Building Product Populations with Software Components

Ommering, Robbert Christiaan van

Rijksuniversiteit Groningen

Building Product Populations with Software Components

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. F. Zwarts,
in het openbaar te verdedigen op
vrijdag 3 december 2004
om 14.45 uur

door

Robbert Christiaan van Ommering
geboren op 25 september 1958
te Geldrop

Promotor	:	Prof.dr.ir. J. Bosch
Beoordelingscommissie	:	Prof.dr. D. K. Hammer
		Prof.dr. J. N. Magee
		Prof.dr. J. C. van Vliet

ISBN 90-74445-64-0

Table of Contents

Chapter 1

Introduction

1.1 The Problem

This thesis studies the creation of software embedded in consumer products such as televisions. The study follows two empirical observations [13], both a consequence of Moore's law [62]:

- Embedded software in consumer products doubles in size every two years.
- *All* consumer products will eventually embed software and follow the same growth curve; only the starting point in time may be different.

As a result, consumer electronics manufacturers such as Philips face the following three challenges:

- How do we ensure that the *quality* of the embedded software remains high while the size and complexity of the software is growing?
- How can we build a large *diversity* of products without unnecessary duplication of effort?
- How can we decrease *time to market* to make sure that we are still (one of) the first to introduce new products?

Ongoing miniaturization and subsequent cost reduction induce a fourth challenge, that of *convergence* between thus far different products:

- Can we *combine* the functionality of existing but different products to create interesting new products, without having to re-implement the software?

This thesis seeks answers to these four challenges. In Chapter 1, we examine the business perspective and the technology trends, leading to our research questions. Chapters 2-8 contain the results of our research in the form of papers published at conferences or in journals. Chapter 9 discusses these results in terms of the original research questions.

1.2 A Business Perspective

Philips televisions have embedded software since 1978. The amount of software is growing roughly with a factor of two every two years. Figure 1 illustrates this graphically for high-end televisions. This growth is not specific for high-end TVs; it also holds for low-end TVs, for video recorders, DVD and CD players, and it is likely to hold for (mobile) phones, digital cameras, washing machines, and shavers as well. The only difference is a shift in time of the growth curve.

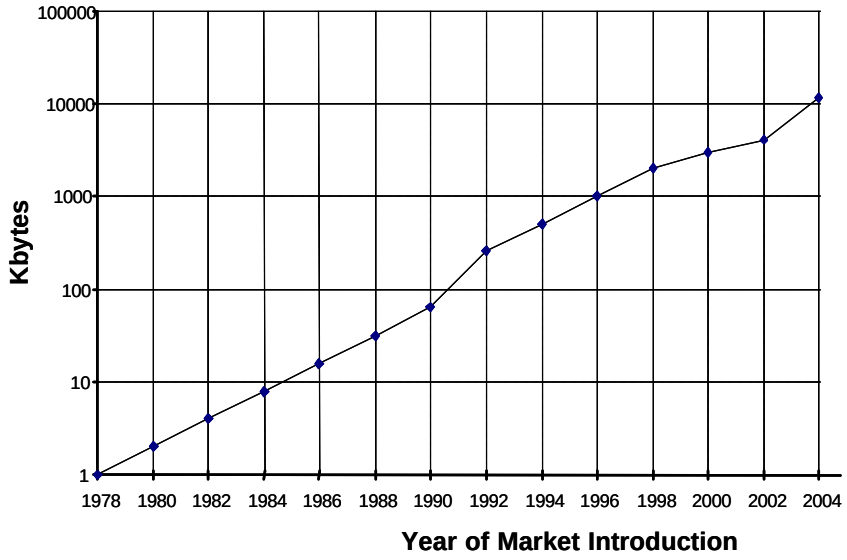

Figure 1. Size of embedded software in high-end televisions.

We expect that every consumer product manufacturer will eventually face the problem of having to build an ever-growing amount of software, and thus becoming – at least partially – a software company [95]. We believe the television is the forerunner in this: a TV with embedded microcontroller entered many households *before* the PC. For that reason, we took the TV as subject of our study.

The first challenge is that with size, complexity grows. Development that starts as a single person activity soon grows into a team effort. Software for current TVs requires more than a hundred developers and more than two years to develop. Clearly, to bring this to a good ending, there must at least be a good architecture and an efficient development process.

But this is not all. Each of the products mentioned above (TV, DVD, mobile phone, washing machine) is not alone, but part of a *family* of strongly related products, differing in price, featuring, supported standards, cultural preferences, and other criteria. Open any catalog of any manufacturer, and you will find at least a dozen different products from which you can choose. Through such a product portfolio, the manufacturer aims to obtain a large market share. The technical consequence of

this is that the software architecture and development process must be such that variants of the product can be made with little effort and as quickly as possible.

The product portfolio is not static but changes over time, setting or following the market trends. This means this it must also be possible to create *new* versions of the product in a short time. As explained above, the initial development of the software for a new television costs more than two years. Commercially, a new range of televisions must come out every year, synchronized with Christmas shopping and major annual sports events. As a more recent trend, this *time to market* must still be decreased: if new products do not come out every 6 months, manufacturers loose shelf space in the shops.

The fourth challenge is that miniaturization and cost reduction enable the creation of integrated products that combine functionality of thus far separate products. A striking example in 2003 was the printer/scanner/copier, of 2004 the mobile phone with built-in digital camera. The original example that was the direct cause of our study was the TV-VCR combination, now largely replaced by the TV-DVD combination. Note that the combination provides functionality beyond that of the parts: e.g. for the phone, taking a picture and sending it to friends or family.

Creating ‘combi’ products is difficult for two reasons:

- It is hard to predict which combinations will be successful beforehand.
- Development of the combined product crosses organizational boundaries.

These effectively rule out the creation of an overall architecture from which the separate and combined products can be derived. The efficient creation of a *product population*, a set of product families from which combination products can be derived, is the main topic of this thesis.

1.3 Technology Trends

This thesis builds on three main trends in software engineering in the past decade:

- Software *architecture* is increasingly recognized as an essential ingredient for building large and complex systems.
- The ‘old’ idea of reusable software *components* is revived through the creation and successful introduction of several component technologies.
- Software *product lines* now receive systematic attention.

Brooks already mentions *architecture* in 1975 as an important element for building large systems [18]. However, it is not until the late 1980s that software architecture becomes a discipline of its own. Shaw [97] pleads for higher-level abstractions than programming languages and abstract data types to build larger scale systems. Schwanke, Altucher and Platoff [96] define architecture as the set of allowed connections between software units, and propose ways to automatically verify an

implementation against an architecture. Perry and Wolf [86] propose a foundation for the study of software architecture, and name multiple views and architectural style as important elements. In 1995, Soni, Nord and Hofmeister propose four architecture views [101], while Kruchten proposes 4+1 views [47]. A special issue on software architecture of the IEEE Transactions on Software Engineering [40], and a book on software architecture by Shaw and Garlan [98] are a further landmark in the acceptance of software architecture as an important discipline.

The idea of reusable *components* is as old as 1968, when McIlroy advocated mass produced software components [54]. However, the *essence* of components: a deployment independent of other components and of the systems in which the components are being used, was not realized until the 1990s, except for special cases such as drivers for an operating system and mathematical and graphical libraries. In 1990, Microsoft's COM started as a foundation for creating compound documents (OLE) [113], and was soon applied to automating programs (DDE) and to implement Visual Basic controls (later called ActiveX) [17]. Around 1996, OMG defined the CORBA Component Model to build large distributed applications. Sun invented JavaBeans as building blocks for Java [42] programs. There is now an active component based software engineering community, and a recent focus of attention is the study of non-functional properties of systems built from components.

Where the component community generally takes a bottom-up view on system construction, the *product line* community essentially takes an integral view on the building of families of products. Already in 1972, Parnas studied the creation of families of programs [85] (spending one year in Philips at the time). Non-software product lines are almost as old as factories; software product lines have also been around for quite some time, CelsiusTech (formerly Philips) being a famous example [21]. Around 1995, the Software Engineering Institute started a program on software product lines [100], the ARES project studied product families [3], and major conferences held sessions on product lines [87]. Most researchers emphasize the analysis of commonality and variability at an early stage, to build systems with a sufficient number of variation points to implement different products; as a result, the product line and component communities are quite disjoint.

Other fields have a relation with our work too, most notably *partial evaluation* and *configuration management*, but we shall not discuss them here. Their role will become evident in the following chapters.

To relate our problem statement to the trends described in this section: we want to build *high quality* software for a *family* of consumer products in a *short time*, and we require an approach that can be extended to build product *populations*, a set of product families from which we can derive 'combi' products. Architecture will help us to manage the complexity and thus the quality of the software, while reusable components (bottom-up) and software product lines (top-down) will help

us to efficiently build families and populations. Figure 2, adapted from [81], illustrates this graphically.

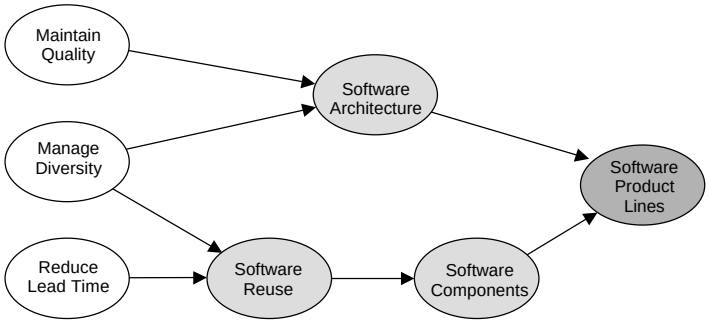

Figure 2. Main drivers for software product lines

1.4 Research Questions

An architecture will only help to improve the quality of a system if that architecture is *explicit* so that it can be analyzed (or at least discussed), and if it *consistently* describes the implementation at some level of abstraction. Much of our earlier work deals with formalizing architecture and verifying implementations against it.

The first in this field was Warshall [110], using an efficient algorithm to calculate the transitive closure of a binary relation. Schwanke, Altucher and Platoff [96] extracted architecture from code and represented it as graphs, so that they could compare the *actual* architecture against the *intended*. They also define aggregation (part-of) relations so that they could inspect the use relation at higher levels. We pursued a similar route in the early 1990s [68][26], using relation algebra to provide a mathematical basis [27]. Holt [35] followed essentially the same approach. Our work finally culminated in an experience paper [15] that shows that verification of architecture *can* be made successful but not without continuous attention. A more promising way is to use an architectural description language in *forward* mode, i.e. to create explicit descriptions of architecture and generate code from that. Darwin [52] is an example of such a language.

To build product families and populations, we need a powerful mechanism to deal with commonality and diversity, and we believe – with many others - that software components provide a solution. A closer investigation reveals two complementary ways to use components for this, as illustrated in Figure 3. One way is to build a product independent framework in which product specific components can be plugged. Another way is to build product independent components that can be combined in product dependent ways. The first way we name *variation*, the second way *composition*. Many researchers in the product line community favor variation, while the component community favors composition. Variation may be good to build families, but we believe that composition is needed to build populations.

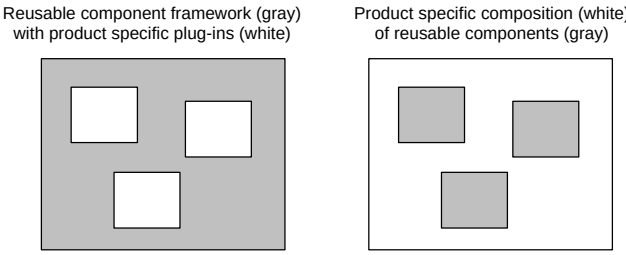

Figure 3. Variation (left) and composition (right) are complementary approaches.

When we started this work in 1996, the available component models were Microsoft's COM, CORBA's component model and JavaBeans. The latter was tied to a Java virtual machine, which we did not have in a television. CORBA was deemed unsuitable for an embedded system, so COM was the only option. Especially the *QueryInterface* concept of COM was useful, as it allows to implement new interfaces while still supporting the old ones, a feature necessary for evolution. The downside of COM is that VTables provide code size and run-time overhead.

So we had two choices: wait 5 years (estimated) for the computing power of a TV to be such that COM can be used, and bridge the gap with other techniques for handling diversity, or adjust the technology so that it fits in a TV, and start gaining experience with component based design. We chose the latter as we believed that components were the future, and we wanted to tune our software development process and organization to component based design as soon as possible. The resulting component technology is called Koala [70].

Summarized, here are the four research questions that we formulated in 1996, and that are the topic of this thesis.

- Can *software architectures* be made more *explicit*, and does this increase the quality of products?
- Can *component technology* help to build product *families* and *populations*, and what design patterns are needed for that?
- Can component technology be applied in *resource-constrained products*, and what consequences does this have on the technology?
- Can this all be made into a business success, and what impact does this have on development *process* and *organization*?

We believe that each of the questions above warrants a study on itself. The value of this thesis is not as much in answering the questions individually and in depth, but rather in combination and integration. We present a method that addresses the four questions above, and that is being successfully applied at Philips at a large scale.

1.5 Way of Working

Research in software engineering differs from research in other fields of computer science in at least two aspects:

- *Scale* is an important source of complexity, making it difficult to conduct realistic (but necessarily small-scale) experiments in the laboratory.
- *Validation* of results is difficult as many factors are involved in the success or failure of a software project.

With respect to scale, Basili distinguishes researchers from practitioners [7]. The researcher tries to *understand* the nature of processes and methods to build systems, while the practitioner *builds* improved systems with this knowledge. The one cannot live without the other: the researcher needs a laboratory to observe and manipulate variables, but these only exist where practitioners build systems. Conversely, the practitioner needs to better understand how to build systems, and the researcher can provide models to help.

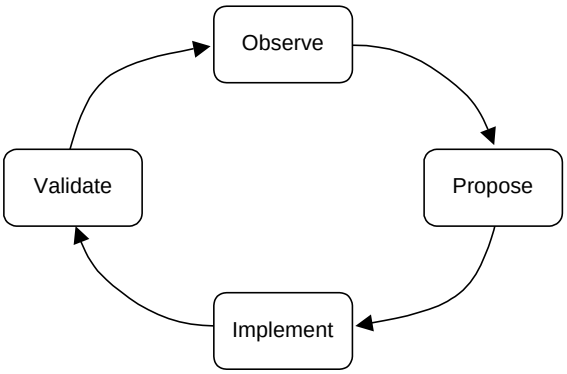

Figure 4. The empirical cycle.

For this reason, we adopted an *industry as laboratory* approach in our software engineering research [65]. We seek a close contact with a development group that builds (large) systems, and then follow an experimental paradigm (Figure 4, see also [7]):

- We *observe* the way that the group is developing software, and – in close cooperation with people from that group – identify problems in their way of working.
- We *propose* – again in close cooperation with people from that group – as hypothesis a set of solutions, and try these out 'in the small' at research.
- We then *implement* these solutions in the development group, and attempt to measure and analyze how well the solution is working in practice.

- This way, we can *validate* our hypothesis, and use the information to repeat our experimental cycle, thus improving our knowledge of both the problem and the solution domain.

As a research group, we have used this paradigm with many development groups and often multiple times with the same development group. This thesis describes the third cycle that we had with the group developing software for televisions. The first two cycles had the following questions as topic:

- Can *formal specifications* and *rapid prototyping* help to improve the quality of software in televisions? The answer was yes, though formal specifications turned out to be difficult to transfer into industry [43].
- Can *architecture* help to manage the increasing complexity and size of the software in televisions? The answer was positive, but specific techniques to handle diversity were needed.

The third cycle started with an explicit question from the development group for techniques to handle diversity; components and explicit architectural descriptions were our answer. We anticipated – and later noticed – that process and organization had to change as well to introduce these techniques properly.

Validation is the other difficult aspect of software engineering research mentioned above. The problem is that the success of any software development project is determined by many variables, among which a number of human factors. It is very difficult to conduct systematic research to the effects of these variables in large-scale projects; it certainly is not feasible to repeat an experiment with other values for a selected set of variables. In fact, in a software business that grows according to Moore's law, *any* new project is bigger than *all* projects before, and therefore a first.

One of the techniques mentioned in [7] is a project-based study, as opposed to a human factors based study, where through variation of teams (*who* is building) and variation of projects (*what* is being built), conclusions can be drawn. As it happens, the method described in this thesis is being applied by over a dozen different teams that operate relatively independently, at different places in the world (Europe, US, Asia). In addition, the method has been applied to three different hardware chassis and two different market segments, each with their own characteristics. As such, our research is more of a *field study* than a *case study*.

Although we really need a *quantitative* validation of the method, a good first step is to have a *qualitative* validation. Quantitative studies are objective and oriented towards verification [7], but even though qualitative studies are subjective, they can help to discover important issues. As a qualitative validation, we can ask the following questions:

- Is the method that we devised *transferable* to industry (a necessary but not sufficient condition for success)?

- Is the development team that uses the method *successful* in building a product line (again, necessary but not sufficient)?
- Are the developers *enthusiastic* about the method, and do they feel that it has helped them to solve their problems?
- Does management believe that the method is instrumental to their success in the future?

A method should be wider applicable than one organization and one domain:

- Can the method be used by other organizations or companies, and/or in other sub domains or domains?

This requires a significant amount of effort to investigate it. A simpler question (again a necessary but not sufficient condition) is:

- Can you identify other teams that experience problems (and are aware of them) that can be solved by the method, according to you and/or the team?

The final qualitative questions (again, none of them conclusive) position the work in software engineering research:

- Are papers on the method accepted at conferences and in journals?
- Do others researchers refer to these, and do they build upon the work?

There are also quantitative measurements that we can perform. We can count the number of products and components and calculate how components are reused over products. We can measure the stability of the code base, and see whether components are reused 'as is' or continuously being modified. We can measure the stability of the interfaces, and determine whether interfaces indeed form stable points in the evolution of the system. All these numbers provide insight, but they would really gain in value when comparing them to other approaches in other companies. Unfortunately, that is outside the scope of this thesis.

1.6 Time Line

Preamble to this study were the following:

- We studied the use of formal specification techniques and rapid prototyping to increase the quality of software in televisions in 1988-1993 [43][69].
- We studied the formalization, visualization and verification of architecture in 1992-1996 (see Chapter 2 for a summary). This work was continued by colleagues until 2002.

The work described in this thesis started mid 1996. The Koala component model was defined in the period September 1996 to March 1997. The Koala compiler was implemented in the period January 1997 to June 1997. The original plan was to

apply Koala to the development of television software in the summer of 1997. Due to other circumstances, this was cancelled, and research received the assignment of designing a new component-based software architecture for televisions in 1998. This architecture, baptized MG-R, created its first product in 2000. By 2002, *all* mid-range and high-end television products were based on MG-R.

The origin of the name Koala is depicted in Figure 5. COLD, IGLOO, ICE and ICE BEAR were a language, a library, a development environment and a graphical user interface respectively for writing formal specifications. Polar and Panda were a graphical language and a graphical editor for COLD. Teddy and Ursa are tools for architecture visualization and verification. Darwin is an ADL for specifying distributed systems, and Kangaroo is an ADL for describing user interfaces.

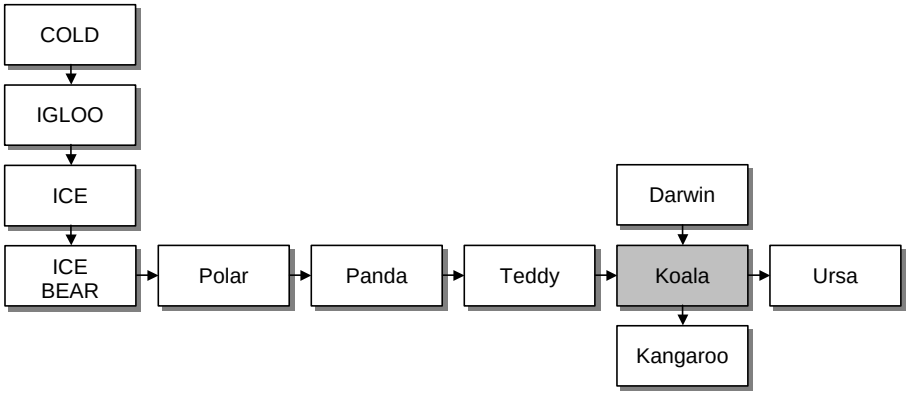

Figure 5. Derivation of the name Koala

1.7 Overview of this Thesis

The main body of this thesis consists of articles that have been published in major journals (chapters 2, 3, 8), conferences (chapters 4, 5, 7) and workshops (chapter 6). The papers have been left ‘as is’, with the exception of some minor error corrections and typographical changes. The articles are provided mostly in chronological order, but where necessary, we deviate from this order to enhance the logical flow of the thesis.

For a quick overview of Koala, read Chapter 3. For a quick overview of the use of Koala to build software for televisions, including process and organization issues, read Chapter 7.

Chapter 2, published in *Computer Languages* in 2001 [79], describes the work on the formalization, visualization and verification of architecture that we did prior to the work on Koala. It shows how familiar concepts in architecture (layers, subsystems) can be provided with a mathematical basis, and how then tools can be built to visualize such structures and to verify whether implementations actually conform to these structures. The conclusions from this work were twofold:

- It is very useful to have a formal notion of architecture, especially if there is a graphical notation accompanying this.
- It is possible to verify an implementation against an architecture 'after the fact', but it is then often too late to correct faults in either implementation or architecture.

This work therefore led to our first research question: can we make architecture explicit beforehand?

Chapter 3, published in *IEEE Computer* in 2000 [70], postulates complexity and diversity as the main issues in the development of software for televisions, and proposes an explicit description of architecture and the (re-)use of components as solution. Koala is then described in two steps: first the basic model (components, interfaces, modules, binding) and then the extended model (function binding, diversity interfaces, switches and partial evaluation).

Chapter 4, published at the *Working IEEE/IFIP Conference on Software Architecture* (WICSA) in 2001 [77], discusses independent deployment as the bare essence of software components. This paper was the direct result of many discussions within Philips, where many development groups claimed to already use components, but in a decomposition paradigm, so in a dependent deployment. The paper introduces a two-component system consisting of a client and a server, where each component can vary in two dimensions: time and space. The resulting four steps are discussed in detail, with ample examples. This material forms the basis of the design of Koala, although we never made this explicit until writing this paper.

Chapter 5, published at the *Second Software Product Line Conference* (SPLC-2) in 2002 [81], explains a topic left open in Chapter 3: why composition is the best paradigm to build consumer products, rather than variation. The answer lies in the intended scope of the software product line. Many familiar examples of variation and composition are given.

Chapter 6, published at the *Tenth International Workshop on Software Configuration Management* (SCM-10) in 2001 [76], compares the use of an ADL to handle diversity with the more traditional approach of configuration management, and clearly separates issues to be handled in the ADL domain from issues to be handled by configuration management tools.

Chapter 7, published at the *International Conference on Software Engineering* (ICSE) in 2002 [80], contains a full overview of our approach, and as such, it is representative for – and has the same title as – this thesis. The paper is structured in terms of Business, Architecture, Process and Organization aspects; one of the important lessons that we learned is that these cannot be treated in isolation. Put more strongly: a component-based architecture is only a success if it serves a clear business goal and it is accompanied by a matching development process and organization.

Chapter 8, published in *Software Practice and Experience* in 2003 [83], answers an important question in the design of (low-level) control software for televisions: how to make the control software ‘composable’. It demonstrates an architectural style of controllers that communicate horizontally to guard the integrity of the signal path. An efficient implementation technique of direct function calls makes this work in a resource-constrained environment.

Finally, Chapter 9 provides a validation of the work reported in this thesis.

Chapter 2

Formalizing Software Architecture

Published as: *Languages for Formalizing, Visualizing and Verifying Software Architecture*, Rob van Ommering, René Krikhaar, Loe Feijs, Computer Languages 27 (2001), p3-18.

Abstract: In this paper we describe languages for formalizing, visualizing and verifying software architectures. This helps us in solving two related problems: (1) the reconstruction of architectures of existing systems, and (2) the definition and verification of architectures of new systems. We define an expression language for formulating architectural rules, a graph language for visualizing various structures of design, and a dialogue language for interactively exercising the former two languages. We have applied these languages in a number of industrial cases.

2.1 Introduction

A well-defined software architecture is an essential element for complex software systems. Ideally, such an architecture is defined in advance, i.e. before the implementation is started (forward architecting). In practice, however, this is not always the case, and often an architecture needs to be reconstructed from an implementation (reverse architecting). Reverse architecting and forward architecting have certain problems in common. Consider the following two scenarios:

- *Reverse architecting*: a software architect is updating an existing software system to fulfill some new requirements. He wants to study the architecture of the software, but no explicit description is available. How can he *extract* the architecture from the implementation, and how can he *verify* that the induced architecture is indeed the one that the implementation satisfies?
- *Forward architecting*: a software architect has just created a *new* software architecture, by defining architectural concepts and rules, together with a decomposition of the software into layers, subsystems and/or components. A large team of software engineers is currently implementing the architecture.

How can he ensure that they indeed follow the rules and decomposition? Can these rules be *formalized* and then be *verified* automatically?

We are concerned with the formalization of software architectures to automatically verify implementations against the architecture. This is useful in reverse architecting, to check whether an extracted architecture indeed matches the implementation. This is also useful in forward architecting, as experience shows that software engineers do not always obey the prescribed architecture. Verifying the implementation against the architecture can then improve the quality of both the architecture and the implementation, as in case of differences, either the architecture or the implementation can be assumed to be wrong.

The approach that we have developed employs three languages: an *expression language* that we use to formulate architectural rules, a *graph language* that we use to visualize design structure, and a *dialogue language* that allows us to experiment interactively with the other two languages. The expression language is based upon an algebra of binary relations in which partitions play an important role. The graph language provides visual representations of graphs with a variety of graphical attributes. The dialogue language uses a desktop metaphor with among other things a relational calculator and various 'drag and drop' techniques to allow for easy experimentation with the first two languages.

This paper is organized as follows. In the section 2.2, we provide a birds-eye view of our approach, based upon a simplified definition of software architecture. Sections 2.3-2.5 discuss the three languages mentioned above. We end this paper with some concluding remarks.

2.2 Formalizing and Verifying Software Architecture

Software architecture can be defined as *the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution* [39]. Basically, this implies that an architecture is an *abstraction of systems* [5]. In this paper, we define architecture as a *specification of a design*, where a design is an abstraction (read: the 'structure') of an implementation. Note that multiple designs can satisfy a single architecture; certain design details can be left as 'implementation freedom' for the designers. Note also that multiple implementations can share the same design; certain coding details can be seen as implementation freedom for the software engineers.

Our definition has operational semantics. An architecture can be defined in a 'legal document', and for every implementation, an independent party can establish whether it satisfies the architecture or not. In agreement with our definition, this proceeds in two steps (see Figure 6). First, the design structure *d* is abstracted from the implementation *i* using an abstraction function *f*. Second, this design *d* is verified against the architecture *a* using a 'satisfies' operator $\subseteq$.

Two domains play a role here, the implementation domain *I*, and the architecture and design domain *D*. We shall treat the architecture and design domain formally, but we shall deal with the implementation domain *I* in an ad-hoc way. Abstracting a design structure *d* from an implementation *i* is a mechanical process, which we implement in the form of a large number of extraction tools. Each tool processes the implementation and reconstructs one design aspect in the domain *D*. Each tool has an implementation that is specific for the design aspect and also for the implementation properties such as programming language and file system conventions. Note that although we do not have a full and formal model of the implementation, we do construct an accurate abstraction of the implementation, which we call the *design*.

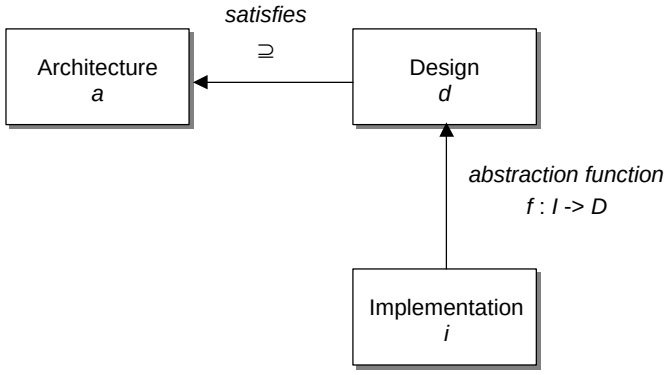

Figure 6. Relation between Architecture, Design and Implementation

Checking a design against an architecture can also be a mechanical process, at least for that part of the architecture that can be formalized. Note that architecture and design are expressed in the same language. In the next section we shall define this language and show an example of the operator $\subseteq$.

It is important to note that we have applied these techniques to those aspects of software architecture that are sometimes called the *modular architecture* [101], i.e. the static decomposition of a system in terms of layers, units, components, files, functions, et cetera. The techniques can in principle also be applied to more dynamical architectural structures, but discussion of that is outside the scope of this paper.

2.3 The Expression Language

The expression language was designed with the following requirements in mind:

- It should allow us to formalize design structures and architectural rules.
- It should be generic so that we can use it for multiple aspects of design and also in multiple application domains.

- It should be possible to build generic tool support for it, so that we can concentrate on software architecture and not on building aspect- or domain-specific tools.

Our expression language is an algebra based on sets, binary relations and a special kind of binary relation called part-of relations. This algebra is called the relation partition algebra (RPA). The difference between part-of relations and partitions is subtle and discussed in more detail elsewhere [29]. We shall give a brief overview of RPA in the next sections.

2.3.1 Sets

The algebra of *sets* is well known. We can calculate the union $S_1 \cup S_2$, the intersection $S_1 \cap S_2$, the difference $S_1 \setminus S_2$, and perform many other operations on sets. There are a large number of algebraic laws that describe the behavior of these operators. Some operations impose some problems, like the complement of a set, but since we always operate in finite and known universes, these problems are not fundamental.

We use sets to model the entities in modular architecture. Examples of such entities are functions, files, components, subsystems, layers, et cetera.

Creating tool support for set operations is straightforward: many libraries offer such facilities. In principle, sets can be created for different types of objects. Our implementation is simpler: we can only deal with sets of *atoms*, where an atom is an object with a unique name but without further content. If we want to assign other properties to objects, we use binary relations, as will be explained in the next section.

2.3.2 Relations

A binary *relation* R on sets S_1 and S_2 can be defined as a set of pairs of elements, i.e. $R \subseteq S * S$. As such, binary relations inherit all the algebraic properties from sets. We can for instance calculate the union $R_1 \cup R_2$, the intersection $R_1 \cap R_2$, the difference $R_1 \setminus R_2$, and perform many other operations. But there is a second side to relations. By viewing them as a mapping from one set to another set, we can calculate the inverse mapping R^{-1}, we can compose two relations into a new relation $R_3 = R_2 \circ R_1$ (R_2 applied *after* R_1), and we can calculate the transitive closure R^* of a relation R. Also, we can calculate the domain *dom*(R) and the range *ran*(R) of a relation.

The algebra of relations has also been studied intensively. A well-known application area is relational databases. The corresponding query languages form in fact an implementation of the operations on relations. A large part of our work on architecture formalization and verification can indeed be implemented in the form of a database (containing the intended and extracted design information) and

queries on the database (formalizing the architectural rules). We shall come back to this in section 2.6.2.

We use binary relations to model various structures in software architecture. Examples are the call relation (or call graph) between functions, the include relation between files, and the use relation between modules or components. Also we model the decomposition of a system with various part-of relations, such as 'functions being part-of files', 'files being part-of components', et cetera. Thirdly, as mentioned in section 2.3.1, we use relations to add properties to objects. A file we usually represent as an atom with as name the path of the file. Other properties, such as a time stamp, are represented as binary relations (actually a function) that relate the files to time stamps.

Our own implementation of relations is based on pairs of *atoms*, the same atoms as mentioned in section 2.3.1. This allows us to handle very large relations, say in the order of 10^6 pairs. Call graphs in industrial systems often have such sizes.

2.3.3 Part-of Relations

A special kind of binary relation is a *part-of relation* that defines which entities are part-of which other entities. In this paper, we only consider functional and acyclic part-of relations. A typical part-of hierarchy in software architecture contains functions, files, components, subsystems, layers, and the system as root. Part-of relations inherit all operations from sets and relations, so again we can take the union $P_1 \cup P_2$ of two part-of relations, and also compose part-of relations: $P_2 \circ P_1$. But the specific interpretation of a relation as a part-of relation also gives rise to new operations such as lifting.

We define the *lifting* operator $\uparrow$ as follows: $R \uparrow P = P \circ R \circ P^{-1}$. The interpretation behind this is the following. If $R \subseteq F*F$ is a call graph on the set of functions F, and $P \subseteq F*M$ is a part-of relation that defines in which module M a function F is defined, then $R \uparrow P$ is the induced usage relation between modules. In other words, $R \uparrow P$ represents the following *uses* relation:

> A module m_1 *uses* a module m_2 if and only if there is a function f_1 in m_1 and a function f_2 in m_2 where f_1 *calls* f_2.

An example of lifting can be seen in Figure 7. The left-hand side of this figure shows the functions *main, a, b, c* and *d*, and the modules *Appl*, *DB* and *Lib*. Through nesting is shown which functions are part-of which modules. Arrows between functions denote the function call graph. The right hand side shows only the modules, and the usage relation between the modules, obtained by lifting the call graph using the part-of relation.

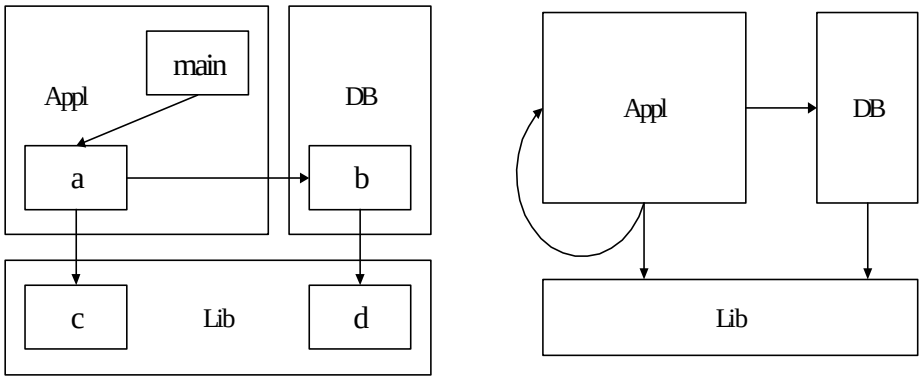

Figure 7. Lifting functions calls to module level

2.3.4 Architectural Rules

The example in Figure 7 allows us to introduce an example of an architectural rule. Suppose that the right-hand side of Figure 7 represents the allowed usage relation U between modules, as defined by software architects. The left-hand side represents the actual call relation C between functions, and the part-of relation P. The architectural rule states that any call between functions should obey the allowed module usage relation. The rule can be formalized as follows:

$$C \uparrow P \subseteq U$$

Note that module *Appl* uses itself in the right-hand side of Figure 7. In general, we are not interested in function calls *within* a module, so the architectural rule can be reformulated as:

$$(C \uparrow P) \setminus I \subseteq U'$$

where I is the identical relation on modules, and U' is the allowed usage relation but without self references.

2.4 The Graph Language

Design structures can be represented as *directed graphs*. In this section we define graphs and views on graphs. We also show how we can extract design structures from implementations and visualize them.

2.4.1 Graphs and Views on Graphs

We define a *graph* G as a tuple (S, R), where S is a set of objects and R is a relation on S*S. Similar to the explanation in section 2.3.1, objects are identified by their name, which is a simple string of characters. Apart from their name, objects have no further content.

We define a *view* V on a graph G as a tuple (S', R', L) where:

- $S' \subseteq S$
- $R' = R \lceil_{car} S'$
- L is a layout function that assigns a position to each element of S.

