification of requirements and Architecture rule.

The Architect primitive operation sub-category groups concepts that are related to operations performed on the System under Design. The results are depicted in Figure A3-7. SDL supports fully Encapsulate, Instantiate and Connect, but little or none for others.

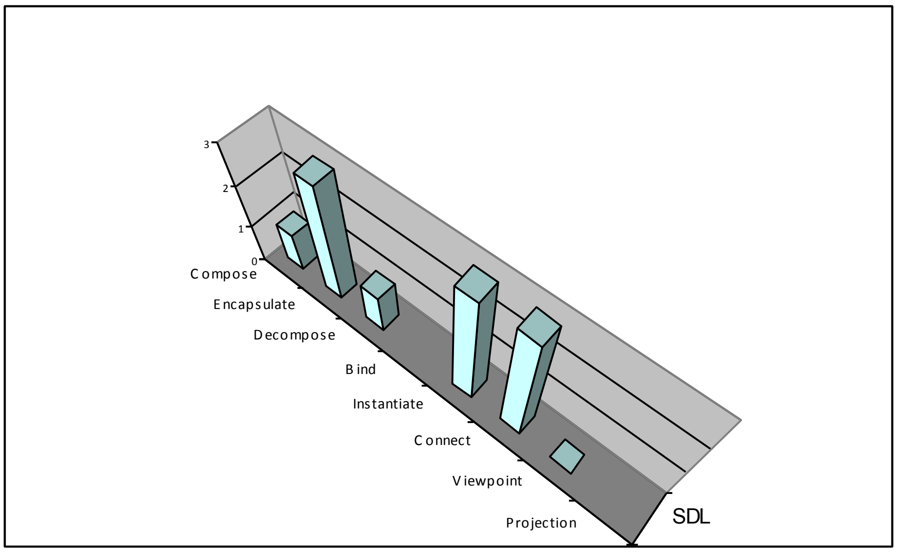


Figure A3-7. Support for concepts of Architect primitive operation.

Concerning the sub-category Primitive Architecture Element, Figure A3-8 shows that SDL provides full support for all the concepts.

Figure A3-9 shows the results related to the sub-category Complex Architecture Element, otherwise, only a little or none support is assessed SDL provides full support for Configuration. Missing support for more complex architecture elements indicates that the notion of large-scale architectures and systems was not taken into account in language development.

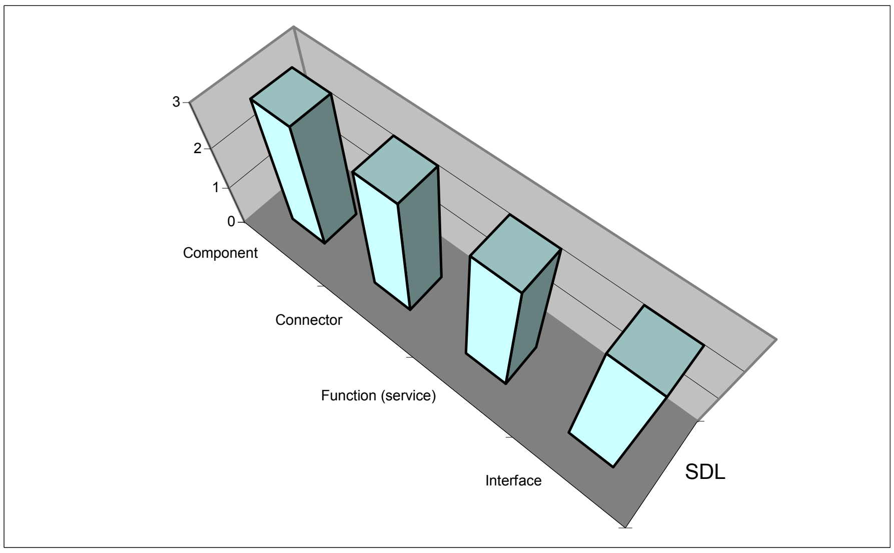


Figure A3-8. Support for concepts of Primitive Architecture Element.

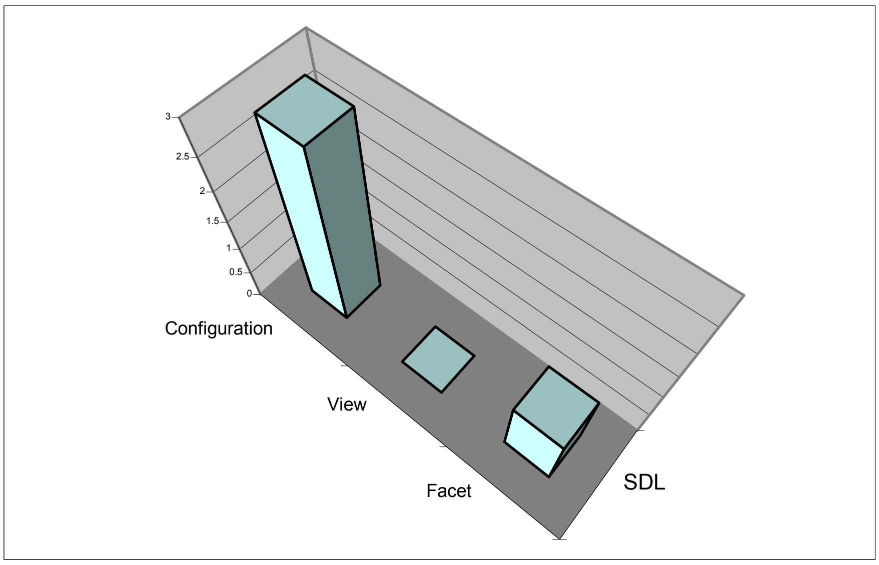


Figure A3-9. Support for concepts of Complex Architecture Element.

4.2 Analysis of Combination of SDL and Other Languages with Respect to a Particular Domain

During the project, a second exercise has been performed in order to understand if the combination of two languages could better support concepts belonging to the System Specification and modeling class.

Single evaluations have been performed on several languages, and then they have been combined. Figure A3-10 shows the result of the combination of SDL with some other language.

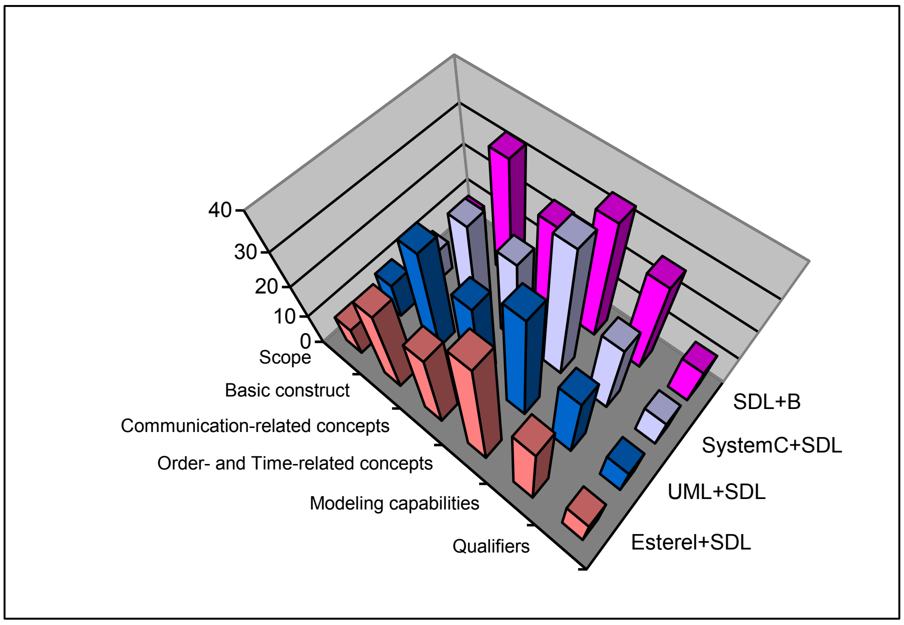


Figure A3-10. Scores of couples of languages.

Figure A3-10 shows that the more realistic combination is of UML and SDL, and this is definitely a very useful combination. Essentially formal properties and assertions reasoning about design remain missing in this combination, but almost everything else is provided. Certainly the best informal system level specification tool.

4.3 Conclusion

This Annex provides some suggestions on the way of analyzing existing languages to verify their usefulness for system level specification and design. A questionnaire build up on that purpose has been described. Finally, the use of the questionnaire has been demonstrated through an example, showing on one side that existing language is enough to describe and/or specify the System under Design: combination of different languages provides a better coverage, and on the other that architecture has not been in the center of requirements for language development.

Annex A4

GUIDELINES FOR SYSTEM-LEVEL PERFORMANCE ANALYSIS

Christophe Gendarme,
Jos van Sas
Alcatel Bell, Antwerp, Belgium

Abstract: The goal of the Performance Analysis research is to investigate how system-level methods and tools support the architecture design phase. During the architecture design phase, system-level modelling can be used to evaluate and optimise algorithms incorporated in a system, enabling an estimation of their influence on the performance of the system. The purpose is to assess requirements for performance analysis methods and tools, based on the needs of an architecture design process. The first phase of the Performance Analysis focuses on system-level modelling of the functional aspects of complex systems (algorithm exploration). The second phase addresses other design aspects (e.g., implementation aspects) to extend the defined requirements.

Key words: System design, Performance Analysis, Property, Abstraction, Switch Core, Flow Control.

1. INTRODUCTION TO PERFORMANCE ANALYSIS

In this annex, we will highlight the issues of Performance Analysis in a current system level design flow. The purpose is to perform an in depth analysis of the design and decision process for one type of system requirement , namely the performance.

The initial focus is to explore the functional aspects of a pilot complex system, enabling an investigation of the requirements for obtaining founded design decisions.

The goal of the Performance Analysis research is to investigate how system-level methods and tools support the architecture design phase. During the architecture design phase, system-level modelling can be used to evaluate and optimise algorithms incorporated in a system, enabling an estimation of their influence on the performance of the system. The purpose is to provide guidelines for performance analysis methods and tools, based on the needs of an architecture design process.

The Switch, which is referred to as Multi-Path Self-Routing (MPSR) Switch System, is basically an input/output buffered switch system. The quality of the System models will be assessed with respect to aspects such as scalability, confidence and accuracy of simulation results, model abstractness versus model adequacy and parameterisation. Those quality aspects will be related to the aspect of simulation performance (simulation speed). As a result, a framework of guidelines for performance analysis methods and tools will be defined. It will support system-level design activities such as algorithm exploration and queue dimensioning in an early stage of the design phase.