The symbol $\lceil_{car}$ means *carrier restrict*, and produces that subset of the relation R that contains all pairs in which the left- and right-hand side are both element of the specified S'. In a formula:

$$R \lceil_{car} S = R \cap (S*S)$$

A view is a *visualization* of *part of* a graph. The visualization is obtained as follows:

- For each object in S' a rectangle with standard size and shape is drawn at the position specified by the layout function. The object's name is drawn inside the rectangle.
- For each pair of objects in the relation R', a straight line is drawn that connects the centers of the rectangles (but the line is drawn *behind* the rectangles).

An example view of part of a design structure is shown in Figure 8.

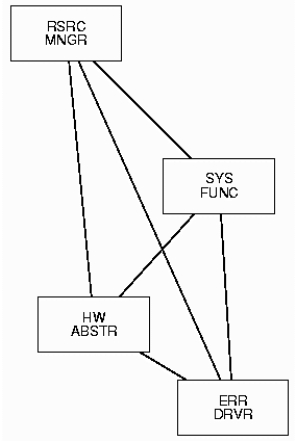

Figure 8. A simple graph showing a design structure

It is interesting to note that we demand a certain completeness of our views. Although the set of objects in a view may be any subset of objects in the corresponding graph, we show a line between objects if and only if the pair of objects is in the relation. Therefore, we never hide a line between objects just to 'clean-up' the picture, a technique often applied when drawing informal pictures of designs. But we can systematically filter out certain connections, using the operations of RPA. An example showing this is beyond the scope of this paper.

A second remark is that in some cases we draw lines instead of arrows between the boxes. Many relations in software architecture are (or should be) a-cyclic. When we draw such relations, our 'normal' drawing direction is top-down. If an edge is to be drawn in upward direction, it is marked in a special way, e.g. by making it thicker. This allows us to spot anomalies in our relations easily. Note that the architectural diagrams drawn in [105] are similar to our pictures.

2.4.2 Teddy

We have built a tool called Teddy [68] that takes a view V consisting of a set D, a relation R and a layout L as input, and that visualizes this view. Figure 9 shows an example use of this tool. As a side note, Teddy uses the term *domain* for sets in its object model for implementation reasons (*Set* is a keyword in the language in which we have implemented Teddy).

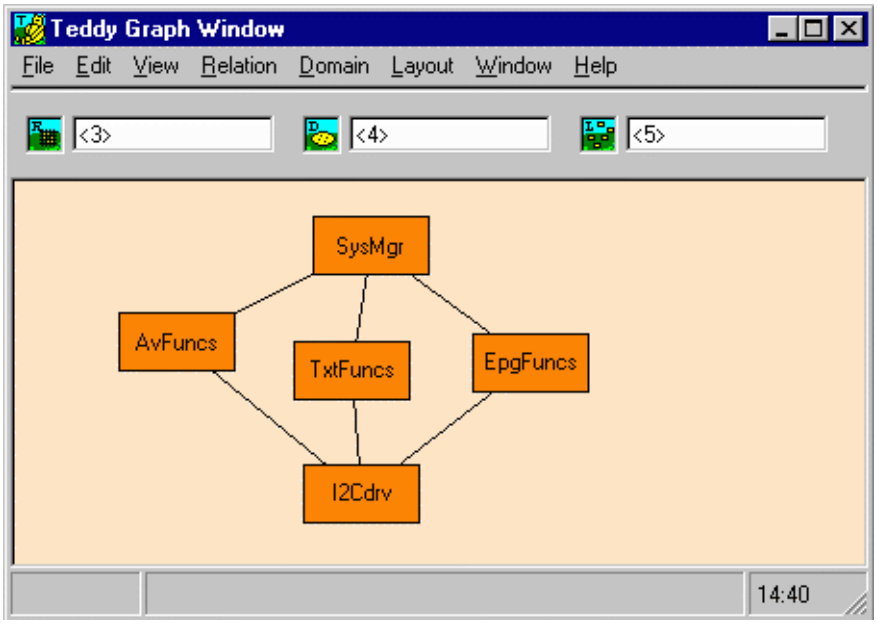

Figure 9. Example use of Teddy

In Figure 9, the three input fields marked R, D and L contain the actual relation, the set and the layout being viewed. Teddy allows the relation R and the layout L to be 'larger' than the set S, as it performs internally a carrier restrict operation (as explained in section 2.4.1) on R and a domain restrict operation ($L \cap (S^*ran(L))$) on L before visualization. So in Figure 9, both the relation and the layout function may contain more objects than the five objects shown. Widening the precondition of Teddy is a practical measure to enhance its usability.

Teddy can also be used to edit the set D, the relation R and the layout L. Dragging a rectangle will change the layout L. Through menus, objects can be added to or

removed from the view, thus changing the set D. Finally, lines between objects can be inserted or deleted, thus changing the relation R. Changing S and L is a frequent activity during architectural verification. The relation R, on the other hand, is usually derived from an implementation and need not be changed (but this does not hold for all relations).

2.4.3 Decorating the Graph

There are various ways in which we can enhance the visual representation.

First of all, we use the *shape* of the objects in the visual graph to represent the kind of object in the design. For instance, we make a distinction between value-based components (containing simple data types, stateless services such as string compare, et cetera) and state-based components. This allows us to visually concentrate on the latter, and ignore the former in certain design discussions. Sharing stateless components is usually harmless, while sharing components (or objects) with state can result in many forms of inconsistency. We used rounded rectangles for value-based components, and normal rectangles for state based components.

We used *dashed lines* for edges to objects representing value-based components, and *solid lines* for edges to state based components, again to diminish the importance of the former components in the design discussions.

We used *thick borders* for components that are used by components outside of the view, and *thin borders* for the rest. This allowed us to create a view of a *subsystem* in the architecture, and to study the export signature of the subsystem. We could have done the reverse as well, and define a special notation for components that *use* external components versus components that don't, but we did not find it very useful (yet).

We defined the *height and width* of individual blocks (this is an extension of the layout function), to make the pictures correspond more closely to the ''original' design pictures.

Finally, we generalized the usage relation to a *multi relation* (with a usage count per pair of objects - this will be explained further below), and visualized the count using the *thickness* of the lines. This allows us to distinguish between 'important' and 'less important' usage between components (though it should be said in all fairness that sometimes a single call to an important function has more impact on the architecture than a large number of calls to less important functions).

The extensions grew directly out of our efforts in practical software development projects to address immediate needs of the software architects. More extensions are feasible:

- different border types (solid, dashed, dotted),

- fill color and/or texture of shapes,
- border color,
- size of the shape (e.g. representing the code size).

2.4.4 Parameterizing Teddy

The extensions to the graphical language mentioned in the previous section could be hard-coded into Teddy, but of course it is much better if the tool is parameterized over these extensions. We therefore make a distinction between *logical* attributes (such as value-based versus state-based) and *graphical* attributes (such as shape), and allow the user to define the mapping.

One technique for doing this is to use labeled graphs, where each node and edge can have multiple labels. We have chosen for a slightly different approach, which is in line with the expression language discussed in Section 2.3. Graphical attributes of objects are controlled by extra relations that map an object to for instance a shape. Graphical attributes of edges are controlled by using multiple relations, one for each edge type. Although slightly more cumbersome for users, limiting ourselves to sets and binary relations reduces the complexity of the expression and dialogue language.

2.5 The Dialogue Language

Languages are systematic ways of expressing interaction, in our case between users and computers. We have defined the expression language (RPA) for representing design structures and architectural rules, and the graph language for visualizing structural information. The third language, the dialogue language allows to use RPA interactively to create design views, and also to replay the calculations when new data becomes available. An RPA calculator is the dominating tool in the dialogue language, but also other tools such as various viewers are part of the language.

2.5.1 The RPA Calculator

The RPA calculator is a desktop interactive calculator that can execute operations on sets and relations. We used the Reverse Polish Notation paradigm, so the calculator has a small stack of sets and relations, and each operation takes zero or more elements from the stack and pushes the result back on the stack. The stack is made visible on the screen (see Figure 10).

Many operations require specific types of arguments and produce specific types of results. For example, the *domain* operation requires a relation as argument and it produces a set as result. And the *union* operation either requires two sets and produces a set, or it requires two relations and it produces a relation. Therefore,

depending on the types of arguments on the stack, buttons in the calculator are enabled or disabled (both functionally and visibly).

Sets and relations can be stored in files and can be loaded from files. We use a simple file format. A set is stored as a file where each line contains the name of an element. A relation is stored as a file where each line contains the names of two elements in a pair, separated by a space. Choosing a simple format makes the writing of scripts to generate and process the files easy.

Drag and drop is also supported in the calculator. A file can be dragged from e.g. an explorer window and dropped at any (visible) location of the stack. Also, any (visible) element of the stack can be dragged and dropped in an explorer window. The drag and drop paradigm can be used to exchange data between different instances of the calculator, and also between the calculator and the other tools, as discussed in the next sections.

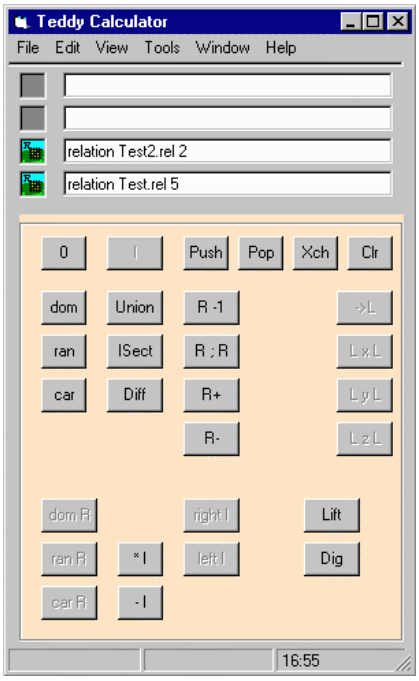

Figure 10. The RPA interactive calculator

2.5.2 The Graph Viewer

We already discussed the graph viewer Teddy in section 2.4.2. The relation, set and layout used by the tools are shown as icons in a tool bar. Using drag and drop, results can be transported from the calculator to the tool bar (and vice versa if necessary). This allows rapid experimentation with data extracted from applications, until the right visual representation is obtained. Needless to say, the

graph viewer can be instantiated as many times as desired. For instance, it is a matter of a few clicks and drags to create a second window that displays the transitive closure of a relation viewed in the first window (shown in Figure 11).

2.5.3 Calculating Layouts

The calculator can perform operations on sets and relations, but Teddy also needs layouts. Therefore, we have implemented a small set of operations on layouts as well. Note that a layout is essentially a mapping of elements to two-dimensional position. It therefore inherits operations on relations and sets, such as *domain* and *domain restrict*.

The operation *union* on layouts needs a further explanation. One aspect is that a layout relation is functional, and so must the union of two layouts be (otherwise an object would have two positions). Another aspect is the location of the resulting objects: merely merging views will probably cause objects to overlap. We therefore defined different flavors of union by considering the bounding boxes of two layouts and stacking them horizontally or vertically, and top/center/bottom (respectively left/center/right) aligned.

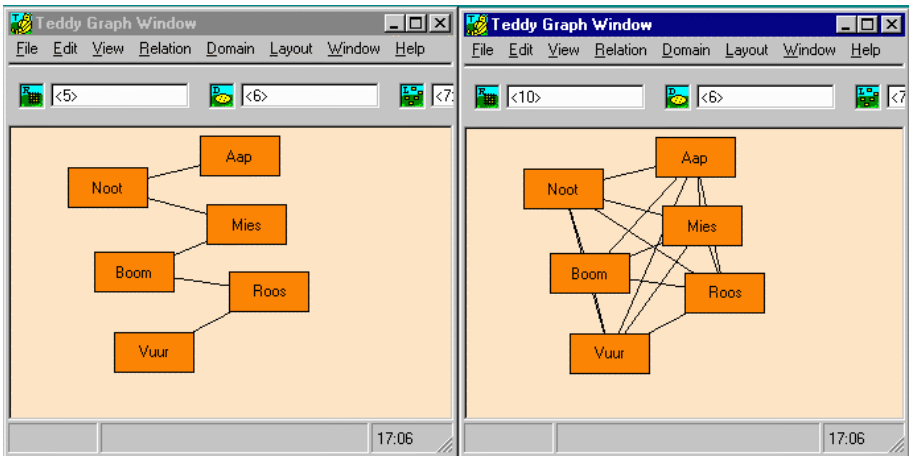

Figure 11. Two graph windows, one showing a relation, the other the transitive closure of this relation

The different union operations allow us to create views of parts of a usage relation by manually adjusting the layout using Teddy, and then to use the union operation to calculate layouts for larger parts of the relation. We can thus create a full overview of a large system step by step.

2.5.4 The VRML Viewer

Drawing graphs in two dimensions will soon result in lines crossing each other. One way of dealing with this problem is by extending the view in the third dimension. We have a special 3D viewer that enables this.

Instead of building 3D viewers ourselves, we rely on existing technology by generating VRML [108] code and viewing that in an appropriate viewer. Of course, we need a 3D layout to generate VRML. We obtain these by generalizing our union operations on layouts into the third dimension, e.g. by also providing a stacking operation in the z-direction. An example can be seen in Figure 12. For more details on our work we refer to [28]. We are not the only ones to represent architecture in with three-dimensional diagrams. See [36] for another 3D visualization of a design structure.

2.5.5 The Module Interconnection Browser

A usage relation can be shown as a graph, but also as a matrix with the elements of domain and range labeling the horizontal and vertical axes, and with ticks in the cells to indicate the pairs in the relation. Moreover, the horizontal and vertical axes can be tree views of the elements in a part-of structure, and the user can interactively expand and collapse nodes. Multi relations can be shown by putting the cardinality of a pair in a cell, instead of a single tick mark. See [14] for more information.

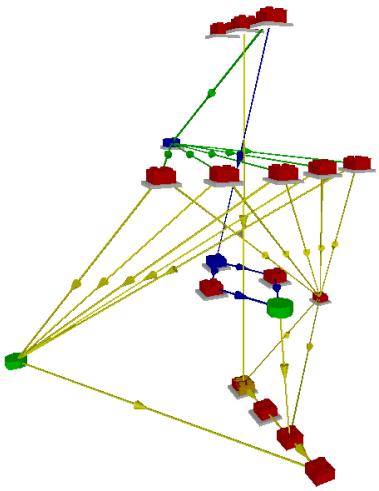

Figure 12. A three dimensional design structure viewer

2.5.6 Recalculating Expressions

The simple use of the calculator and the various viewers involves extracting structural information from an implementation, manipulating this information

using RPA, and viewing the results using various viewers. Sometimes, we want to repeat the calculations with different inputs. To achieve that, the calculator remembers the operation together with the result, so that it can re-evaluate the operation if the inputs change. Also, when interactively dragging results from the calculator to the viewers, it is possible to specify whether to drag *by value* (i.e. an (unlinked) copy of the result is created) or *by reference* (the result is liable to recalculation). We shall give some examples.

Suppose we want to view different relations with the same layout. We can drag the layout used in one instance of the graph viewer *by reference* to the layout icon of another instance of the graph viewer. If we now update the layout in any of the windows interactively, then the other window will be updated automatically.

Suppose that we are viewing a relation in one window, and we use the calculator to calculate the transitive closure of the relation and show it in another window (see Figure 11) using the same layout (and the same subset of nodes). The layout can now be changed in the first window, and the layout of the second window changes immediately. If an edge is added to the first window, the transitive closure is recalculated and immediately displayed in the second window.

As another example, we show a relation in two graph view windows and one 3D window. The two graph view windows show disjoint subsets of the elements, the 3D window shows all elements. The 3D layout is calculated by stacking the 2D layouts in the third dimension. We can now use the two graph view windows to quickly edit the 3D layout.

2.6 Concluding Remarks

2.6.1 Applicability

In this section we want, first of all, to look back and see how the described languages satisfy the needs of software architects as sketched in the introduction. We have the expression language, the graph language, and the dialogue language. We may consider the dialogue language as an extended and interactive combination of the expression language and the graph language.

Both in forward and in reverse architecting it is important to *formalize* rules and *verify* them automatically. A typical formalized rule is of the form R `is-a-subset-of' E, which is the same as R \ E = ∅ (using the RPA operator `\' for relational difference and using the constant ∅ for empty relation). Here we have an expression for the real relation R, as obtained by extraction and further calculations from the source code under development, next to an expression for the expected relation E (for example E may tell which layers are allowed to make direct usage of certain other layers). The verification is done by first extracting R. Most details of this extraction are outside the scope of the present paper and we just refer to [27] for more details. The next step in the verification is to evaluate R and E, perform

the calculation of `\' and look whether the result is empty. Although, in principle, this concludes the verification, we learned from practice that usually the result is not empty and then the user wants to find out more: which pairs in the relation are *violations* of the rule, and *why?* To see the *violations* and to trace them back to the overall design is precisely where the graph language comes in, and to formulate the usual sequence of additional questions needed to find out *why*, is where the dialogue language comes in.

The approach has been used in several Philips projects, both in the research department and in the industrial divisions. Amongst the analyzed systems we mention television systems, several telecommunication systems and also medical systems. A number of these cases are described in [27] en [46].

2.6.2 RPA versus Databases

We already mentioned the close relation between the use of relational databases and RPA in the field of architecture formalization and verification. There are two points where RPA provides advantages:

- *theoretical*: by considering the special role that part-of relations (usually a tree and not a graph) play in our analysis (e.g. lifting), we obtain a specialized set of operations and properties called Relation Partition Algebra.
- *implementation*: we often have sets and relations with an extremely high cardinality, and it pays to create a dedicated implementation of the operations, instead of relying on standard data base technology. Also, databases are usually not strong in calculating the transitive closure, which is something we regularly use.

Apart from these, the use of RPA is very extensible: one can just add relations and rules without having to change the model. For more information on the pure mathematics of Relation Partition Algebra, we refer to [29]. In this paper, we only discuss the language-oriented aspects and their applications.

2.6.3 Related Work

For reverse engineering, we can refer to earlier publications of ourselves that focus on the software architecture aspects [27], [46] and on the algebraic laws of our expression language [29]. Similar work on reverse engineering includes the following:

- Kazman and Carriere [44] developed a workbench to support the extraction and fusion of architectural views. Both *uses relations* ('functions call functions') and *part-of relations* ('classes define functions', 'files contain functions') are recognized. The architectural views are similar to the graph language of Section 2.4.

- Chikofsky and Cross [20] give an overview of the field of reverse engineering and design recovery. Their terminology is most helpful, for example to distinguish between *reverse engineering* (analysis and abstraction) and *re-engineering* (renovation and alteration).
- Holt [35] uses operators based on Tarski's relational algebra, supported by a language called Grok. Grok scripts are similar to the expression language of Section 2.3.
- Murphy at al. [63] use high-level abstractions called *software reflexion models*, which can be used to determine where the engineer's high-level model does and does not agree with the source model. Operations such as lifting are used in an implicit way. The reflexion model itself is formalized in Z.

For forward architecting, one could argue that it is better to define the software architecture in an *architecture description language* (ADL). Such a description can then be used to control the implementation build process so that no one can violate the architecture. Architectural description languages indeed exist (Aesop [1], Darwin [52], Koala [72], Rapide [102]), but they are not commonly used yet, and if used, they concentrate on decomposition only. General architectural rules may be built into the semantics of the ADL, but domain specific rules usually cannot be added. Furthermore, you might want to tolerate local and/or temporary deviations of the architecture, but a rigid ADL compiler might not allow that.

2.6.4 Acknowledgements

The authors want to thank Piërre van de Laar and André Postma for their contributions to this paper.

Chapter 3

The Koala Component Model

Published as: *The Koala Component Model for Consumer Electronics Software*, Rob van Ommering, Frank van der Linden, Jeff Kramer, Jeff Magee, IEEE Computer, March 2000, p78-85.

Abstract: A component-oriented approach is an ideal way to handle the diversity of software in consumer electronics. The Koala model, used for embedded software in TV sets, allows late binding of reusable components with no additional overhead.

3.1 Introduction

Most consumer electronics today contain embedded software. In the early days, developing CE software presented relatively minor challenges, but in the past several years three significant problems have become apparent:

- The size and complexity of the software in individual products are increasing rapidly. Embedded software roughly follows Moore's Law, doubling in size every two years.
- The required diversity of products and their software is increasing rapidly.
- Development time must decrease significantly.

What does all this software do? At first, it provided only basic control of the hardware. Since then, some of the signal and data processing has shifted from hardware to software. Software has made new product features possible, such as electronic programming guides and fancy user interfaces. The latest trends show a merge with the computer domain, resulting in services such as WebTV.

No longer isolated entities, CE products have become members of complex product-family structures. These structures exhibit diversity in product featuring, user control style, supported broadcasting standards, and hardware technology – all factors that increase complexity.

Today's dynamic CE market makes it impossible to wait two years between the conception and the introduction of a new product. Instead we must create new products by extending and rearranging elements of existing products. The highly

competitive market also requires us to keep prices low, using computing hardware with severely constrained capabilities.

3.2 The Challenge

How can we handle the diversity and complexity of embedded software at an increasing production speed? Not by hiring more software engineers – they are not readily available, and even if they were, experience shows that larger projects induce larger lead times and often result in greater complexity. We believe that the answer lies in the use and reuse of software components that work with an explicit software architecture.

3.2.1 Why Software Components?

Software reuse lets us apply the same software in different products, which saves product development effort. Software reuse has been a goal for some time [54]. The classical approach of defining libraries can be highly successful in limited domains, such as scientific and graphical libraries. However, while stimulating low-level code reuse, libraries do not help much in managing the similarities and differences in the structure of applications.

Developers devised object-oriented frameworks to create multiple applications that share structure and code [25]. The framework provides a skeleton that they can specialize in different ways. This approach makes application development faster, as long as the applications share similar structures. But changing the structure is difficult because it is embedded in the framework. Also, a strong and often undocumented dependency exists between components and the framework because of implementation inheritance.

Component-based approaches let engineers construct multiple configurations with variation in both structure and contents [8][103]. A software component is an encapsulated piece of software with an explicit interface to its environment, designed in such a way that we can use it in many different configurations. Classical examples in desktop application software are the button, the tree view, and Web browser. Well-known component models are COM/ActiveX, JavaBeans, and CORBA.

3.2.2 Why an Explicit Architecture?

Many component models are used with one or more *programming languages* – such as Visual Basic and Java – to construct configurations out of sets of components. Such an approach has one disadvantage: difficulty in visualizing and therefore managing the structure of the configurations. You can use visual tools to design the structure and even generate skeleton code from it, but keeping such designs consistent with the actual code often proves difficult. Although we can use

round-trip engineering techniques to extract the design information from the actual code, wouldn't it be better if the structure was made explicit in the first place?

With an architectural description language (ADL), you can make an explicit description of the structure of a configuration in terms of its components [32]. This description makes both the diversity of the product family and the complexity of the individual products visible; thus it serves as a valuable tool for software architects.

3.2.3 The Perfect Marriage?

We believe that a component model combined with an architectural description language will help us develop CE product families. COM inspired us, but we soon found the following requirements specific to our domain:

- Most of the connections among our components are constant and known at configuration time. To limit the runtime overhead, we wish to use *static binding* wherever possible;
- High-end products will allow for the upgrading of components in the near future. We would like the components we described earlier to be *dynamically bound* into such products, which will have looser resource constraints
- We need an explicit notion of *requires* interfaces.

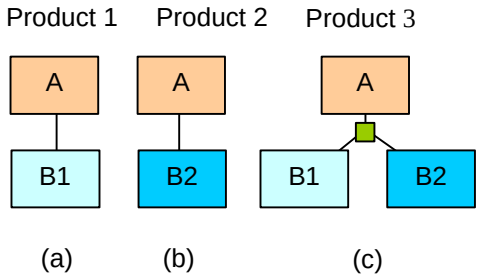

Figure 13. Taking the binding knowledge out of the components

Figure 13 illustrates the need for *requires* interfaces. In Figure 13 (a), if component A needs access to component B1, it would traditionally import B1, but this puts knowledge of B1 inside A, and therefore A cannot combine with B2, shown in Figure 13 (b). One solution would be to let A import an abstract component B and have the configuration management system choose between B1 and B2. But this would not allow us to create product 3, shown in Figure 13 (c), where A is bound to either B1 and B2 depending on some condition to be determined at runtime.

The solution is to take the binding knowledge out of the components. Component A is then said to *require* an interface of a certain type, and B1 and B2 *provide* such an interface. The binding is made at the product level.

Darwin [51][52], although originally designed for distributed systems, provides most of what we need from an ADL: an explicit hierarchical structure, components with *provides* and *requires* interfaces, and bindings. However, it did require modification in order to support:

- The easy addition of glue code between components (without having to create auxiliary components) and
- A diversity parameter mechanism that allows many parameters to be defined and that also permits code optimization depending on the parameter settings.

We therefore created the Koala model and language, which Philips software architects and developers currently use to create a family of television products.

3.3 The Koala Model

In designing Koala, we sought to achieve a strict separation between component and configuration development. Component builders make no assumptions about the configurations in which their component is to be used. Similarly, configuration designers are not permitted to change the internals of a component to suit their configuration.

3.3.1 Components

Koala components are units of design, development, and – more importantly – reuse. Although they can be very small, the components usually require many person-months of development effort.

A component communicates with its environment through *interfaces*. As in COM and Java, a Koala interface is a small set of semantically related functions. A component provides functionality through interfaces, and to do so may require functionality from its environment through interfaces. In our model, components access all external functionality through *requires* interfaces – even general services such as memory management. This approach provides the architects with a clear view of the of the system's resources use.

For example, in a TV, a *tuner* is a hardware device that accepts an antenna signal as input, filters a particular station, and outputs it at an intermediate frequency. This signal is fed to a high-end input processor (HIP) that produces decoded luminance and color signals, which in turn are fed to a high-end output processor (HOP) that drives the TV screen. Each of these devices is controlled by a software driver that can access hardware through a serial (I2C) bus. Therefore, each driver

requires an I2C interface, which must be bound to an I2C service in a configuration.

Figure 14 graphically represents a TV software platform that contains these drivers and some extra components. We deliberately designed Koala's graphical notation to make components look like IC chips and configurations look like electronic circuits. Interfaces are represented as pins of the chip; the triangles designate the direction of function calls. The configuration in Figure 14 binds the tuner and HIP driver to a fast I2C service and binds the HOP driver to a slow I2C service.

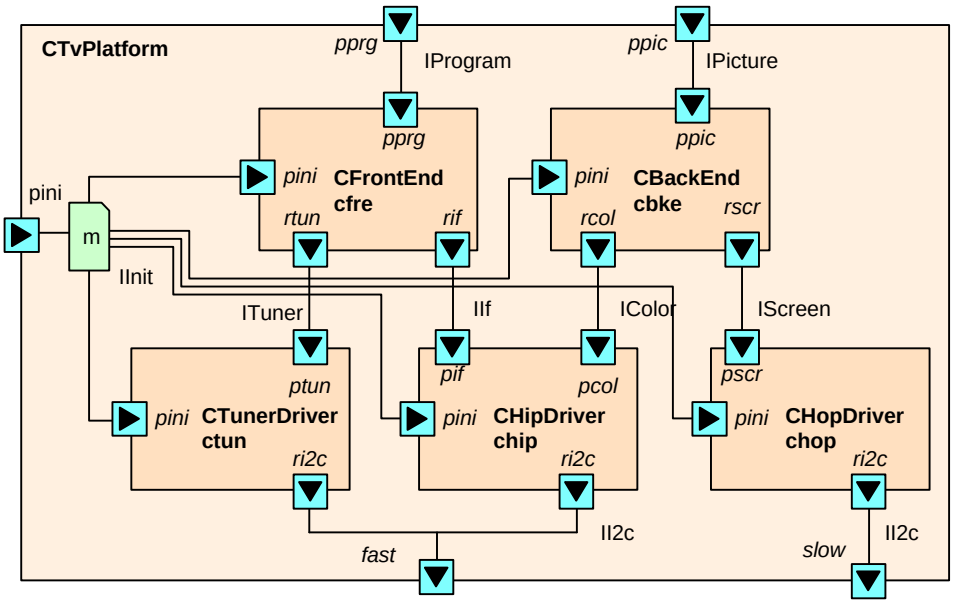

Figure 14. A TV platform in Koala's graphical notation.

3.3.2 Interface Definitions

We define an interface using a simple *Interface Description Language* (IDL), in which we list the function prototypes in C syntax. For instance the `ITuner` interface is defined as follows:

```
interface ITuner
{
   void SetFrequency(int f);
   int  GetFrequency(void);
}
```

`ITuner` is an example of a *specific* interface type, which will be provided and/or required by only a few different components. The `IInit` interface (also present in Figure 3) is an example of a more *generic* interface: it contains functions for initializing a component, and most components will provide this interface.

3.3.3 Component Descriptions

We describe the boundaries of a component in a textual component description language (CDL). The tuner driver is defined as follows:

```
component CTunerDriver
{
    provides ITuner ptun;
             IInit  pini;
    requires II2c   ri2c;
}
```

Each interface is labeled with two names. The long name – for example `ITuner` – is the *interface type name*. This globally unique name refers to a particular description in our interface repository. The other name – for example `ptun` – is a local name to refer to the particular interface *instance*. This allows us to have two interfaces on the border of a component with the same interface type – for instance, a volume control for the speakers and one for the headphones – as long as the instance names are different.

3.3.4 Configurations

A configuration is a set of components connected together to form a product. All *requires* interface of a component must be bound to precisely one *provides* interface; each *provides* interface can be bound to zero or more *requires* interfaces. Interface types must match.

3.3.5 Compound Components

A typical component may contain 10 interfaces, and a typical configuration tens of components. Hence, it is not convenient to define system configurations directly in terms of basic components. Therefore, as in Darwin, we introduce *compound components*. Figure 14 shows an example, the TV platform. Here is an incomplete definition:

```
component CTvPlatform
{
  provides IProgram pprg;
  requires II2c slow, fast;
  contains
    component CFrontEnd cfre;
    component CTunerDriver ctun;
  connects
    pprg      = cfre.pprg;
    cfre.rtun = ctun.ptun;
    ctun.ri2c = fast;
}
```

Each contained component has a *type name* – for example, `CTunerDriver` – and an *instance name* – for example, `ctun`. The globally unique type name refers to the reusable component definition in our component repository. The instance name is local to the configuration.

We have to extend the binding rules to cater to compound components. The rules are very simple if we take the triangles into account. *An interface may be bound by its tip to the base of precisely one other interface*. Conversely, *each base may be bound to the tip of zero or more other interfaces*. In plain English, there must be a unique definition of each function, but a function may be called by many other functions.

3.3.6 Modules

In Figure 14, each subcomponent provides an initialization interface that has to be called when initializing the compound component. We cannot just connect all initialization interfaces to that of the compound component - that violates our binding rule (What would be the order of calling?). We could define a new component to perform the initialization, but this non-reusable component would pollute our component repository.

We have therefore chosen another solution. A *module* is an interfaceless component that can be used to glue interfaces. We declare modules within a component and connect them to interfaces of the component or of its subcomponents. The module has access to any interface whose base is bound to the module. The module implements *all* functions of *all* interfaces whose tip is bound to the module. We also use modules to implement basic components, forming the leaves of the decomposition hierarchy.

3.3.7 Implementation

In Koala, components are designed independently of each other. They have interfaces to connect to other components, but this binding is *late* – at configuration time. By running the compiler at configuration time, we can still deploy *static binding*!

The implementation of static binding is straight forward, using naming conventions and generated renaming macros. A simple tool (also called Koala) reads all component and interface descriptions and instantiates a given top-level component. All subcomponents are instantiated recursively until Koala obtains a directed graph of modules, interfaces and bindings.

For each module, Koala generates a header file with renaming macros as shown in Figure 15. A function `f` in interface `p`, implemented in module `m` of component C, is given the *logical* name `p_f`. Koala chooses a *physical* name `c_p_f`, where `c` is a globally unique prefix associated with C. To map logical to physical, Koala generates the following macro in the header file for `m`:

```
#define  p_f  c_p_f
```

Similarly, a module `n` in component D refers to a function `f` in the *requires* interface `r` by its logical name, `r_f`. Koala calculates the binding and generates the appropriate renaming macro – in our case:

```
#define  r_f  c_p_f
```

The names `p_f` and `r_f` are local to modules and the name `c_p_f` is globally unique. This is an example of static binding. Koala also supports limited forms of dynamic binding in the form of switches.

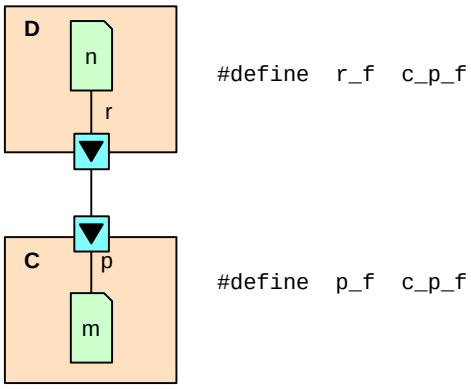

Figure 15. Implementation of static binding.

3.4 Handling Diversity

Koala has some extra features that are aimed at handling diversity efficiently: interface compatibility, function binding, partial evaluation, diversity interfaces, diversity spreadsheets, switches, optional interfaces, and connected interfaces.

3.4.1 Interface Compatibility

An interface of type `ITuner` can be provided or required by more than one component. For instance, both a European frequency-based and an American channel-based television front-end can be connected to both a high-end and an economical tuner driver if they support the same interface. Treating interface definitions as ‘first class citizens’ ensures that component builders do not change the interface to suit only one implementation.

As a consequence, we declare an interface definition to be immutable – it cannot be changed once it has been published. But it is possible to create a new interface type that contains all the functions of the previous interface plus some additional ones. With strict interface typing, a tuner driver providing the new interface cannot be connected to a front end requiring the old interface – without adding glue code. Because we expect this to be a common design pattern, we permit an interface to

be bound to one of a different type if the provided interface supports at least all of the functions of the required interface.

3.4.2 Function Binding

When two interfaces are bound, their functions are connected on the basis of their name. Sometimes we must bind functions of different names efficiently, perhaps even from different interfaces. We can implement glue functions in C, but this introduces a runtime overhead. To solve this problem, we introduce *function binding*.

Remember that developers ultimately implement functions in modules. Normally, Koala generates a renaming macro in the header file; a developer implements the function by hand in a C file. We allow a function to be bound to an expression in CDL. Koala will then generate a macro that contains the C equivalent of that expression as its body. The expression may contain calls to functions of interfaces bound to that module.

For example, suppose that for some reason we must bind a *new* front-end requiring `ITuner2` to an *old* tuner driver providing `ITuner`. The interface `ITuner2` has an extra function `EnableOutput`. A different component, the HIP, also can perform this function.

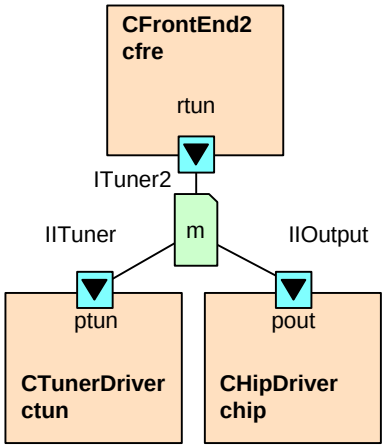

Figure 16. Function binding in Koala.

Figure 16 shows how Koala performs function binding. Koala function binding can implement module `m` as follows:

```
within m {
 cfre.rtun.SetFrequency(x) =
    ctun.ptun.SetFrequency(x);
 cfre.rtun.GetFrequency() =
    ctun.ptun.GetFrequency();
 cfre.rtun.EnableOutput(x) =
    chip.pout.EnableOutput(x);
}
```

Bacause Koala can shortcut the renaming macros, this is more efficient than implementing the functions in C. However, the real benefit of function binding comes with partial evaluation.

3.4.3 Partial Evaluation

Koala understands a subset of the C expression language, and can partially evaluate certain expressions – so `1 + 1` will be `2`, and `1?f(x):g(x)` will be `f(x)`. This plays an important role in our diversity management.