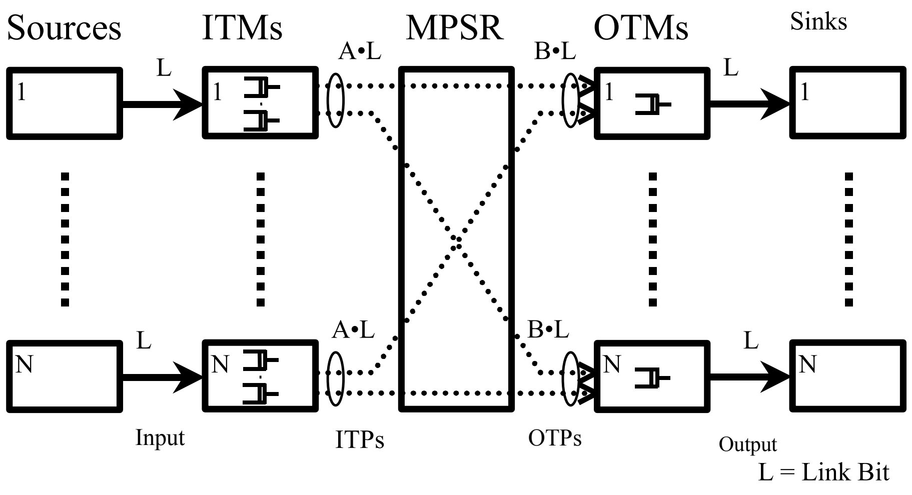


Figure A4-1. Schematic representation of the MPSR Switch System.

Figure A4-1 gives an overview of the MPSR Switch System. In principle, the MPSR Switch System involves an equal number (N) of inputs and outputs. The input traffic available for the N inputs originates from N independent Sources. However, each Source actually represents a number (M) of Links, which all induce independent input traffic on the involved input. The aggregate bit rate with which files of packets are induced on a single input is indicated with the link bit rate (L).

The input traffic is organised in such a way that the destination of all packets of a file is the same. After passing through the MPSR Switch System, the packets of a file should be available at the correct destination output without changing the order of the packets of that file. However, the sequence of packets of the considered file can be interleaved with packets of other files. For every output of the MPSR Switch System, a Sink is available, which can handle output traffic with at bit rate equal to the link bit rate. As a result, a number of MPSR Switch Systems could be interconnected.

Due to the varying size of the files of packets induced on the inputs of the MPSR Switch System, the size of the packets is varying too. Two types of files are distinguished. Long files consist of a number of packets with a maximum packet size and possibly an additional packet of some other size that is larger than a minimum packet size. Short files consist of only one packet of which the size is equal to the minimum packet size. In addition to the varying size of the files, the destination varied uniformly over the N possible destinations. To handle the properties of such input traffic, the MPSR Switch System includes input queues and output queues for buffering traffic.

The input buffers of the MPSR Switch System are located in the Input Termination Modules (ITM), whereas the Output Termination Modules (OTM) incorporate the output buffers. To overcome the problem of head-of-line blocking, any ITM includes a queue for every destination OTM. To transfer information between the ITMs and the OTMs, the MPSR Switch Core provides a so-called Virtual Ingress-Egress Pipe (VIEP) between every input queue of an ITM and every output queue in the OTM. Physically, only a single connection between an ITM or OTM and the MPSR Switch Core is available. These connections are called Input Termination Port (ITP) and Output Termination Port (OTP) respectively. Scheduling mechanisms are involved to ensure proper utilisation of the physical connections.

To reduce a possible loss of packets in the MPSR Switch System, an aggregate bit rate of A times the link bit rate is available at every ITP. An OTP can handle traffic with a bit rate equal to B times the link bit rate. Both so-called bandwidth factors A and B are larger than 1. Next to the size of the queues, the bandwidth factors A and B should be dimensioned in such a way that no packet will be lost at the OTMs and the loss of packets at the ITMs is reduced to a minimum.

2. MODELLING REQUIREMENTS

Designing a MPSR Switch System, from generation to generation, involves the identification of several system parameters, which can be considered invariants, for a dedicated product.

The Internet Protocol Core Router switch has the following performance characteristics. If the input traffic is constant bit rate (CBR), the output traffic has a known delay vs load distribution. The switch can be assumed to be lossless if the total load stays below a given upper bound.

The required system performance is expressed in terms of e.g.:

- average and minimum latency
- minimum throughput
- maximum drop probability.

In order to achieve the required performance, a bandwidth negotiation and allocation mechanism must be optimized. This mechanism is required in order to: limit the total input traffic towards the switch (to ensure lossless behavior), limit the traffic between the switch and each individual line card.

The resources available on the line cards are 1. bandwidth towards the switch and 2. the available buffer space on each line card. Because of the limitation of the total load on the switch, the bandwidth of the line cards to and from the switch is resource shared between all line cards. In order to evaluate the maximum achievable performance as a function of available buffer space and bandwidth, the buffer and bandwidth management algorithms need to be incorporated in the model, and the algorithms must be exploration and optimized.

Below, the case for a 2,5 Gbits/s (OC-48) is detailed as an example. Table A4-1 gives an overview of all system parameters that are available for the MPSR Switch System.

In general, a functional model of a system, which concerns an abstract functional representation, enables to reason about the functional properties incorporated in that system. In addition, the functional model should be able to answer whether some configuration of values for the system parameters satisfies the performance requirements according to the performance metrics. The functional model is called to be adequate if it satisfies the functional properties of the system and enables to provide a properly founded answer to this question.

A functional system-level model focuses on the conceptual aspects of the functionality incorporated in the considered system. In order to develop a compact model, which supports reasoning in an easy way about the conceptual functional properties, such a system-level of abstraction is chosen. To ensure that the functional system-level model is sufficiently adequate, discussability presents an important modelling requirement.

As indicated in Table A4-1, the system parameters N and M can vary depending on whether a small or large MPSR Switch System is considered. As a result, the modelling requirements for developing a functional system-level model include the capability of easily changing any system parameter. Such a parameterisation enables to develop a template functional system-level model for a future generation of MPSR Switch Systems that will be based on the same conceptual functional properties. However, the support of parameterisation serves another goal. Increasing the size of the MPSR Switch System (i.e., N becomes 256 or 2048), results in a decrease in simulation performance. The functional system-level model should enable an investigation of how conclusions for a large MPSR Switch System can be drawn based on simulations performed with a small MPSR Switch System. The consideration of how to support parameterisation therefore presents the result of an important modelling requirement.

Table A4-1. The system parameters of the MPSR Switch System.

System Parameter	*Notation*	*Typical*
Bandwidth factor A	A	2.5
Bandwidth factor B	B	1.35
Maximum size of a long file	MaximumLongFileSize	65536 Bytes
Maximum size of a packet	MaximumPacketSize	1500 Bytes
Minimum size of a long file	MinimumLongFileSize	1500 Bytes
Minimum size of a packet	MinimumPacketSize	40 Bytes
Number M of Links per Source	M	Range: 1 – 16
Number N of ITMs (OTMs)	N	Range: 1 – 64
Markov State transition probability α	Alpha	0.9
Markov State transition probability β	Beta	0.98
The link bit rate	LinkBitRate	2.488 Gb/s
Time delay for transferring a packet through the MPSR Switch Core	SwitchDelay	100 μs
Total size of all input queues of an ITM	TotalOutputQueueSize	256*1024*58 Bytes
Total size of the output queue of an OTM	TotalInputQueueSize	4096*58 Bytes

Another modelling requirement concerns support for estimating the confidence and accuracy of the values obtained for the performance metrics after some simulation. In general, simulating for a longer time results in more accurate results. Without knowing the simulation time for obtaining accurate results, confidence intervals can be used to enable drawing properly founded conclusions. Supporting the use of confidence intervals results in automatically adjusting the simulation time for obtaining results with a certain confidence. Additionally, it is possible to assess how accurate the estimated result is in comparison with the real value. Since simulation performance decreases due to increasing a system parameter like N, modelling tools should scale properly. This means that simulation speed should decrease in a proper way according to the increase of N.

3. MODELLING ADEQUATELY AT A SYSTEM-LEVEL OF ABSTRACTION

To enable thorough investigations on whether a particular configuration of system parameters satisfies the performance requirements for the MPSR Switch System, the development of an adequate functional model at a system-level of abstraction is involved. This section presents a detailed discussion on two aspects that are encountered when modelling the MPSR Switch System at a system-level of abstraction. The first aspect concerns the allocation of memory resources regarding the input queues and output queues, whereas the second aspect concerns the abstraction from scheduling mechanisms. Additionally, this section presents how to support the use of parameterisation.

3.1 Queue Filling Levels

In order to develop an adequate model at a system-level of abstraction, it is important to understand the functional property of how memory resources are allocated for a packet that is temporarily buffered in an input queue or output queue. If this functional property is not modelled adequately, the results of all performance metrics are affected. This Section presents the assumed memory allocation strategy, which is equal for both types of queues.

To ensure that the order in which packets of a certain file are transferred through the MPSR Switch System is not changed, the incorporated queues use a FIFO policy. However, because the bandwidth with which packets are received into a queue and the bandwidth with which packets are sent out of that queue may differ, two independent activities are concurrently involved with respect to the occupation of memory resources regarding the queue. A queue input handler is concerned with receiving packets into a queue, whereas sending packets out of a queue is performed by a queue output handler.

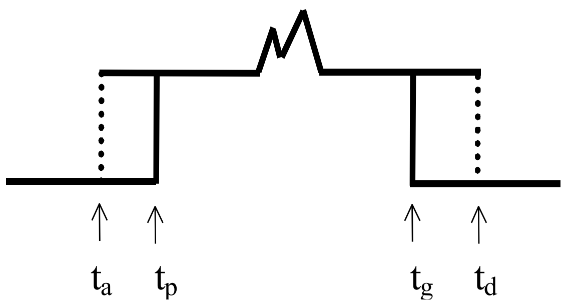


Figure A4-2. Changes in queue filling level during some period of time.

Figure A4-2 indicates how the filling level of a queue may change during some period of time. At time t_a, the head of a packet is arrived. After some time duration t_p - t_a, which should be equal to the duration of receiving the packet (i.e., the size of the packet divided by the rate with which it is sent), the packet is received completely at time t_p. Since the tail of the packet is received at time t_p, the packet is only completely available in the queue at t_p. From this time t_p on, the queue's output handler may decide to send the packet out again. Assume that the queue output handler decides to do so at time t_g. The head of the packet is sent out at time t_g, whereas the packet is completely sent at time t_d. The duration t_d - t_g is equal to duration of sending the packet out. It is remarked that t_d - t_g might not be equal to t_p - t_a due to differences in the available bandwidth and that t_g can only be equal to t_p if just a single packet is involved.

According to Figure A4-2 the packet is available as a whole in the queue for time t_g - t_p. It is possible to assume that the queue is merely occupied with a packet during the period [t_g, t_p]. However, this results in the implicit assumption that some extra memory is available to receive a packet during the period [t_a, t_p] and also some extra memory for sending the packet out during the period [t_g, t_d]. In the physical implementation this might result in unnecessary movements of information between different memories. Based on this assumption, it is even possible that a packet can be sent out faster then that it is received into the queue. This situation occurs if the bandwidth with which packets are sent out of a queue is larger than the bandwidth with which packets can be received and the condition $t_g = t_p$ holds.

To overcome the indicated unrealistic situation, it is assumed that the queue is occupied during the period [t_a, t_d]. So, at time t_a, the necessary amount of memory for the packet must be reserved or allocated (if there is enough room in the queue, otherwise the packet is discarded). This amount of memory will not be available for any other packet until time t_d, at which it can be de-allocated since the packet is then completely sent. Applying this assumption results in a much more conservative model, which might enable to draw conclusions regarding the memory needed during receiving or sending a packet.

3.2 Abstracting from Scheduling Mechanisms

Several physical resources are shared to transfer information through VIEPs between the ITMs and OTMs. A similar observation is possible for the communication between the Links of a Source and the corresponding ITM. Due to the sharing of physical resources, the physical implementation involves the use of a scheduling mechanism [1]. Several different types of scheduling mechanisms could be applied to obtain the required effect. However, the exact scheduling mechanism that is chosen for the final implementation is, in principle, not important for the conceptual functional aspects of the MPSR Switch System. As a result, modelling the MPSR Switch System at a system-level of abstraction includes abstracting from any possible solution for the scheduling mechanisms.

The effect of the scheduling mechanisms that are applied for the MPSR Switch System is the reservation of a physical resource for one specific sender and receiver combination during a specific time. To obtain this effect, such scheduling mechanisms divide the time into time slots of which a number will be reserved that is in accordance with the available bandwidth. Between the ITMs and the OTMs, the results of this negociation mechanisms are responsible for reserving more or less time slots (i.e., bandwidth) for some VIEP. Between the Links of a Source and the corresponding ITM, however, the resulting bandwidth for a single Link to ITM combination remains equal to the link bit rate divided by M.

Considering the implementation of scheduling mechanisms that use time slots, it is important how fast the average number of reserved time slots results in the assigned bandwidths. The smaller the time slots are, the more optimal the scheduling mechanism is. In the implementation of the MPSR Switch System, packets are subdivided in cells of 58 bytes to come close to the results obtained with an optimal scheduling mechanism. The optimal scheduling mechanism can be modelled by separately assigning the appropriate bandwidth to each individual sender and receiver combination [1]. Abstraction from the scheduling mechanisms as applied in the model is therefore based on the use of an individual concurrent scheduling activity for every sender and receiver combination. Each scheduling activity ensures the use of the correct bandwidth available for the considered sender and receiver combination.

Due to abstracting from scheduling mechanisms, the level of abstraction for the communicated information can be chosen independent from the actual implementation. Because the occupation of the input queues and output queues is based on the availability of packets, using the abstraction level of packets in the model is an appropriate choice. A packet is discarded completely if one or more of its cells would not fit into a queue. In the case a packet is lost, the file to which it belonged is not lost.

3.3 Concurrency

A system commonly consists of a number of different elementary active resources. Considering the example of the MPSR Switch System, a single MPSR Switch Core can be distinguished next to an equal number of ITMs and OTMs. During operation of the system, the different elementary active resources simultaneously exhibit some specific behaviour due to the incorporated functionality. To model such parallel behaviour, object-oriented modelling languages offer the use of some number of instances of object-classes. In the case that more than one elementary active resource is available of the same type, several instances of a single object-class can be used to reuse modelled functions. Support for parameterising the number of elementary active resources consists of instantiating a varying number of instances from a specific object-class.

Similar to other object-oriented modelling languages offer, the use of instantiating a varying number of parallel processes, which are represented by process objects, is beneficial. However, process objects can only share data information by explicitly communicating that information through channels. Next to multiple process objects, it offers the use of concurrent activities within a single process object for modelling concurrently exhibited behaviour. Concurrent activities may share data information that is available within the involved process object.

4. MODELLING THE FUNCTIONAL SYSTEM ARCHITECTURE

To develop a comprehensible model of the functional system architecture that is flexible enough for use with distinct configurations of values for the system parameters, it is essential to recognise how a particular function is used at several places in a system. Figure A4-3 depicts the functional system architecture for the MPSR Switch System. The dashed boxes in Figure A4-3 denote the use of equal functions in distinct functional blocks. All N Source blocks represent exactly the same function. A similar recognition is possible for the functions represented by the ITM blocks, OTM blocks and Sink blocks. It is furthermore indicated that the function of transferring packets through a VIEP is equal for all VIEPs.

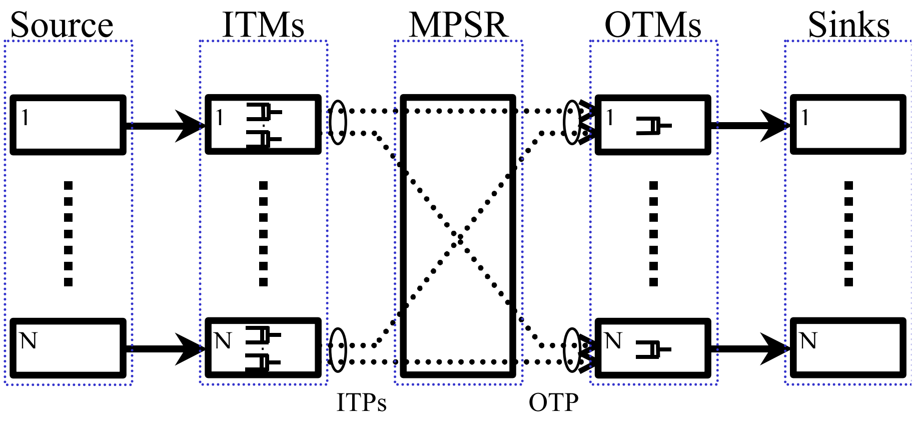


Figure A4-3. Functional system architecture of the MPSR Switch System.

The symmetric construction of the MPSR Switch System enables to take the use of equal functions in different functional blocks into account when modelling the functional system architecture. The functional blocks depicted in Figure A4-3 represent the elementary active resources of the MPSR Switch System, which have an own locus of control. Using multiple instances of the process class that describes some type of functional block enables to reuse modelled functions. In addition to using multiple instances of the same process class, the model offers the use of a number of similar concurrent activities for the purpose of reusing modelled functions.

A disadvantage of using multiple instances of the same process class is the individual initialisation of the instance parameters (E.g., the identifier of the instance). Considering the required scalability of the model, instantiating and initialising multiple process objects is not favourable. Using a number of similar concurrent activities instead, instantiation of only a single process object is involved. Initialisation of a scalable number of similar concurrent activities can be performed automatically.

5. MODELLING THE BEHAVIOUR EXHIBITED BY THE FUNCTIONAL BLOCKS

The developed model of the functional system architecture involves modelling the behaviour of the functional blocks correspondingly. The utilisation of concurrent activities for reusing modelled functions entails describing the initialisation of (similar) concurrent activities and the activities themselves. The examples are limited to some core descriptions, ITMs, OTMS, and Sinks can be easily derived from the Sources and MPSR ones.

5.1 The Process Part

This Section presents an elaborate discussion on the process part of the model of the MPSR Switch System. Figure A4-4 gives an indication of the functions included in the functional blocks of Figure A4-3. Next to the equally tinted states for the Sources, the equally tinted curled arrows denote the use of equal functions in different functional blocks. The symmetric construction of the MPSR Switch System enables to model such equal functions using similar concurrent activities within the involved process object.

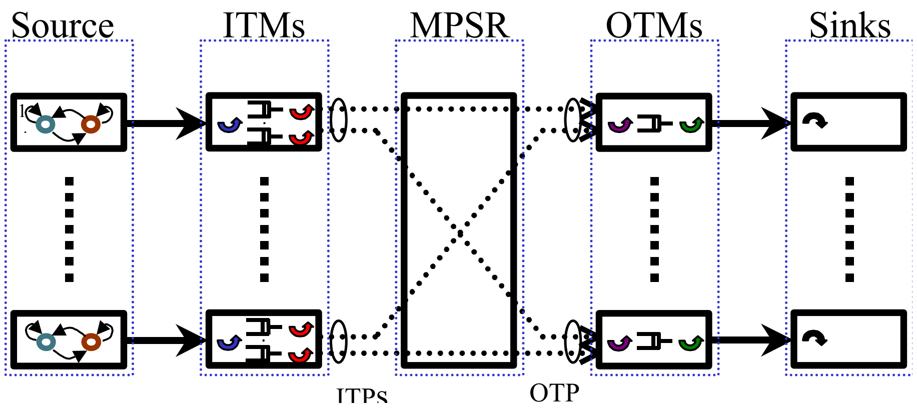


Figure A4-4. Behaviour in the MPSR Switch System.

5.2 Modelling the Behaviour for Sources

Any Source depicted in Figure A4-4 actually represents the input traffic that is available from M independent Links. All M Links of a specific Source send packets to the corresponding ITM with an aggregate bit rate equal to the LinkBitRate (L). Considering the physical implementation of M Links sending packets to one ITM, a scheduling mechanism and a queue is involved to smoothen the input traffic offered to the ITM. The physical connection between the queue and the ITM, which offers the bit rate of LinkBitRate, induces a load equal to 1. This worst case situation can be modelled adequately by abstracting from the scheduling mechanism and using M similar concurrent activities that represent the M Links. The bit rates with which packets from an individual Link of a Source are sent to the corresponding ITM is equal to the LinkBitRate divided by M, ensuring an aggregate load of exactly 1. As a result, no smoothening queue needs to be modelled.