3.4.4 Diversity Interfaces

To be reusable, components should not contain configuration specific information. Moving all configuration specific code out of the component may provide an almost empty component that, while reusable, is not very usable! We believe that non-trivial reusable components should be parameterized over *all* configuration specific information.

We could add a parameter list to a component definition, but this technique only works well with a few parameters. We expect components to have tens and maybe hundreds of parameters – see for instance the property lists of ActiveX components.

Property lists are indeed suitable, but an implementation in terms of Set and Get functions does not allow for optimization when certain parameters are given constant values at design time. Therefore we reverse roles: Instead of the component *providing* properties to be filled in by the configuration, we let it *require* the properties through the standard interface mechanism. Such interfaces are called *diversity interfaces*.

Figure 17 shows how to give a television front-end a diversity interface. A parameter in the interface `div` of `CFrontEnd` could be a Boolean function `ChannelMode()`, indicating whether the component should operate in frequency or in channel mode. The function is implemented in a module `m` that belongs to the configuration.

Koala can implement `ChannelMode` as a C function, which makes the diversity parameter dynamic. It can also bind `ChannelMode` to an expression that can be calculated at configuration or compile time. Koala will then assign the result value – for example, `true` – to the function so that the C compiler can, for instance,

remove the `else` part of if statements referring to the parameter, resulting in less code. Koala will generate an extra macro of the form `#define div_ChannelMode_CONSTANT 1` that can be used to conditionally exclude certain pieces of code that are unreachable given this diversity parameter setting.

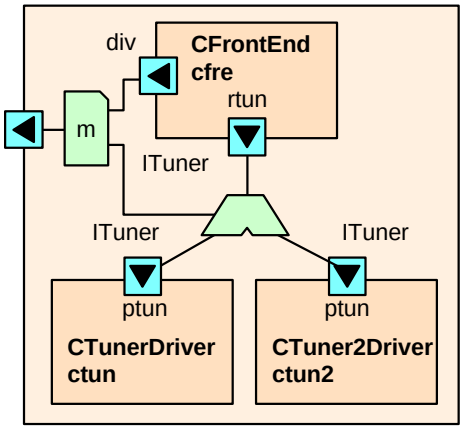

Figure 17. Diversity interfaces and switches.

3.4.5 Diversity Spreadsheets

Setting up an *object-oriented spreadsheet of diversity parameters* provides the most interesting use of Koala's function binding and partial evaluation. Koala can express parameters of inner components in terms of parameters of parameters of outer components. For instance, it can express `ChannelMode` of the television front-end in the module `m` in terms of the *region* diversity parameter of the TV platform (US = channel mode, Europe = frequency mode). Koala can express this region parameter again in terms of a product diversity parameter at a yet higher level. Using the product parameter in the front-end component would be a violation of our principle of configuration independent components, which mandates that the component designer have no knowledge of specific products.

The spreadsheet approach allows for even more elegant diversity calculations. Consider the use of Real Time Kernel (RTK) threads. Each component will create zero or more of such threads. Some RTKs require the thread descriptor blocks to be allocated statically. If we let each component provide an interface that contains the number of threads required, we can use Koala to add all the numbers at compile time and bind the result to the diversity parameter of the RTK!

3.4.6 Switches

Koala can use diversity interfaces to handle the internal diversity of a component. But what about the *structural diversity* in the connections between components?

Koala already provides for this: You can use function binding with conditional expressions to route the function calls to the appropriate components. Koala's partial evaluation mechanisms allow these connections to be turned into normal function calls if it can evaluate the condition at compile time. For us, this design pattern occurs so frequently that we decided to make it a special Koala construct – the *switch*.

Figure 17 demonstrates the use of a switch. The front-end connects to the first or second tuner driver depending on the switch's setting. An interface, which could be a diversity interface, controls the switch itself. If the switch setting is known at compile time, Koala's partial evaluation techniques will optimize the switch to a direct function call. Moreover, Koala removes unreachable components from the configuration automatically. These measures allow for a late yet optimal binding of components. Koala also permits multiple interfaces to be switched simultaneously and between more than two targets.

3.4.7 Optional Interfaces

Our product family has a set of components that all provide a basic set of interfaces, but some of them provide extra interfaces. A set of tuner drivers may, for instance, all offer frequency and channel selection interfaces, but some of them may also offer advanced search interfaces. If we design another component to be connected to one of this set, this component may want to inquire whether the tuner driver actually connected to it supports searching – this knowledge is not a component but a configuration property. To do so, the component declares an optional *requires* interface that may, but need, not be connected at its tip.

A component with an optional *requires* interface `r` can use the function `r_iPresent()` to determine whether the interface is indeed connected. Koala will set this function to `TRUE` if the interface is connected to a non-optional provides interface of another component, and to `FALSE` if it's not connected.

A component can also *provide* optional interfaces. Such an interface is automatically extended with an `iPresent` function, which the component must implement to inform others whether it actually implements the interface. The `iPresent` of such an optional *provides* interfaces may depend on the availability of hardware, or on the `iPresent` function of optional *requires* interfaces that the component needs for its implementation.

We modeled optional interfaces after COM's query interface mechanism. Again, partial evaluation allows Koala to optimize the code if it can determine its presence at compile time.

3.4.8 Connected Interfaces

A configuration consists of a given component instantiated recursively. If Koala can determine after switch evaluation that certain components are not reachable, it will not include them. To start this process, at least one module must be declared to be present.

A component that is reachable can use the function `iConnected` to determine whether a provided interface is actually being used in that configuration. The component can skip time-consuming initializations or exclude parts of the code if certain interfaces are not used.

3.5 Coping with Evolution

Koala supports the software development of a product family of up-market television sets. The model is used by over 100 developers at different sites spread all over the world, which raises some *process* issues in component-oriented development.

3.5.1 The Interface Repository

Developers store interface definitions in a global interface repository, where each interface type has a globally unique name. An interface definition consists of an IDL description and a data sheet, a short text document describing the semantics of the interface.

The repository is Web-based and globally accessible. Changes can be made only after they have been approved by the interface management team. The following rules constrain the evolution:

- Existing interface types cannot be changed;
- New interface types can be added.

In practice, we allow for exceptions to the first rule, but only if *all* components using that interface can be changed at the same time – which is usually impossible.

3.5.2 The Component Repository

Developers store component definitions in a global component repository, where each component has a globally unique long name, used in component descriptions, and a globally unique short name, used as prefix for function names. Note that C itself has no name space facility other than file scope.

The repository is also Web-based. Each component has a CDL description, a data sheet (a short document describing the component), and a set of C and header files. These header files are only for use by the component itself; Koala handles all connections between components.

Changes to the repository can only be made after approval by the architecture team. The following rules apply:

- New components can be added;
- An existing component can be given a new *provides* interface, but an existing *provides* interface cannot be deleted;
- An existing component can be given a new *requires* interface but it must then be optional – an existing *requires* interface cannot be deleted (but it can be made optional!).

A compound component is a reusable component just as any of its constituents. As with hardware, we sometimes call a compound component a *standard design*. Our component repository is flat: It contains both basic and compound components. Though component instances are encapsulated in compound components, the corresponding types are not. Therefore, it is possible to construct a second compound component with the same basic components in a different binding.

3.5.3 Configuration Management

The repositories are under the control of a standard configuration management system. This system is used to manage the *history* of components and temporary branches – for example, where a developer repairs a bug while another adds a feature. Koala handles all permanent diversity, either by diversity interfaces or by variants of components stored under a different name.

3.6 Concluding Remarks

More than 100 software developers within Philips are currently using Koala. It lets us to introduce component orientation in a domain that is still severely resource constrained. It offers explicit management of *requires* interfaces, a graphical notation that is very helpful in design discussions, and an elegant parameterization mechanism. Its partial evaluation techniques can calculate part of the configuration at compile time while generating code for that part that must be determined at runtime. We do not claim that the underlying component model is unique, but we do believe that its diversity features and partial evaluation techniques are both novel and beneficial. Furthermore, we believe that the approach will facilitate the future transition to standard platforms such as COM.

Acknowledgement

Koala is a result of the Esprit project 20477 (ARES).

Chapter 4

Independent Deployment

Published as: *Techniques for Independent Deployment to Build Product Populations*, Rob van Ommering, WICSA 2001: The Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 28-31, 2000, p55-66.

Abstract: When building small product families, software should be shared between different members of the family, but the software can still be created as one system (with variation points) with a single architecture. For large and diverse product families (product populations), the software can no longer be developed in one context and at one moment in time. Instead, one must combine software components of which the development is separated in space and in time, each with their own evolution path. In other words, we need independent deployment of components.

We discuss four aspects of independent deployment in this paper. Two of these aspects - upward and downward compatibility - deal with variation in time. The other two - reusability and portability - deal with variation in space. For each aspect we indicate the relevance, provide some examples, and list several techniques to deal with it. The paper can thus be seen as a guide for product population development.

4.1 Introduction

If you are building a single product, then management of *complexity* is most likely your major concern. The obvious technique is to *divide and conquer*: split the system into well-defined components, build and test the components in isolation, and integrate the components into a system. Many engineers talk about components in this sense. There are of course other (or additional) ways to deal with complexity.

If you are building a *small* product family, then management of *diversity* will be your major concern. A proven technique is to build a variant-free architecture [88] with variation points [41]. Variation points can be implemented e.g. as compiler constants, a registry with properties, functions that can be 'plugged in' (like the

compare function in a sort function), or plug-in components. The latter solution results in a component framework, where diversity is handled by the plug-in components, and where complexity may be managed by splitting the framework itself into components.

If you are building a *large* product family, where the individual products have many commonalties but also many differences, then we believe that it is no longer feasible to create a single variant free architecture in which all software is developed in one context and at one moment in time. Instead, software developed for one product (a television) often needs to be combined with software created for a second product (a DVD player) to create a third product (a TV-DVD 'combi').

Hence, independent deployment of software becomes *the* major concern when architecting large product families (which we call product *populations* [73]). We use the term 'deployment' in the wide sense here; it includes design, development and evolution of the software.

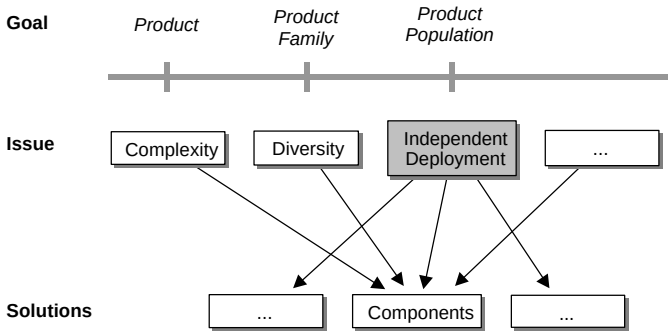

Figure 18. Independent deployment as problem, components as one of the solutions

Figure 18 shows the spectrum that ranges from single product development to product population development. It lists the major concern for each point in the spectrum as described above, and it indicates that there exists a multitude of solutions for each concern.

This paper is about independent deployment of software, and about techniques for achieving independent deployment. Some of these techniques can be called component technologies, others are just common sense engineering. The main questions we want to address are:

- Can I produce a non-trivial piece of software without knowing the specific client or the specific version of the client of that software in advance?
- Can I produce such a piece of software without knowing the specific underlying servers or the specific versions of such servers in advance?
- If so, what techniques do I have to apply?

This paper is organized as follows. In section 4.2, we define the notion of independent deployment and discuss its business relevance. It will turn out that we have to deal with *four* aspects, which we will baptize *upward compatibility*, *downward compatibility*, *reusability* and *portability*. These are discussed in sections 4.3-4.6, together with possible techniques and a number of examples. Section 4.7 puts yet a different light on the issues, and section 4.8 ends this paper with concluding remarks.

4.2 Independent Deployment

In this section we define independent deployment and we categorize it into four aspects that we can discuss separately. We briefly describe each of these aspects in this section - the next sections discuss them in more detail.

4.2.1 What is Independent Deployment?

Suppose we build systems that consist of two parts: a *client* C and a *server* S. Both are just pieces of software that call each other's functions - no distribution of computation is implied by the terms client and server. We can readily construct such systems if we develop the client and server *at the same time and in a single project*, by fine-tuning their mutual requirements. The resulting cooperation is indicated with arrow (1) in Figure 19.

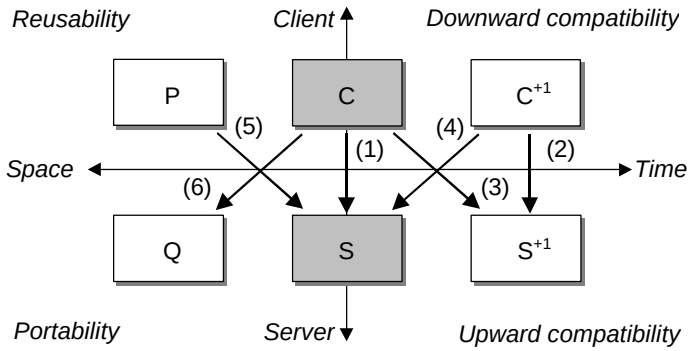

Figure 19. Dependent deployment (1-2) and independent deployment (3-6).

Now suppose we have to upgrade the system to satisfy new requirements. We create a new version of the client, C^{+1}, and a new version of the server, S^{+1}. Again we do so in a single project, and we succeed because we can fine-tune the (new) mutual requirements of the client and the server. The resulting cooperation is arrow (2) in Figure 19.

Both scenarios are state of practice. We shall now consider four other scenarios that deal with the construction of systems out of pieces of software that have *not* been developed together. Two of these scenarios concern a separation in *time* (the

right-hand side of Figure 19), the other two a separation in *space* (the left-hand side of Figure 19)[1].

Combining a new version of the server with an old version of the client (arrow (3)) is only possible if the client is *upward compatible* with respect to the new server. One way of achieving this is to make the new server *backward compatible* with the old server (as explained further in section 4.2.2). Although many people are familiar with this compatibility issue, achieving it is still difficult. Upward compatibility is discussed further in section 4.3.

Combining a new version of the client with an old version of the server (arrow (4)) is only possible if the client is *downward compatible* with respect to the server. Most software does not satisfy this criterion. One of the design goals of Microsoft's COM was to solve this problem. Downward compatibility is discussed further in section 4.4.

Combining a server with a different client (one that was not taken into account when creating the server) is called *reusability* (of the server). Arrow (5) illustrates this combination. Although it might look easy, in practice most software turns out to be applicable only in the specific context for which it has been designed. Reusability is discussed further in section 4.5.

Combining a client with a different server (one that was not taken into account when the client was built) is called *portability* (of the client). Arrow (6) illustrates this combination. Again, in practice, most software is not portable, but instead makes so many assumptions about the environment that it can only be used in the context in which it has been developed. Portability is discussed further in section 4.6.

Please note that the four scenarios sketched above need not occur in isolation. A software component is usually a client of certain servers, and at the same time a server for certain clients. If a component is reused in a new product, then it may have to work with the *same* versions of certain clients and servers, with *newer* versions of other clients and servers, and also with *other* clients and servers than in the previous product, all at the same time. We only describe pure scenarios in this paper to keep the discussion simple.

---

[1] Although time and space are both represented along the horizontal axis, they do *not* span the same dimension. A three-dimensional drawing, with space and time along the X and Y-axis, and the client-server dimension along the Z-axis, would have been better (but is more difficult to draw and comprehend).

4.2.2 Intermezzo on Compatibility

For people getting confused about the terms *backward*, *forward*, *upward* and *downward* compatibility, Figure 20 shows our definition of these terms (see also e.g. [111]).

A client is *upward compatible* with respect to a server if it also runs on a newer version of the server. Note that this cannot be fully guaranteed by the client if the features of the new server are not yet known at the time of building the client.

A client is *downward compatible* with respect to a server if it also runs on an older version of the server.

A server is *backward compatible* if *all* clients (even hypothetical ones) that ran on the old server also run on the new server. Note that this makes *all* clients of the old server upward compatible. The reverse is not necessarily true: clients can be made upward compatible without the server being fully backward compatible.[2]

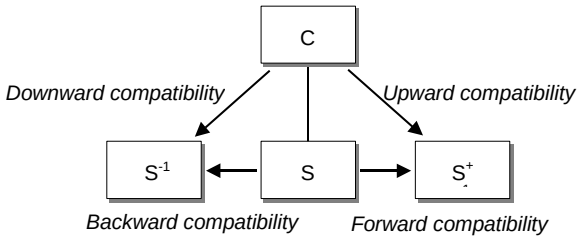

Figure 20. Types of compatibility.

A server is *forward compatible* if *all* clients that run on the server will also run on the new version of the server. There are fewer examples of this kind of compatibility, but here's one: a prerelease of a certain bit of software is usually forward compatible with the final release.

4.2.3 Business Relevance

Why do we worry so much about independent deployment, and should we really? The bottom line is that most of us are still developing software in single projects and for single products. Sometimes, projects are *distributed*, but then different subprojects are *co-developing* instead of developing independently. We cannot easily transfer functionality from one product to another.

To give an example, we can create high-end analogue TVs, we can create set-top boxes and we can create DVD players. But we cannot easily create a high-end *digital* TV with *built-in* DVD player. Independent deployment *in space* helps us to

[2] In one of our architectures, the *golden rule* of compatibility is to make servers backward compatible. The *silver rule* of compatibility is to ensure that all existing clients are upward compatible.

reuse MPEG decoders and DVD players in TVs - this is the traditional goal of software reuse activities. Independent deployment *in time* helps us to decouple development projects and hence reduce risks.

As an example of the latter, suppose we are developing a new version of an Electronic Programming Guide (EPG) 'middleware' for inclusion in a new product. The EPG software is not ready in time, but the underlying software is, and so is the application on top of it. Both have additional features that can't be missed. Can we combine them with the old version of EPG?

More examples will be given in the next sections.

4.3 Upward Compatibility

We shall now go into the technical details of independent deployment. The first question we ask ourselves is: *can an old version of a client work with a new version of a server*? We call this upward compatibility of the client.

4.3.1 Relevance of Upward Compatibility

The relevance of the question above is sketched in Figure 21, where a product consists of a server S and two clients C_1 and C_2. To build a new product, a new version of S, called S^{+1}, may be needed together with a new version of C_2, called C_2^{+1}. But there is no time to update C_1, so C_1 must still run on S^{+1}.

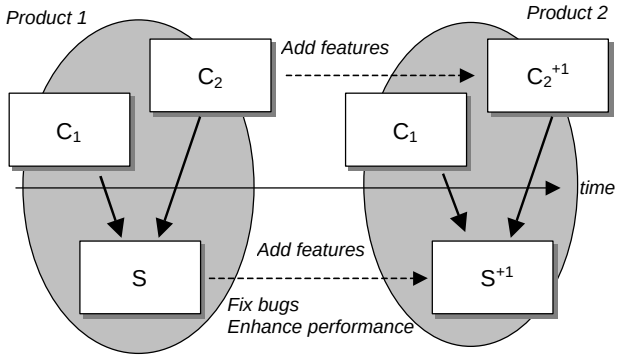

Figure 21. An upward compatibility scenario.

As a second example, suppose C_2 isn't there. Then still we might want to use S^{+1} because of bug fixes and performance enhancements, without being forced to update C_1.

4.3.2 Examples of Upward Compatibility

Windows 95 is designed to be backward compatible with Windows 3.1, so most Windows 3.11 programs will be upward compatible to Windows 95. See section 4.3.6 for some interesting exceptions.

Successive versions of Windows have seen subtle but also significant changes in the driver model. As a result, many hardware drivers only work on specific versions of the operating system. Some (older) hardware may even not be supported any more for new versions of Windows. Upgrading Windows on your PC is therefore rarely a simple job.

Microsoft's Internet Explorer has known several versions since its first release. Web pages that have been created for older versions of IE should still be viewable in IE5.5.

4.3.3 Backward Compatibility of the Server

The main technique for achieving upward compatibility of clients is to make the new version of the server *backward compatibility* with the old version (see section 4.2.2 for our definition of compatibility). This implies that one makes *conservative* extensions only to the component. For example, one can add functions to a component without changing the semantics of existing functions.

This form of compatibility is formalized in the field of *sub-typing theory*. In a nutshell, to be backward compatible, you should *provide more* and *require less*. This applies to data types (simple types, unions, structs), to functions (*in* and *out* parameters), to interfaces (functions, constants, types), and to components (provides and requires interfaces). A full discussion of this is outside the scope of this paper (see e.g. [89]).

The next two sections describe two ways of realizing backward compatibility.

4.3.4 Versioned Interfaces

The first way is to regard the interface of a component as a single interface, consisting of all the functions that the component exports. An extension of the component then changes the interface - a backward compatible extension is achieved if functions are added to but not removed from this interface. One still has to be careful: what is syntactically an extension can still be semantically incompatible, as illustrated by the following example.

In one of our systems, the Real Time Kernel interface did not contain watchdog facilities. To add those, *IRealTimeKernel* was given an extra function to 'feed' the watchdog. The watchdog was *on* by default. This implies that existing applications that do not feed the watchdog are reset after a certain time has elapsed. The default should have been *off* to make the change upward compatible.

To manage changes to component interfaces properly, version numbers should be attached to the interface. A client can then be specified to require at least a certain version of that interface. Note that newer clients may not work together with servers that are too old. This is in fact a downward compatibility problem, as explained further in section 4.4.

Disadvantage of this approach is that it is not possible to make *incompatible* extensions to the component's interface without breaking existing clients. Some people increment the minor version number of an interface for compatible changes, and the major number for changes that are incompatible.

4.3.5 Immutable Interfaces

The second solution for achieving backward compatibility is to allow a component to provide multiple interfaces. If the interfaces are kept small, then in principle it is possible to regard interface definitions as immutable, i.e. once published, they never change anymore. Microsoft COM uses this approach. If interfaces are large, then immutability can probably not be achieved (see also Figure 22).

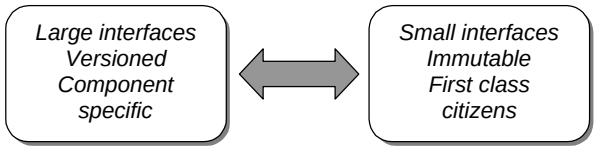

Figure 22. The interface granularity dilemma.

Microsoft COM allows component builders to realize backward compatibility by adding new interfaces while still supporting the old one. Old clients query for the old interface and will therefore still work, while new clients query for the new interface and can thus access the new functionality.

Interesting aspect of this approach is that it is possible to actually *change* the way a component is to be addressed, by adding an incompatible interface, without breaking down existing clients.

4.3.6 Client Detection by the Server

The design goal of Windows 95 was that Windows 3.11 programs were to be upward compatible. In practice, many Windows 3.11 programs do *not* run properly on Windows 95. One reason for this is that the programs make illegal use of the API (e.g., they still use memory *after* they have released it). To achieve some upward compatibility, Windows 95 recognizes such programs, and patches them on the fly when starting them.

4.3.7 Server Detection by the Client

Yet another solution is to let the client detect the version of the server and adapt it self accordingly. This is one of the approaches to achieve downward compatibility as described in section 4.4.6. Of course, this technique is not applicable if the new version of the server is not yet known at the time of creating the client. But it can be used when sufficient information is available with respect to the new version of the server. Note that if the new server is already available for a longer time, clients

are usually developed for the *new* server, and then face the *downward* compatibility problem instead of the upward compatibility problem.

4.4 Downward Compatibility

The second question we ask ourselves (see Figure 23) is: *can a new version of a client work with an old version of a server*? We actually recognize two cases here: the new version of the server already exists when the client is developed, or the new versions of the server and client are developed together.

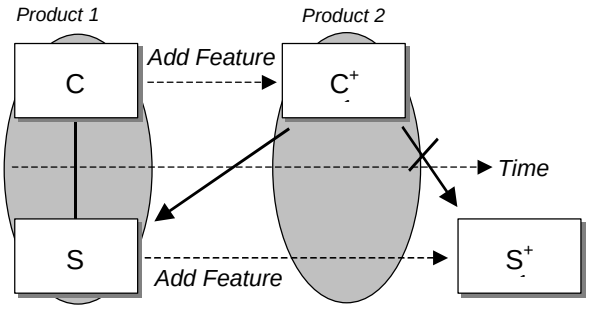

Figure 23. An example of required downward compatibility.

4.4.1 Relevance of Downward Compatibility

The relevance of downward compatibility may be less intuitive. State of practice is to ignore this problem. Let's think of some examples where it *is* useful.

If the new server is already available when the client is developed, it may still be desirable to be downward compatible with respect to the old server. Third parties that combine the client and server may choose for the old version of the server for qualitative (less buggy), economic (less expensive) or other (smaller footprint) reasons.

If the client and server are developed together, the following scenario can arise. Suppose that a new version of a client has *two* new features, one for which the new version of a server is needed, and one that also works with the old version of the server. The new server is not ready in time, or still buggy, and cannot be deployed in a product. But we do need the second feature. Can we use the new client with the old server?

4.4.2 Examples of Downward Compatibility

Suppose you haven't upgraded to Windows Millenium yet, but you want to use a new software application, released *after* W-ME. Can you use it with your old Windows 95, or do you have to upgrade your operating system as well? Most users would not appreciate the latter...

From a given version onwards, Windows 9x supports alpha blending in its graphic subsystem. Attempts to use this functionality on older versions of Windows results in strange artifacts. So, programs utilizing alpha blending should check the version of Windows.

Suppose you create a web site that should look stunning on IE5.5. Can you make it such that it still looks reasonable on older versions of IE?

4.4.3 Rely on Backward Compatibility

If the new version of the server is available at the time of building the client, the client may choose to only use the functionality as provided by the old server. This makes the client automatically work on both versions. This is not always a desirable solution, since features of the new server cannot be utilized.

4.4.4 The Multiple Implementation Technique

Another way of handling downward compatibility is to have specific client implementations for specific versions of the server. This technique is often used for drivers in Windows. The driver model is subtly different in subsequent versions of Windows, therefore many hardware manufacturers deliver specific drivers for specific versions of the operating system.

This is - of course - not an ideal solution. It puts a multiple maintenance problem at the client side, and it also burdens the creator of the system (in case of the Window driver example, the owner of the PC).

4.4.5 The Configuration Technique

Another technique is to specify the version of the underlying server as a configuration parameter to the client. This can be a compile-time parameter, which will work if all components are compiled into a product, but which results in the solution sketched in the previous section if components are deployed binary.

The disadvantage of this technique is still the burden that is put on the creator of the product combining the client and server.

4.4.6 The Version Checking Technique

Another technique is to let the client ask at run-time for the version number of the server. Depending on this number *and on knowledge about the server*, the client can then decide which functionality of the server it may use.

An example concerns web sites: it is quite common to start a web page with a query for the version of the underlying browser, and then have specific elements on the page, or even specific pages, for specific versions of the browser.

Disadvantage of the version checking technique is that knowledge about the server is built-in into the client. This hampers portability, as we will see in section 4.6.

4.4.7 The Capability Query Technique

The third and most powerful way of determining whether an underlying server implements a certain functionality is to explicitly ask the server for it. The query should be *function oriented*, and not *server version oriented*, as in the previous section. COM supports this from the ground up, using *QueryInterface* whenever functionality of a server is required.

If this technique is implemented properly, then ultimately, also the portability problem can be handled (see section 4.6).

4.5 Reusability

The third question we ask ourselves is: *can we reuse a server with a different client*? We suppose here that the server was not designed with the different client in mind.

4.5.1 Relevance of Reusability

This has been the goal of the reuse community for quite some time now. A more difficult reuse problem is actually the *portability* question as discussed in section 4.6, but let's first concentrate on what we call reusability in this paper.

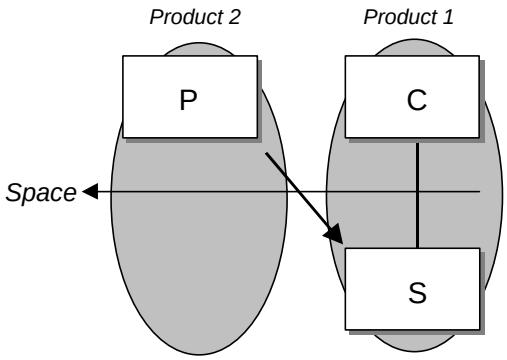

Figure 24. An example of reusability.

Figure 24 shows an example of reusability. A server S has been designed together with a client C. Can it now also be used together with a client P? The problem lies in the following dilemma (see also [103] section 4.1.8):

- To make software reusable, context specific decisions should not be taken in the software.
- To make software usable, a lot of decisions should be taken in the software.

Can we make software that is reusable and usable at the same time? We believe that the answer lies in parameterization.

4.5.2 Examples of Reusability

Windows is a platform and is therefore designed to be reusable, i.e. to create many applications on top of it.

The Internet Explorer is basically an ActiveX control that can be used for different purposes then just browsing the web. In fact, Microsoft uses it to browse through its new HTML Help format. The help pages are written in (dynamic) HTML. Seamless integration with help content on the web is also possible.

One can think of ample examples of desired reusability in our own (Philips) domain, the most urgent ones being the reuse of digital TV reception and decoding (from the set-top box domain) and reuse of storage facilities (tape, DVD, hard disk) in the classical analogue TV domain.

4.5.3 Configuration Interfaces

The most intuitive way to parameterize software is to provide one or more *configuration interfaces*. Such interfaces contain *Set* and *Get* functions through which properties of the software can be accessed. The properties influence the behavior of the software. The property mechanism makes the software reusable. If suitable default values are defined, the software can also be quite usable.

Visual Basic is *the* example where components (ActiveX controls) are usable and reusable at the same time. Controls usually have a large list of properties, some of which are set at design-time, some at run-time, and others will retain their default value. The Visual Basic environment in fact cleverly integrates design-time and run-time; to change parameters at design-time, controls also *live* at design-time.

The configuration approach has its disadvantages. Parameters can only be changed programmatically, and quite some code (and time) may be spent on initializing the system at run-time. Also, there is no means to remove redundant code, i.e. code that is not reachable under the configuration settings in a specific product.

4.5.4 Diversity Interfaces

We speak of *diversity interfaces* if the software 'takes action' to retrieve values of certain parameters from its environment. There are various implementations of this idea. The simplest one is the use of the *#define* in the C/C^{++} preprocessor. This allows fine-tuning of the software *before* being compiled, so that all kinds of compile time optimizations are possible.

A variation of this is the use of a *registry* that contains key-value pairs with settings for a variety of software. Microsoft uses this technique intensively. The Windows

Explorer can be tweaked using all kinds of registry settings, e.g. to add context menus. Microsoft Word can be convinced to perform live scrolling when moving the scroll bar[3]. Of course, it can be argued that such settings are persistent information of the tools, and that this information should only be changed through option menus of the tools, but state of practice is that the registry needs tweaking now and then.

4.5.5 Plug-In Functions

A common technique is to parameterize a piece of software over a function. A classical example is a sort algorithm that is parameterized over a function that compares two elements of the set to be sorted. This allows the function to be written in a polymorphic way, so that it can handle elements of different types.

Templates (like in C^{++}) allow parameterization over data (see the previous section), over functions (as described in this section) and over classes (as described in the next section).

Another frequent example of the use of plug-in functions is in components that can notify other components of certain events occurring. Such components are usually parameterized over functions that handle the notifications. Another term for the handler functions is 'call-back' functions.

4.5.6 Plug-In Components

The ultimate flexibility may be achieved by parameterization over *components*. When using such software, fine tuning then takes place in the form of providing *plug-in components* that may implement quite some functionality. Examples can be found in the architecture of medical equipment [91], where for instance different types of geometry are handled by plug-in components.

We need to be careful in our vocabulary here. There is quite some consensus nowadays that components are containers of classes (e.g. a DLL or an EXE), and that classes can be instantiated into objects (see [103]). Usually it is the *objects* that have interfaces and can interact with each other.

In this terminology, the parameter is rarely a component, usually a class, but sometimes it is an actual object.

As another example, the standard Windows Explorer allows for a number of such plug-ins, to be defined for all or specific subsets of files. They can bring up context menus, add property pages, display yellow pop-up windows, or change icons depending on the contents of the file.

---

[3] Set *LiveScrolling* to "1" in "HKEY_CURRENT_USER\ Software\ Microsoft\ Office\ 8.0\ Word\ Options"

4.6 Portability

The fourth and last question we ask ourselves is: *can we reuse an existing client on top of a different server*? The name portability is sometimes confusing, as we will see in section 4.6.6.

4.6.1 Relevance of Portability

Independent deployment of the fourth kind - *portability* - is the most difficult to achieve. Still, if one achieves it, it is also the most powerful, as it subsumes downward compatibility as a special case. State of practice is (again) to ignore the question, or to solve it by standardizing on a platform (see section 4.6.3). Figure 25 shows schematically how a client C that has been originally designed to cooperate with server S, now has to be used in combination with server Q.

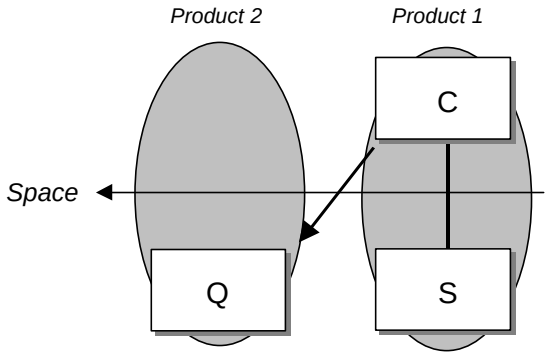

Figure 25. An example of portability.

4.6.2 Examples of Portability

Windows applications are rarely available on other platforms than Windows. In fact, it could be seen as a Microsoft strategy *not* to make their applications portable. There are some exceptions, however. Word is for instance also available on some Apple machines.

Can a web site that has been designed for Internet Explorer also be viewed with Netscape? Often, the solution sketched in section 4.6.5 is adopted.

There are ample examples of required portability in Philips again: for instance the reuse of Teletext and Electronic Program Guide (EPG) software, where the problem mainly lies in the porting of this software to the different infrastructures of the various products.

4.6.3 Platforms

The usual 'solution' is to *standardize* on a platform (cf. Windows). If everyone uses the same underlying platform, then we do not have a portability problem. There are various reasons why this approach does not work well in all cases.

First of all, no choice of any platform is eternal. You may for instance select a real time kernel (RTK), and then discover one day that the company has been bought by another company, and the RTK is being discontinued. Or, you might have selected the perfect RTK and then customers insist that you provide the software also on another RTK.

Secondly, platform versus application is only *one* division of the software, but the portability issue also arises *within* the platform and *within* the application.

4.6.4 Adaptation Layers

Mismatch between the required interface of a client and the provided interface of a server is often solved by inserting an adaptation layer. However, this is a patch, and not a fundamental solution. It is worrying how large a role adaptation layers play in current architectural discussions. Some people see them as a fact-of-life.

4.6.5 Server specific Clients

The solution that many web sites deploy for working well with Netscape *and* Internet Explorer is to query for the identity of the server, and then have specific pages for specific servers. This solution is quite similar to the one described in section 4.4.6

4.6.6 A Client Server Architecture

Another solution is to standardize on an interface or an interface suite between client and server. Any client that supports this interface suite can then be used with any server that requires it.

A good example of this is Microsoft's ActiveX compound document functionality (previously called OLE [17]). A large set of interfaces is defined that allows OLE containers to embed instances of OLE servers. Examples of OLE containers are Word, Excel and PowerPoint. The same applications are also examples of OLE servers.

The compound document functionality is successful with respect to the *servers*. It is relatively easy to create another server of which objects can be embedded in one of the OLE containers. Creating an OLE *container* is much more difficult. There are not many examples of OLE containers, so in our terminology, the *reuse* of OLE servers is limited.

As a final note, we admit that the word *portability* - one of our four aspects - does not optimally cover the client server architecture described here. When we talk about portability we usually mean the deployment of a client with a server that have *not* been designed under a single architecture. The next sections will elaborate on this.