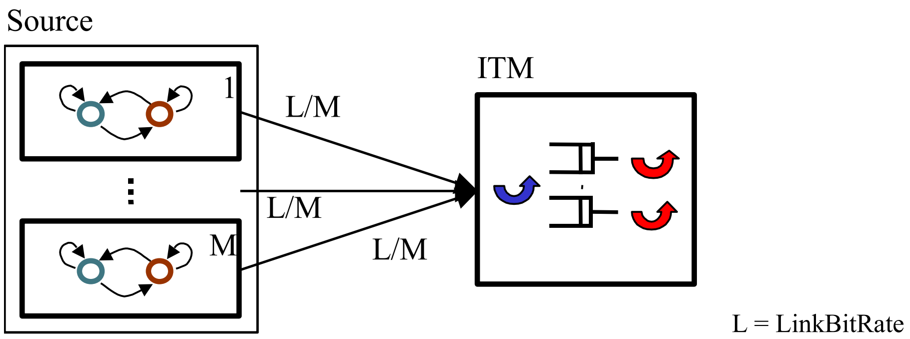


Figure A4-5. Modelling the structure of a Source using similar concurrent activities.

Figure A4-5 indicates how the structure of a Source, which actually represent M Links, is modelled using similar concurrent activities. Since all M Links of a Source are independent, autonomous activities are needed to generate the corresponding input traffic. To model the input traffic generation for any Link, a 2-state Markov mechanism is used. The process object Sources therefore involves the initialisation of M independent 2-state Markov mechanisms per Source. Since the 2-state Markov mechanism concerns a (probabilistic) finite state machine of which only one state can be active at a time, the process object Sources involves NxM similar concurrent activities at any time during simulation of the model.

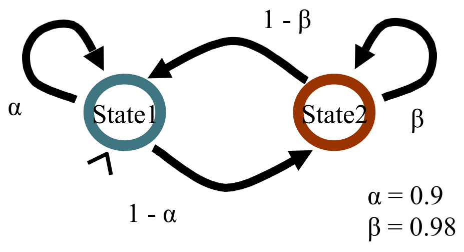


Figure A4-6. The 2-State Markov mechanism.

Figure A4-6 depicts the 2-state Markov mechanism that is used for generating the input traffic originating from a Link. Input traffic generation starts from the initial state: State1. In State1, long files are generated of which the destination is chosen according to a discrete uniform destination distribution with parameters [1, N]. A long file is subdivided into a number of packets, which all have the same destination.

The size of a long file ranges from MinimumLongFileSize (1500 Bytes) to MaximumLongFileSize (64*1024 Bytes). The size of all packets for a long file, except the last one, is equal to MaximumPacketSize (1500 Bytes). The last packet is sized between MinimumPacketSize (40 Bytes) and MaximumPacketSize. The size of the generated long file is chosen according to a discrete uniform distribution with parameters [MinimumLongFileSize, MaximumLongFileSize] in such a way that the last packet will not be smaller than MinimumPacketSize. After the generation and transmission of a long file, the 2-state Markov mechanism may continue traffic generation from State2 based on a state transition probability of 1 - α.

In State2, short files are generated for which the destination is chosen using a discrete uniform destination distribution with parameters [1, N] again. Such a short file consists of one packet of which the size is equal to MinimumPacketSize (40 Bytes). After generation and transmission of a short file, the 2-state Markov mechanism may continue traffic generation from State1 according to a state transition probability of 1 - β.

The use of similar concurrent activities results in an efficient reuse of modelled functions: only two methods need to be specified for modelling the behaviour of generating input traffic for all Links of any Source. These methods represent the behaviour exhibited according to the possible states of the 2-state Markov mechanism.

5.3 Modelling the Behaviour of the MPSR Switch Core

Figure A4-7 indicates the activities available in the MPSR Switch Core. Every individual VIEP concerns an independent activity for transferring packets from an input queue in an ITM to the correct output queue in an OTM. Since not all VIEPs may transfer packets at the same time, dynamic creation and termination of similar concurrent activities is used to model the behaviour of the MPSR Switch Core. As a result, the number of similar concurrent activities that is available during simulation of the model ranges from 1 to NxN. A smaller number of concurrent activities may improve simulation speed. Since a new similar concurrent activity is created whenever needed, no special initialisation procedure is involved for the process object MPSR.

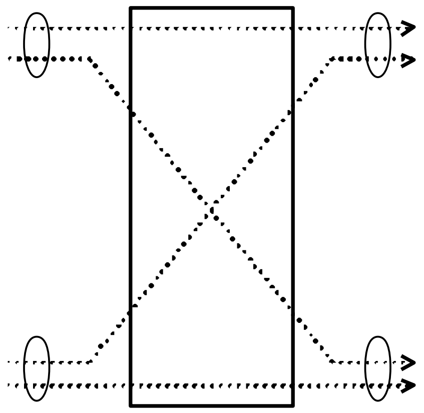


Figure A4-7. VIEP activities included in the MPSR Switch Core.

6. SUMMARY

To develop a proper understanding of the conceptual functional properties on which the MPSR Switch System is based, this annex presented an initial model. Appropriate validation is enabled by developing a compact model at a system-level of abstraction satisfying the modelling requirement of discussability. To satisfy the modelling requirement of parameterisation, it is indicated how to support the development of a parameterised model. The applied modelling strategy, which is based on similar concurrent activities, enables an investigation on whether conclusions regarding a large MPSR Switch System can be drawn based on the simulation results of a small model.

Two aspects that are encountered during the development of this model concerned the queue filling levels and the abstraction from scheduling mechanisms. A comparison of two options for modelling the occupation of input queues and output queues resulted in the conclusion that one of them is more close to the functional property defined for the MPSR Switch System. Abstracting from the scheduling mechanisms that are implemented in the MPSR Switch System enabled to concentrate on the conceptual functional aspects by disregarding design decisions concerning the implementation of such scheduling mechanisms.

Based on the modelling strategy of a parametrisable number of similar concurrent activities, the model of the functional system architecture is presented. The behaviour exhibited due to the functionality incorporated in the MPSR Switch System is partly modelled in a process part and partly in a data part. The process part models all conceptual aspects of the behaviour exhibited by the elementary active resources incorporated in the MPSR Switch System. Based on the use of aggregate data objects, the data part concerns the detailed modelling aspects of all major data operations invoked by the process part. The compactness of the process part improves concentration on the conceptual functional properties.

ABBREVIATIONS

Abbreviation	*Description*
ID	Identity
IP	Internet Protocol
ITM	Input Termination Module
ITP	Input Termination Port
L	Link Bit Rate
MPSR	Multi-Path Self-Routing
OTM	Output Termination Module
OTP	Output Termination Port
VIEP	Virtual Ingress-Egress Pipe

REFERENCES

[1] Keshav, S. An Engineering Approach to Computer Networking; ATM Networks, the Internet and the Telephone Network. Reading, Massachussetts (U.S.A.):Addison.

Annex A5

BLUESTONE: A CASE EXAMPLE

Kari Tiensyrjä
VTT Electronics, Oulu, Finland

Abstract: In order to illustrate the concepts in a practical way, this annex presents an artificial but realistic example of an application of the SDCM to an electronic product development. The story told here starts with the idea of a need and ends with a set of information needed to synthesise the information needed to make a product to meet that need. The example is based on extensive reuse of existing system intellectual property (SIP), part of which comes form the company itself, while part comes from external sources. For the purposes of the example we conjure up a company called Bolderbits Inc. They are in the business of making consumer electronic products and have a track record in wireless communications. They develop and market advanced mobile phones. A Bluetooth equipped model is their latest one. The example will follow the design of a new product, which they will call "BlueStone".

Key words: System design, user and domain requirements, functionality, functional architecture, architecture template, hardware architecture, software architecture, system design process, system under design, system IP, reuse, platform, Bluetooth.

1. SDP: A FLOW FROM IDEA TO SYNTHESIS

In this part of the example we will coarsely outline how the conceptual model of the System Design Process (SDP) is instantiated in the case of BlueStone.

The example is constructed to include enough detail to help illustrate our conceptual model. To do this the set of facets will be specialised along with the other elements in the model: e.g. the activities, artefacts and roles.

For this example we will define the following team, called BlueTeam, consisting of actors (with names taken from a well known cartoon), and list their roles, activity responsibilities, and artefact responsibilities in the example process:

- Mark: Market Analyst
- Dino: Product Manager
- Fred : System Architect
- Barney: Software Designer
- Wilma: Hardware Designer

There are a number of activities that will take place in the process, the names of which are those used in Bolderbits:

- Market Definition: Where the idea happens, identification of a need, enumeration of price, volume and margin. The resulting artefact set is User and Domain Requirements Specification.
- System Design - Refinement (SD-R): Refinement of the idea into a set of informal and formal requirements. The resulting artefact set is Technical Requirements Specification.
- System Design - Partitioning (SD-P): Exploration of potential solutions leading to a design proposal. The resulting artefact set is System/Architecture Specification.
- System Design - Synthesis (SD-S): Design activities in various technical domains (e.g. HW and SW). The resulting artefact set is System/Architecture Description.

The core of BlueTeam is summarised in Table A5-1, and Figure A5-1 shows the top-level view of the design process BlueTeam will carry out. Although the market definition as a facet and the market analyst as a respective actor have been included, explicit information about business goals has been left to the reader. Actual system implementation has neither been addressed.

Table A5-1. Core Blue Team.

Resource	*Actor*	*Role*	*Facet/Activity*	*Artefact Set*
Mark	Market Analyst	Analyser	Market Definition	User and Domain Requirements Specification
Dino	Product Manager	Manager	Requirements Analysis	Technical Requirements Specification
Fred	System Architect	Architect	Conceptual Design	System/ Architecture Specification
Barney	Software Designer	Designer	Functional Decomposition	System/ Architecture
Wilma	Hardware Designer	Designer	Platform Analysis	Description

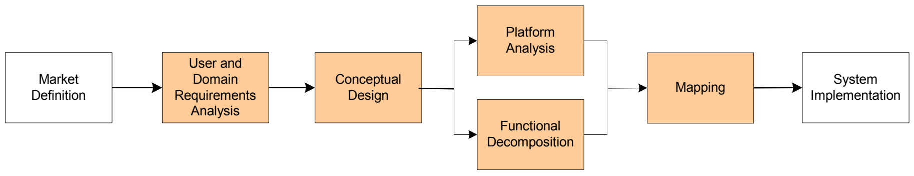


Figure A5-1. Facets of BlueStone design process.

The descriptions given above are not intended to be exhaustively complete but they should illustrate the nature of the activity. These can be seen as top-level activities in a hierarchy that could be decomposed using the SDP conceptual model process composition view.

There is a close linkage between the SDP and the SUDM produced as a result of that process so it will not be possible to maintain a strict distinction between them. It is important to remember that the progress of the design process is not directly correlated to the layers of abstraction. Table A5-2 shows the way information might build up in the layers by plotting the information content in each layer at a set of points or phases in the design process.

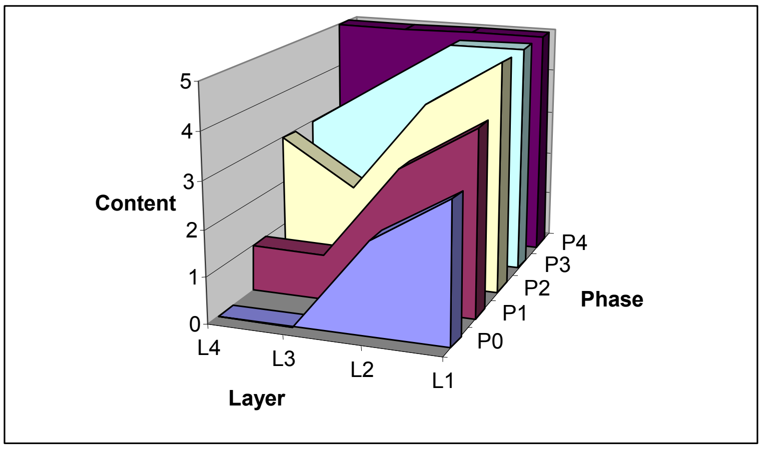


Figure A5-2. Information Content Profile as System Design Progresses.

The y-axis is normalised so that, for each layer, 5 indicates “complete”. It is not unusual for the design to progress at the “lower” layers faster than the higher ones. By definition the design is not complete until all the required information is available. A well behaved design process will address the higher layers first and the check-points or toll-gates will require information to be available at the higher layers early on. This follows from the fact that the cost expended and time used to work in the higher layers tends to be less than in the more detailed lower layers. In many cases the higher layer information does exist but is not made explicit. This may be because there is a paucity of ways to express, share and verify this information.

As will be seen in the following example, the design progresses rather quickly to layer L3. Furthermore, it will be noticed that information is added to the higher layers whilst, apparently, working in a lower layer.

The three-dimensional nature of the realisation of a design process means the narrative that follows will inevitably mix the design process and system under design aspects of the conceptual model. The authors have chosen to unfold the story in a SUD layer first order. This has the effect that the process is less visible. Even so there will be one or two points where the process dimension shows through, for example, where the partitioning of functionality between the handset and headset reveals that one of the requirements cannot be completely fulfilled.

2. SUD: A WIRELESS HEADSET FOR A MOBILE PHONE

This section describes the example System Under Design (SUD). The system will be described according to the layers of abstraction presented in the conceptual model. It should be noted that the relationship between the design process and the SUD model (SUDM) is not defined by these layers. The next section will show the relationships based on the concept of **views** between SDP and model subsets of the SUDM presented at the end of the chapter.

2.1 User and Domain Requirements

Although the heading gives away the end result, we start the design process off at as an early stage as possible to fully illustrate the model. The starting point, stated in as simple and brief terms as possible is the following need:

```
A system that allows a person to use a mobile
phone without touching it.
(Motivations to have such a product include e.g.
safety, comfort, regulations, mobility, privacy,
life-style, etc.).
```

This can be expanded into a list of informal elements of functionality, each of which can be thought of as providing a service to a user of the system:

```
FR1. Initiate a call
FR2. Accept an incoming call
FR3. Reject an incoming call
FR4. End a call.
```

With these come some elements of usability:

```
NFR1. Plug-and-play: easy to start to use
NFR2. Light in weight
NFR3. Power supply included
NFR4. Not physically connected to mobile phone
NFR5. Easy to operate.
```

An assumption about a possible way to meet the stated need has crept in with the fourth item in the list. There is an implicit partitioning between the mobile phone and the means used to meet the need. It has been assumed that a solution includes some physical addition to the phone rather than just a new facility or function that is added to the phone. Whilst we are aiming to avoid discussion of the relationship between SDP and SUDM it is interesting to note at this point that the system design process has clearly already started! In fact, in our definition the process started as soon as a germ of an idea came into being.

For Bolderbits, the proposal to use their mobile handset model as basis is quite natural. The system architecture is apparent, if only in very rough terms, as there has been a partitioning into two main parts: the handset and the headset as depicted in Figure A5-3. As the handset is taken to already exist, the headset can be identified as the system under design (SUD).

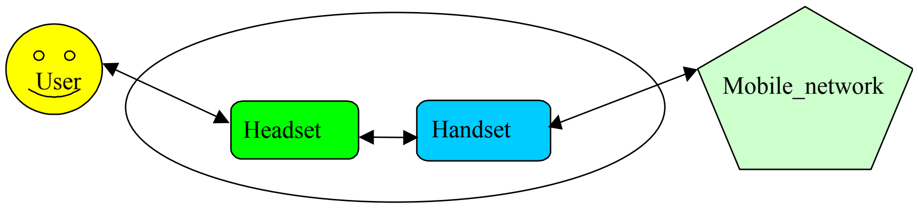


Figure A5-3. BlueStone partitioning into headset and handset.

Once this first partitioning step is taken (phone + physically separate add-on), there are some (derived) requirements that can be identified which relate to the interface of the user with the device:

```
DR1. Earpiece and microphone
DR2. Activate/deactivate with push button
DR3. Control with voice commands
DR4. Audio feedback/response to commands
DR5. Visual indication of status (e.g. active/inactive)
DR6. Audio and/or visual indication of low power supply capacity.
```

Based on the above, there is obviously need for other interfaces:

```
I1. Non-physical connection to mobile phone
I2. Power supply change or automatic loading.
```

So far no quantitative information has been revealed so the following performance measures are included in the example:

```
P1. Sound quality at least as good as mobile phone (assume GSM EFR for this example)
P2. Range (distance between ear and phone)<5m
P3. Time to set up system <20 seconds
P4. Minimum operation/standby time 24 h / 336 h with full power supply.
```

Among possible connection technologies, Bolderbits studies e.g.

- Infra-red, but rejects it due to e.g. line-of-sight requirement
- FM radio, but rejects it due to e.g. missing call privacy
- Etc.

Bolderbits decides to set Bluetooth as a preferential choice of connection technology. In order to ensure co-operability with Bluetooth equipped mobile phones of other manufactures, the company sets the respective Bluetooth specification (www.bluetooth.com) and especially its Part K:6 Headset Profile as a requirement. Consequently, the system is a wireless headset that connects to a mobile phone via Bluetooth channel, as depicted in Figure A5-4.

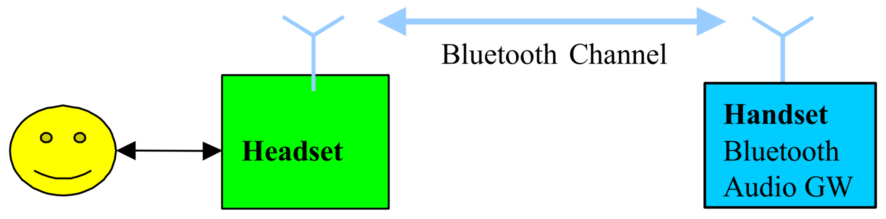


Figure A5-4. Bluetooth communication channel.

BlueStone Use Cases (UC)

The main interactions of the user and the handset with BlueStone are depicted in the use case diagram of Figure A5-5.

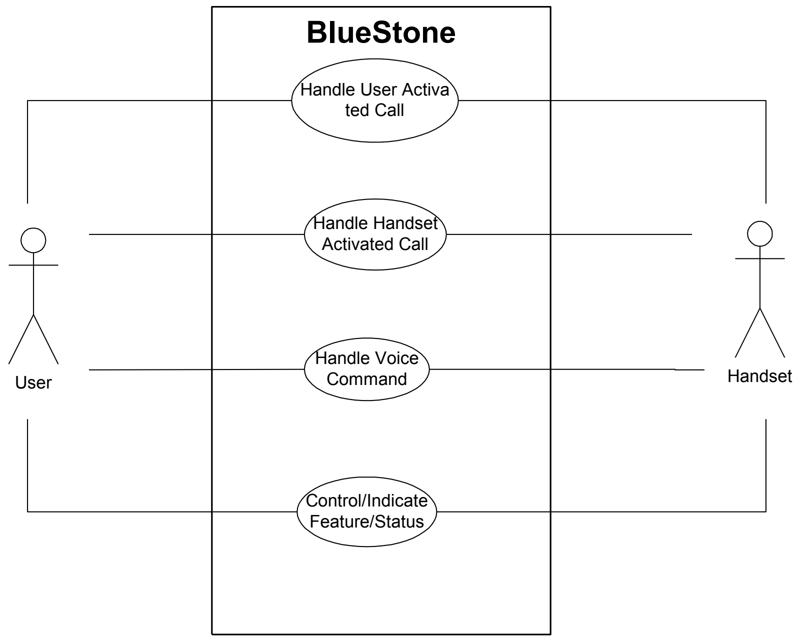

BlueStone
Handle User Activa ted Call
Handle Handset Activated Call
Handle Voice Command
Control/Indicate Feature/Status
User
Handset

Figure A5-5. Use case diagram for BlueStone.

UC1: Handle User Activated Call

- Pre-condition: Active headset (i.e. paired)
- The user initiates a call and selects the recipient by using voice commands.
- During on-going call, the user can control the audio volume of the earpiece by the button.

- The handset closes the call connection when the recipient ends the call
- The user ends the call by pressing the button (for longer than a certain time 1).