4.6.7 Explicit Context Dependencies

The real solution to the portability problem might lie in the author's *Rule of Portability*:

> *80% of the software does not **need** more than 20% of the underlying software.*

Unfortunately, a variant of Murphy's law states:

> *Only 20% of the software does not **use** more than 20% of the underlying software.*

The only way to make sure that 80% of the software actually does not use more than 20% of the underlying software is to *make* all context dependencies explicit and to *manage* them explicitly. We believe that it is not sufficient to study link maps for context dependencies. Instead, software developers must make explicit choices in using certain servers, and software architects should monitor such explicit choices.

Interfaces to the environment are sometimes called *requires interfaces*. To really make them work, the build environment should not allow a build if functionality of the environment is used without having specified it explicitly.

4.6.8 Third Party Binding

One step further is explicit third party binding. Now it's no longer the client specifying which *server* is to be used, but only which *service*, and a third party (the product creator) selects a particular server and binds it to the client. The binding is done *late* in the production process, i.e. when the product is created, *after* the clients and servers have been built. But it can still be done *early* in the compile/link/run process (called *static* binding, and explained in the next section), or *late* (called *dynamic* binding, and explained in section 4.6.10).

4.6.9 Static Binding Techniques

The most common static binding technique is to use parameters in the configuration management system to select between component variants. Such parameters may for instance describe the underlying hardware platform or the product to be created. Disadvantage of this approach is that it hides architectural

information in the CM system, and that it is impossible to postpone the choice to run-time.

The technique deployed in Koala [72] is to use *#define* to map names of required functions to names of provided functions. In fact, Koala is more general since it also allows to postpone the binding to run-time when desired.

Yet another technique is to map names of required functions to names of provided functions in the linker, via symbol mapping. This technique also allows for postponing binding decisions until run-time, by inserting selector functions in between.

Note that in all cases, the conceptual level of indirection between the required and the provided function is mostly absent at run-time.

4.6.10 Dynamic Binding Techniques

Dynamic binding techniques rely on a level of indirection that is present at run-time. The simplest solution is to use *function pointers*; a technique often used to bind callback functions. To make the binding easier, function pointer *tables* can be used.

		<n> objects	polymorphic
static	f(x, y)	f(o, x, y)	
dynamic	*fp(x, y)	*fp(o, x, y)	o->f(o, x, y)
dynamic	t->f(x, y)	t->fp(o, x, y)	o->t->f(o,x,y)

Figure 26. Various kinds of binding.

Often, it is not a *function* or *class* that must be bound, but rather a specific *object* of a specific class. The solution is to add the object handle to the functions, usually as first parameter. Finally, if a specific object of *any* class is to be used (as long as it supports a certain function or interface), then a polymorphic solution has to be used, resulting in yet another level of indirection.

Microsoft's COM deploys polymorphic interface binding.

4.7 The Quality Dilemma

We have mostly discussed *technical* issues up to now. We shall now discuss one *quality* issue with respect to independent deployment (see Figure 27).

Suppose client C has been developed together with a server S, and the combination P = C + S has been tested and released as a product. Since then, a new version of the server, S^{+1}, has been released with some bug fixes and added features. Somewhat later, a new product P' has to be released where client C needs a minor

update, C'. Do you use the old S in the product, or the new S^{+1}? In the latter case, you profit from the fact that some bugs have been solved, but have other bugs been introduced when adding features to S?

Now suppose that the new product also requires a slight modification of S. Do you add this to S^{+1}, or do you create a new derivative of S, resulting in a branch S'? In the latter case, what happens if you want to create yet another product P" that requires a slight extension of C' and S'?

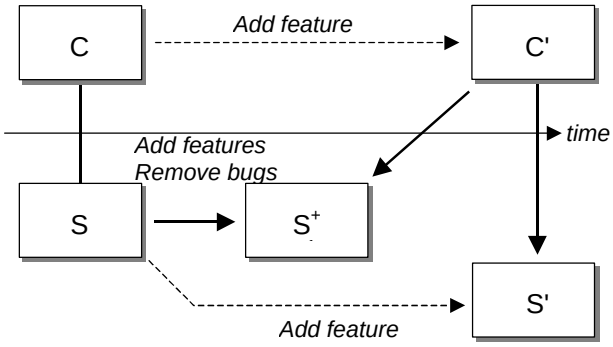

Figure 27. The quality dilemma.

The message may be clear: if the (short-term) advantages of using S^{+1} in P' do not outweigh the risks, branching will occur, and the development team of S will fall into a (long-term) multiple maintenance trap. Careful management is desired.

4.8 Concluding Remarks

4.8.1 Independent Deployment Revisited

We have discussed independent deployment of software in terms of four issues: upward compatibility, downward compatibility, reusability and portability. We have provided a number of techniques (see Figure 28) to deal with these issues, and have given examples. We do not claim that our inventory of techniques is complete.

Note that in practice, combinations of these issues occur, as software usually has multiple clients and uses multiple servers. We have discussed the issues in isolation to keep the discussion clear. Note also that some of the techniques address more than one issue. We have not tried to give the reverse mapping here.

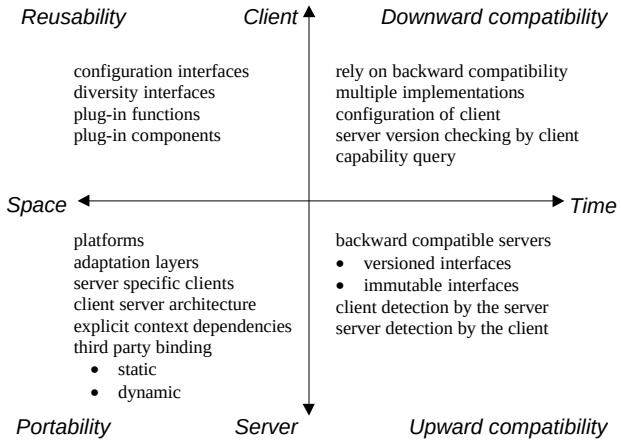

Figure 28. Summary of techniques

4.8.2 Related Work

Perry [89] also discusses compatibility, and defines the notions more formally than we found necessary to do for this paper. He concentrates on the use of configuration management and version control systems to manage compatibility, whereas we are interested in 'compatibility in the large', i.e. for product populations. See [76] for a more elaborate discussion on these topics.

Some examples have been drawn from Microsoft's COM as designed for OLE (see [17] chapter 1), where one of the design goals was to allow clients to work with servers even if they do not know of each other's existence. See [113] for a seminal paper on this topic.

4.8.3 Acknowledgements

I want to thank Hugh Maaskant and Chritiene Aarts for discussing and reviewing this paper.

Chapter 5

From Variation to Composition

Published as: *Widening the Scope of Software Product Lines – from Variation to Composition*, Rob van Ommering, Jan Bosch, Second Software Product Line Conference, San Diego, August 2002, LNCS 2379, p328-347.

Abstract: Architecture, components and reuse form the key elements to build a large variety of complex, high-quality products with a short lead-time. But the balance between an architecture-driven and a component-driven approach is influenced by the scope of the product line and the characteristics of the development organization. This paper discusses this balance and claims that a paradigm shift from variation to composition is necessary to cope with an increasing diversity of products created by an ever-larger part of an organization. We illustrate our claim with various examples.

5.1 Introduction

As a software engineering community, we have a long history in building software products: we have been doing this for half a century now. *Architecture* has always been an important success factor [18], though it has not received the attention that it deserves until the last decade. *Components* (or modules) serve as a means to cope with complexity (divide and conquer), or to streamline the evolution of systems [85]. And we are already *reusing* software for a long time: operating systems, graphical and mathematical libraries, databases and compilers are hardly ever written from scratch for new products.

It is therefore not surprising that people developed the vision that the future of software engineering lies in the use of reusable components: new applications would be built by selecting from a set of existing components and then clicking or gluing them together. McIlroy was one of the first to advocate this idea [54], and many have followed since. We have indeed seen some successes in this area, for instance the use of ActiveX controls (components) and Visual Basic (glue) to quickly build interactive applications for PCs. But we have also seen many failures, especially when applied *within* companies to create a diversity of products.

This is where software product lines enter: proactive and systematic approaches for the development of software to create a variety of products. Again, software product lines have been around for decades, but only recently receive the attention that they deserve. Note that product lines are not specific to software only: they are also commonplace in for instance the car, the airplane and the food industry [21].

Most software product line research focuses on an early analysis of commonality and variation [5], followed by the creation of a variant-free architecture [88] with a sufficient number of variation points [41] to deal with diversity. While this works well for small product families in a small organization, we have experienced problems when applying this to larger ranges of products, especially when development crosses organizational boundaries.

This paper is about widening the scope of product lines to encompass a larger diversity of products (a *product population* [73]), where the development takes place in different organizational units but still within the same company. We believe that the successful building of product populations is the next hurdle to take in software engineering. Figure 29 illustrates this, showing on the one hand that the focus of our community shifts over time from products to components to families to populations, and on the other hand that we are seeking a balance between variation (as in product families) and composition (as in software components).

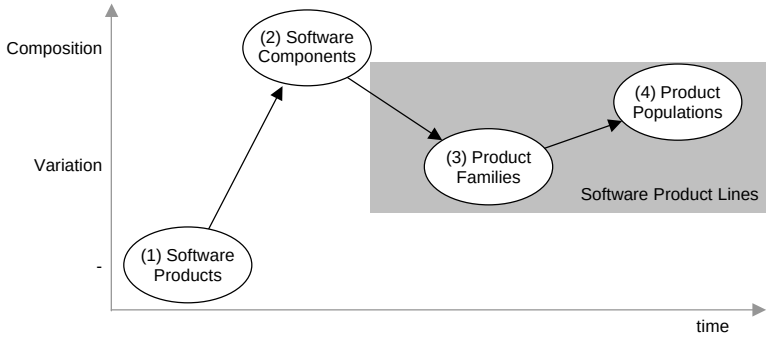

Figure 29. Towards product populations

This paper is organized as follows. First we recapitulate the main drivers for software product lines. Then we discuss the influence of product scope and characteristics of the organization on software product lines. We proceed with introducing variation and composition as separate dimensions, and then explain variation and composition in more detail. We end with some concluding remarks. We take examples from the domain of one of the authors, the control software for consumer electronics products, but include examples from other domains as well.

5.2 Why Use Software Product Lines?

Let us first summarize *why* we want to create software product lines. In Figure 30, we show the main four drivers as we identified them within Philips at the left hand side. We show three key solutions in the middle. But it's the combination of these three that forms the most promising solution to our problems: software product lines.

In the next subsections, we discuss each of the drivers and their relation to the solutions. Please note that arrows in Figure 30 show main dependencies only; arguments can be found for other dependencies as well, but adding them would obscure the main flow of the picture.

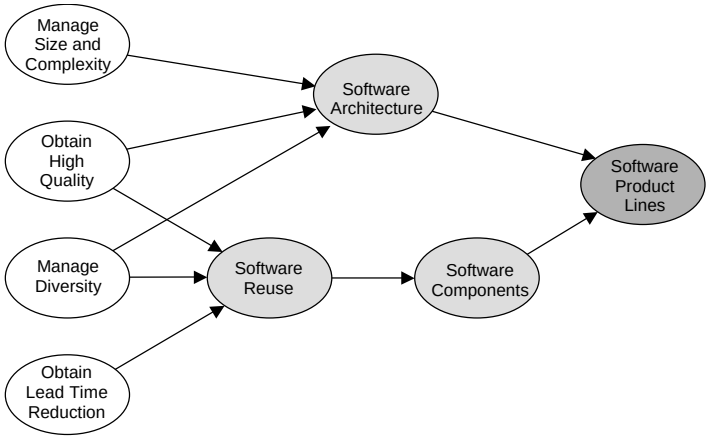

Figure 30. Main drivers for software product lines

5.2.1 Manage Size and Complexity

Software plays a role of ever-increasing importance in many products of today. In Philips televisions, software was introduced in the late 70s, when a 1-kiloByte 8-bit microprocessor was first used to control the hardware signal processing devices. The size of the software grew following Moore's law closely: today, a TV typically has 2 MegaBytes of software. The same trend can be observed in other products, though the point in time where software starts to play a role differs. Typically, televisions run 10 years behind on PCs, while shavers run 10 years behind on televisions.

The best – perhaps only – way to manage size and complexity is to have a proper architecture. A software architecture defines, among other things, a decomposition of a system in terms of components or modules [39]. One common misconception (at least in our surroundings) is the assumption that components defined to manage size and complexity, automatically help to satisfy other drivers as listed in Figure 30 as well, most notably diversity. Experience has shown this not to be the case: defining components to manage diversity is an art per se.

5.2.2 Obtain High Quality

The second driver in Figure 30 is quality: the more complex systems become, the more difficult it is to maintain a high quality. Yet for systems such as televisions, high quality of the software is imperative[4]. Users of a television do not think they are communicating with a computer, so they will not accept crashes. Also, although lives may not seem to be involved, a TV catching fire can have disastrous effects, and it is partly the software that prevents this from happening.

Quality can be achieved by having a proper architecture that guarantees (or at least helps to satisfy) certain quality attributes. Quality can also be achieved by reusing software that has already been tested in the field.

5.2.3 Manage Diversity

The third driver in Figure 30 is the first one specific for product lines: managing the efficient creation of a variety of products. For consumer products, we have 'small-scale' diversity within a family of televisions: the output device, the price setting, the region, the user interface, the supported standards, interconnectivity et cetera. We have 'large-scale' diversity if we want televisions, VCRs, DVD/CD players/recorders, etc. to be part of one product line approach. There is indeed sufficient commonality to justify the sharing of software between those products. We use the term *product population* to denote the large-scale diversity. Other examples of product populations can be found in the medical domain [50] and for printers and scanners [9].

There are actually two schools of thought for managing diversity, which we believe are two extremes in a spectrum of solutions. One is to build an architecture that expresses the commonality between products, with variation points to express the diversity. The other is to rely on reuse of software components for the commonality, but to use different compositions of these components to implement diversity. This paper is about balancing between these two solutions, variation and composition, depending on the scope of the products and the characteristics of the organization.

5.2.4 Lead-time Reduction

The fourth driver in Figure 30 is lead-time reduction. Although in principle also relevant for single product development, we think that it is especially relevant in product line contexts, since the goal of product lines is to produce a range of products fast. The main solution to obtain lead-time reduction is reuse of software. Experience has shown that higher-level languages, modern development methods,

[4] Repairing software shipped in televisions sold to a million customers is as difficult as in a rocket sent a million miles into space.

better architectures and advanced software development environments have indeed improved software productivity, but not with a factor that is sufficient to obtain the speed required in product lines.

5.2.5 Does Software Reuse imply Software Components?

Figure 30 may suggest that software reuse implies the deployment of reusable software components. Indeed, reuse of software components is one (compositional) approach to build product populations, and one that we will examine in this paper. But we should note that we can also have reuse *without* software components, for instance when using inheritance in an object-oriented framework. And, perhaps surprisingly, we can also build a product line that utilizes software component technology without actually reusing the components! An example of the latter is a framework with component plug-ins, where the plug-in components implement *diversity* rather than *commonality*. Figure 31 illustrates this: the left hand side shows a reusable framework with product-specific plug-ins, while the right hand side shows reusable components, composed in a product-specific way.

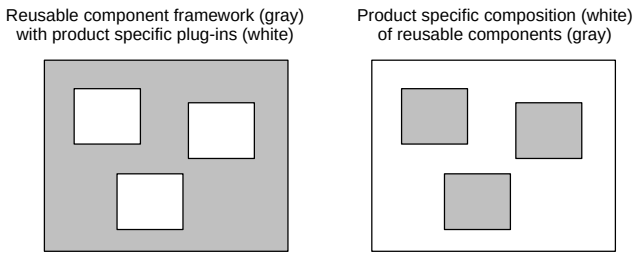

Figure 31. Variation (left) and composition (right) are complementary approaches

5.2.6 Finding the Balance

In the rest of this paper, we concentrate on the balance between the use of variation and composition in software product lines, depending on the scope of the product population and the characteristics of the organization building it.

5.3 The Influence of Scope on Software Product Lines

As we have said before, many software product line approaches advocate an early analysis of commonality and variation of the products to be built, resulting in a variant-free architecture with variation points. This is indeed a valid approach, but there may be circumstances in which the preconditions for such an approach are not satisfied. We shall discuss two categories of these in the following sections: (1) product related and (2) organization related.

5.3.1 From Product Family to Product Population

Traditional product lines cover families in which products have many commonalities and few differences. This enables the creation of a variant-free architecture with variation points to implement the differences. The variation points typically control the internals of a component, but they may also control the presence or absence of certain components, the selection of a concrete component to implement an abstract component (a place holder), or even the binding between components.

A typical product family in the consumer domain is the set of televisions (direct view, flat screen, projection) produced by a single company[5]. Let us now extend the product range to include video recorders, CD and DVD players, CD, DVD and hard disk recorders, audio amplifiers, MP3 players, tuners, cassette players, et cetera. These products still have sufficiently in common to warrant reuse of software. But they are also sufficiently different to require different architectures.

The latter is actually not a fundamental problem. In principle, it seems perfectly possible to create a single architecture for a 'super consumer product', being the combination of all products mentioned above, and to strip this architecture for any subset to be sold. While this is certainly possible in hindsight, it is very difficult to predict which combinations of audio and video elements will actually be interesting next year, and to include that already in the architecture.

Instead, the market itself thrives on novel combinations of existing functions (for instance a mobile telephone with built-in digital photo camera). We believe that in such an evolving 'compositional' market, the software architectures must also be compositional. Put differently, variation can be used to implement diversity known a priori, while composition can be used for open-ended diversity.

5.3.2 From Business Line to Product Division and Beyond

The second argument in favor of composition has its roots in organization. A large company like Philips is typically organized in product divisions (Semiconductors, Consumer Electronics), which are split into business units (video, audio), which again are split into business lines (Flat TV, projection TV). See Figure 32 for an illustration.

Within a business line, setting up a software product line is generally relatively easy. One person is in charge of the business, which has its own profit and loss calculation, and a product line is demonstrably an efficient way to create a variety of products. The products are reasonably alike (e.g. all high-end Flat TVs). Moreover, the product road maps in a business line are usually carefully aligned,

[5] For technical reasons, low-end televisions usually belong to a different family than high-end televisions.

thus enabling the creation and use of shared software. A single architecture with variation points is very feasible.

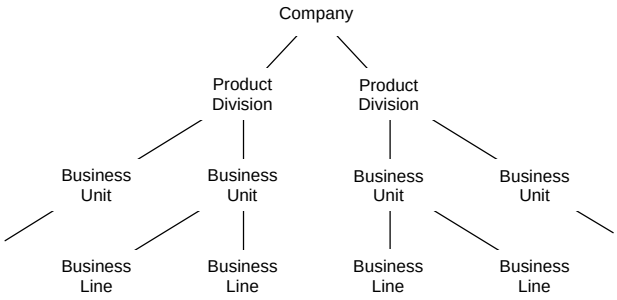

Figure 32. Schematic organization of a large company

Compare this with a business unit. Although – of course – still one person is in charge, profit and loss responsibility is delegated to the business lines, making sharing software between business lines at least cumbersome. Each business line manager has his own agenda for planning software development. Protection of IP (intellectual property) is sometimes an issue, but more frequently the misalignment of roadmaps causes problems: one business line is changing software with its product release still far away, while another product line needs a mature version of the software for its immediate next release. A careful and explicit road mapping is required here [78]. Products show more differences, but in general a single architecture with variation points is still be feasible.

Problems get worse at the level of a product division. Not only the products become more different (e.g. TV versus set-top box), the managers more apt to look after themselves instead of after their common interest, and the roadmaps harder to align, but also the cultures between organizations become different. In Philips, the TV business is a traditional and stable consumer business (low profit per product but high number of products, targeted directly at the person in the street), while the set-top box business is new and evolving (currently targeting at large providers who deliver the boxes free with their services). TVs are typically programmed in C in a few MegaBytes, while set-top boxes are programmed in C^{++} in many MegaBytes.

Finally, product divisions act as independent companies (at least in Philips), and while sharing of software between them is in many cases infeasible (as between medical systems and consumer electronics), it may in certain situations still be desirable. Philips Semiconductors produces for instance many of the chips for Philips Consumer Electronics, and it also sells the same chips to third parties, so certain software can be better leveraged if developed by the semiconductors division. This puts extra demands on protection of IP. Cultures between product divisions are even more different, and the levels of software engineering maturity may differ.

5.3.3 How to Share Software in Large Organizations?

It should be obvious by now that the top-down creation of a single architecture with variation points to cover all consumer products within Philips is infeasible. Instead, we should find a way to let different business lines create the software components in which they are specialized (e.g. the business line DVD creates a DVD unit), and find ways to 'arbitrarily' compose them.

Should we then strive for a bottom-up approach as advocated by many believers in software component markets, i.e. (2) instead of (4) in Figure 29? The answer is no: we still need a careful balance between top-down and bottom-up, as we need to make sure that components fit easily, as explained in the next section.

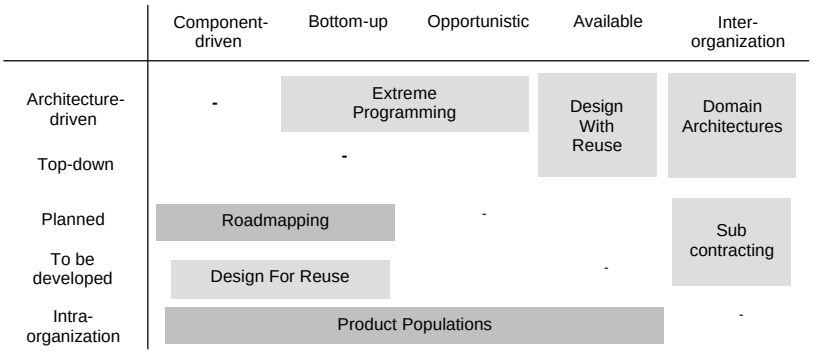

Figure 33. Balancing between architecture- and component-driven

5.3.4 Balancing Architecture- and Component-Driven

Classical product and product line approaches ((1) and (3) in Figure 29) are architecture-driven, top-down, planned, intra-organization, with all software to be developed in the project. Component market evangelists ((2) in Figure 29) on the other hand, claim that software development should be component-driven, bottom-up, opportunistic, inter-organization and using components that are already available. Figure 33 shows both sets of in fact opposite characteristics along different axes. Note that they are not all dependent – we can make cross combinations:

- *Extreme programming* could be seen as an opportunistic bottom-up yet architecture-driven method.
- Top-down, architecture-driven development deploying available software is sometimes called *design with reuse*, while on the other hand component-driven bottom-up development is sometimes called *design for reuse*.
- A top-down, architecture-driven development spanning multiple organizations is possible but only if there is consensus on a *domain architecture*.

- The planned development across organizations of new software is often called *subcontracting*, and should not be confused with the development of reusable software[6].

Our vision is to achieve a component-driven, bottom-up, partly opportunistic software development using as much as possible available software to create products within our organization; we call this a *product population* approach. In reality, life is not so black and white. More specifically, we believe that such an approach should be:

- Component-driven (bottom-up) with a lightweight (top-down) architecture, since absence of the latter would result in architectural mismatches [33].
- Opportunistic in using available software but also planned when developing new software, the latter carefully roadmapped [78].
- Intra-organization but spanning a large part of the organization.

In a way, building product populations is more difficult than producing and utilizing software components on a component market. In the latter case, laws of economics apply: component vendors can leverage development costs over many customers (as compared to few in product populations), while component buyers can find second sources in case vendors does not live up to expectations. One would expect that the same economical model could be used *within* an organization, but in practice this does not work: contracts within a company are always less binding than between companies.

5.3.5 Third Party Component Markets

Does this mean that we not believe in third party component markets [109] then? Yes, we do, but not as the only solution.

First of all, most software components on the market tend to implement generic rather than domain specific functionality. While this ensures a large customer base, it does not help companies like Philips to build consumer products. Put differently: of course we reuse generic components such as a real-time kernel from third parties, but there is still a lot of software to be created ourselves. This situation may change with the arrival and general acceptance of TV middleware stacks.

Secondly, many companies distinguish between *core*, *key* and *base* software. Base software can and should be bought on the market. Key software can be bought but is made by the company itself for strategic reasons. Core software cannot be bought because the company is the only company – or one of the very few –

[6] Within Philips, we have seen a single team that had as mission the development of reusable software act in practice as two teams subcontracted for two different product projects.

actually capable of creating such software. But there may still be a need to share this software within the company!

Thirdly, many components out on the market are still relatively small (buttons, sliders), while the actual need for reuse is for large components. Of course there are exceptions: underlying Microsoft's Internet Explorer is a powerful reusable HTML displaying *and* editing engine that can be used in different contexts (such as Front Page and HTML Help). But such examples are still exceptions rather than a rule.

Finally, to be accepted in a market, functionality must be mature, so that there is consensus in the world on how to implement and use such functionality. This process may take well over 20 years – consider operating systems and windowing systems as examples.

5.4 The Dimensions of Variation and Composition

Figure 34 shows variation and composition as two independent dimensions. The variation axis is marked with 'As is' (no variation), parameterization, inheritance and plug-ins, as subsequently more advanced techniques of variation. The composition axis is marked with no composition (yet reuse), and composition. The table is filled with some characteristic examples, which we briefly discuss in the subsections following the table. In the sections following, we zoom into variation and composition with more elaborate examples.

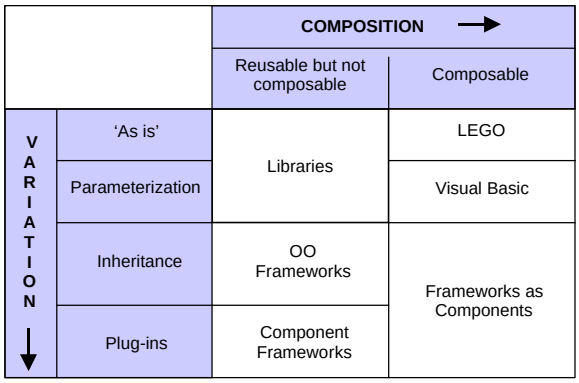

Figure 34. Variation and composition as two independent dimensions

5.4.1 Plain 'Old-Fashioned' Libraries

The classical example of software reuse is through libraries. Examples are a mathematical library, a graphical library, a string manipulation library, et cetera. The functions in the library are either used 'as is', or they can be fine-tuned for their specific application using a (procedural) parameter mechanism.

Typically, software deploying the library includes the header file(s) of the library, and thus becomes specifically dependent upon the library. It is possible to abstract

from the library by defining a library-independent interface; POSIX is an example. The build process may then select the actual library implementing this interface. However, in practice, most software built on top of libraries is still specifically dependent upon those libraries.

This is especially true for the library itself. Often, a library uses other libraries, but few of them allow the user to select which underlying libraries to use. Moreover, while some libraries are relatively safe to use (they define specific functions only such as ‘sin’ and ‘cos’), other libraries may introduce clashing names (String), or have clashing underlying assumptions (both define the function ‘main’).

The conclusion is that while libraries are a perfect low-level technical mechanism, more advanced techniques are necessary to achieve the variation and composition required for product lines.

5.4.2 Object-Oriented Frameworks

An object-oriented framework is a coherent set of classes from which one can create applications by specializing the classes using implementation inheritance. An example is the Microsoft Foundation Classes framework [92]. The power lies in the use of inheritance as variation mechanism, allowing the framework to abstract from specific behavior, to be implemented by the derived classes. A base class usually has a very intricate coupling with its derived classes, the base class calling functions implemented in the derived class and vice versa.

This intricate coupling is also one of the weaknesses of this approach. A new version of the framework, combined with old versions of the derived classes may fail due to subtle incompatibilities; this is known as the *fragile base class problem* (see [103] for a treatment). Also, since many frameworks are applications themselves, combination of two or more frameworks becomes virtually impossible.

5.4.3 Component Frameworks

A component framework is (part of) an application that allows to plug-in components to specialize behavior. Component frameworks resemble OO frameworks; the main difference is the more precise definition of the dependencies between the framework and the plug-in components. The plug-ins can vary from quite simple to arbitrarily complex, in the latter case enabling the creation of a large variety of applications.

Component frameworks share some of the disadvantages with OO frameworks. First of all, the plug-in components are usually framework dependent and cannot be used in isolation. Secondly, the framework often implements an entire application and can therefore not easily be defined with other frameworks.

5.4.4 LEGO

The archetype non-software example of composition is LEGO, the toy bricks with the standard 'click and play' interface. Indeed, LEGO bricks are very composable, but they support no variation at all! This results in two phenomena.

First of all, it *is* possible to create arbitrary shapes with standard LEGO bricks, but one must use thousands of bricks to make the shapes smooth (see Figure 35 for a pirate's head). As the components are then very small compared to the end product, there is effectively not much reuse (not more than reusing statements in a programming language).

To solve this, LEGO manufactures specialized bricks that can only be used for one purpose. This makes it possible to create smooth figures with only few bricks. The down side is that specialized bricks can only be used in specialized cases.

Figure 35. The pirate's head with ordinary LEGO bricks[7]

A solution could be parameterized bricks, which indeed is a mechanical challenge! Maybe a LEGO printer, a device that can spray plastic, could be an answer here![8]

5.4.5 Visual Basic

The archetype software example of composition is Visual Basic. Actually, Basic is the non-reusable glue language, where ActiveX controls are the reusable components (later versions of VB also allow to create new ActiveX controls).

One of the success factors of Visual Basic is the parameterization mechanism. An ActiveX control can have dozens (hundreds) of properties, with default values to alleviate the instantiation process. Another parameterization technique is the event

[7] From http://www.lego.com

[8] Strictly speaking, a LEGO printer would be the equivalent of a code generator, which is a technique for building product lines that we do not tackle in this paper, although it can be very fruitful in certain domains (cf. compiler generators). See also [23].

mechanism, which allows the signaling of (asynchronous) conditions, where the application can take product specific actions.

Still, ActiveX controls are usually quite small (the Internet Explorer excepted), and it is not easy to create application frameworks (e.g. supporting a document view control paradigm such as the Microsoft Foundation Classes do).

5.4.6 Frameworks as Components

The ultimate solution for variation *and* composition lies – we believe – in the use of (object-oriented or component) frameworks as components. We have seen that frameworks provide powerful variation mechanisms. If a framework only covers part of an application domain, and if frameworks can be arbitrarily combined (making them *true* components), then we can create a multitude of products by selecting and combining them and then specializing them towards a product.

There are some examples of OO frameworks used as components [11], but no examples of component frameworks used as components yet. The basic mechanism for achieving composition is to make every context dependency explicit and bindable by a third party, as we will see in a later section.

5.5 Variation Further Explained

Let us zoom into variation first, and then discuss composition in more detail.

5.5.1 Why do we Need Variation?

The basic argument why we need variation is the one underlying all product line development: we need to implement diversity. We can achieve this with reusable components, but an interesting dilemma pops up then. The more useful we make a component (read: larger), the less reusable it becomes. Conversely, the more reusable we make a component, the less useful it may become, the ultimate being the empty component: extremely reusable while not very useful! See also [103], Fig 1.1).

The solution for this dilemma is *variation* as a way to fine-tune components when instantiating them into a product. The archetype example is the Visual Basic ActiveX control, which usually comes with a long list of properties that can be modified at design- and/or run-time. These properties also have default values: this alleviates the product creation; otherwise the burden on the designer may be too high.

5.5.2 Types of Variation

A variation point can be implicit or explicit, open or closed, unbound or bound, and flexible or fixed [11]. Not all sixteen combinations are meaningful; see Figure 36 for an overview. When development starts, a variation point is implicit; it must be

made explicit during analysis and design. An explicit variation point can be open if the set of possible values can still be extended, or closed otherwise. A variation point is unbound if a value has not been selected, and bound otherwise. A variation point is fixed if it's bound and the value cannot be changed anymore.

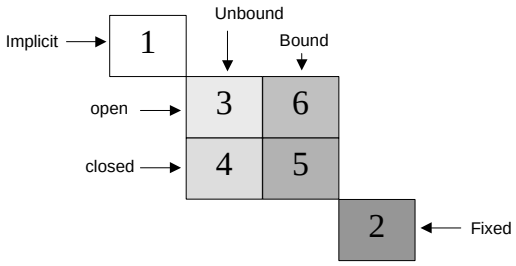

Figure 36. A lifecycle for variation points

Traditional single product development starts at (1) in Figure 36 and then immediately jumps to (2). Classical product family development proceeds through (3), (4) and (5) (identify a variation point, identify its values, and then choose a value) before ending in (2). There is a tendency to make (5) the end point of development, allowing the variation point still to change at the customer site. There is even a tendency to make (6) the end point of development, allowing the customer to extend the set of values.

5.5.3 Example 1: Switches and Options

In our previous television software architecture, we distinguished between *switches* and *options*. A switch is a compile-time variation point (an #ifdef), closed, bound and fixed somewhere during development time. An option is a run-time variation point, a value stored in non-volatile memory and used in an if-statement, so closed yet unbound during development, and bound in the factory or at the user's home.

These variation points were used for adjustments of hardware, for presence or absence of components, and rarely also for selecting between components. The set of variation points was very specific for the product family, and could not be used to achieve the large-scale structural variation that we needed to widen our family.

5.5.4 Example 2: The Windows Explorer

The windows explorer is an example of an application with many explicit, open, unbound and flexible variation points. It can be tuned into almost any browsing application, though it cannot be used as component in a larger application. In other words, it supports a lot of variation, but no composition.

The explorer can be tuned at a beginner's level and at an advanced level [59]. In the first case, simple additions to the windows registry suffice to change the file association (the application to be run for a certain type of file), the context menus

or the icons. In the second case, handlers must be registered for specific shell extensions, and these handlers must be implemented as COM components. When such a handler crashes, the explorer crashes, with all its windows including the desktop[9].

The folder view of the explorer can be modified by name space extensions; some FTP clients use this to add favorite sites to the name space. The folder view is part of the explorer bar, which can also host other views (search, favorites, history). The right pane normally hosts a standard list view control, but it can also host the Internet Explorer to view (folders as) web pages, or any other user defined window.

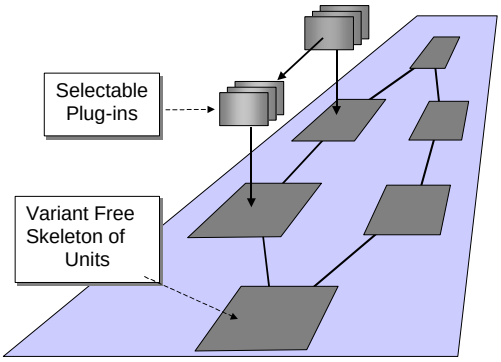

Figure 37. An architecture in the medical domain

5.5.5 Example 3: A Framework Architecture

Wijnstra [112] describes a software architecture in the medical domain that deploys component frameworks to manage diversity. Figure 37 provides a schematic overview of this architecture. It consists of a fixed structure of *units*, COM components (mostly executables) running on a Windows NT computer. Each unit is a component framework in which COM components can be plugged to implement the diversity of the product family. Plug-in components may talk to each other, but the initial contact is made through the main units.

In this architecture, the fixed structure of units[10] implements the common part of the family, while the plug-ins take care of the differences between the products. It is therefore an example of variation rather than composition (see the left hand side of Figure 31). Of course, this is true if the plug-ins are relatively small and specific for one product only. In real life, some plug-ins are rather large and shared between multiple members of the family. In fact, if one can create products by taking

[9] It is possible to run the desktop in a different process to prevent such crashes.

[10] Some units are optional.

arbitrary combinations of plug-ins, then this system exhibits signs of composition rather than variation.

5.6 Composition Further Explained

We shall now discuss composition as a technique for managing diversity.

5.6.1 What is Composition?

We speak of *composition* if two pieces of software that have been developed *without* direct knowledge of each other, can be combined to create a working product. The components may have been developed in some common context, or they may share interfaces defined elsewhere. The essential element is that the first piece of software can be used without the second, and the second piece without the first.

The underlying argument for using composition to handle diversity is the large number of combinations that can be made if components are not directly dependent on one another. Note that composition excludes the notion of plug-ins, since these are directly dependent on the underlying framework, and cannot be used in isolation.

A nice analogy may be the problem of describing a language, where an infinite number of sentences must be produced with a finite (and hopefully small) set of ‘components’. Typically, such components are production rules in a grammar, where the production rules function both as a composition hierarchy as well as a set of constraints on the composition.

Garlan has illustrated how difficult composition can be in his well-known paper on Architectural Mismatches [33]. It appears that two components being combined must at least share some common principles to work together efficiently. We call this a lightweight architecture, but leave it to another paper to elaborate on how to create such an architecture.

Szyperski [103] defines components as ‘having explicit context dependencies only’, and being ‘subject to composition by third parties’. In Koala [72], the former are called (explicit) requires interfaces, while the latter is called third-party binding. Tony Williams recognized the same two elements in an early paper leading to OLE [113], though an interface is called a ‘base class’ there, and the third party a ‘creator’.

In the following subsections we provide examples of compositional approaches.

5.6.2 Example 1: The ‘Pipes and Filters’ Architectural Style

One of the oldest examples of composition to create a diversity of applications with only few components, is the pipes and filters architecture of Unix. With a

minimum of common agreements: character based I/O, a standard input and output stream, and a new line as record separator, with a small set of tools such as *grep*, *sed*, *awk*, *find* and *sort*, and with essentially one simple composition operator, the '|', one can program a large variety of tasks, including small databases, automatic mail handling and software build support.

Pipes and filters are considered an architectural style in [98], and indeed there are more examples with similar compositional value. Microsoft's DirectShow [58] is a streaming architecture that allows the creation of 'arbitrary' graphs of nodes that process audio and video signals. The nodes, called *codecs*[11], are implemented as COM servers, and may be obtained from different vendors. The graph can be constructed manually with a program that acts as a COM client. A simple type mechanism allows automatic creation of parts of such a graph, depending on types of input and output signals. Other companies have similar architectures, e.g. Philips with the Trimedia Streaming Software Architecture (TSSA) [106].

A third kind of pipes and filters architectures can be found in the domain of measurement and control. LabView [64] is an environment in which measurement, control and automation applications can be constructed as block diagrams and subsequently compiled into running systems. See also [16].

5.6.3 Example 2: Relation Partition Algebra

Software architectures are normally defined a priori, after which an implementation is created. But it is also possible to 'reverse engineer' an implementation to extract a software architecture from it, if the architecture has been lost. Even if the architecture has not been lost, it is still possible to extract the architecture from the implementation and verify it against the original architecture. Discrepancies mean that either the implementation is wrong, or the original architecture is 'imperfect' (to say the least).

Tools can be built to support this process, but a large variety of queries would result in a large range of tools. As it turns out, mathematical set and relation algebra can be used to model 'use' and 'part-of' relations in software effectively; queries can be expressed mostly in one-liners. A compositional tool set for this algebra is described in [79]; a similar technique is also deployed by [35].

5.6.4 Example 3: COM, OLE and ActiveX

Microsoft's COM [94] is a successful example of a component technology, but this does not automatically imply that it supports *composition* from the ground up. COM can be seen as an object-oriented programming technique, its three main functions, *CoCreateInstance*, *QueryInterface* and *AddRef/Release*, being

[11] For *co*ding and *dec*oding of streams of data.

respectively the new operator, the dynamic type cast, and the delete operator. COM's main contribution is that it supports object orientation[12] 'in the large': across language, process and even processor boundaries.

If a COM client uses a COM service, the default design pattern is that it creates an instance of that service. It refers to the service by CLSID, a globally unique identifier that is nevertheless specific for that service, thus creating a fixed dependency between the client and the service. In other words, clients and services are (usually) not freely composable. To circumvent problems thus arising, some solutions have been found, most noteworthy the *CoTreatAs*, the *Category ID* and the connectable interfaces. The latter provide a true solution, but are in practice only used for notifications.

Microsoft's OLE [17] *is* an example of an architecture that allows free composition, be it of document elements (texts, pictures, tables) rather than software components. An OLE *server* provides such elements, while an OLE *container* embeds them. An application may be an OLE server and container at the same time, allowing for arbitrarily nested document elements. An impressive machinery of interfaces is needed to accomplish this, as a consequence, there are not many different OLE containers around (though there are quite a few servers).

Microsoft ActiveX controls are software components that are intended for free composition. In general, they do not have matching interfaces, making it necessary to glue them together. The first language for this was Visual Basic; later it became possible to use other languages as well. An impressive market of third party ActiveX controls has emerged.

5.6.5 Example 4: GenVoca, Darwin and Koala

GenVoca [8] is one of the few approaches that supports explicit *requires* interfaces to be bound by third parties. GenVoca uses a functional notation to combine components and is equipped with powerful code generation techniques. It has been applied to e.g. graph processing algorithms, where different combinations of the same set of components can be made to create a variety of applications.

Darwin [52] is a research language for the specification of distributed software architectures. It supports components with typed *provides* and *requires* interfaces, and it allows compound components to instantiate and bind other components. Darwin has been applied to the control and monitoring of coal mining, and - as Koala - to control software for televisions (see below).

Koala [72] is an industrial architectural description language with a resource friendly component model used for the creation of embedded control software for consumer products. Koala supports both composition and variation. Koala

[12] Be it without implementation inheritance, a 'mishap cured in .NET'???

components can be instantiated and their (typed) interfaces can be bound by compound components as shown in Figure 38, thus forming a hierarchy of components. Variation is supported in the form of diversity interfaces, a special form of requires interfaces that resemble properties in ActiveX controls, and switches, a kind of pseudo dynamic binding. Various products are already on the market, created from different combinations of different subsets of a repository of reusable components. Koala supports adaptation in the form of glue modules as a fundamental technique for binding components.

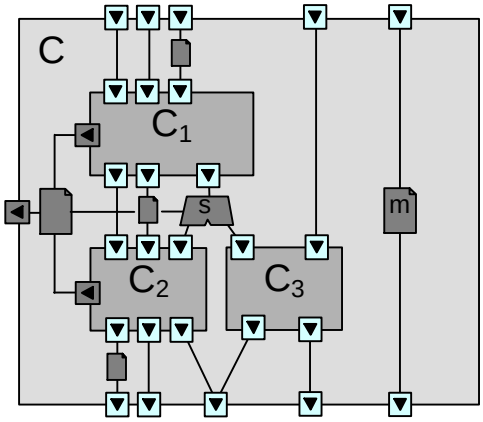

Figure 38. The Koala component model

5.6.6 Composability is Actually a Form of Variation!

We have discussed variation and composition. We presented them as fundamentally different, but it is possible to unify the concepts. In all examples that we have seen, components become 'freely composable' when context dependencies have been turned into explicit variation points. This may be in the form of an input stream or of connectable (requires) interfaces. Some approaches make every context dependency explicit (e.g. Koala), while others only make certain dependencies explicit (like in the pipes and filter architectures).

5.7 Concluding Remarks

Let us first recapitulate. Managing complexity and diversity while maintaining high quality and short lead-times have become important software development issues. Reuse, architecture and components are key solutions to achieve this. Originally, we envisaged reuse to happen 'automatically' once software components were available on the market. Later we discovered that sharing of software actually requires a pro-active, systematic approach: that of software product lines. The first product lines rely heavily on (limited) *variation* in an otherwise variant-free architecture. With product scope increasing, and software

development starting to cross-organizational borders, we believe that more powerful techniques are required to handle diversity, and discuss *composition* as one such a technique.

This does not mean that we're back to third party component markets. Instead a careful balance between architecture-driven top-down and component-driven bottom-up development is required, and we explain why and how. Also, composition does not replace variation, but rather they should be seen as two independent dimensions. We typify various points in the two-dimensional space with archetype examples, before explaining variation and composition individually in more detail. To make things as concrete as possible, we illustrate variation and composition with a rich set of well-known examples.

Within Philips, we have moved from variation to composition in our architectures for certain sets of consumer products. We find that we can handle diversity better, and cover a larger range of products even though development is decoupled in time and space across organizations. We find that few other companies use composition, which surprised us originally. We then discovered that the few companies that do use composition also cover a wide product scope, or cross-organizational boundaries, or exhibit both phenomena.

5.7.1 Acknowledgements

We would like to thank attendants of ARES workshops, the SPLC conference, and the Philips 'Composable Architectures' project for many fruitful discussions on this topic.

Chapter 6

Configuration Management

Published as: *Configuration Management in Component Based Product Populations*, Rob van Ommering, 10th International Workshop on Software Configuration Management, May 14-15, Toronto, 2001, Canada, also published in LNCS 2649, p16-23.

Abstract: The ever-increasing complexity and diversity of consumer products drives the creation of product *families* (products with many commonalties and few differences) and product *populations* (products with many commonalties but also with many differences). For the latter, we use an approach based on composition of software components, organized in packages. This influences our configuration management approach. We use traditional CM systems for version management and temporary variation, but we rely on our component technology for permanent variation. We also handle build support and distributed development in ways different from the rest of the CM community.

6.1 Introduction

Philips produces a large variety of consumer electronics products. Example products are televisions (TVs), video recorders (VCRs), set-top boxes (STB), and compact disk (CD), digital versatile disk (DVD) and hard disk (HD) players and recorders. The software in these products is becoming more and more complex, their size following Moore's law closely [13]. Additionally, the market demands an increasing diversity of products. We have started to build small software *product families* to cope with the diversity within a TV or VCR family. But we believe that our future lies in the creation of a *product population* that allows us to integrate arbitrary combinations of TV, VCR, STB, CD, DVD and HD functionality into a variety of products.

This paper describes the configuration management approach that we devised to realize such a product population. We first define the notion of product family and product population, then describe some of our technical concepts. We subsequently discuss five aspects of configuration management, and end with some concluding remarks.

6.2 Product Family and Population

We define a *product family* as a (small) set of products that have many commonalties and few differences. A software product family can typically be created with a single variant free architecture [88] with explicit variation points [41] to control the diversity. Such variation points can be defined in advance, and products can be instantiated by 'turning the knobs'. Variation points can range from simple parameters up to full plug-in components. An example of a product family is a family of television products, which may vary - among other things - in price, world region, signal standard and display technology.

We define a *product population* as a set of products with many commonalties but also with many differences [73]. Because of the significant differences, a single variant free architecture no longer suffices. Instead we deploy a composition approach, where products are created by making 'arbitrary' combinations of components. An example product population could contain TVs, VCRs, set-top boxes, CD, DVD and HD players and recorders, and combinations thereof. These products have elements in common, e.g. a front-end that decodes broadcasts into video and audio signals. These products are also very different with respect to for instance display device and storage facility.

It is true that any product population of which *all* members are known at initiation time, can in principle be implemented as a product family, i.e. with a single variant free architecture. In our case, two reasons prevent us from doing so. First of all, our set of products is dynamic - new ideas for innovative products are invented all the time. Secondly, different products are created within different sub organizations, and it is just infeasible - both for business as well for social reasons - to align all developments under a single software architecture.

6.3 Technical Concepts

In this section we briefly describe the software component model that we use, and show how components are organized into packages and into a component based architecture.

6.3.1 The Koala Component Model

We use software components to create product populations. Our component model, Koala, is specifically designed for products where resources (memory, time) are not abundant [70], [72]. Basic Koala components are implemented in C, and can be combined into compound components, which in turn can be combined into (more) compound components, until ultimately a product is constructed. The Koala model was inspired by Darwin [52].

Koala components have explicit provides *and* requires interfaces. Provides interfaces allow the environment of the component to use functionality

implemented within the component; requires interfaces allow the component to use functionality implemented in the environment. All requires interfaces of a component must be bound explicitly when instantiating a component. We find that making all requires interfaces both explicit and third-party bindable, greatly enhances the reusability of such components.

Provides and requires interfaces of components are actually instances of reusable *interface definitions*. We treat such interface definitions as first class citizens, and define them independently from components. Such interface definitions are also common in Java, and correspond with abstract base classes in object-oriented languages.

Diversity interfaces, a special kind of requires interfaces, play an important role in Koala. Such interfaces contain sets of parameters that can be used to fine-tune the component into a configuration. We believe that components can only be made reusable if they are heavily parameterized, otherwise they either become too product specific or they become too generic and thus not easily or efficiently deployable.

Note that the Koala terminology slightly differs from that used by others. A Koala component *definition*, or more precisely a component *type* definition, corresponds closely with the notion of *class* in object-oriented languages. A Koala component *instantiation* corresponds with an *object*. What Szyperski calls a *component*, a binary and independently deployable unit of code [103], resembles mostly a Koala package as described in the next section, i.e. a set of classes and interfaces. It is true that we distribute source code for resource constrained reasons, but users of a package are not allowed to change that source code, only compile it, which inherently is what Szyperski meant with 'binary'.

6.3.2 Packages

We organize component and interface definitions into *packages*, to structure large-scale software development [75]. A package consists of public and private component and interface definitions. Public definitions can be used by other packages - private definitions are for use in the package itself only. This allows us to manage change and evolution without overly burdening the implementers and users of component and interface definitions.

Koala packages closely resemble Java packages, where component definitions in Koala resemble classes in Java, and interface definitions in Koala resembling interfaces in Java. Unlike in Java, the Koala notion of package is *closed*; users of a package cannot add new elements to the package.

6.3.3 The Architecture

With Koala we have created a component-based product population that consists of over a dozen packages, each package implementing one specific sub domain of

functionality [74]. Products can be created by (1) selecting a set of packages appropriate for the product to be created, (2) instantiating the relevant public component definitions of those packages, and (3) binding the diversity and (4) the other requires interfaces of those components. Product variation is managed by making choices in steps (1) - (4).

6.4 Configuration Management

We now come to the configuration management aspects of our product population development. We distinguish five issues that are traditionally the domain of configuration management systems, and provide our way of dealing with these issues. These issues are:

- Version management;
- Temporary variation;
- Permanent variation;
- Build support;
- Distributed development.

The issues are discussed in subsequent subsections.

6.4.1 Version Management

In the development of each package, we use a conventional configuration management system to maintain a version history of *all* the programming assets in that package, such as C header and source files, Koala component and interface definitions, Word component and interface data sheets, et cetera. Each package development team deploys its own configuration management system - see also the section that describes our distributed development approach.

A package team issues formal *releases* of a package, where each release is tagged with a package version identification (e.g. 1.1b). The version identification consists of a major number, a minor number, and a 'patch letter'. The major and minor version numbers indicate regular releases in the evolution of the package - the 'patch letter' indicates bug-fix releases for particular users of the package.

Each formal release of a package must be internally consistent. This means that in principle *all* components should compile and build, and run without errors. In practice, not every release is tested completely, but it *is* tested sufficiently for those products that rely on this specific version of this package.

Customers of a package only see the formal releases of the package. They will download the release from the intranet (usually as ZIP file), and insert it in their local configuration management system. They will only maintain the formal

release history of the package, and will not see all of the detailed versions of the files in the package.

Each release of a package must be backward compatible with the previous release of the package. This is our *golden rule*, and it allows us to simplify our version management - only the last release of each package is relevant. Again in practice, it is sometimes difficult to maintain full backward compatibility, so sometimes we apply our *silver rule*, which states that all existing and relevant clients of the package should build with the new release of the package.

6.4.2 Temporary Variation

For temporary variation we use the facilities of traditional configuration management systems. We recognize two kinds of temporary variation.

Temporary variation *in the small* arises if one developer wants to fix a bug *while* another developer adds a feature. By having developer specific branches in the CM system, developers do not see changes made by other developers *until* integration time. This allows them to concentrate on their own change before integrating changes made by others into their system.

Temporary variation *in the large* arises just before the release of a product utilizing the package. Note that packages are used by multiple products - this is what reuse is all about. We cannot run the risk of introducing bugs into a product coming out soon when adding features to a package for a product to be created somewhere in the future. This is why we create a *branch* in the package version history for a specific product just before it is being released. Bug fix releases for such a product are tagged with 'patch letters', e.g. 1.2a, 1.2b et cetera. As a rule, no features may be added in such a branch, only bugs may be fixed.

Although the branch may live for a number of weeks or even months - it may take that long to fully test a product - we still call this a temporary branch since the branch should be closed when the product has been released. All subsequent products should be derived from the main branch of the package. There is also *permanent variation* in our product population, but we handle that completely *outside* of the scope of a configuration management system, as described in the next section.

6.4.3 Permanent Variation

For permanent variation, i.e. the ability to create a variety of products, we do not deploy a traditional configuration management system, but rely on our component technology. We have a number of arguments for doing so:

- It makes variation explicit in our architecture, instead of hiding it in the CM system

- It allows us to make a late choice between compile time diversity and run-time diversity, whereas a CM system typically only provides compile time diversity.
- It allows us to exercise diversity outside the context of our CM system (see also section 6.4.4).

The second item may require further clarification. Koala diversity interfaces consist of parameters that can be assigned values (expressions) in the Koala component description language. The Koala compiler can evaluate such expressions and translate the parameters into compile-time diversity (#ifdefs) if their value is known at compile time, or run-time diversity (if statements) otherwise. This allows us to have a *single* notion of diversity in our architecture. We can then still decide at a late time whether to generate a ROM for a *single* product (with only the code for that product), or a ROM for a set of products (with if statements to choose between the products). Note that always generating a ROM for a single product complicates logistics, but on the other hand always generating a ROM for multiple products increases the ROM size and hence the bill of material.

Put differently, we expect that once (binary) components are commonplace, the whole configuration issue will be handled at run-time, and not by a traditional CM system.

With respect to the first item mentioned above, we have four ways of handling diversity [75]:

- Create a single component that determines by itself in which environment it operates (using *optional* interfaces in Koala, modeled after *QueryInterface* in COM).
- Create a single component with diversity interfaces, so that the *user* of the component can fine-tune the component into a configuration;
- Create two different (public) components in the same package, so that the *user* of the package can select the right component;
- Create two different packages for different implementations of the same sub domain functionality, so that *users* may select the right package (before selecting the right public component, and then providing values to diversity parameters of that component).

It depends on individual circumstances which way is used in which part of the architecture.

6.4.4 Build Support

The fourth issue concerns build support, a task normally also supported by traditional CM systems. We deliberately uncoupled the choice of build environment from the choice of CM system, for a number of reasons.

First of all, we cannot (yet) *standardize* on one CM tool in our company. And even if different groups have the same CM system, they may still utilize incompatible versions of that system. So we leave the choice of a CM system open to each development group (with a preferred choice of course, to reduce investments). This implies that we must separate the build support from the CM system.

Secondly, we do not *want* to integrate build support with CM functionality, because we want to buy the best solution for each of these problems, and an integrated solution rarely combines the best techniques. So our build support currently utilizes our Koala compiler, off-the-shelf C compilers and a makefile running on multiple versions of make. We deploy Microsoft's Developer Studio as our IDE.

Thirdly, we want to be able to compile and link products while *outside* the context of a CM system, e.g. when in a plane or at home, or in a research lab. Although not a compelling argument, we find this approach in practice to be very satisfying, because it allows us to do quick experiments with the code at any place and at any time.

6.4.5 Distributed Development

The fifth and our final issue with respect to configuration management concerns distributed development. Recently, many commercial CM systems provide support to distribute the CM databases over the world. We are not in favor of doing that for the following reasons.

First of all, it forces us to standardize again on *one* CM system for the entire company, and we have shown the disadvantages of this in the previous section.

Secondly, we want to be able to utilize our software *outside* the context of the CM system, and relying on a distributed CM system does not help here.

Thirdly, we think this approach doesn't scale up. We envisage a 'small company approach' for the development of packages in our company. You can 'buy' a package, and download a particular release of that package, *without* having to be integrated with the 'vendor' of that package in a single distributed CM system. Imagine that in order to use Microsoft Windows, you would have to be connected to their CM system!

This concludes the five issues that we wanted to tackle in this paper.

6.5 Concluding Remarks

We have explained our component-based approach towards the creation of product populations. We briefly sketched our Koala component model and the architecture that we created with it. We then listed five configuration management issues, some of which we may have tackled in different ways than is conventional. The issues are:

- Version management of files and packages;
- Temporary variation for bug fixes, feature enhancements and safeguarding products;
- Permanent variation as explicit issue in the architecture;
- Build support separated from configuration management;
- Distributed development using a 'small company model' rather than a distributed CM system.

The approach sketched in this paper is currently being deployed by well over a hundred developers within Philips, to create the software for a wide range of television products.

Chapter 7

Building Product Populations

Published as: *Building Product Populations with Software Components*, Rob van Ommering, International Conference on Software Engineering, Orlando, US, May 2002, p255-265.

Abstract: Two trends have made reuse of embedded software for consumer electronics an urgent issue: the software of individual products becomes more and more complex, and the market demands a larger variety of products at an increasing rate. For that reason, various business groups within Philips organize their products as product families. A third trend is the integration of functions that until now were only found in separate products (e.g. a TV with Dolby Digital sound and a built-in DVD player). This requires software reuse between product families, which - when organized systematically - leads to a product population approach.

We have set up such a product population approach, and applied it in various business groups within our organization. We use a component technology that stimulates context independence, and allows the composition of new products out of existing parts. We use an architectural description language to explicitly describe the architecture, and also to generate efficient bindings. We have aligned our development process and organization with the new 'compositional' way of working. This paper outlines our approach and reports on our experiences with it.

7.1 Introduction

The last decade has seen a growing interest in software architecture to build complex, high-quality systems. Also, various component technologies have emerged, significantly improving software reuse. These two phenomena are combined in software product lines [49][24], allowing companies to efficiently create a variety of complicated products with a short lead-time. This paper reflects our experiences in setting up a component based software product line in the field of embedded software for consumer electronics (CE).

By 1996, Philips already had quite some experience in developing a large range of televisions in all regions of the world. The hardware was reasonably modular, but the software relied mainly on 'ancient' principles to handle diversity: compiler switches, run-time options, and - if the differences became too large - 'copy and edit'. It was clear to us that this way of working could not be continued.

Moreover, new products loomed on the horizon, containing new and increasingly complex combinations of existing functionality (see Figure 39) such as TVs with built-in VCR or DVD players or recorders, with enhanced sound, and implementing digital TV standards (formerly the task of a separate 'set-top box'). This required integration of pieces of software developed at different points in time, and in different parts of the organization. In other words, we needed a compositional approach, allowing to 'arbitrarily' combine existing software assets.

Figure 39. Example consumer products

At that time, various component technologies existed (COM, Corba, JavaBeans), but none was suited for resource-constrained environments such as televisions (which typically run 10 years behind on a PC in computing resources). So we could either find another solution for our diversity problem, or adopt a component technology and adapt it to our specific needs. As only the latter would bring us on the learning curve of applying component technology, our research question became (in 1996):

> *Can we benefit from component technology in resource-constrained environments **now**, so that we can already adjust our development process and organization? And how will the latter be affected?*

As a result, we created the Koala component technology ('96-'97) [72] and subsequently used it to set up a product line architecture for CE products ('98-'99). This product line has been running for two years now, with several products out on the market. In this paper we provide an overview of the overall approach.

This paper is organized as follows. In sections 2-5 we sketch an outline of our approach, starting from the *b*usiness context, going via *a*rchitecture to development *p*rocess and *o*rganization (we call this scheme BAPO). Section 6 lists our experiences and compares them to other work. We complete our paper with concluding remarks, acknowledgements and references.

7.2 Business

In this section we sketch our business context.

7.2.1 The Product

A television consists of mechanics, electro-optics, electronics and software. Over the past 15 years, the size of the software in CE products has followed Moore's law closely (see Figure 40). In the beginning, the software just switched devices on and off, but this was soon extended to detection and control loops, data processing (Teletext, Electronic Programming Guide) and advanced user interfaces (menus, animation, 3D graphics).

It typically takes 100 software engineers 2 years to build the software for a high-end television. This is partly due to the large number of control algorithms that have to be implemented, and partly to the resource constraints that have to be taken into account. The Bill of Material (BoM) is an important issue in CE products, as it largely determines the price; development cost can be divided by the number of products sold (millions). Note that the systems are (still) *closed*: the code is burnt into ROM and can only be updated by a service engineer by replacing the ROM.

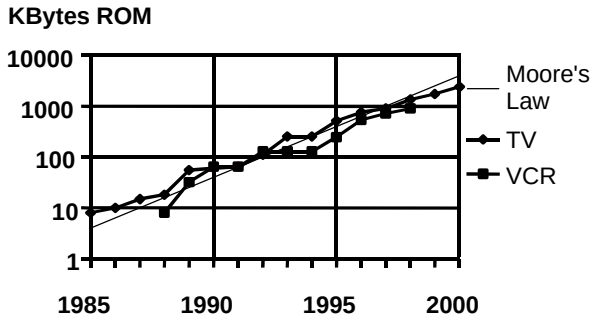

Figure 40. Growth of embedded software in CE products

7.2.2 The Product Family

Televisions are sold in many variants. There is variation in screen size, picture and sound quality, data processing features, user interface, output device (tube, projection, LCD), broadcasting standards and interconnectivity. There is also significant diversity for different regions in the world, caused by for instance language issues and cultural differences. The result is essentially a matrix of product types at different price points and for different regions.

The software is influenced by most of these factors. The original strategy for handling this diversity was to create a particular high-end TV for one region first, then extend the functionality to other regions, and then subset this for lower price points. Two kinds of diversity parameters were recognized: run-time *options* and compile-time *switches*. Options are stored in a non-volatile memory that is programmed in the factory, allowing the use of a single ROM for multiple product types. Switches operate at the source code level (mostly #ifdef), and require recompilation when setting values differently.

7.2.3 Problems in Handling Diversity

Various problems arose in handling diversity. We name a few:

- The distinction between options and switches requires an early decision between compile-time and run-time diversity. But the selection of which features go into a ROM and which ROMs are used for which products should preferably be done at a late point in time (depending on code size and factory logistics).
- The list of parameters grows over time, as new features are added. More importantly, the 'language' in which parameters are expressed is often of the wrong level, for instance relating to specific driver parameters or to specific products.
- It was difficult to design-in new features, especially when optionally replacing components, or when inserting code between existing components (cf. instrumented connectors as defined by Balzer [6]).
- Last but not least: the approach did not scale to product populations. Software used from other organizations was usually 'copied and edited', instead of 'reused as is'.

7.2.4 The Product Population

The products that we envisage in the near future - combinations of old and new functionality such as improved sound and picture (home cinema), storage (VCR, DVD, hard disk), digital TV, interactivity et cetera - all require the ability to combine existing parts in new ways. In the hardware we are already succeeding in this; we have for instance plug-in modules for Dolby Digital, for digital reception (a 'set-top box'), and for storage. We want the software to be equally flexible.

The classical approach to building product families is to define an overall architecture for the family and introduce variation points a priori for modeling the required diversity. A good example can be found in [112], which contains a variant-free architecture [88] with a plug-in component mechanism as variation points. However, for at least two reasons we cannot define such an overall architecture in practice:

- It appears to be difficult in practice to define *the* architecture of future CE products in advance, as so many non-software factors play a role, for instance hardware availability, hardware technology, market demands, company strategy, et cetera.
- Different parts are produced in different sub-organizations, each with their own goals, time scales, history and culture. Even agreement on for instance naming conventions proves to be a burden. It is out of the question that consensus can be reached on a single global architecture.

We have coined the term *product population* [73] for this problem domain, where we want to build a set of products with many commonalties but also with many differences, with development of parts spread over different sub-organizations within a larger organization. We claim that we need a component technology that stimulates the development of freely combinable components, while at the same time we recognize that we must define at least some architectural issues globally.

7.3 Architecture

This section discusses the Koala component model, together with some typical design patterns.

7.3.1 Components

Figure 41 shows a Koala component, with two 'provides' interfaces at the top, a code module *in* the component, and two 'requires' interfaces at the bottom. An interface is a small set of semantically related functions (as in COM and Java), and serves as the unit of binding. The triangles denote the direction of function calls. A code module implements all functions in interfaces bound with the tip (of the triangle) to the module, and may use any function of interfaces bound with the base to the module. There may be more than one code module in a component, bound to different interfaces. To delay the decision on the actual binding technique between components, modules use *logical names* to implement and use functions in interfaces. We currently use C as our implementation language.

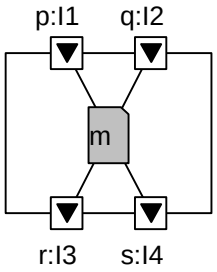

Figure 41. A Koala component

Each interface has an instance name (e.g. p in Figure 41) and a type (I1). A component can provide more than one interface, each interface implementing one aspect of the component. Different components may provide interfaces of the same type, making them interchangeable with respect to those aspects. Interface types are managed separately from components (see section 7.4.3).

Koala's striking feature is that *all* communication of a component with its context is routed through requires interfaces that are bound by a third party, even access to for instance the underlying operating system. This makes components to a large

extent context-independent: they rely on *services* only, rather than on specific *servers* (read: implementations of services).

7.3.2 Connectors

Figure 42 shows three ways of binding interfaces of components: a straight connection between interfaces (1), the use of a *switch* (2), and a code module gluing two interfaces (3). Each form is a special case of its right neighbor, and can be expressed in terms of that.

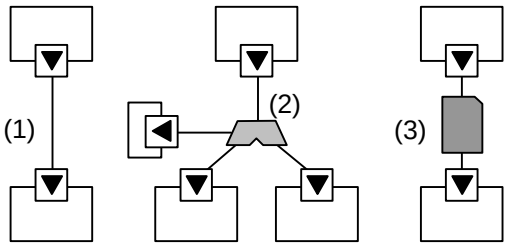

Figure 42. Koala's forms of binding

A *straight connection* between interfaces couples every function in the 'tip' interface to the function with the same name in the 'base' interface. For this, it is sufficient if the type of the tip interface is a *super*-type (i.e. a *sub*-set) of that of the base interface. We introduced this feature to allow for interface *instances* to grow over time (see section 7.3.8); it turns out to be convenient for diversity interfaces also (see section 7.3.5).

A *switch* is a pseudo dynamic binding of one interface to one out of a set of other interfaces. The setting of the switch is controlled by a function of yet another interface, for instance of a component that configures the system (the small one in Figure 42). A switch can have more than two positions, and can bind more than one interface at the same time. If Koala can determine the position of the switch at compile-time (as explained in section 7.3.4), it reduces the switch to a straight connection.

A *glue module* allows to insert code between the called functions (in the tip interface) and the implementing functions (in the base interface). We anticipate that in product populations components will not always fit perfectly, hence this facility. It also allows to insert tracing and logging code, and for instance thread-synchronization code (see section 7.3.7). Glue functions can be implemented in a simple expression language (a subset of C) in Koala, or directly in C. As far as Koala is concerned, there is no difference between a code module as shown in Figure 42 and the glue module in Figure 42. A switch is equivalent to a conditional expression in a glue module.

7.3.3 Architectural Description Language

Koala's composition process is recursive: a connected set of components is again a component (see Figure 43 for an example). A *configuration* is a top-level component in this hierarchy with no interfaces on the border. Only configurations can be compiled and linked into executables.

To make things work, components and interfaces are described in an architectural description language. An *interface definition* declares the prototypes of functions, parameters (read: nullary functions) and constants (read: parameters with a value assigned in the interface definition). A *component definition* declares provides and requires interfaces, the modules and instances of other components that it contains, and their inter connections.

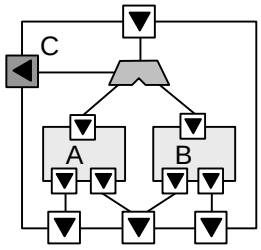

Figure 43. A compound Koala component

The Koala compiler reads all definitions, instantiates a designated top-level component, and generates code for all connections. For a large part, this code will consist of #defines that equate logical names to physical names, as explained in the next section. Where necessary, C code is generated to resolve binding at run-time (e.g. for some of the switches). Note that due to the code generation, descriptions in Koala are by definition consistent with the actual implementation of the products.

Koala's ability to deal with product *populations* lies in the fact that *all* knowledge about the connection between components such as A and B in Figure 43 are property of the compound component C. By creating a different compound component, say C', a different combination of a different subset of the reusable components A, B, … can be made for a different product.

7.3.4 Partial Evaluation

Koala has a *partial evaluation* mechanism that uses constant folding to simplify expressions wherever possible. Basically, all types of binding (see Figure 43) are translated to function bindings using Koala expressions and/or C code. Then, the Koala compiler simplifies the Koala expressions as much as possible - it cannot optimize the C code. As a result, many of the bindings are reduced to a simple #define, while others result in simple pieces of C code (e.g. if-statements).

A specific example can be found in Figure 43, which implements a variant component C that behaves as either A or B, depending on a function (say *f*) in the gray interface. If *f* is assigned a constant in a Koala expression at some outer-level in the component hierarchy, then Koala reduces the switch to a straight connection with no overhead whatsoever. If, on the other hand, *f* is defined as a piece of code that calculates the setting of the switch at run-time, then Koala generates C code for the switch to implement the run-time binding.

This feature of Koala removes the distinction between compile-time switches and run-time options, at least for the builders of reusable components: they just rely on functions in interfaces. Of course, at some outer-level, the decision between A and B still must be made. But it can be made by the person creating the *product*, hence *late* in the development process. If that person still fixes the choice of the switch to say A, then we speak of *late compile-time binding*. Alternatively, the person can assign code that reads the value from a non-volatile memory, making the choice between A and B a run-time option.

7.3.5 Handling Diversity

Technical diversity has two aspects: diversity *within* a component, and diversity of the connections *between* components. We shall discuss both in turn.

Diversity *within* a component concerns the ways in which a component should behave if applied in different products. No two products demand exactly the same of a component. This requires some sort of component parameterization. We believe that non-trivial components often require a long list of diversity parameters (some people call them *properties*), most of which are set at design time (or retain their default value), while others are set at run-time.

Koala as described above already has all of the features needed to implement a powerful diversity parameter mechanism. Diversity parameters are grouped into *diversity interfaces*, which are in fact just requires interfaces. Glue code in the form of Koala expressions (or C code if necessary) can be used to assign values to parameters. Koala's partial evaluation mechanism simplifies the Koala expressions as much as possible, so that the original generic code can be fine tuned into resource-friendly specific code.

Figure 44 shows an example of a parameterized component A that embeds a parameterized component B. The parameters of B are assigned values in terms of the parameters of A. This allows to use different 'languages' for parameters at different level. For instance, component B could be a 'picture in picture' component with a window border with programmable color. Component A could be a TV platform that is dependent on the region. The module m can then specify the color in terms of the region.

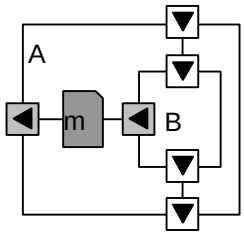

Figure 44. Components with diversity parameters

One particular use of interface sub-typing is to make the types of diversity interfaces of components as specific as possible, i.e. specifying precisely which diversity parameters the components require. These (different) *requires* interfaces can be connected to a single (large) interface at the product level, containing the union of all parameters, and provided by a (product specific) configuration component. This makes very explicit which component uses which diversity parameter, compared with traditional techniques where all components include the global diversity parameter file.

Diversity of the connections *between* components is expressed in terms of switches (conditional Koala expressions) in the bindings between components (see Figure 43 for an example). As already explained above, Koala switches are used both for compile-time and run-time diversity. When creating the product, some switches are reduced to straight connections, while for others code is generated.

7.3.6 Notifications

One recurring design pattern is the use of notifications to signal the occurrence of (asynchronous) events. Possible implementation techniques range from callback functions to notification managers in the infrastructure. Our design goal for components is to make as few assumptions about the environment as possible *and* be resource friendly, therefore we reduce a notification to its bare minimum: an outcall through a bindable requires interface.