UC2: Handle Handset Activated Call

- Pre-condition: Active headset (i.e. paired)
- The handset initiates a call by alerting user via the headset to the earpiece. The user can accept or reject the call by using voice commands.
- During on-going call, the user can control the audio volume of the earpiece by the button.
- The handset closes the call connection when the recipient ends the call
- The user ends a call by pressing the button (for longer than a certain time 1).

UC3: Handle Voice Command

- Pre-condition: Active headset (i.e. paired)
- Voice commands are available, when a headset is active (i.e. the headset is paired with the handset) and no call activity is on-going. The headset relies on the respective speech recognition features of the handset. The responses from the handset are received via the earpiece.

UC4: Control/Indicate Features/Status

- Assuming a powered-off headset, the user turns the power on by pressing the button (for longer than a certain time 2). The headset indicates successful operation by LED and audio tone.
- In order to activate a headset for operation after power on, a paring process with a handset for authentication and encryption is needed. The user starts the paring and provides the required PIN-code at the handset. The headset indicates successful operation by LED and audio tone.
- Assuming a powered-on headset, the user turns the power off by pressing the button (for longer than a certain time 3). The headset indicates successful operation by LED and audio tone (later turning them off).
- During on-going call, the user can control the audio volume of the earpiece by the button.

All of the above can be seen to fall into the “User and Domain Requirements” but they also fall into “Functionality”. This is consistent with the observation that the System Design Process had started before the set of lists above were complete. It could be said that, as soon as one tries to write down something about a system, one is designing it.

In this example the Wireless Headset is the System Under Design (SUD). The development of the mobile handset, not untypically, is carried out in another department who will be responsible for adding the Bluetooth capability to the phone including the Audio Gateway implementation of the Bluetooth Headset profile.

The following sub-sections will describe the SUD in terms of the layers of abstraction presented in the SDCM.

2.2 BlueStone at Layer L1 Abstraction

To illustrate this layer the SUDM for the wireless headset product called BlueStone will be expressed in terms of states, variables, relations and properties. The following variables (short names in brackets) can be identified:

1. `Configuration: Uninitialised/Initialised (Boolean) (C)`
2. `Power: on/off (Boolean) (P)`
3. `Activity: Active/Inactive (Boolean) (A)`
4. `Call: ongoing/No call (Boolean) (N)`
5. `Capacity: ok/low (Boolean) (B).`

The sets of values these may take represent the states of the SUD. Put more formally, the total number of states is the cardinality of the Cartesian product of the sets of values. In this case, as all the variables are Boolean the total number is 32.

There are some properties that can be expressed as relations between the variables that begin to say interesting things about the SUD (let us call it “device” for simplicity):

1. `An uninitialised device cannot have an ongoing call.`

   `C=false :- N=false  (:- means implies)`
2. `An inactive device cannot have an ongoing call.`

   `A=false :- N=false`
3. `A powered off device cannot change its configuration.`

```
   P=false :- const(C)=true (const(V) is a predicate
   that is true if V does not change).
4. A powered off device cannot be active.
   P=false :- A=false
5. A device with low capacity will be powered off.
   B=true :- P=false.
```

These properties reduce the total number of possible states of the SUD. The reader should note that the states described here do not relate to a real system but the system under design. In other words we are dealing with a model of the real system, so a characteristic of the model is that it has a total number of states = Card($\wp$(Values)) and a reduced number of possible states due to the properties that have been, so far, identified.

The variables and properties depicted here are only a subset of those that could be identified in the real system under design. The state-space and design-space would soon get out of hand, even when the reduction by identification of properties is included. The common way to handle this growing complexity is to partition the design space. The result is the creation of one or more sub-systems. Each of these has a set of interfaces, a set of variables and a set of properties. The paradox is that the overall complexity of the model (SUD) is increased by the partitioning operation but, taken alone, each sub-system will have a lower complexity (fewer variables and hence states). Interfaces will be formed when interactions between variables span the partitioning boundaries. Some variables, for example the power state, will be shared across the boundaries.

In the BlueStone example we will partition the SUD into three sub-systems. Two of these are the User_sub-system and the Communication_sub-system. Both of these are sub-systems of the component kind as depicted in Figure A5-6. The third is a sub-system of kind connector that joins the other two.

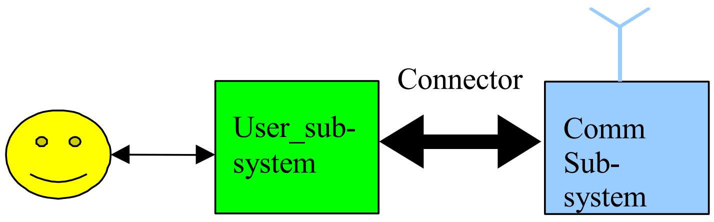


Figure A5-6. Partitioning of functionality of BlueStone.

This functional partitioning separates the concerns of the user interface from those of the radio link to the headset. The result of the decision to divide the system in this way is the need to have some way of connecting the two parts. For now this will be kept loosely defined as a connector.

2.3 BlueStone at Layer L2 Abstraction

The functionality that was described in an informal way by the requirements above is translated into services and mathematical functions. In this layer the details of the architecture of the system can begin to be seen. It is important to recognise that a real system design process might iterate and produce a number of candidate architectures, however only one top-level view will be presented in this example.

The functions will be represented here in a form of pseudo-code. The following conventions will be used in the pseudo-code.

Upper_case_initial = identifier/name
Lower_case_initial = operator/function/keyword

```
system is {Bluestone}
environment is {Mobile_phone, User}
system + environment ⇔ join {Mobile_phone, User}
with Bluestone
state System_state of Bluestone is {Off,
Initialised, Active, Incoming_request, Incoming_
active, Outgoing_initiate, Outgoing_active}
function User_sub-system of Bluestone is
{Power_on_off, Initialise, Activate, Indicate_
status, Initiate_call, Accept_call, Reject_call,
End-call}
interface User_interface of User_sub-system is
{Button, LED, Microphone, Earpiece}
connect (Bluestone, User) by (LED, Button,
Microphone, Earpiece)
NB. LED, Button etc. represent here logical
functionality (services), not the electrical
components.
function Communication_sub-system is
{Establish_link, Establish_control, Establish_
audio, Release_audio, Release_control, Release_
link}
interface Communication_interface of
Communication_sub-system is {Bluetooth_radio}
NB. Bluetooth_radio represents here logical
functionality (services), not the electrical
component.
```

```
connect (Bluestone, Mobile_phone) by
{Bluetooth_radio}
connect (User_sub-system of Bluestone,
Communication_sub-system of Bluestone) by
Connector.
```

To summarize, the example system has, as depicted in Figure A5-7, at L2 layer three sub-systems (or indications of such) Communication_sub-system and User_sub-system, which are connected by a connector. Furthermore, the Communication_interface of the Communication_sub-system has become a separate sub-system Bluetooth Radio reflecting the decision to use external System Intellectual Property (SIP) for its implementation.

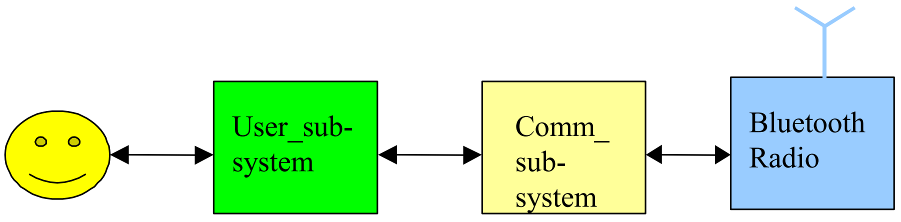


Figure A5-7. Functional architecture of BlueStone.

In the Architecture (functional) view the following can be seen: state = System_state of Bluestone, which lists different modes of the system. Additionally, there are functions (services) of the two sub-systems. These can be expressed using Action Semantics as shown below for the User sub-system.

```
A property:
Capacity=low :- Power=off
An equation:
Capacity = Check_battery(voltage,threshold)
Where the Check_battery function is defined by:
       x>y :- Check_battery(x,y) = true
       x=y :- Check_battery(x,y) = false
       x<y :- Check_battery(x,y) = false
A maplet:
Button|---- Power // pressing the button will
affect the power state
Battery_voltage|---- LED // the LED will indicate
a low battery voltage
Causal chain:
```

```
Button|----    Power|----    System_state    |----
Radio_state |---- BT_link_state
// Pressing the button will turn on the power and
initialise the system. The radio will operate and
a Bluetooth link with the handset will be
established.
Thread of maplets is shown in Figure A5-8.
```

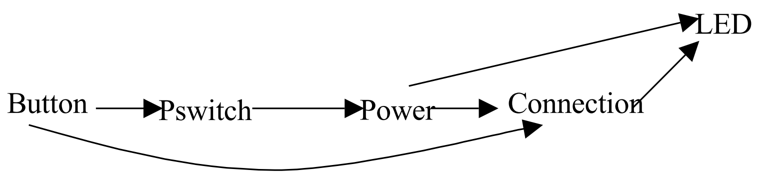


Figure A5-8. Thread of maplets.

This example shows that the value of the Button variable can affect both the value of the power switch (Pswitch) variable and the Connection variable. Similarly, the LED variable is influenced by both the Power and the Connection variables.

The causal chain and thread of maplets span the whole system. When the interaction crosses the boundary of variables, there will emerge interfaces between the sub-systems and connectors.

One thing that is evident from the chain of maplets is that the button affects the value of both Power and the Connection variables. However, the power affects the connection too. The button functionality needs to provide different changes in the variable to distinguish what the user wants to change. This detail is left until later but the requirement has been defined here.

2.4 BlueStone at Layer L3 Abstraction

At this layer the structure of the SUD is described in terms of modules. The Bluetooth platform that is the candidate for the system offers two options that can be seen as different architectural templates. The first divides the functionality into two parts: host controller and device controller. The host handles the higher layer of the Bluetooth protocol stack and the profile functionality while the device performs the lower layers and implements the air interface. The second implements all the functionality on one unit. These are described as the two processor and single processor configurations, respectively due to the fact that the partitioning involves software and the software needs a processor to run. They are also known as hosted and embedded scenarios. In the SDCM the host and device controllers or embedded controller can be seen as modules and the task in the design process is to partition the functions into modules. Figures A5-9 and A5-10 show the resulting mappings for the two architectural templates.