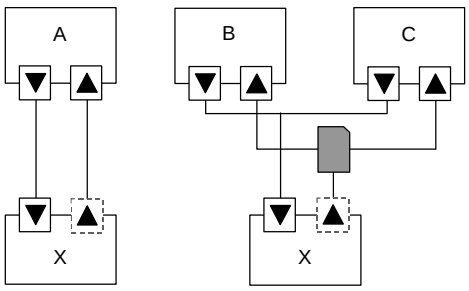

Figure 45. Notification Interfaces

The left side of Figure 45 illustrates this. Component X can notify the environment through an outgoing interface, and Component A can implement this interface. The

outgoing interface is dashed, meaning that it need not be connected (if no-one is interested in the notification). Koala ensures that X can observe whether there is a connection, to prevent X from calling into void.

By convention, functions in notification interfaces have no return values, and implementations may not perform any significant processing in such functions. We discuss mechanisms to decouple the processing from the notification in the next section.

Koala only allows *one* interface to be connected to a requires interface. If more than one component is interested in receiving the notification, then glue code has to be added, as shown in the right hand side of Figure 45. A simple multicast can be used if B and C are interested in *all* notifications. More glue code is needed if they can only handle notifications that follow their own actions.

The mechanism sketched above is very light in weight and for us sufficient in 90% of the cases. Note that we prefer to choose the lightweight solution as default and add complexity only when necessary. However, our architecture *does* have components with more elaborate notification mechanisms. We use for instance call back functions to notify the arrival of Teletext pages.

7.3.7 Multithreading

Software in CE products typically contains many activities that are relatively independent. A real-time kernel provides the ability to program asynchronous tasks (a.k.a. threads), and there is a body of knowledge on how to satisfy real-time constraints [45]. However, resource limitations prevent the creation of too many tasks (say more than a dozen), whereas the number of activities easily exceeds one hundred. As a consequence, task creation becomes a *system* issue rather than a *component* issue.

An easy way out is to define the threads of the system a priori in the architecture and build components to use specific threads (e.g. of high or low priority). But in a product population this approach is not desired, as it assumes too much knowledge in advance. Our basic solution is therefore to let components use *logical* threads that are mapped to *physical* threads at the product level by using Koala's diversity mechanism. This also enables the fine-tuning of the performance of the system at a late stage in the development process.

To be able to map *multiple* logical threads to a single physical thread, we introduce the notion of pumps and pump engines. A pump is a logical message queue with a single dispatch function; a pump engine is a physical message queue with a (physical) thread. Messages sent to a pump are actually handled by the pump engine to which the pump is allocated. Components create pumps on virtual pump engines obtained through diversity interfaces. At the product level these are bound to physical pump engines. Note that this makes part of the execution architecture visible in the Koala bindings!

Figure 46 demonstrates the use of pumps to decouple notification processing. Component A issues a notification that is handled in component C by sending a message to a pump. At a later point in time, the pump message is handled by the pump engine allocated through a diversity interface, and on that pump engine's thread, C can make a down call to B. The component F contains the pump engines defined at the product level.

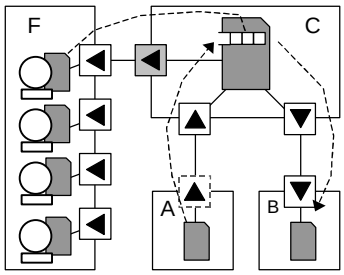

Figure 46. Decoupling notifications

If two components containing pumps are mapped to the same pump engine, then their activities are *implicitly* synchronized, and therefore need no *explicit* synchronization. But this mapping is only defined at the product level, so a component builder has no access to this information, and would be tempted to make his component thread-safe. As this would increase the complexity and resource usage, by default components need not be thread safe. Instead, the product builder should add synchronization where needed.

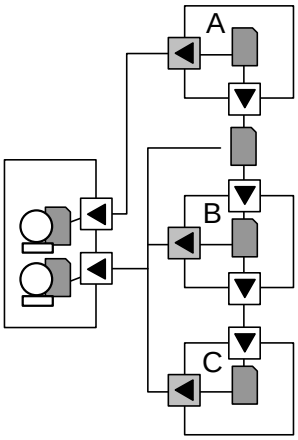

Figure 47. Synchronizing components.

Figure 47 illustrates this. Components B and C are mapped to the same pump engine and do not require synchronization. But A is running on a different engine, so when A calls B, the inserted glue code must synchronize with B's and C's pump engine.

7.3.8 Coping with Evolution

An important design criterion for Koala was the ability to add functionality to a component without disturbing existing users of that component. Consider a component C that is used in product P and to be used in product Q as well. For Q, functionality must be added to C. Of course, existing products P out in the market will never see the change to C, but if a variant of P is to be produced, then preferably the new version of C must be used (e.g. to avoid double maintenance), and must still work in that context.

We have defined several rules of evolution.

- Interface definitions may *not* be changed (after they have been frozen in development). New 'versions' of an interface definition must have a new name.
- Component definitions *may* change, internally, of course, but also with respect to the externally visible interfaces. The following three kinds of modification are allowed:
  - *Add* a provides *interface* of a new type (e.g. an ITuner2 next to an ITuner);
  - *Widen* a provides *interface* to a sub(!)-type;
  - *Narrow* a requires *interface* to a super-type.

A fourth kind of modification, removing a requires interface, is not allowed, as the Koala compiler will complain when a non-existing interface is bound. It is however allowed to have a requires interface that is internally not connected. Also note that what we describe above is a form of sub-typing, more specifically of covariance and contra-variance. See e.g. [103] (section 6.3) for more information.

7.4 Development Process

We developed the Koala component model and the accompanying product population architecture initially within Research. When we started to apply this within our business groups, we found that the established software development processes needed adaptation. We report the main changes in this section.

7.4.1 Composition versus Decomposition

The first and most important change is a psychological one. Software developers traditionally start from a (single) system specification. They decompose the system into subsystems, the subsystems into components (see the left hand side of Figure 48), and then implement the components and integrate them into subsystems and ultimately the system.

We want components to be combined in multiple ways into subsystems, and subsystems in multiple ways into systems. In other words, we want *composition* rather than *decomposition* (see Figure 48), or a *graph* rather than a *tree* as design hierarchy.

One fundamental difference is that in a decomposition approach it does not matter *where* a certain feature is implemented, as long as it is implemented somewhere in the system. In a composition approach this *does* matter, since some components may be present in other systems, while others may not be included.

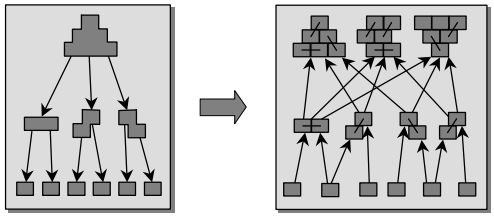

Figure 48. From decomposition to composition

So, developers have to start thinking in terms of components that can be clicked together in different ways to obtain different sets of functionality. Moreover, they cannot ask for *the* product specification anymore, since one of the major risks is that components become too product specific. Koala provides several means to separate product-specific information from components, e.g. the use of requires and diversity interfaces.

The other end of the spectrum is that developers fall into a 'genericity' trap, and start developing the ultimate reusable component. Careful roadmapping of the product population must prevent this, as will be explained in section 7.4.6.

7.4.2 Documentation

Traditionally, software documentation consists of a requirements document, a global design defining the various subsystems, and per subsystem a detailed design defining the components. The documentation is written in the future tense ('the system *shall*'), *before* the software is created, and is rarely updated if changes are made to the design during development. Documentation of reusable components, however, should be written in the present tense ('the component *does*'), and should reflect the *actual* implementation instead of the original design.

For that reason we write *data sheets* for components, inspired by datasheets for electronic devices such as ICs. A datasheet is a document of typically 10-20 pages that describes the component from an external (user) point of view. The data sheet starts with a Koala picture of the component, then lists the main 'selling' features of the component, lists the interfaces, and provides observable implementation properties of the component such as code size and performance data. It concludes with application notes, showing typical usage of the component. The data sheet is

written *before* the development starts, and is updated when the component has been completed.

Component datasheets list interfaces by name and type, but refer to *interface data sheets* for the semantics of those interfaces. An interface data sheet is a document of typically 10-20 pages that starts with an overview of the concepts behind the interface (a 'model'), and then lists the functions with informal descriptions of their semantics (much in the style of Java documentation). The document concludes with application notes (how to *use* the interface) and optionally implementation notes (how to implement the interface). Note that we describe *all* interfaces in interface data sheets, even if an interface is only provided or required by a single component.

A third category of documents is the *component implementation notes*, which describe how the component is built internally out of other components and glue code. Of course, this document refers to the data sheets for the subcomponents. Furthermore it lists the major design decisions. It does *not* explain the implementation in great detail; for that we just add comments to the code.

7.4.3 Repository

Component and interface definitions are stored in a repository that does *not* reflect the design hierarchy. Definitions of compound components are stored next to definitions of their subcomponents, and both are equally reusable. Put differently, component *instances* can be nested, but component *definitions* cannot. The preferred way of creating new products is to start by selecting and combining large compound components. Only when a large compound component does *not* satisfy the requirements, should a product creator turn to more basic components.

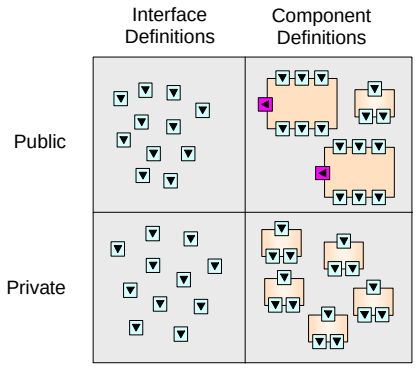

Figure 49. A package

We do however have some structure in our repository, and that is in the form of *packages*. A package is a set of interface and component definitions, where each definition is labeled public or private (see Figure 49). Public definitions can be used in other packages; private definitions are limited to the package itself. As a

rule, a package is developed by a single team in a separate project at a single site. The scoping thus introduced helps to streamline the development process: changes to private definitions can be made more easily than changes to public definitions.

A package can contain a single large public compound component that acts as a subsystem in products, and a set of smaller private components from which the compound component is constructed. Typically however, a package contains more than one public compound component, each with a different combination of a different subset of the smaller private components, so that it implements a different set of requirements. Also, packages sometimes contain small public components that can be used as glue or as 'plug-ons'. See [75] for more information on our ways of handling diversity.

7.4.4 Configuration Management

The established way of managing the software in our organization was to put all software in a single configuration management (CM) system, and use it to manage *revisions* (versions in time) and *variants* (versions in space). The CM system also manages the *build* process and *multi-site development*. We feel that product populations put different requirements on the CM strategy, and describe here how we deviate in our approach.

Our software development is essentially multi-organization, multi-site. Each site hosts one or more projects, and each project develops one or more packages. As a rule, a project either creates packages with reusable software (called 'subsystems' for historic reasons), or packages that contain end products. As each package is developed at a single site, changes *within* the package are much easier to achieve then changes that involve multiple packages.

Each project has its own CM system that manages the sources of the packages allocated to that project. Each project publishes its packages on the intranet in two forms: daily as browsable directory structure, and - roughly - monthly in the form of tested releases downloadable as ZIP file. The first is to enhance communication and discussion, the second helps to safe guard development. Each team downloads releases of other packages on demand, and imports them into their own CM system, thus maintaining a fine-grained history of their own packages and a coarse-grained history of other packages.

The CM systems are used to manage revisions (versions over time) and temporary variants (one developer fixes a bug while another adds a feature). We do *not* use our CM systems to manage permanent variation: we use Koala instead! The most important reason is that this brings variation into the realm of the architects instead of the configuration managers. Furthermore, Koala has powerful facilities for handling variation, such as switches that can be set at a late point in the development process or left as run-time option.

We also want to keep our build process separate from our CM system (which is not possible if the CM system handles variation). Reasons vary from practical (more efficient, easier to use off-line, e.g. in the plane!) to strategic (no lock-in on CM vendors). More information on our CM approach can be found in [76].

7.4.5 Development Environment

Our development environment consists of a layered set of tools. The use of each tool is in principle optional, though each tool depends on all of the previous tools (and in practice, most developers use all of the tools).

- The Koala compiler, which reads component and interface definitions and generates C code and header files.
- 'KoalaMaker', which produces a makefile to compile Koala components into object code.
- A set of master makefiles to control the build process on different platforms.
- Microsoft Developer Studio, for editing and debugging code.
- A small set of Developer Studio plug-ins to help developers perform frequent tasks.

The first three tools run on Unix and PC. The first two tools are optimized for speed, and process the code for a product in less than a minute (on a 256MB 450MHz Pentium III, with 500 component and 1000 interface definitions). A large part of the software runs and can be tested on a PC; in fact, we use Koala's diversity facilities to support our software on multiple platforms.

7.4.6 Integration and Testing

Testing is done at four levels: testing of individual components, of consistency *within* a package, of consistency *between* packages, and of products. We shall discuss each in turn.

Within a package, each Koala component is tested individually by building a simple test application around the component that allows to exercise the component's functionality. Koala's features of binding and gluing make this much simpler then in conventional approaches. Functionality required by a component is provided either by using other (already tested) components, or by stubbing the functions, depending on the complexity of the functionality. The test application is stored in the repository just as any other component. The component developer is responsible for defining and performing the tests.

Packages are tested for *internal* consistency by thoroughly testing all the public components of the packages with various diversity settings before each release. For functionality required by the package, we use (tested) releases of other packages if

there is a *strong* dependency (as explained below), or stubs if there is a weak dependency. Package testing is the responsibility of a separate team within the same project.

A package *p* has a *strong dependency* to a package *q* if it uses so much of the functionality of *q* that development cannot proceed without having *q* available. We discourage strong dependencies between packages because they sequentialize development. Figure 50 shows a simplified example. TV services cannot be built without a TV platform, and the Electronic Program Guide and Teletext Services cannot be built without a Teletext platform. But applications (user interfaces) can be built on a UIMS by stubbing the TV functionality. Note that all packages need the (computing) infrastructure.

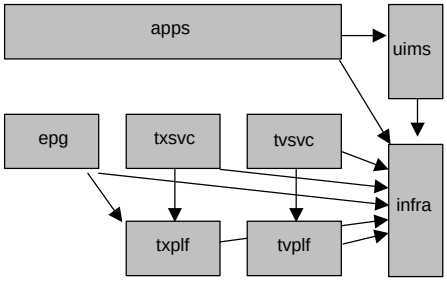

Figure 50. Strong dependencies between packages

To check the consistency *between* packages, an integration of the packages is required. In principle, every product project performs such an integration. To prevent different teams from running into the same problems, we have a separate project that performs a pre-integration, by building one or more reference products.

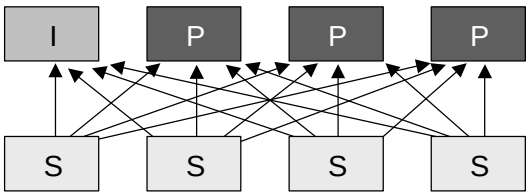

Figure 51. Subsystem deliveries to products

Figure 51 illustrates this approach. The subsystems (boxes marked with ‘S’) are delivered *independently* to products (marked with ‘P’). This decouples development processes considerably. When a subsystem reaches a stable version, products that can incorporate the version may do so, while products that are in a critical phase of release may keep on using an older version. The advantage is speed: in traditional platform approaches, a subsystem innovation must await platform integration before being made available to products. The downside is the added complexity of guaranteeing compatibility between multiple versions of subsystems. See [77] for an inventory of problems in independent deployment.

The last form of testing is at the level of products. Software and hardware are submitted to a long series of tests before the product is taken into production. Each product is tested individually before release. Our product population approach currently does not give benefits here, other than that new products which are minor variants of existing products need less testing for the parts that have been unchanged.

7.4.7 Roadmapping

We favored composition over decomposition in section 7.4.1, but in fact a careful balance between the two is required when building product populations. Too much emphasis on decomposition may result in components that are too product specific, whereas too much emphasis on composition may result in components that are too generic and (partly) never used. The (pure) composition approach has more disadvantages:

- Features cannot be removed since there may be products that use or plan to use them.
- Product plans can only be made if the components are already available, but new product features usually require updates of the (reusable) components.

The answer is roadmapping: the planning which components are developed when and by whom, and how they are used in products in the next few years. For hardware components this is already common practice in our organization; for software components we are currently installing such a process.

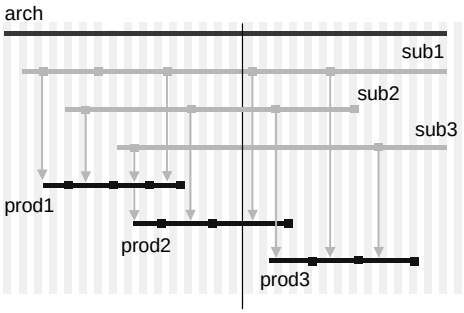

Figure 52. Roadmapping subsystems and products

The basics of roadmapping are sketched in Figure 52. The top horizontal line depicts the architecture, which constantly evolves over time (the horizontal axis). The gray lines represent subsystem packages and their evolution. The bottom (black) lines represent products. The vertical arrows denote dependencies (deliverances). Figure 52 is symbolic: it does not depict a real-life situation.

At this moment we have more than 20 subsystem packages and over 10 products. As a result, we cannot represent all releases and all dependencies in a single

picture. Moreover, we do not want to maintain such a road map centrally, as it concerns too much information. Therefore, we are distributing the roadmap over the projects, letting each project specify the roadmap of its packages. This is done in XML [115] and is published by the projects on the company intranet. Simple tools can then be used to check the consistency of the overall roadmap.

More information on our roadmapping can be found in [78].

7.5 Organization

In the previous section we have seen how our approach influenced the development process. The software development organization also had to be adapted to support building product populations. This section describes how we did that.

7.5.1 Four Types of Organization

Figure 53 shows four types of organization that we encounter in our company. The first, top-left, is classical. Different teams are set-up for creating different products. Each team creates *all* of the software for one product. If there is any reuse, then it happens by copying code from one product project to another.

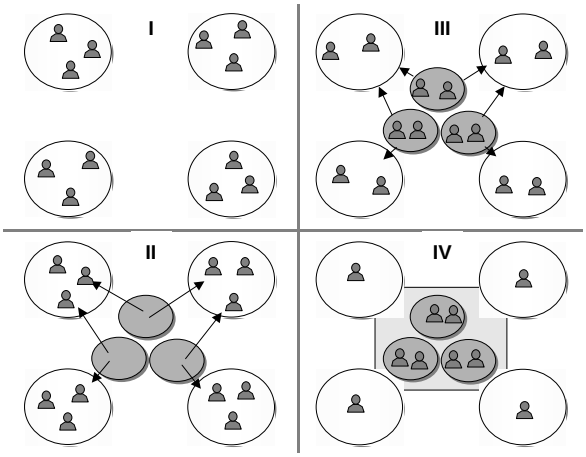

Figure 53. Types of organization

The second type, at the bottom-left, consists of product teams and capability teams. The capability teams act as internal software house: they lend their people to the product projects, and these people are physically located at the site developing the product. It is more efficient than the first type, since we can create experts in certain areas, such as video, audio, or data processing, or user interfaces. Reuse happens because capability team members bring their software with them when they join a product project.

In the third type of organization, at the top-right of Figure 53, the members of capability teams remain located at their own site, but are directly paid by the

product projects. The capability teams strive for the creation of a single software package; the members paid by the product projects have as task to ensure the suitability of the package for the specific products.

A variant on this type is when product teams send their people to work with capability teams for a period of time, again to ensure that the software matches the product's requirements.

The fourth type, sketched at the bottom-right, contains an internal department that creates and sells reusable components within the company. The capability teams still consume requirements from different product teams, but it is their own responsibility to convert them into functionality.

We have made these four types explicit, since many believe the fourth type to be the true answer to software reuse, whereas the first and second type are actually occurring in most organizations. Our organization is currently of the third type. Our ambition is to move towards the fourth type of organization, but we have strong doubts whether that form of internal software reuse will actually be feasible on the short term. We believe that product population development requires a mix of types III and IV.

7.5.2 Introducing the Approach

We have introduced this product line successfully into part of our organization: it is now being used in four business groups for the creation of high-end TVs with classical picture tubes, Flat TVs, projection TVs, and mid-range TVs. We have not succeeded yet in integrating the software development of for instance DVDs and VCRs. We feel that it may be useful to sketch the way we have introduced the product line, and list some of the success factors.

The Koala component model was developed by research on the specific request of our business group TV to find a solution for handling diversity. Darwin [52] served as leading example, and to a lesser extent Microsoft COM [57]. It was very important to create a solution that caused no overhead; otherwise it would not have been accepted (in fact, use of Koala tends to result in *less* code as compared to traditional approaches). Also important was to limit the complexity of Koala and associated tools. The compiler consists of 13000 lines of C++ code, and the use of Koala is taught in one day.

The product line architecture was also set-up in Research, again on the specific request of our business group TV. Only few people were involved in the beginning (5-10); most software developers were still maintaining and evolving the previous generation of software. The architecture team consisted of three people, a domain expert, a software technology expert, and a person with a strong feeling for the business aspects. Two sponsors supervised the work, the TV software development manager, and the overall software technology officer.

After showing initial feasibility, the team was rapidly expanded and moved from research to the business groups. The researchers also temporarily joined the business groups. In a relatively early phase, a lead product was selected. We chose one with high visibility but low risk if things went wrong. The development was multi-site from the beginning, which we handled by carefully aligning our architecture with the organization. The four places where there still was a mismatch between architecture and organization all turned out to cause extra complications.

The first – lead – product actually did not make it to the market for commercial reasons, but the second product did. We now have two business groups that have products with software based on our approach, and two that will follow soon.

7.6 Experiences and Related work

In this section we report on our experiences with the approach described in the previous sections, and we relate our work to other work found in literature.

7.6.1 The Overall Approach

We have created a software product line by starting from business requirements, defining the architecture, and subsequently tuning the development process and organization towards this (BAPO). Jacobson et al. inspired us in doing so [41]. BAPO is part of our method on component oriented platform architecting (COPA); the work described in this paper serves as one of the two cases in our COPA tutorial [2]. Our (necessary) involvement with process and organization caused many a battle within our organization, as traditionally these are the realms of different sets of people entirely. We believe that we succeeded in reaching our business goal with our architecture and component technology, but we still have a tough job in preventing our development process and organization from falling back to classical product development.

7.6.2 Composability and Variability

Two important issues arise in product lines: *composability* and *variability*. The former concerns how well two pieces of software fit if they have not been developed together; the latter concerns the degree in which one piece of software can be adapted to fit into an application. To give an example, the architecture described in [112] scores high on variability through the use of component plug-ins, but low on composability. Jan Bosch [12] addresses these problems; see also chapter 6 of [22].

7.6.3 Koala

Koala was directly derived from Darwin [52], as a result of the ARES project [3]. For us, Darwin's main feature was *requires* interfaces that can be bound by third

parties, as it allows us to write components with no implicit assumptions about their context. GenVoca [8] also promotes decoupling of components but through a parameter mechanism. Microsoft COM [57] supports *connectable interfaces*, though in practice these are used in specific cases only, such as for notification. CORBA 3 [99] supports *receptacles* for the late binding of used components.

Actually, the CORBA 3 component model closely mirrors Koala: *facets* for provides interfaces, *receptacles* for requires interfaces, and *attributes* for diversity interfaces. CORBA3 also defines *event sinks* and *sources*, for which we use 'normal' interfaces in Koala, be it that we model a *provided* event as a *requires* interface.

Of course, Koala is not the only architectural description language (see [55] for a recent overview). We believe that Koala is the ADL that is most suited for product families or populations. Also, to the best of our knowledge, Koala is the only architectural description language that is widely applied in an industrial context.

Koala's lightweight binding mechanism, designed for resource-constrained systems, has two interesting side effects. First of all, it allows us to route *all* context dependencies through requires interfaces bindable by a third party, thus enhancing the context independence of components. Secondly, it allows us to apply component technology at large and at small scale, thus obtaining both reuse in the large and reuse in the small.

We anticipated in 1996 that non-proprietary technologies such as Microsoft's COM [57] would be applicable in TVs in five years time, and created Koala to bridge the gap. Now, in 2001, we foresee a prolonged use of Koala for a number of reasons. Most importantly, as long as we're building closed systems, Koala perfectly suits our needs, without introducing *any* overhead whatsoever, allowing us to apply component technology at a small grain size. Secondly, Koala has elements not readily found elsewhere, such as requires interfaces and an explicit description of architecture. Finally, we *will* use off-the-shelf component technologies in our high-end products in the near future, but the use of Koala will then migrate to mid range and low-end products.

7.6.4 Organizing Reuse

We have shown in Figure 51 how subsystem teams deliver their packages to products: there is no integration team in between that first integrates all subsystems before products get the software. There are actually different strategies for organizing reuse, as we illustrate in Figure 54. We shall also compare this with Figure 53.

The top left strategy is the traditional one: different products are created without coordination and with at most ad-hoc reuse. Organization types I and II in Figure 53 can be used for this.

The bottom left strategy is proposed in [105]: there are only product teams creating products, but there is a central architecture team that ensures that components are reused and made reusable. Here too, organization types I and II are applicable. This is a very promising approach, although we have doubts on how well it scales to large organizations.

The top right strategy is used in many product line approaches, for instance in [112]. The software consists of separate subsystems, but these are integrated by a central platform team before being made available to product teams. Organization type IV must be used here. The advantage is better control of quality; the disadvantage is the extra step in the process: it will now take long for a new feature to ripple through the process. Also, the approach does not scale well to very large organizations.

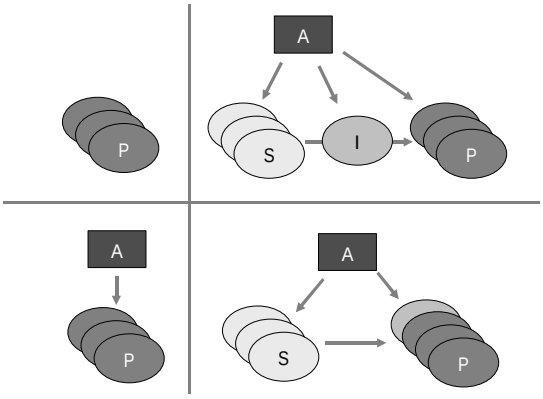

Figure 54. Strategies for organizing reuse

Our strategy is the bottom right one, and can be used both with organization types III and IV. We believe that this is the only strategy that scales well to large organizations.

Jan Bosch recognizes four types of organization [11]. The (single) development department is not applicable to our case, as Philips is traditionally already organized in terms of business units, lines and groups. This is in fact the second type recognized in [11]; it corresponds with type I in Figure 53. Our types II-IV can be mapped to the multiple domain-engineering units in [11], but we make a further distinction with respect to the location of the people and the product specificity of the software.

7.7 Concluding Remarks

Products in the consumer electronics domain show an increasing integration of functions. While different types of products can be treated as individual product families, their integration requires reuse of software *between* the families. We have argued that this asks for a compositional approach. Key element here is *context*

independence, or more precisely the ability to use a component in contexts other than the one in which it was originally developed. Our component model stimulates context independence through the explicit modeling of requires and diversity interfaces. The model is simple to learn, and provides our developers with an easy to use terminology to handle diversity and evolution.

When introducing this technology, it appeared that changes had to be made to the software development process, i.e. the way the software is created, documented and managed. We have described some of the key differences in our paper, and have also shown how the organization must be made to match this process.

The component model was created in 1996 and 1997, the product line architecture set up in '98 and 1999, and in 2000 and 2001 we have set a small number of products on the market with this technology. In the coming two years, this number will increase strongly. Between 100 and 200 software developers are currently involved, distributed over 10 sites.

The work described in this paper is the result of the prolonged attention of Hans Aerts, Hans van Antwerpen, Erik Kaashoek, Henk Obbink, Gerard van Loon, the author, and many others. I would like to thank Chritiene Aarts and Hugh Maaskant for reviewing this paper.

Chapter 8

Horizontal Communication

Published as: *Horizontal Communication: a Style to Compose Control Software*, Rob van Ommering, Software Practice and Experience, 2003.

Abstract: Consumer products become more complex and diverse, integrating functions that were previously available only in separate products. We believe that to build such products efficiently, a compositional approach is required. While this is quite feasible in hardware, we would like to achieve the same in software, especially in the low-level software that drives the hardware. We found this to be possible, but only if we let software components communicate horizontally, exchanging information along software channels that mirror the hardware signal topology. In this paper a concrete protocol implementing this style of control is described, and many examples are given of its use.

8.1 Introduction

Philips produces a large variety of consumer electronics, such as televisions, video recorders, CD and DVD players and recorders, audio amplifiers, and many more. All of these products contain embedded software to control hardware devices, to process signals and information (such as Teletext or Closed Captioning), and to provide a graphical user interface. Consumer products exhibit characteristics also found in many other fields:

- The *size* and *complexity* of the embedded software is growing;
- The required *diversity* of products is growing;
- The *time to market* must become shorter.

But they also have some typical properties of their own:

- Consumer products are *resource constrained*, to keep prices competitive;
- There is an *increasing integration* of functions that were previously unrelated.

One example of the latter is a TV with a built-in VCR, DVD or hard disk as storage medium. As technology improves, we see that traditionally disjoint domains such as audio and video acquisition, reception, storage and reproduction are merging – this is sometimes called *convergence*. It gives rise to a whole new range of products, especially when combined with interoperability and mobility (telecommunications).

Our problem may be evident: we need to create *more* product software in a *shorter* time without diminishing the *quality* of the software, and preferably without increasing the required number of developers. We should profit from the fact that we already have a lot of software available, but we need the ability to reuse such software easily and systematically.

A proven way to reuse is to install a *product line*: a pro-active and systematic approach for developing a set of related products [21]. A common approach is to define an overall variant-free architecture [88], and to rely on predefined variation points [41] to implement diversity. While this works well for a *product family*, a set of products with many commonalities and few differences, we have found that for a *product population*, a set of products with many commonalities but also many differences, variation is not necessarily the best way to handle diversity [80]. Especially when development crosses organizational boundaries, we feel that *composition* must become the dominant development paradigm.

Composition as development paradigm is not new [54]. In [81], we have made an inventory of the use of variation and composition. Although the archetype example of LEGO has often been used as the case *against* composition, there are many positive examples of using software composition to build a wide range of applications. Well-known are the pipe-filter paradigm as for instance used in Unix command shells [93] and in streaming software [106], and the use of ActiveX controls and Visual Basic to rapidly create PC applications [60]. As certain categories of software apparently lend themselves naturally to composition, what about using composition to write control software?

In this paper a television is used as an example. Figure 55 shows a typical architecture for the software in a television. An infrastructure abstracts from the computing hardware, providing an operating system API, while TV device drivers and a so-called TV platform abstract from the domain hardware providing a TV API. On top of both APIs, services and applications are built. Our focus is on the TV platform, an integration layer that coordinates all TV devices and that provides an API that is simple to use (‘tune main picture to channel X’).

Designing the TV platform turns out to be one of the most difficult tasks. It is not that the individual problems are intrinsically complex, but the number of problems that have to be solved is very large, and the order in which their solutions have to be executed at run-time is often critical. This makes development very complicated even for a single product, let alone for an entire product population.

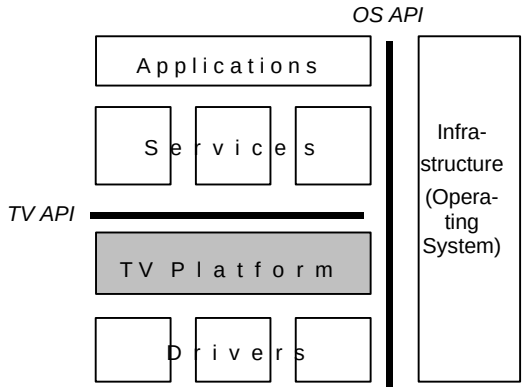

Figure 55. The software architecture in a television

The signal processing hardware in a TV lends itself for easy composition, a phenomenon that is eagerly deployed by our hardware engineers to create a large variety of products. The same must be achieved in software to prevent software development from getting on the critical path. To further complicate matters, our solution must be resource efficient. The 'Bill of Material' is an important issue in consumer products, and as a result the computing resources in a TV are typically 10 years behind on what is found in a PC.

In this paper an architectural style is proposed to design control software such that it can easily be composed. This style is denoted as *horizontal communication*. Essentially, communication channels are introduced in software that mirror the signal flow in hardware and that are used to exchange information about these signals. A device-independent protocol is defined and third party binding of software components is used to create a variety of products. This protocol is described in this paper, together with a particular (efficient) implementation technique that suits our resource constraints. Many samples are shown of its use.

This rest of this paper is organized as follows. Section 8.2 describes the component technology and architecture that we use for building consumer electronics software. Section 8.3 explains typical software control tasks in a TV, and suggests a particular style for composing control software. Section 8.4 discusses the protocol that we currently deploy in analogue televisions. Section 8.5 shows the steps we needed to take to introduce this protocol into our organization. Section 8.6 lists our experiences with the protocol, while section 8.7 discusses related work and section 8.8 provides concluding remarks.

8.2 Component Technology and Architecture

First the component technology in which our solution has been implemented is summarized and the architecture in which our solution fits is described.

8.2.1 Koala

Koala is a software component model with an architectural description language (ADL) [72] aimed at the creation of a product population of resource-constrained products [80]. Koala's ADL has a graphical syntax of which examples are shown in Figure 56 and a textual syntax that is not used in this paper. Koala has its roots in Darwin [52] and in Microsoft COM [57].

Figure 56 shows a basic and a compound Koala component. The basic component provides four interfaces and it requires three. An interface is a small set of semantically related functions, graphically represented by a square with an embedded triangle; the tip of the triangle shows the direction of function call. Each interface is typed; a separate interface definition describes the syntax and semantics of the functions. A component may provide and/or require multiple interfaces of the same type. Code modules m_1 and m_2 in Figure 56 implement functions of provides interfaces by using functions of requires interfaces. Each function can be implemented either in C or in an expression language of Koala (for which C code is generated then). A provides interface can also be connected to a requires interface directly.

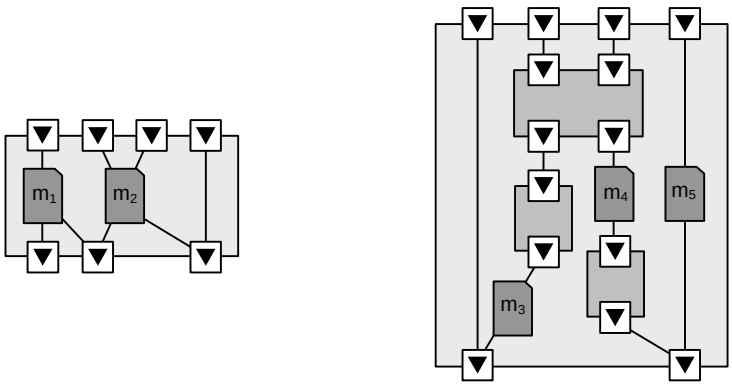

Figure 56. A Koala basic and compound component

A compound Koala component instantiates other components and binds their interfaces. Binding can be done by a straight connection or by inserting glue modules (such as m_3 and m_4). In the first case, the interface types must match; more specifically, the interface connected with the base (of the triangle) must contain all of the functions of the interface connected with the tip, and possible more (a form of subtyping). In case of a glue module, functions in interfaces connected with the tip must be implemented in C or in Koala expressions, using functions in interfaces connected with the base. A compound component may also implement functionality directly (in module m_5) or defer the implementation to a requires interfaces.

A Koala *configuration* is a component without interfaces on its border. Such a component can be instantiated, compiled, linked and subsequently run. The instantiation is performed by a Koala compiler, which recursively instantiates and binds subcomponents until a full decomposition tree is obtained. It then optimizes this tree by shortcutting chains of interfaces, by partially evaluating functions implemented in the Koala expression language, and by subsequently removing unreachable components. The result is a skeleton of bindings that connect functions implemented in C, and Koala generates #defines and auxiliary functions to make those connections.

The Koala *repository* contains all interface and component definitions. These include multiple configurations so the repository in fact spans a composition graph. This is the main value of Koala: as an ADL it provides full encapsulation of component instances, and as a component model it provides full reuse of component definitions. Making requires interfaces explicit and relying on third party binding (by the compound component) ensures that components are sufficiently context independent to be indeed usable in multiple products. Koala's partial evaluation mechanism removes most of the fat inherent for reusable components, thus satisfying the resource constraints in our application domain.

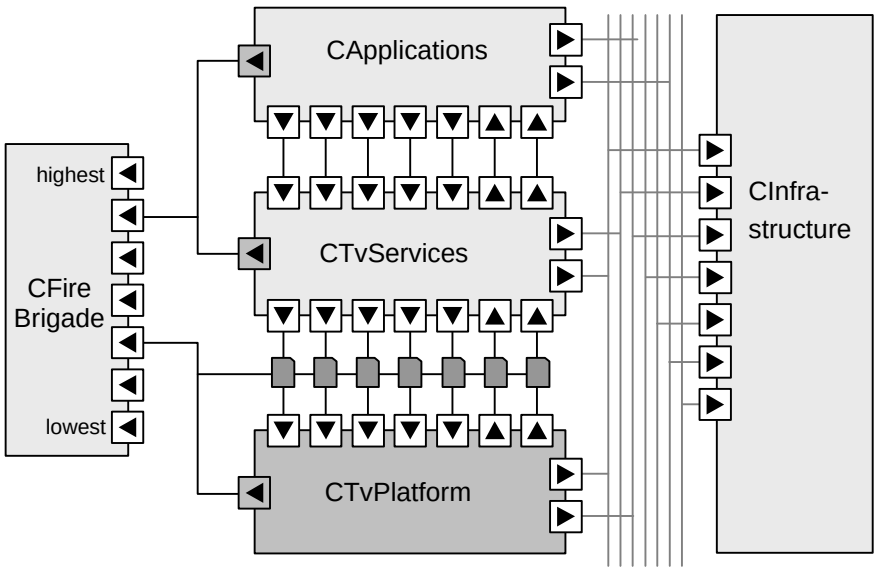

Figure 57. TV architecture in Koala

8.2.2 Modular Architecture

Figure 57 shows the software architecture for a television as introduced in Figure 55 but now in Koala notation. The infrastructure and application layer have become Koala components. The multiple service components have been integrated into a single compound component, as have the TV platform and its drivers.

Putting drivers *inside* the TV platform component is possible because the drivers are completely shielded by the TV platform. The fire brigade component and the modules between the services and the platform are explained in the next section. Figure 57 is taken from real-life but is (of course) severely simplified.

Note that some interfaces between platform, services and applications are directed upward. These are *notification* interfaces for reporting (asynchronous) events. Koala binding is used as the subscription mechanism for notifications, providing us with a very cheap implementation, and allowing us to visualize notifications in our diagrams.

8.2.3 Execution Architecture

The software in a TV has to perform many activities, operating on different time scales. The highest frequency activities, with response times below 1 ms, are handled in the interrupt domain, in close cooperation with the hardware. Activities with response times between 1 and 100 ms are handled in high priority real-time kernel threads. Other activities are handled in medium or low priority threads.

The computing hardware in a TV does not allow us to create many threads (10 is a typical maximum). We do not want to have many threads anyway because of the potential synchronization problems that may arise. We therefore perform most of our activities on pumps and pump engines. A *pump* is a message queue with a virtual thread that processes the messages by calling a function associated with the pump with the message as parameter. Anyone can send messages to the pump, including the pump function itself. Messages can be delivered immediately or after a time interval. The former is often used to decouple threads, while the latter enables the creation of timed loops.

Multiple pumps can share a single physical thread. The thread is encapsulated in a *pump engine* on which pumps can be created. The message queues of the different pumps are actually integrated into one large message queue attached to the pump engine. The pump engine drives the pumps connected to it by processing the messages in sequential order (first arrived, first served). This makes pumps running on the same pump engine synchronous with respect to each other, a feature that allows us to omit synchronization operations between such pumps. Of course, pumps running on different pump engines are fully asynchronous.

We use the Koala binding mechanism for the allocation of pumps to pump engines. Components may require one or more ‘virtual pump engine’ interfaces (gray in Figure 57). Through each such interface, the component gets access to the handle of a single pump engine on which it can create pumps. A standard set of seven pump engines is provided in the component *CFireBrigade*, with priorities ranging from lowest to highest. The system builder may use it to allocate pump engines to the various components.

In Figure 57, the designer of the TV platform has decided that all pumps in the platform run on a single pump engine. This is realistic since the platform contains only activities with roughly the same deadline and in general a very short execution time. As a positive side effect, using only a single pump engine relieves the designer from having to synchronize between activities within the TV platform. The TV services and the application also require one pump engine each.

The system builder decides to let *CTvServices* and *CApplications* share the same pump engine, while *CTvPlatform* deploys another one. As a consequence, different threads may enter the TV platform (which has no synchronization built-in); therefore an external synchronization has to be added in the form of a set of glue modules that lock the pump engine of the TV platform before entering. This 'monitor' concept ensures that at any point in time only one thread is active in the TV platform, and that simplifies the protocol to be discussed.

8.3 The Control Problem

In this section some typical control tasks of a TV platform are listed, and then traditional solutions and our proposed solution are given.

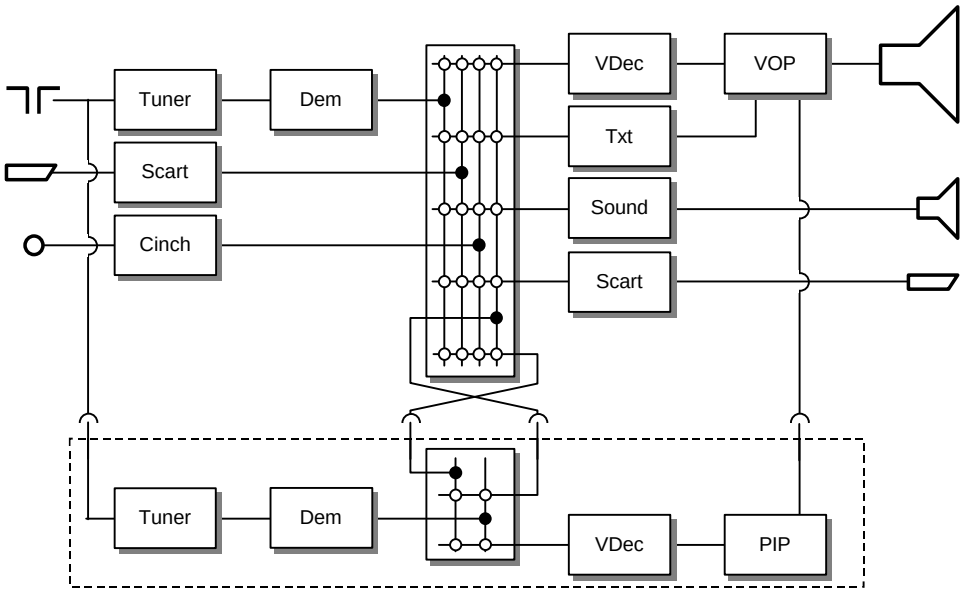

Figure 58. A simplified television

8.3.1 A Simplified TV

Figure 58 contains a schematic overview of a television. An antenna signal is processed by a tuner and a demodulator, and then fed into a switch matrix. Two other inputs of the switch matrix are connected to a SCART and a cinch input. Four

outputs of the switch matrix are connected to respectively a video decoder, a Teletext or Closed Captioning decoder, a sound decoder and amplifier, and a SCART output. The output of the video decoder is fed to the video output processor (VOP), which can also superimpose an image generated by the Teletext display hardware.

The optional double window module (the dashed rectangle in Figure 58) is connected via four sockets. It contains a second tuner and demodulator, a second switch matrix and video decoder, and a Picture In Picture (PIP) module for downscaling the image. The PIP output is fed to the third input of the VOP to be superimposed on the other two images. The two switch matrixes are mutually connected to be able to use a SCART or Cinch input as source for PIP, and/or the second tuner as source for Teletext.

8.3.2 Typical Control Tasks

We shall now use the TV as shown in Figure 58 to describe some typical design challenges when building a TV platform. The following paragraphs describe product requirements that have to be satisfied. In section 8.4, these requirements are dealt with one by one.

Our first task concerns *tuning*. When the frequency of a tuner is changed, the tuner temporarily produces noise. This leads to undesired artifacts on screen and in the speakers. Therefore, the screen should be blanked and the sound muted before the frequency is changed, and they should be restored only after the tuner's output is stable again. Activities such as Teletext decoding should also be stopped during the tuning activity.

Some downstream devices actually need some time to complete their operation before the tuner output may be dropped. A PIP module for instance samples the image and stores it in an internal memory, which is then used to generate the output. It is a product requirement that the PIP output should be frozen instead of blanked when changing the frequency of the PIP tuner. Freezing without synchronization may result in a PIP that contains half of an old scene, half of a new scene. As this is visible for a prolonged time, the PIP must be allowed to complete sampling the image before the tuner output is dropped.

The presence of a switch matrix adds complexity. The blanking, muting, freezing and stopping must only occur for down-stream devices that are actually connected to the tuner being tuned. For devices connected to the other tuner, or to a SCART or cinch input, no action is required.

The switch matrix participates in yet another way. When the matrix is used to select another source, the switches in the matrix temporarily drop their output signal, resulting again in undesired artifacts. So, switches too should request for permission to drop the signal first.

The product sketched in Figure 58 in its extended form contains two switch matrixes that should be set to the right position in the right temporal order to select the proper sources for the outputs of the TV. Controlling these two switch matrixes may still be relatively easy, but a real-life TV contains dozens of switches at different locations in the topology, making the source selection process a difficult task in itself. And remember that whenever a switch is changed, downstream devices should be given the opportunity to halt gracefully.

A typical characteristic of analogue TVs is that measurements of different properties of the signal take place at different locations in the signal path. The aspect ratio of the broadcast, for example, is transmitted through the Teletext signal. Typical values are 4:3 and 16:9. In the latter case, the image contains black bars at the bottom and top, which can be made invisible by the VOP by vertically scaling the image.

The basic detection that there *is* a video signal occurs at different locations in the signal path. The demodulator performs a fast but unreliable detection, while the video decoder performs a slow but reliable detection. If both are present in the signal path, then use of the second one is obligatory. If only the first one is present in the signal path, for instance when the tuner signal is fed to the SCART, then use of that one is obligatory. Certain countries forbid the reproduction of sound when there is no video signal, as this would make the TV useable as receiver for military radio.

As the last product requirement named here, some TVs only enable screen and speakers if there is both sound and video. Again, this requires coordination of different detectors at different locations in the signal path.

8.3.3 The Problem Statement

We are now in a position to formulate our problem statement.

- Many control tasks in a TV coordinate different devices in the same signal path, and are therefore strongly dependent upon the topology of the signal flow in hardware.
- This topology is fundamentally different for different products; see for instance the addition of the plug-in module in Figure 58.
- Even for a single product, the overall topology is apt to change during development up to the very last moment.
- Finally, signal paths change even at run-time in complicated ways, given the many switches present in a real-life TV.

Therefore an architecture or architectural style is sought that allows us to easily cope with changes in the topology. In the next section, two classical solutions that

do not have this property are investigated. In section 8.3.5 a solution is proposed that does satisfy our requirement.

8.3.4 Traditional Solutions

Figure 59 shows two solutions that we have used in the past to create control software for a TV, and that we believe most software developers will propose as a first intuitive solution to the control problem. Of course, the architectures shown in Figure 59 are greatly oversimplified.

The first solution is a hierarchical control system. The left hand side of Figure 59 shows a four-layer approach, with at the bottom device drivers and at the top the overall control. The middle two layers group successively larger sets of devices together. The grouping is usually based on adjacency in the topology, or on providing the same kind of functionality. The control hierarchy is a pure tree; in every layer a certain hardware device is handled by one component only.

The hierarchical control system has two disadvantages. The most important disadvantage is that low-level control problems that pass group boundaries must be solved at high levels in the design. A second disadvantage is that simple changes in the topology often influence multiple components in the middle layers. Consider for instance a spare switch in the matrix that is going to be used in the audio/video featuring of a new product. This requires changes in both the source switching and the A/V featuring components.

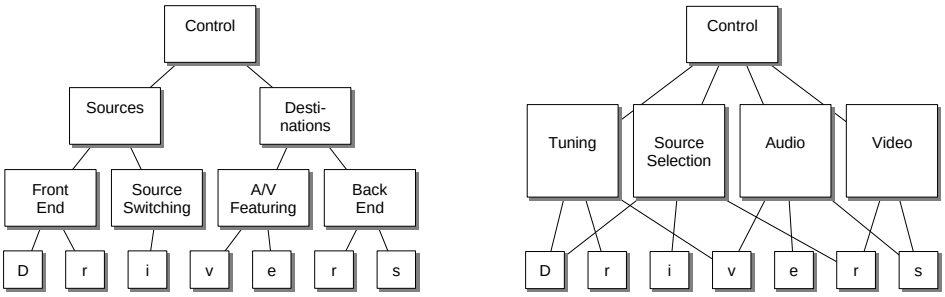

Figure 59. Traditional solutions to the control problem

A second solution is to create an aspect-oriented control system, as shown on the right hand side of Figure 59. Here, the middle layer consists of components that are responsible for a single aspect in the TV, for instance tuning, source selection, audio or video. The control hierarchy now forms a graph. An advantage of this approach is that software engineers can specialize in one aspect. Also, low-level control problems can be handled at low levels. However, as in the previous approach, this approach is very sensitive to changes in the topology.

What we really need is a solution where we can create *large* software components that are either dependent upon a specific piece of hardware (and can therefore be

reused when the hardware is reused) or not dependent on hardware at all. The actual dependency on the topology of the hardware should be isolated in a small part of the architecture.

8.3.5 The Proposed Solution

Figure 60 shows our solution: a set of large and relatively independent software components (gray) that control individual hardware devices (black), and that are coordinated by a small software control layer (white) as the only component with knowledge of the hardware topology. We found such an approach to be feasible only if we allow the gray components to communicate with each other. Of course, components should not have specific knowledge about their neighbors, so they should communicate in terms of a standard protocol. Since most control tasks relate devices on a particular signal path, we mimic the signal paths in software, and let components communicate through those.

The right hand side of Figure 60 translates this solution to Koala. The gray components have traditional vertical control interfaces (for instance to change the frequency of the tuner), but they also have horizontal communication interfaces, one for each signal output and one for each signal input. The top-level control component becomes a Koala compound component that instantiates gray components and binds their horizontal communication interfaces in correspondence with the topology of the hardware. Changes to the topology are therefore isolated in this component: for different products we can have different compound components that instantiate and bind the same basic components in different ways. Vertical control interfaces of the basic components are generally bound to interfaces at the border of the compound component.

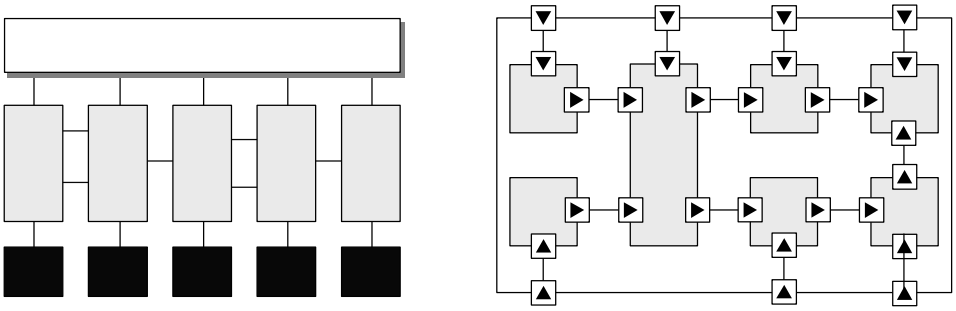

Figure 60. Proposed solution: horizontal communication

We see our proposal as an architectural style: it resembles for instance the pipe-filter style but it communicates control rather than data. In the next section one specific example of this style is introduced, the horizontal communication protocol that we use in the domain of analogue televisions. Furthermore, one particular implementation of this protocol is discussed, using direct side-calls instead of messages.

8.4 Horizontal Communication

Our horizontal communication protocol is now introduced step by step. We start with a simple tuner and output device to solve the signal drop and restore problem, and then we subsequently add features to the protocol until we have solved all of the problems listed in section 8.3.2.

8.4.1 Synchronous Drop Request and Restore

In Figure 61, component *A* is the tuner driver, *B* the 'intelligent' component controlling the tuner, *C* the 'intelligent' component controlling the video output, and *D* the video output driver. Components *B* and *C* are connected through a unidirectional horizontal communication protocol implemented as Koala interfaces.

Suppose that the top-level control software (not shown in Figure 61) wants to change the frequency of the tuner. It therefore calls the function *Tune(f)* (1) in the control interface of *B*. Through its horizontal interface, *B* requests *C* for permission to drop the signal (2). While in this call, *C* calls its driver to blank the output (3) and then returns *true* to indicate its approval. When the drop request returns, *B* is ready to change the frequency in the tuner driver *A* (4). Assuming that the result is immediate, *B* then informs *C* that the signal has been restored (5). *C* responds by unblanking the output again in *D* (6) before returning.

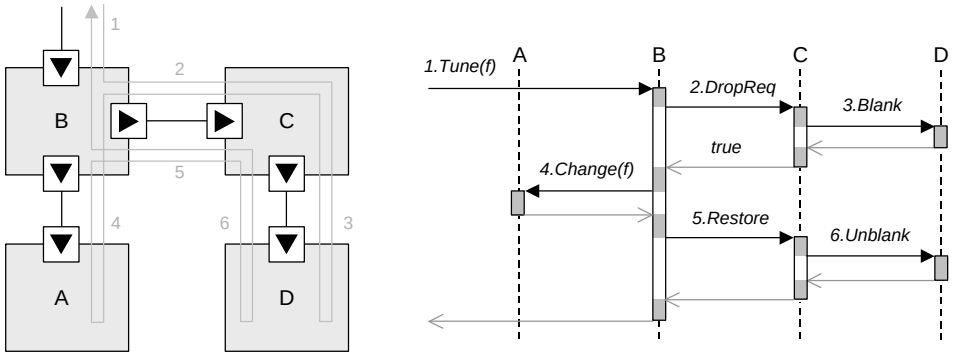

Figure 61. The synchronous drop/restore protocol

Note that the whole activity is performed on a single call. It is thus an atomic action, since only one thread is allowed to be active in the TV platform at any point in time (as explained in section 8.2.3). Although *B* communicates with *C* during this call, it has no knowledge of the existence of *C*. All that it knows is the abstract communication protocol, in terms of drop requests and restores. The fact that *C* is connected to *B* is knowledge confined to the higher-level compound component implementing the platform.

Note also that the protocol is unidirectional and fully synchronous. In the next section, the protocol is extended to include asynchronous acknowledgements and

restores, forcing us to make the protocol bi-directional. In all these cases, asynchronous means ‘executed later in time’ rather than ‘executed in parallel’.

8.4.2 Asynchronous Drop Request and Restore

There are two cases in which the drop/restore protocol may be asynchronous. The first is when *C* is not immediately ready to approve of the drop request, for example in case of the PIP component as described in section 8.3.2. To indicate this, *C* may respond with *false* to the drop request, but then it has the responsibility of calling an acknowledge function in *B* somewhere later in time. The second case occurs when *B* is not immediately ready after changing the frequency in the tuner driver. *B* may then postpone its call to the restore function of *C* to a later point in time.

Figure 62 shows the situation in which both driver *D* needs time to blank and driver *A* needs time to recover from the change. The protocol is then split into three atomic actions. In the first step, the call to *Tune(f)* (1) causes *B* to request *C* whether it may drop its signal (2). After starting the blank operation in *D* (3), *C* returns *false* to indicate that the request is not yet acknowledged. This completes the call to *Tune*, leaving the TV platform in an intermediate state. Note that *B* must remember the new frequency value *f*, for later use in the *Change* command.

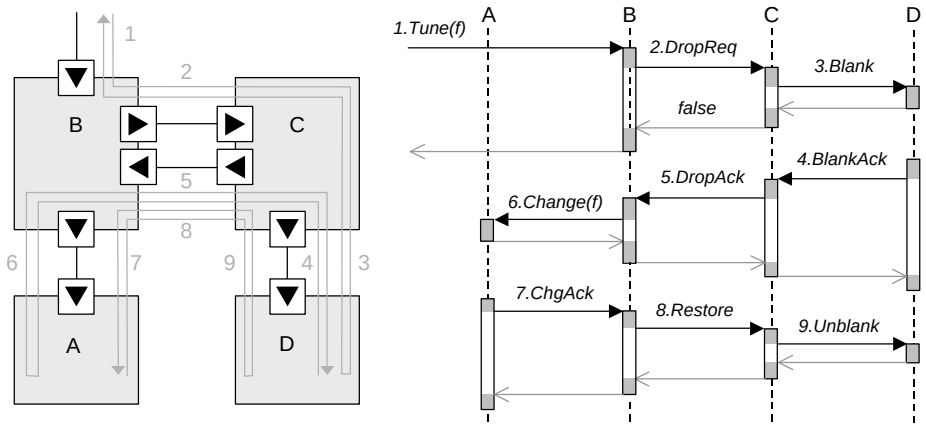

Figure 62. Asynchronous drop request and asynchronous restore

After a while, *D* performs an up call (4) to indicate that the blanking has succeeded. This call is performed on a pump, which may be triggered by a timed message or by a message sent from within an interrupt. *C* uses this call to acknowledge the drop request to *B* (5), who in turn uses the call to change the frequency in *A* (6). Since changing the frequency now takes time, this completes the second step, leaving the TV platform in yet another intermediate state.

After again some time, *A* performs an up call (7) to report the completion of the change. *B* uses this call to inform *C* that the signal has been restored (8), and *C* uses that call to unblank *D* (9). This (finally) completes the protocol.

Some remarks should be made at this stage. We assumed that *A* and *D* are the cause for the delay, and that they perform an up-call when the action is completed. Another option is that *A* and *D* are just dumb drivers, while *B* and *C* contain the intelligence when to continue the operation. They may deploy timed messages and possibly driver polling to achieve his.

Note that *C* may postpone the signal drop but may not refuse it. *C* must either return *true* to indicate immediate approval, or *false* to indicate delayed approval. In the latter case, *C* has to call the drop acknowledge function up-stream somewhat later in time to further progress the protocol.

Leaving the platform in an intermediate state adds some complexity to the platform behavior: what happens for instance if a new call to *Tune* is made while the platform is in the first or second intermediate state? In the first state, *B* should just store the newly required value for the frequency locally, and use that value later in time as parameter to the *Change* function. In the second state, *B* may just call *Change* with the new frequency value directly, since the drop request has already been approved.

A traffic light paradigm may be used to clarify the protocol. In normal circumstances, the ‘wire’ between *B* and *C* is green, indicating a valid signal. A drop request call makes the wire orange. If the call returns *true*, the wire turns red; otherwise the wire remains orange. In the latter case, the drop acknowledge call colors the wire red. The restore operation turns the wire green again.

The tune operation is now easy to implement. If the wire is green, *B* should issue a drop request to *C*. If the wire is orange, *B* should store the new frequency value and return. If the wire is red, *B* should change the frequency of driver *A*. There are still some complications, but they are not discussed here.

As a final note, we assume that unblanking does not take time, so there is no ‘delayed’ unblanking. However, if *C* is not at the end of the signal path, it does have the responsibility to call the restore operation of its downstream neighbor, and it may postpone that call. This is not discussed further in this paper.

8.4.3 Asynchronous Drop Request / Synchronous Restore

Examples have been given of the drop/restore protocol where the drop and restore were both synchronous and both asynchronous. The situation where the drop is synchronous but the restore is asynchronous can simply be derived from these. The situation where the drop is asynchronous and the restore is synchronous deserves some extra attention. Figure 63 shows the protocol in this situation. The novel aspect here is that the restore operation is called *during* the drop acknowledge

operation. In other words, a function in component *C* (and in our example also *D*) is called again during an outcall from that same component! Although this requires some careful programming, we believe that this is not harmful. We shall see other examples of such 'reflective' calls in the sections following.

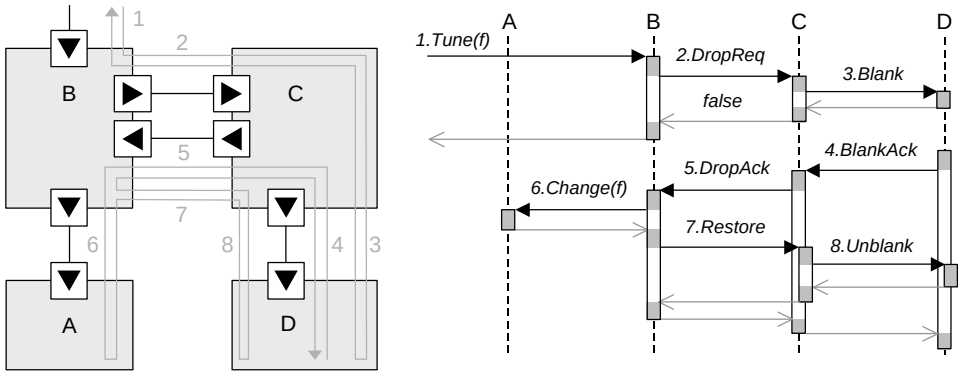

Figure 63. Asynchronous drop request and synchronous restore

8.4.4 The Fork

The previous sections studied the communication between a tuner and a video output component. What happens, however, in the case of *two* output components to a single tuner? In hardware this is (almost) trivial, but in software a fork component must be added in between. Figure 64 shows such a configuration, where the shorthand notation (• and °) is used for a pair of Koala interfaces as introduced before. As a further simplification, the driver components have been omitted; this will not change the essence of our explanation.

Starting again from the top-level control software calling the function *Tune(f)*, component *A* issues a drop request to fork *F*, which first forwards this request to output component *B*. Assuming that *B* answers positively (i.e. returns *true*), *F* subsequently forwards the drop request to the second output component *C*. If *C* also answers positively, then *F* can return *true* to *A*, which can in turn change the frequency in its driver (not shown in Figure 64). *A* then calls the restore command in *F*, which forwards this to *B* and *C* respectively.

The synchronous case only is given in Figure 64. If one of the output components returns *false*, delaying the approval of the request, then fork *F* must keep track of this and return *false* as well (after having called the drop request of both components). Component *A* cannot proceed with the tune operation then. Fork *F* must now wait for the component that has returned *false* to call a drop acknowledge in *F*. On receipt of that, *F* can call a drop acknowledge in *A*, which can in turn change the frequency and call the restore operation. Fork *F* forwards the restore to *B* and *C* just as sketched in Figure 64. Note that the restore can be called synchronously or asynchronously, but this adds no extra complexity here.

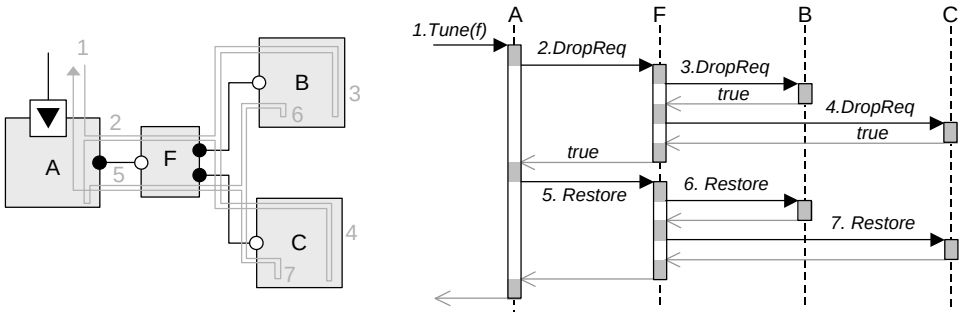

Figure 64. A fork with synchronous drop requests and restores

If both output components delay the approval of the drop request, then fork *F* must remember this, and keep count of the drop acknowledges that *B* and *C* send later. Only on the second acknowledge, *F* may forward this acknowledge to *A*. The protocol then proceeds as described above. This case is illustrated in Figure 65.

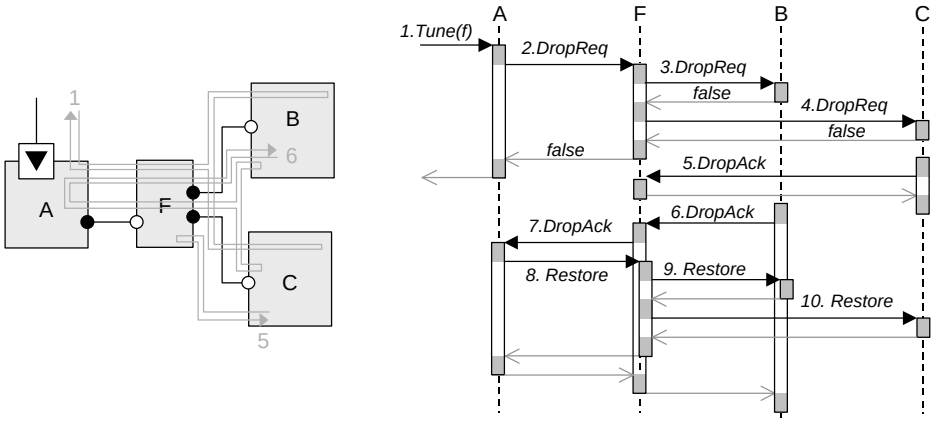

Figure 65. A fork with asynchronous drop requests and synchronous restores

Naturally, the protocol can easily be extended to forks with more than two outputs.

8.4.5 The Switch

While a fork connects <n> inputs to one output, a switch connects one of <n> outputs to one input. Figure 66 shows a binary switch *S* that connects either tuner *A* or tuner *B* to component *C*. The switch is currently in position *A*. The result of two actions are discussed. The first is a *Tune* operation on *A* (1). Since the switch is in position *A*, it passes the drop request of *A* to *C* and returns its answer to *A*. The restore command of *A* is handled similarly. The second action is a *Tune* operation on *B* (6). Since the switch does *not* connect *B* to component *C*, it can handle the drop request and the store by itself. In fact, the switch serves as an output stub here,

answering any drop request with *true*, and returning any restore command immediately.

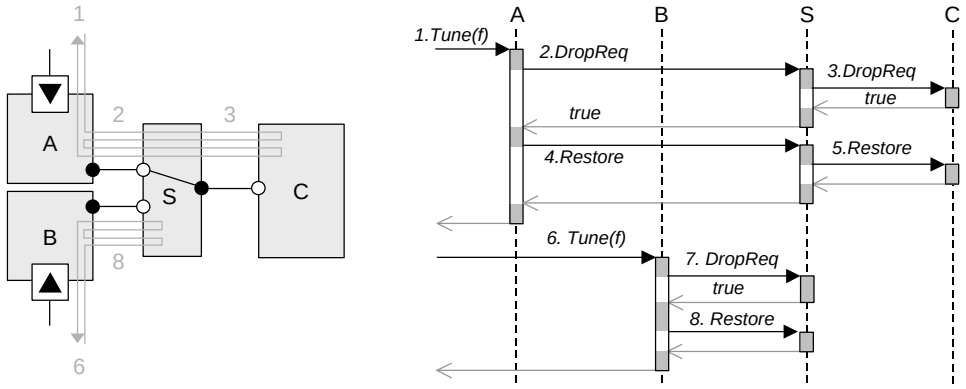

Figure 66. A switch forwarding the drop request protocol

The switch has an extra complexity: it can also change position. Before it does so, it must request permission from down-stream devices using the drop request protocol as defined above. Figure 67 shows how this proceeds. The top-level control software calls the *Switch(i)* command to select input *i*. The switch requests component *C* for permission to drop the signal. In this case, *C* accepts the request immediately. The switch can then change position, after which it sends a restore command to *C* to indicate that the signal is valid again (assuming that A's output is still valid).

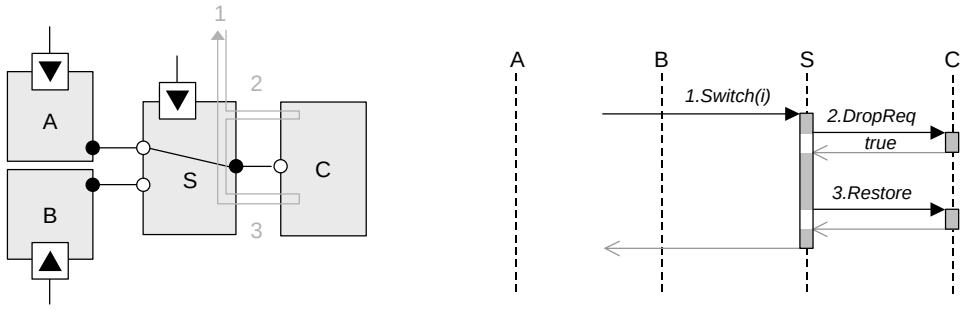

Figure 67. A switch issuing the drop request protocol

Some of the subtleties of the switch protocol are now discussed.

If component *C* delayed the approval of the drop request, then *S* may not change the position of the switch yet. Instead, *S* comes in an intermediate state, where it must remember the desired new position. When *C* issues a drop acknowledge somewhat later in time, *S* can change to the desired position, and hence return to its 'stable' state.

The intermediate state of the switch adds some complexity to the protocol. Firstly, if a second *Switch* command is called in the intermediate state, then *S* must just overwrite the desired position with the new information, so that the switch can go to the new state immediately when the drop acknowledge arrives.

Secondly, if a *Tune* command is called in *A* with the switch in its intermediate state, then *A* issues a drop request to the switch. But since the output wire of the switch is already in an 'orange' state (see the traffic light paradigm of section 8.4.2), forwarding *A*'s drop request to *C* is not necessary. But later on, the switch must forward the drop acknowledge of *C* to *A*, so that the tuner can continue tuning. The result of the tuning action will usually not reach *C*, since by that time, *A* will be have been disconnected.

Thirdly, if a *Tune* command is called in *B* with the switch in its intermediate state, and *B* has an asynchronous restore, then the switch may reach its new position before *B* is ready. In that case, the switch cannot send a restore to *C*, but instead must wait for the restore of *B* first.

The switch protocol has been fully implemented, but not all the details are included in this paper. Although this section considered only a binary switch, extension of the protocol to n-ary switches is straightforward.

8.4.6 The Matrix

Switches occur regularly in practice, but mostly in the form of a so-called switch matrix. A switch matrix has <m> inputs and <n> outputs, and allows every output to be connected independently to any of the inputs. We can model an <m> by <n> switch matrix simply by <m> <n>-forks followed by <n> <m>-switches, as is illustrated in Figure 68, where m=3 and n=2.

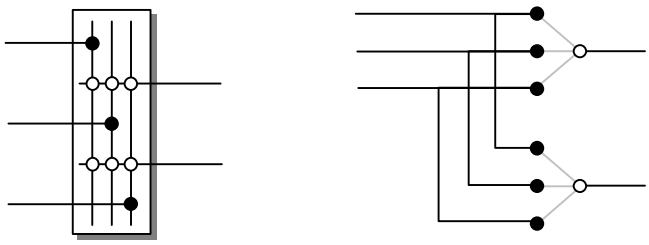

Figure 68. A matrix modeled as forks plus switches

When implementing the protocol, it turns out to be more convenient to implement the matrix directly, instead of as a combination of forks and switches. The code is only slightly more complicated than that of a fork or switch, and the result is much more efficient while still easily comprehensible.

8.4.7 Source Selection

The previous sections dealt with only one problem: that of properly blanking the picture and muting the sound when a tuner changes frequency or a switch changes position. In section 8.3.2 it was explained that in a real-life TV, setting the switches is a non-trivial issue due to the large number of switches and the complicated (and product-dependent) topology of the connections. This section considers the use of switches to select sources more carefully. For that, an example is needed that is more complex than the one in Figure 66, yet simple enough not to bury ourselves in details. Figure 69 shows such an example. The destination device *E* can be connected to each of the four source devices *A*-*D* by means of a cascading set of switches S_1-S_3. Even in this example it would be fairly trivial to implement source selection in the traditional ways as described in section 8.3.4, but our solution generalizes easily to more complicated situations.