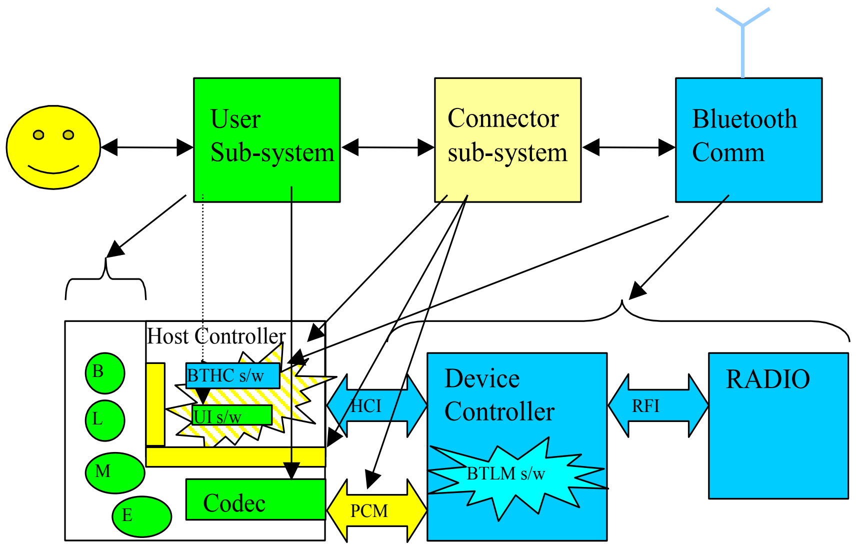


Figure A5-9. BlueStone based on hosted Bluetooth platform.

The major modules in this hosted platform partitioning are: Host Controller, Device Controller and Radio. The BTLM (Bluetooth Link Manager) and BTHC (Bluetooth Host Controller) software are part of the platform. The UI (User Interface) software is developed by BolderBits. There are some sub-modules shown within the Host Controller. These are the voice codec (containing audio AtoD and DtoA converters), a button (B), LED (L), microphone (M) and an earpiece (E). The host and device controllers are connected by the Host Controller Interface (HCI) and Pulse Code Modulation (PCM) interfaces. The Radio is connected by an interface called RFI (Radio Frequency Interface). The software is divided into two parts. The platform includes the Link Manager software (BTLM) that runs on the device controller. The Host Controller includes a processor that runs the Host Controller (BTHC) software and the User Interface software. Both of these would need to be sourced or developed separately from the Bluetooth hosted platform.

The mapping of functions in the functional architecture are shown by arrows pointing at the modules in the physical architecture. It is interesting to note that the connector sub-system maps to a number of sub-modules. The rectangles in the Host Controller adjacent to the software “star” represent peripheral input-output sub-systems.

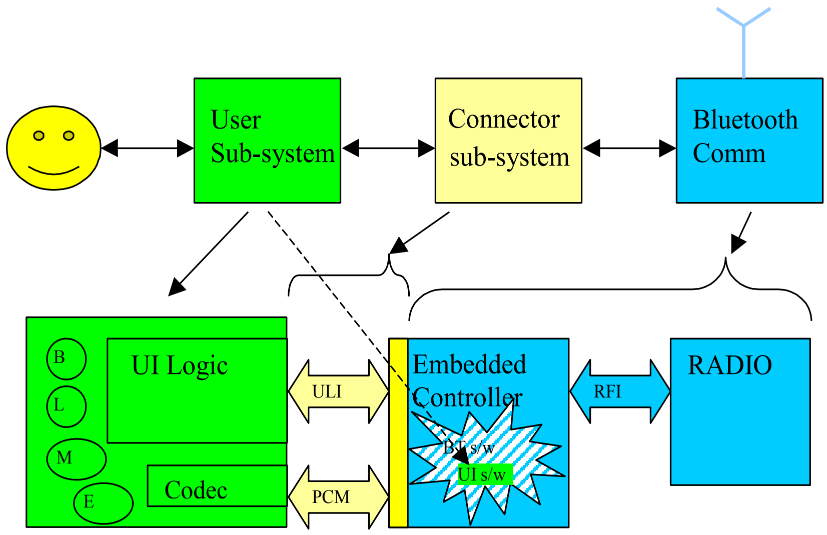


Figure A5-10. BlueStone based on embedded Bluetooth platform.

The embedded platform partitioning has a codec and a User Interface Logic module (UI Logic) which are connected to an Embedded Bluetooth Controller by the User Logic Interface (ULI) defined by BolderBits and the PCM interface. In this alternative all the software runs on one processor that is part of the platform and resides in the Embedded Controller.

The selection of the appropriate option for the platform depends on a number of factors including system cost and size. These favour the embedded platform but there are performance requirements that must be met. The use of the embedded option introduces a constraint: the UI software has to run on the processor in the embedded platform. From this issue we can identify that the System IP (SIP) provider must supply information about the available capacity on that processor. They also need to provide open (or at least well documented) interfaces to peripheral devices such as the codec and the UI logic. The same goes for the run-time environment: the SIP user will want a reliable way to add functionality to the platform to realise their product.

It can be argued that the performance issue introduces the need to have a notion of time. The reader may have noticed that time has not been mentioned at all so far. The system description (in some facet of the SUDM) needs to be elaborated with information about events and the time from one event to the next. An example below tries to depict that.

```
From the requirements: the time to set up the
system must be <20s
The constraint will be:


Button {t1,off} {t2,on} // NB t1 < t2
Connection {t3,idle} {t4,connected} // NB t3 > t2
, t4 > t3
// Where tn are ordered tags : no notion of time
as yet
Now we can introduce time by saying that the tags
are in the set of natural numbers and that their
values represent the time of the event in
microseconds. The constraint can be written as:
t4 – t2 < 20000000
Assuming that the sound quality can be expressed
in objective terms, such as maximum bit error rate
on the PCM stream, this requirement can be
expressed in a similar way.
The battery for the headset was not shown in the
partitioning figures, but it is interesting to
consider a derived constraint on the battery
management sub-system, if we introduce a
requirement that the system should provide a
audio/visual warning of low battery voltage and
shut down if the battery voltage falls below a
certain critical level.
```

The embedded controller based system is realized in our example by Bolderbits licensing a platform for embedded Bluetooth applications. One way to look at a platform is as the set of services it offers. These services exist in different facets, for example, the physical architecture that has been presented so far in this story. Another important facet of a platform contains the system design know-how. This may take the form of rules or constraints that must be met by any design that uses the platform. A very concrete example of this would be an ASIC design platform consisting of a cell library, design rules and rule checking scripts (e.g. a DRC (Design Rule Check) run-set). In the context of our example the platform supports the addition of user software to run on the embedded processor but that software may not violate the constraint that says that the CPU must be available to the Bluetooth software so that it can service the hardware. The Bluetooth specification divides time into slots with a period of 625us. The software, in our example, requires access to the processor once every two slots. This constraint is ensured in the implementation by a real-time assertion checker. In an ideal world there would be some way to check the rule in a static way but this would require tools that, as far as the author knows, are not commercially available.

2.5 BlueStone at a Diversion

At this point in the story Bolderbits carries out a review of their design. Consider the original requirements and their allocation as depicted in Table A5-2:

Table A5-2. List of Bluestone derived requirements.

Id	*Requirement*	*Allocated to*
DR1	Earpiece and microphone	UIL
DR2	Activate/deactivate with push button	UIL
DR3	Control with voice commands	???
DR4	Audio feedback/response to commands	???
DR5	Visual indication of status (e.g. active/inactive)	UIL
DR6	Audio and/or visual indication of low power supply capacity.	UIL (audio?)

DR3 and DR4 are not completely satisfied by the UIL module or at least for the sake of this example let us assume this is the case so far. The reader will have to take it as fact that the handset has voice dialling and which can be activated by a magic word. The requirements imply that all the features of the phone should be accessible via the headset. The battery life requirement means that the radio link cannot be active all of the time the phone is switched on. This means that any magic word recognition would have to be implemented in the headset (as well as the phone). After evaluation of the platform it is apparent that it does not have enough spare processing capacity to implement even a simple speech recognition facility. The requirement for voice control needs to be modified if it is to be met. So, something that could have only been discovered at layer L4 has an effect on the design in layer L1.

This, admittedly fabricated, example has illustrated the multi-dimensional nature of the design space. The layers in the SUD and the activities or check-points in the SDP cannot be described in a linear fashion. It has been possible with implementation design, such as at RT level synthesis, to draw lines in the sand and define more or less strict boundaries between activities. To do the same in system design, it will be necessary to find sets of assumptions that can be used to reliably define similar boundaries. Having found these boundaries then tools can be created that operate within them.

It is not clear if UIL should provide the audio feedback or if the phone should generate tones and send them over the radio link. The partitioning of this requirement can be realised by considering the properties and states and the SUD. Strictly according to the SDCM, this design task could operate at layer L3 and above as all the required information is available there. By analysing the way Bolderbits do their product design we have identified an opportunity to improve their design process. This particular example is not critical to the choice of platform but Bolderbits may have found themselves committed to a sub-optimal solution and unable to change the design without incurring a large cost.

To illustrate the action semantics involved consider these example power consumption estimates of the embedded platform in different states:

```
Standby:  10uA  average  (for  single  Bluetooth
chip)
Parked:  1.2mA average (100ms period)
Connected:   20mA  average  (voice  active)  for
BT+ 20mA for UIL.
```

Given that the required standby time was 336h compared to the talk time of 24h it follows that the current consumption needs to be lower in the system standby state. When the system is in this state the headset is in a state in which is ready to receive an incoming call. This requires the connection between the handset and the headset to be able to react within a less than a second or so. To achieve this with Bluetooth the devices need to be participating in a link which is parked. This means they will rendezvous every period to maintain synchronisation. The period can be selected to trade off power consumption with system latency. One solution for the magic word facility would be to implement it by detecting sound at the microphone on the headset and then connecting to the handset to send the speech for recognition. The chain of maplets would be:

```
Microphone|----Threshold|----Connection|----Power.
```

Here the "Power" system variable would have possible values = {off, standby, on}. This variable should not be confused with the state of some switch in the implementation of the system. In terms of implementation it is more likely to be a value stored in a memory or flip-flop. Both "standby" and "on" draw current from the battery. As stated earlier, to minimise to latency, the Power state called "standby" is implemented by keeping the Bluetooth link to the handset in the "parked" state.

The argument for the audio feedback is rather simpler to work through. An unpaired headset (i.e. one that has lost its authentication to connect to the handset) or, more simply a headset out of the range of its handset, will still need to provide the audio indication of low battery level etc. So this requirement must be met within the headset and some way must be provided in the implementation to send tones to the earpiece.

```
Check(Battery,Threshold)|-----Audio.
```

Rather than, for example:

```
Check(Battery,Threshold)|-----Connection|----
Tone|----Audio.
```

Where “Tone” is a variable in the handset sub-system in the context of the original layer L1 model.

2.6 BlueStone at Layer L4 Abstraction

Having chosen to use the embedded Bluetooth platform as a basis for the headset, Bolderbits are faced with the task of specializing that platform towards their end product. In addition to the user logic hardware and user interface software it becomes apparent that there will need to be some additional software in the handset. The handset software team takes responsibility for the implementation so this example will not include any details except to mention that the software is needed to support the configuration of the headset. For example, the authentication and encryption mechanism in Bluetooth uses a secret key (PIN codes) to facilitate the pairing process. As the headset has no way to enter numeric data by itself, the phone is used to set the PIN code via a data link (RFCOMM) via the Bluetooth interface. The Bluetooth platforms includes the Generic Access Profile, which implements the pairing process but the initiation and data input have to be added by the platform user. This illustrates that a platform can be seen, in a way, as an incomplete design where certain refinements are left to the platform user. The differences between this System IP (SIP) and Implementation IP (IIP) such as a virtual component (i.e. a combination of synthesisable RTL, scripts and verification test benches) lie in the fact that the refinement of the IIP is more or less automatic whereas the SIP calls for much more manual intervention. The other main issue is the fact the form and content of SIP is much less well defined. The industry has no clear definition of what to expect from SIP. The situation is as it was before VSIA began to influence the IIP domain.

The layer L4 in the SDCM is characterised by the SUD being elaborated with model artefacts representing notions such as queue, shared variable, buffer etc. The hardware models take the form of logical block diagrams and, in the analogue domain, schematics. The digital functionality is refined down to FSMs and data flow graphs that are amenable to automatic synthesis. The mapping from the functional architecture to the hardware architecture is shown in Figure A5-11.

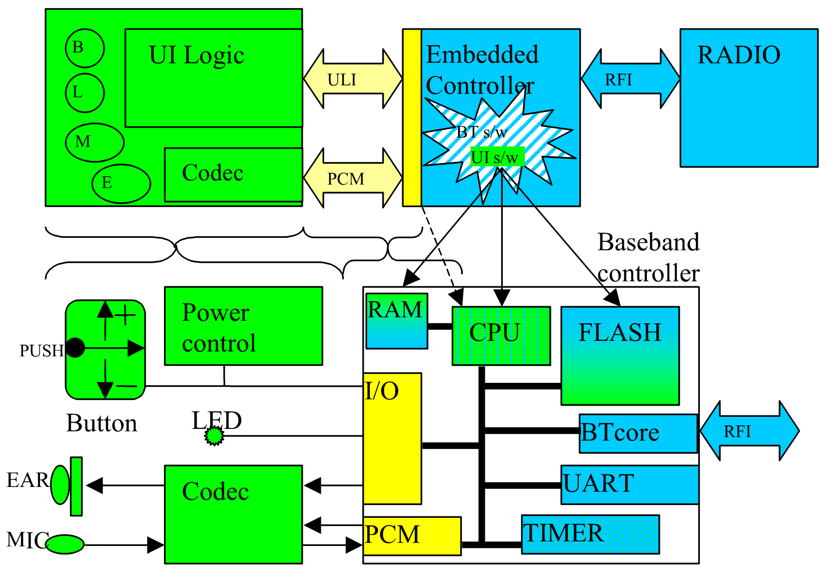


Figure A5-11. BlueStone mapping onto hardware architecture.

The mapping is shown by the braces and arrows. This figure also shows the mapping from the software functional view into the hardware structural view. The User Interface software is executed on the CPU in the baseband controller and it consumes memory in the RAM (read-write data) and Flash (code and read-only data). The connector maps mainly onto the IO block and the PCM block but there is an element of the connector that maps to software that runs on the CPU. This is shown as a dotted line in the figure.

The mapping of functions can be explored by looking at the Button element in the functional architecture:

Table A5-3. Button functions.

Function	*Button action*
Power on/off	PUSH
Pair	PUSH
Activate	PUSH
Volume up/down	+/-
Accept call	PUSH
End call	PUSH

The volume up/down function introduces the need for a hardware button that has 3 actions. This comes from the need to indicate whether the user is requesting an increase or decrease in volume level at the earpiece. This request also needs to be distinguished from a request to end the call. It would not be sensible to expect a voice command to be used as the commands would be heard by the other party in the phone call.

There is another conflict in the mapping concerning the "power off" and other functions activated by the "Push" action. This is resolved by defining a simple protocol that says that the power will only go off if the button is depressed for more than a certain length of time. This can be implemented using the timer hardware/software function in the platform.

The overall function of the button can be represented by a finite state machine (FSM) model. A data flow graph is a more appropriate model for the functionality of the PCM interface and the mapping of the voice path and the tone generation path into the single hardware resource at the codec. The actual switching could be done in the software domain by selecting PCM data from the relevant buffer. Or it could be done in hardware by implementing a logical switch for the data source going over the PCM interface. The software option is easier to provide in the platform if the more general case is considered.

The software architecture is best represented as a stack diagram shown in Figure A5-12.

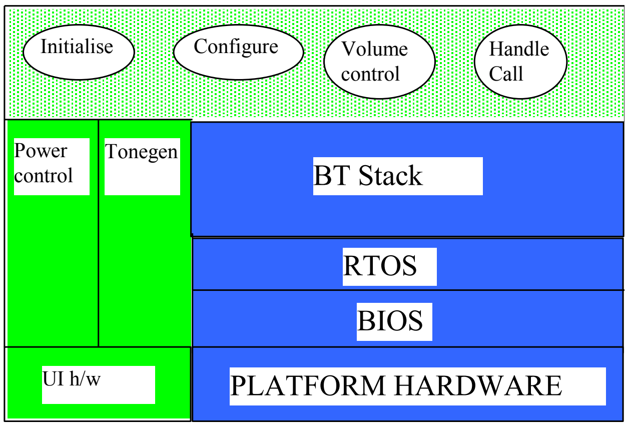


Figure A5-12. BlueStone software architecture.

The software included as part of the Bluetooth platform forms the bulk of the functionality. This sits on top of a Basic Input Output System (BIOS) and the Real Time Operating System (RTOS). The BIOS contains driver code for the hardware contained in the platform. More important is the existence of packaged know-how which allows the platform user (Bolderbits) to add hardware and software functionality. This is shown on the left and above the BT Stack software. The top layer in the software stack diagram represents the application developed by Bolderbits to meet the needs of their customer. It is here that the integration with the software and the handset is realised. The software above the User Interface hardware (UI h/w) implements the drivers for the added hardware, such as the codec and the power control hardware. Software interfaces exist at the boundaries of the various boxes in the stack diagram. For example the "Tonegen" block or module presents an application programmer interface (API) to the top level code. This API may be facilitated by the programming language environment, i.e. through the function call mechanism or via an operating system function such as message passing. The platform includes guidance for how the user should apply these in the implementation of their system. For example, the power control function is affected by a number of variables that might not all be implemented in one software unit. In this case the message passing mechanism is the best to use.

As was highlighted earlier, the platform imposes certain constraints on the system designer in order to preserve its invariant. For example, part of the invariant of the Bluetooth platform is: the system complies with the Bluetooth standard. One constraint that has to be met is that the CPU has enough processing power available to fulfil the tasks that implement the BT Stack. On the hardware side the RAM is of a fixed size in the embedded controller for a given variant of the platform. The selection of variant is therefore a function of the data usage of the application software. This is another example of the interaction across the abstraction layers in the System Design Process.

3. SDP AND SUD INTERACTION

Now that the design process and the SUD have been described it is possible to provide an overview of the relationship between SDP and SUD. This is depicted in Figure A5-13.

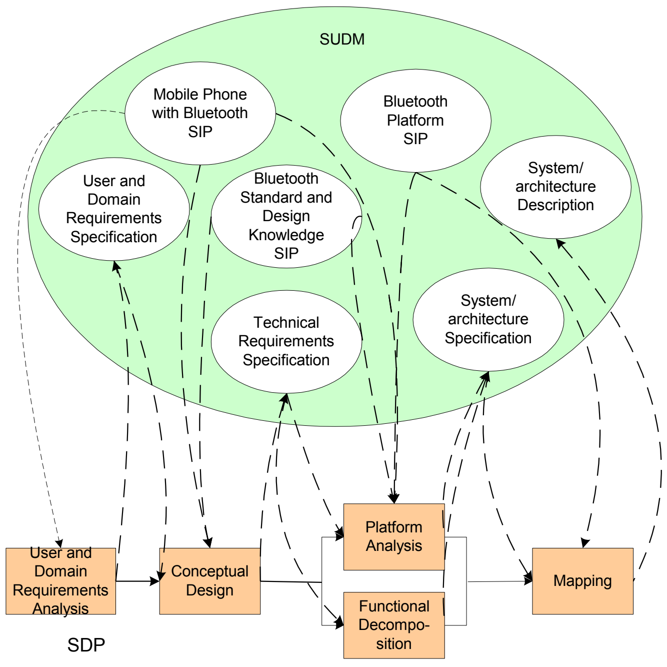

SUDM
Mobile Phone with Bluetooth SIP
Bluetooth Platform SIP
System/ architecture Description
User and Domain Requirements Specification
Bluetooth Standard and Design Knowledge SIP
Technical Requirements Specification
System/ architecture Specification
User and Domain Requirements Analysis
Conceptual Design
Platform Analysis
Functional Decompo-sition
Mapping
SDP

Figure A5-13. BlueStone SDP/SUDM linkage.

4. SUMMARY

This chapter has described a walk-through of an artificial but realistic example in order to show how the SDCM can be instantiated in practice.

When applying the SDCM, the user is expected to instantiate both the design process, modelling methods, languages and the specific artefacts according to the needs of her/his organisation. This requires effort from the user, but the payback will come from improved reuse capability of the organisation. This example has tried to show some aspects of the instantiation of the SDCM.