The source selection protocol works as follows. Component *E* provides a connection interface through which an application can ask *E* to select a particular source, say *D* (the actual function is *Connect(D)*, marked C_D in Figure 69). Component *E* has no knowledge of the hardware topology other than that it has a single input, so it invokes a similar function C_D on its up-stream input interface. Component *E* is actually connected to switch S_3, which forwards the command to its first up-stream input interface, which in turn is connected to an output of switch S_1. Switch S_1 then also forwards the command to its first input, making the request eventually arrive at source component *A*.

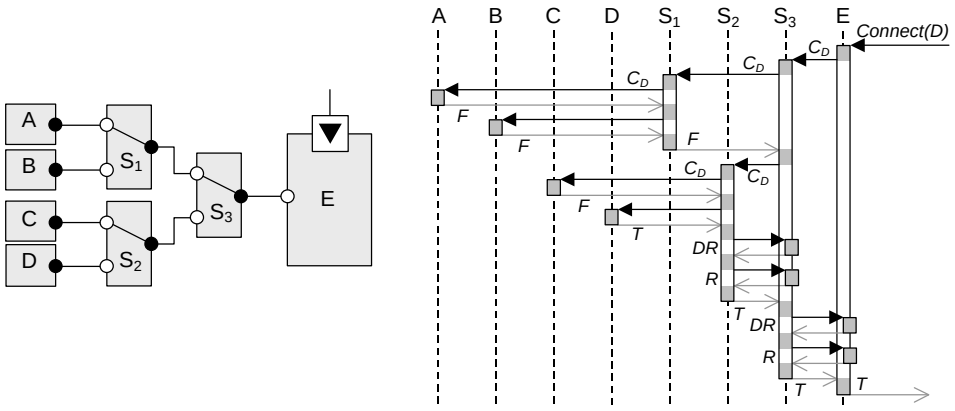

Figure 69. A more complicated switch topology

Since component *A* does not have the identity *D* (the value of the parameter), it returns *false* (marked *F* in Figure 69) to the connect request. Switch S_1 then tries it second input, which also fails, so S_1 has to return *false* itself. This causes S_3 to try its second input, connected to S_2. Switch S_2 then first tries component *C*, which fails, and secondly component *D*, which succeeds, returning *true* (marked *T*).

At this point in the protocol, switch S_2 will change its position to *D*. But before doing so, it must first issue a request down-stream whether it may drop the signal (as described in section 8.4.5). Since S_3 currently does not connect *E* to switch S_2 , S_3 can approve the drop request immediately. After S_2 has changed position it issues a restore command (provided of course that *D* has a valid signal), which is again handled by S_3. After the restore, S_2 can return true to the connect request.

Switch S_3 can then change position in a very similar way, also communicating with its down-stream neighbor in the way described above. At the end, the *Connect(D)* command returns *true* to the application, indicating a successful connection.

Again a number of remarks can be made. First of all, the source selection protocol is in fact a simple depth-first search algorithm, which may seem inefficient in a resource-constrained environment. To resolve this, the identities of the set of sources directly or indirectly connected to each input could be cached in every switch, so that each switch can directly route the request to the right input. Although this would improve speed, it also adds complication to the protocol, and in practice it has not been necessary to implement this solution.

Second, the reader may have noticed that the protocol sets the switches in a convenient order, namely up-stream to down-stream. Changing the down-stream switch S_3 first would cause E to be temporarily connected to C, which would result in undesired visible and audible artifacts.

Third, the reader may wonder whether source components should have an identity that is unique for the whole product population. This is actually not necessary, since the identities can be assigned at the platform level, i.e. at the level of the compound component in Figure 60.

Fourth, the *Connect* function can return *false* if an application requests a connection to a non-existing (or not implemented) source. If a connection *does* succeed, then as side effect, due to possible sharing of switches (not shown in Figure 69), other destinations may get connected to a different source. This would for instance be the case if the output of switch S_2 was also connected to a Teletext module. For that an extra priority parameter has been introduced in the *Connect* function, where a connection can be made only if the request priority is higher than the current priority of the signal path. This would allow an Electronic Program Guide to become available in the background (transmitted through Teletext) on a different channel if the user is not deploying the tuner. This priority mechanism is currently not deployed.

Fifth, what happens if there are cycles in the topology (such as in the extended version of Figure 58)? This could be solved by adding cycle-detection to the depth-first search algorithm. Since cycles do not occur very often, we decided to solve the problem first by deploying the specific order in which switches query their inputs, and secondly by inserting glue modules between our communication interfaces at the platform level to break any remaining cycles.

Finally, the connect interface was added to the destination component in Figure 69. In practice signal paths sometimes merge, for instance in a TV with PIP. In such cases, the inputs of the component that merges the signals should then be connected to the appropriate sources. One convenient way to handle this is to insert a so-called *destinator* at the appropriate locations in the topology. A destinator is a small component that is transparent to most of the horizontal communication protocol; it that it provides a connect interface with which the up-stream switches can be requested to connect the path in which the destinator is inserted to a particular source.

8.4.8 Properties

The previous sections have discussed the down-stream negotiation of one specific property of the signal: its availability. There are many more properties of the signal to be measured and used in control loops. To make life more difficult, properties that are measured down-stream are sometimes needed up-stream, or even in a different branch of the signal path. Our solution to this problem follows the style of the previous solutions: the information is communicated along the signal path mirrored in software. The added complexity here is that the initial direction of the communication is up-stream until the source of the signal path is reached, where the information is subsequently reflected to reach all the branches of the current signal path.

Figure 70 shows a sample configuration where component *D* measures a certain property of the signal that is subsequently needed by components *C* and *E*. Component D invokes the function *ChangeProperty* (*CP*) up-stream. Fork *F* routes this call to component *A*, which is the root of this signal path. During this call, component A sends an *OnPropertyChanged* command (*OPC*) down-stream. Fork *F* forwards this to *C* and then to *D*, while *D* forwards the information to component *E*. Thus it reaches all components that may possibly need the information.

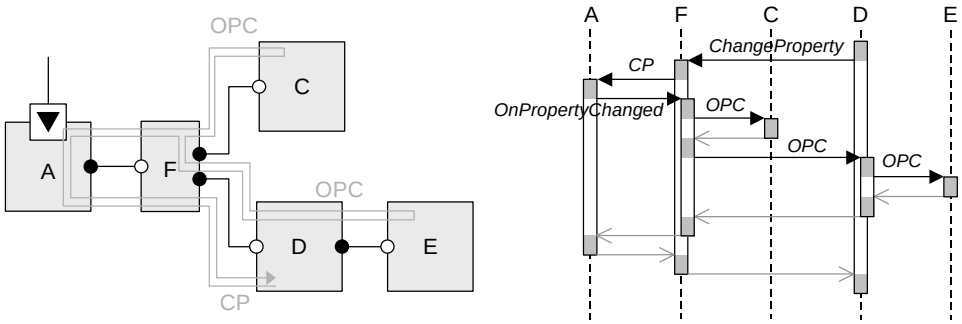

Figure 70. Communicating properties in a branched signal path

The *OnPropertyChanged* command has a parameter that indicates which property of the signal has changed. A second parameter is used to communicate the new

value of the property. It was anticipated that property handlers in components would sometimes need the values of other properties of the signal as well. As we do not want each individual component to cache such information, the second parameter actually contains a pointer to a structure with *all* properties of the signal. This structure is maintained by the source component.

Non-source components need not do any processing on the reception of an up-stream change property command on any of their outputs. They only need to forward this call to the appropriate input. In Koala, this can be done very efficiently, since Koala allows the binding of individual functions, which it then can optimize using #define directives. As a result, a change property call will reach the source directly if the signal is not routed through switches, or else it will need only one intermediate function call per switch.

A similar argument holds for the *OnPropertyChanged* command. If a component does not need access to *any* property of the signal, it can just forward the call using Koala function binding, which effectively removes the component from the signal path, at least for this function. However, as soon as the component needs *one* property, it must insert a function to check for this property. By splitting properties into groups (e.g. video, audio, data) and having a separate 'on property changed' command for each of these groups, better use can be made of the Koala function binding optimization. In practice, we have not found it necessary to do this.

Note that component *D* receives an *OnPropertyChanged* command during its outcall of *ChangeProperty*! Again, this requires some careful programming, but it would allow component *D* to process the change of the property that it measures in its own *OnPropertyChanged* handler.

8.4.9 Capabilities

Sometimes a certain signal property can be measured at different locations in the signal path, be it that one measurement is more reliable than another. In such cases, the most reliable measurement available should be used. This requires additional information to be communicated along the signal path: not only the *values* of the properties, but also the *capabilities* of the devices on the signal path should be known. An example is given in Figure 71, where both *A* and *C* can measure a certain property needed in *D*, but *C* does it more reliably than *A*. Now it depends on the setting of matrix *M* whether *C*'s information is available for *D*: it *is* if *C* is connected to *A*, otherwise it is not.

First, the two measurements are made distinct properties of the signal, say α and β. A simple solution would be to let a component that must decide between using α or β, query up-stream whether α and/or β are actually being measured currently. The query would be routed to the source of the signal path, where it is reflected to all branches until the information is obtained. As this would be quite inefficient, a form of caching was introduced here. Each source component maintains the set of

properties that its signal path is currently capable of measuring. This set is communicated down-stream with every *OnPropertyChanged* command (where the capabilities are typically needed) as a third parameter. Components can thus make use of this information whenever they handle a property change, and select the most appropriate property for their use.

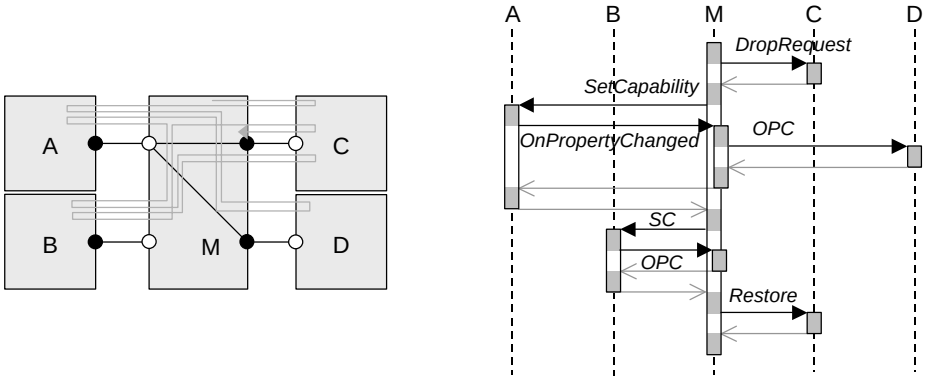

Figure 71. Changing capabilities on the signal path

The set of properties changes whenever a switch (matrix) changes position. Let us disconnect C from A and connect it to B in Figure 71. After a successful drop request to C and a change of position, the switch (matrix) calls a *SetCapability* command up-stream for each input that has been connected to or disconnected from outputs. A parameter of this function contains the new set of capabilities. To handle this efficiently, each switch (matrix) must cache the capabilities of each of its outputs. While the command travels up-stream, components can add new properties to the set, for instance in fork or matrix components, or in components that measure properties themselves.

When the command reaches the root of the signal path, the source component communicates the new set down-stream to all branches. Although typically no property has changed, the function *OnPropertyChanged* is still used for this occasion, since property change handlers may decide on different actions now that new capabilities have become available and/or old capabilities have been removed. The right hand side of Figure 71 shows an example trace of this protocol.

From Figure 71, the reader may derive that *D* is informed of the new capabilities, but *C* isn't. Component *D* is informed because *C* is disconnected, possibly reducing the capabilities of the signal path starting at *A*. Component *C* does not receive an *OnPropertyChanged* command for disconnecting from *A* because it has approved of the drop request earlier, and is thus not listening. However, when *C* gets connected to B somewhat later in time, it still doesn't get an *OnPropertyChange* command, since it is still not listening. But it needs access to the new capabilities at some point in time. Therefore, the new set of capabilities is added as parameter to

the *Restore* command that completes the transaction. The restore command also communicates the property values of the signal starting at *B* to *C*.

Many components take some time to perform measurements of properties, so when they are connected to a new signal, or when for instance a tuner changes frequency, there is a period of time in which the signal path has the capability of measuring the property, but the value of the property is not yet known. We therefore apply a three-valued logic for each property: (0) the property cannot be measured, (1) the property can be measured but hasn't been yet, and (2) the property has been measured. How components handle these three values is not discussed here.

As a final remark, the reader may understand that the capability caches, located at each source component and for each output of a fork or matrix, must be initialized. A *GetCapability* function that travels down-stream is used for this. Initialization is not discussed further in this paper.

8.5 Introducing the Protocol

Our horizontal communication protocol was designed on paper in only a few weeks. It was based on the novel idea that control software needs to be as composable as the underlying hardware. The paper study looked promising, but we needed to convince ourselves, and also our developers, that this approach would indeed work in practice. How we did so is the topic of the following subsections.

8.5.1 Simulating the Protocol in Visual Basic

We decided to build a prototype of the protocol in Microsoft Visual Basic [60]. We implement each software component, together with its underlying hardware device, as a VB *control*. We implement a product, a configuration of such components, as a VB *form*. Note how we map our reuse paradigm on that of VB, where controls are the reusable assets that can be combined in different ways to create different forms.

Each VB control in our simulation contains a simple *model* of the hardware device plus its software driver. A tuner for instance has only two state variables: the current frequency of the hardware and the frequency as desired by the software. In stable states, these two will have the same value; in transitory states, they may be different. Similarly, a switch is also modeled with two state variables, the actual position of the underlying hardware switch, and the position as desired by the software.

The VB controls and forms provide a *view* on these models. Figure 72 shows an example configuration consisting of a single tuner (named Tuner1) and a single output component (named Hop1), connected by a wire. The tuner control shows the values of its two state variables, while the Hop control simulates an image. This

image contains a number that represents which source is currently connected to the Hop (in the simple configuration of Figure 72, this cannot change).

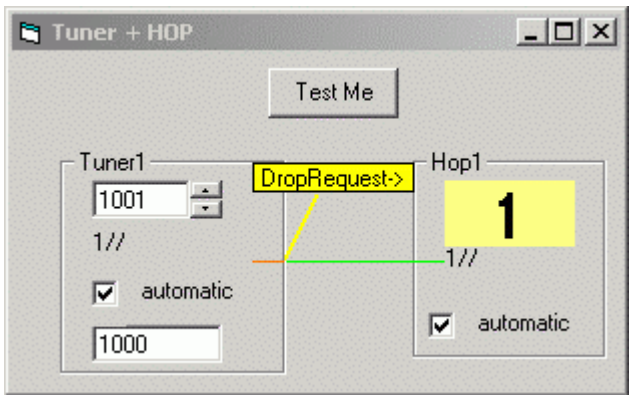

Figure 72. An example simulation

With buttons and check boxes we can simulate the actions that the application can perform. We can for instance change the desired frequency of the tuner (which has just been set to 1001 in Figure 72). We can also interactively change the behavior of the simulated components. The 'automatic' check boxes in Figure 72 can for instance be used to make the simulated devices handle drop requests and restores synchronously (checked) or asynchronously (unchecked).

To connect an output of control *A* to an input of control *B* in our simulation, we insert an intermediate object *S* of type *Signal* (see Figure 73). This object is in fact owned (and created) by *A*, and assigned to a variable representing the input in *B* by the configuration C (the form). Note that in our real implementation, we do not have these signal objects.

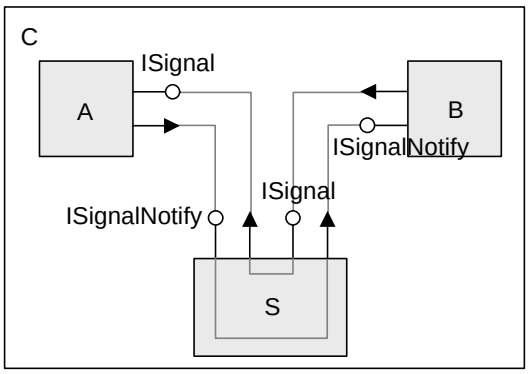

Figure 73. A signal object between two components

The signal object implements two interfaces, one of type *ISignal* and one of type *ISignalNotify*. The down-stream device uses the *ISignal* interface for up-stream communication. The signal object implements this interface by calling the

corresponding interface type in the up-stream device. Similarly, the up-stream device uses the *ISignalNotify* interface for down-stream communication. The signal object implements this interface by calling the corresponding interface in the down-stream device.

To make composition extremely simple, we use VB6's reflection capabilities. A control can query its container for its own position. If we create graphical representations for the inputs and outputs of a component, then the control can find the absolute coordinates of these. If we layout devices such that inputs are co-located with outputs, then we can let a control ask the form for the objects to be assigned to each of its inputs. The form passes the coordinates to each of the other controls, until it obtains the actual signal object involved. This allows us to let the controls wire themselves at initialization time. To make it even more convenient, we introduced a wire as an extra 'pass-through' device, as can be seen in Figure 72.

The next problem is how to visualize the different up- and down-stream calls. For that, we add another control to the form: a yellow flag as shown in Figure 72. Whenever a signal object processes a command, it sets the position of this flag to the location of the output that owns the signal object. The text of the flag is set to the command being processed. When the simulation is run, this results in a flag that moves across the form just as the flow of control is traversing the components. The protocol can be single stepped to study it more carefully.

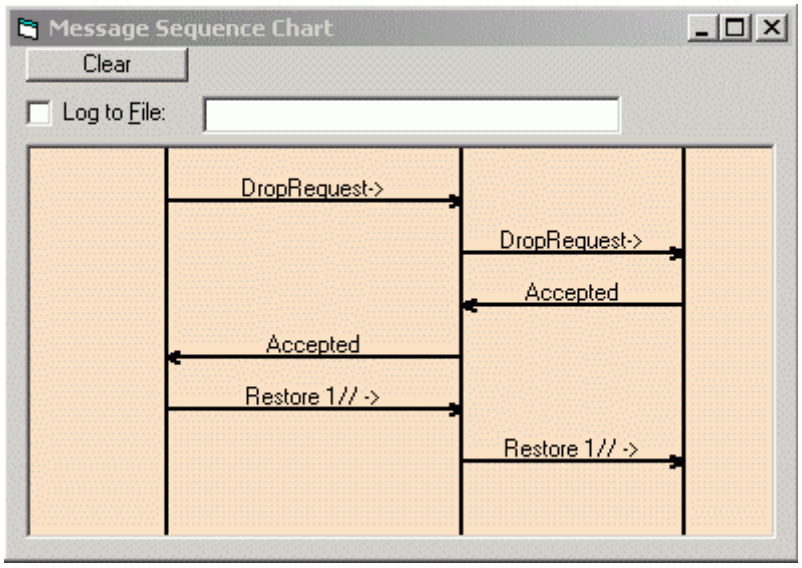

Figure 74. A message sequence chart

Another way to visualize the communication is to generate a message sequence diagram on the fly. To obtain this, we can also instrument the signal objects. First we assign a horizontal position to each control depending on the top and left coordinate of each control. Then we draw an arrow for each message that passes

each signal object, labeled with the name of the message. The result for a simple tune operation can be seen in Figure 74.

The prototype with the two techniques for visualization turned out to be sufficient to convince our designers that the protocol would work. In fact, when writing this paper a few years later, we still use the simulation to verify all examples that we have shown. Our simulation can be downloaded from [71].

8.5.2 Testing the Protocol

Running and verifying the protocol by hand is one way to gain confidence, but it is very difficult to walk through all possible scenarios by hand. We therefore added a random test generator to each form (activated by a button labeled 'Test Me', see Figure 72). This generator randomly selects an action that is allowed in the current state of the protocol, and calls it. Assert statements in the code then verify whether the protocol behaves as expected (using for instance the traffic light paradigm of section 8.4.2). The generator repeats this in a continuous loop until cancelled by the user.

One could argue that the quality of this test depends on the quality of the assert statements in the code, and that is true. We used this test to quickly find a number of errors in our initial implementation. After correcting them, the test now runs for hours without halting. While this gives a fair amount of confidence that the protocol behaves as it should, it still does not provide us with certainty.

8.5.3 Formal Verification of the Protocol

We have started with a formal verification of the protocol, using the Labeled Transition System Analyser (LTSA) [53]. Figure 75 shows a simplified version of the protocol with only the drop request ('dr'), drop acknowledge ('da') and restore ('re'). In the figure, '.t' stands for returning true, '.f' for returning false, and '.r' for a void return. We have used this technique to verify simple properties, such as that the protocol will indeed ensure that the screen is blanked for a configuration with a tuner and an output component.

```
HORCOM = ( dr -> ( dr.t -> re -> re.r -> HORCOM
                 | dr.f -> da -> ( re -> re.r -> da.r -> HORCOM
                                 | da.r -> re -> re.r -> HORCOM
                                 )
                 )
         ).
```

Figure 75. Part of HorCom specified in LTS

While the use of LTSA was extremely helpful to increase our insight into the protocol, the full discussion of the use of LTSA to model HorCom falls outside the scope of this paper. Besides, we have only succeeded in modeling a small part of the protocol; modeling the full protocol will take quite some effort. We invite

readers of this paper to help us in this analysis, or to come up with alternative ways of describing and/or verifying the protocol. One such an alternative technique was used by Uchitel *et al.* [107], who generated a model of part of our protocol from a set of scenarios.

8.6 Experiences

In this section some of our experiences with the protocol are listed.

8.6.1 Managing Complexity and Diversity

We have implemented the protocol in the software architecture that is currently being used for all of our analogue up-market televisions, and that is starting to be used for our mid-range TVs as well. The protocol consists of a single pair of interfaces, one for up-stream and one for down-stream communication, with six respectively four functions. Of these ten functions, three handle drop/restore, two source selection, three properties and two capabilities. Over 60 components implement these interfaces, with in total 700 signal inputs and outputs. The protocol currently handles 30 signal properties and associated capabilities (this is small enough to implement a set as a 32-bit word).

With a single set of components, we are able to create TV platform software for 15 different topologies of the hardware. Using Koala's partial evaluation mechanism, we create ROM images for five groups of products; the rest of the diversity is handled at run-time. Note that the decision which products to group into a single image is largely determined by the logistics of the factory, not by the software architecture! We would like to give some examples of different television topologies, but the complexity of those are such that they cannot be included in this paper. Hopefully, Figure 58 gives the reader at least some impression of the complexity.

Although we have no exact metrics on the performance of the protocol, we have experienced no major performance problems. The platform control typically consumes only 25% of the CPU cycles. Although the load approaches 100%, the duration of a channel change is still not determined by software, but rather by hardware circuits catching up with the new signals.

During a single channel change, we observed 370 calls of functions in our protocol with a maximum nesting of 11, which is roughly the maximum length of the signal path of a particular platform. Remember that Koala can optimize certain function bindings. We do need sufficient stack space for function call nesting, since also within a component functions may call each other. We have calculated that in certain situations the nesting may be up to 50 levels deep. But in all fairness it should be said that traditional designs (as depicted in Figure 59) also tend to have deep call chains due to the many levels of management.

A typical experience before introducing the protocol was that solving a bug in the TV platform software resulted in the introduction of more than one new bug – no person could oversee all of the consequences anymore. This has been solved now: control aspects have become much more local. However, there is also a downside to this. We have in fact replaced a classical top-down control system with what is essentially a distributed control system. Such systems are known to be harder to understand and debug if something is wrong.

8.6.2 Direct Function Calls

The reader may wonder why we use direct function calls instead of sending messages. In our systems, the typical overhead of sending a message is 100 µs. Strangely enough, this overhead does not reduce significantly when faster processors are used. Such processors typically have more registers to save, and although the speed of processors doubles every two years, the speed of memory increases much more slowly. To handle the latter, caches are often introduced, but context switches often result in cache misses. To put this in perspective: we have measured over 350 function calls during a single channel change in a TV.

There is another way of looking at our protocol. Traditionally, we draw the signal flow from left to right in our diagrams, and the control flow from top to bottom. But we could also draw the signal flow from top to bottom and align it with the control flow: a screen needs an output processor, which needs a decoder, which needs a source selector, which needs a demodulator, which needs a tuner, which needs an antenna (see Figure 58). In that sense it is perfectly logical that a down-stream (read: higher) component asks an up-stream (read: lower) component to select a source. Similarly, when an up-stream component obtains new information about the signal, it should notify the down-stream components about this. This is why we call our interfaces *ISignal* and *ISignalNotify* in Figure 73.

This explains why we use direct function calls up-stream. Most software architectures, however, decouple notifications (our down-stream calls) by using a kind of message passing. We could do have done that as well, but we have found that using direct function calls here is not harmful, provided that one uses some simple rules. In our protocol, the *Restore* can be nested in a *DropAcknowledge*, the *DropRequest* can be nested in a *Connect*, and an *OnPropertyChanged* can be nested in a *Connect*, a *ChangeProperty* or a *SetCapability*. Other forms of nesting are not allowed, nor is nesting within nesting.

8.6.3 Using Templates

Although the protocol is fairly simple, it is by no means trivial. If a dozen developers or more have to implement the protocol, chances are high that at least one of them will make a mistake somewhere. This may bring down the entire

application – a chain is as strong as its weakest link! For that reason, we sought to decouple the protocol implementation from the components.

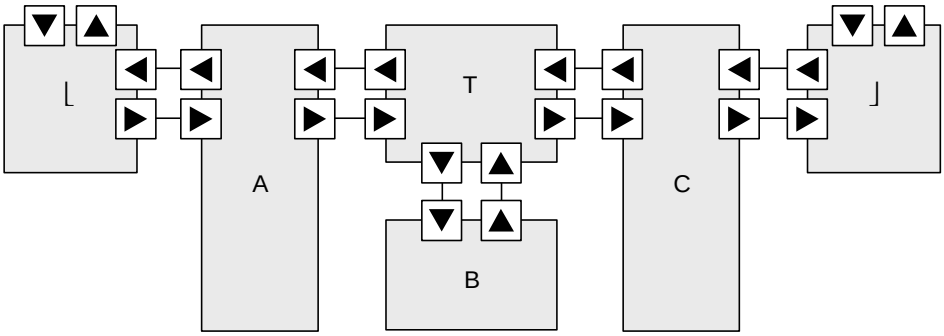

Figure 76. Separating HorCom from components

This turns out to be quite possible. We shall not discuss it in detail here, but essentially we create reusable T-pieces (component *T* in Figure 76) that can be inserted in the signal chain and placed on top of a more traditional component (*B*). The vertical interfaces of *T* are more easily understandable for most software engineers. Moreover, the developer of *B* need not worry about parts of the protocol not relevant for *B*, such as for instance source selection and capabilities.

The T-piece allows us also to insert traditionally programmed (read: legacy) control components into our signal path. We also wrote an L-piece (and its mirror image, both shown in Figure 76) that allows us to convert (chains of) components communicating horizontally to a more traditional vertical API.

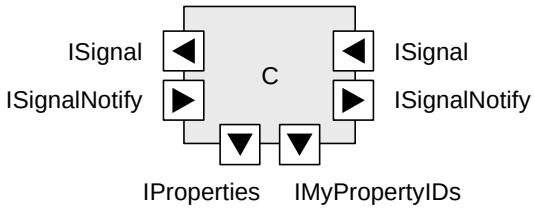

Figure 77. Making properties bindable

If all of the horizontal communication implementation is localized T- and L-pieces, then it is easy to change these to experiment with different strategies. We tried for instance to replace the change property strategy, where each property change traverses all components in the signal path, by a subscription mechanism. This greatly reduces the ‘call storms’ that occur whenever a property changes, but it replaces these with other ‘call storms’ whenever the topology changes. Although we certainly saw the advantages, we decided not to implement subscription in our second version of the protocol.

As a final remark on the reusable T-pieces, they also allow us to implement generic tracing facilities, in the same way as that we extended the signal objects in our simulation (Figure 73). Without T-pieces, tracing is still possible by adding glue modules or small components in the Koala bindings of inputs to outputs, but it is more cumbersome.

8.6.4 Binding Properties

The reader may remark that although we have localized measurement and use of properties, still global knowledge of properties seems to be embedded in the components. Furthermore, it seems difficult to get an overview of which component deploys which property. Most components only measure or need access to a small number of properties. How can we make that explicit, and sufficiently decouple components from the global list of properties? To achieve this, we let components require *two* additional interfaces to handle properties (see Figure 77). One is of type *IProperties* and contains general functions for dealing with properties, capabilities and sets of these. The other is of a type specific for the component (*IMyPropertyIDs* in Figure 77), and contains the IDs of the properties required by this component. We can bind this subset to an interface providing all IDs of all properties in the system, using the interface subtype capability of Koala. We can also rename certain properties in the process, using Koala's function binding. This provides us with much greater flexibility with respect to the measurement and use of properties.

8.6.5 Three Types of Reuse

The horizontal communication protocol has helped us significantly in managing the diversity of a product family of up-market televisions. Software components were coupled one to one to hardware components, and could be reused whenever the hardware was reused. Then came a new generation of mid-range TVs, where all hardware was different. Our managers were disappointed that we could not directly reuse the up-market TV software. We have several answers to this.

For our first answer, recall Figure 55. We split the software into three parts, one depending on the computing hardware, one on the domain (TV) hardware, and one hardware-independent. We envisage the majority of reuse to happen in the third part (services and applications), and we expect that part to grow significantly in the near future. Unfortunately, it is the TV platform that is the most complicated to build currently (as explained in section 8.1), and that part had to be rebuilt for the new generation of mid-range TVs. The fact that the computing hardware was also different did not help much either.

Still, the *interfaces* of the TV platform and infrastructure could be kept the same, and although this does not show up in measurements of reused lines of code, it certainly has a large positive impact on development time. Unfortunately, in our

development much lead-time is lost due to changes of or uncertainties in hardware specifications, so it is difficult to make the gain obtained by reusing interfaces explicit.

While we see the first type of reuse as reuse of services and applications, the second type of reuse is the reuse of software coupled to hardware as enabled by our horizontal communication protocol. So as long as hardware is being reused, the software can be reused. In practice, there is much hardware reuse in *space* (different products) but less in *time*. Although new functions shift over time from high-end TVs via mid-range to low-end TVs, their concrete implementation does not remain the same, due to miniaturization and other cost-saving measurements.

This brings us to a desired third type of reuse: of components within the TV platform. We already gave an example of such components in Figure 76, where we introduced reusable T- and L-pieces. Many other components can be made reusable too, such as power managers, channel tables and certain control algorithms. While we highly appreciate such reuse, the global structure of our TV platform software remains a mirror image of the (product specific) hardware architecture. The third type of reuse is visible only at deeper levels in this hierarchy.

8.7 Related Work

We believe our approach to be quite unique, but we can name corresponding work that has at least influenced our thoughts.

8.7.1 Unix Make

The first influencing factor is already quite old and maybe unexpected in this context. The Unix ‘make’ facility allows developers to specify the compilation of their software by just listing how a file depends on other files and how it should be compiled. The ‘make’ tool collects all build rules and executes them in the right order. Our protocol also allows to specify signal drops, source selection, property and capability handling locally, while still executing these in the right order. Note that in ‘make’, missing a single dependency in one of the rules may result in an erroneous rebuild, just as in our protocol an error in one of the components may cause the entire application to fail.

8.7.2 OO Design Patterns

Our protocol resembles certain object-oriented design patterns [31]. It corresponds most closely to the *Chain of Responsibility* pattern, where a request is decoupled from the object that actually implements it by passing it along a chain of objects until one handles it. In our case, the request is sent through a chain of signal processing components. All parts of the protocol exhibit this behavior: the drop/restore, the source selection, the property change and the capability management. In fact, to bring this one step further, we even contemplated letting

'normal' commands such as *Tune* or *VolumeUp* traverse the signal chain. This would make it easy to change the frequency of the tuner connected to the main stream – in the current situation at least some knowledge about the signal path is still required.

A second pattern that we use is the well-known *Observer* pattern. Down-stream devices 'observe' in fact the signals as provided by up-stream devices, and they are notified by the latter whenever changes occur. The observer pattern has proved its usefulness at many an occasion, and this certainly includes our situation.

Our third party binding, where components do not know their direct neighbors but the *compound* component makes the connection, is a special case of the *Mediator* pattern. But where a mediator is usually seen as an object *between* the others, our compound component (usually) is not active anymore after it has made the connection. In that sense, it resembles more of a *Broker*, be it that a broker is usually passive and accessed by the components, whereas our compound component takes the initiative.

8.7.3 Dynamic Architectures

Our protocol enables the creation of a dynamic architecture: a software architecture that adapts itself to changing circumstances. To understand this, consider Figure 58 again, where the dashed rectangle represents a plug-in module with extra functionality. We could store the control software for this module in a ROM on the module itself. Then, when an end customer buys this extension module and plugs it into his television, the driver code could be downloaded and connected to the other code using some form of dynamic binding. Our protocol would then ensure proper operation of the module in the existing TV. Although we do not deploy this idea currently, we certainly consider it as an option for future products.

8.7.4 Blackboard Architectures

Boasson created a blackboard architecture for air traffic control and air defense systems [10]. This architecture exhibits a large degree of uncoupling, to the extent that individual sensors and actuators can be added to or removed from the system without breaking the system. It also supports having different sensors that provide the same information with different degrees of reliability, while control algorithms are able to select the most appropriate sensor for their use. This closely resembles our approach. Novel in our approach is that our components do not communicate through a general blackboard, but rather through specific communication channels that mirror the hardware signal flow. Also in implementation our protocol differs, as we use direct function calls to communicate between components.

8.7.5 Coordination Languages

A related field is that of coordination languages (see for instance [34]). The underlying claim there is that programs consist of *computation* and *coordination*, and that these could be separated into different languages. 'Normal' programming languages are largely computation languages; coordination languages on the other hand focus on the specification of and the interaction between components, such that they can be (dynamically) composed into systems.

A distinction is made in [84] between data-driven and control-driven coordination models. Data-driven models rely on some shared data space such as a black board or a tuple space. Control-driven models deploy ports or interfaces, which are connected by and channels or connectors. Our solution falls in the latter category, as we set-up a dedicated (though possibly dynamic) communication structure between components. Many examples of control-driven coordination languages are mentioned in [84], among which Darwin, the predecessor of Koala.

8.8 Concluding Remarks

This completes the description and discussion of our style and protocol. For the reader's convenience, we present a short summary of the paper in the next subsection, after which we try to generalize what we have learned. We end with some acknowledgements.

8.8.1 Summary

In a television, the low-level software that directly controls the hardware turns out to be the hardest to build. The software is particularly dependent upon the topology of the signal flow in hardware. This topology is different for different members of the product family, it is apt to frequent changes until late in development, and it also changes at run-time. The TV hardware itself allows for easy composition, even with parts of other CE products such as DVD players or hard disk recorders. So a natural question is: can we make the low-level control software as composable as the hardware?

We have realized this by letting software components that control particular hardware devices have input and output ports that mirror those in hardware, be it that the software exchanges meta information rather than video and audio signals. This allows us to build a control architecture whose hierarchy exactly follows the structure of the hardware: core cells build chips which populate layout cells which constitute printed circuit boards which are combined into products. As a consequence, software reuse follows hardware reuse, or put differently, if new hardware needs to be developed, we also have time to develop new software.

Instead of a traditional top-down control hierarchy (with bottom-up notifications), we now get a right-to-left (up-stream) hierarchy with left-to-right (down-stream)

notifications. For such a hierarchy we have devised a set of interfaces that allow us to handle the disappearance and recurrence of signals, the selection of sources, and the measurement and usage of properties of signals. The two major advantages of our approach are the automatic correctness of the order of operations in a single product, and the ease with which we can construct a variety of products from a given set of components.

Although the protocol may feel natural to some people, our experience is that most software developers find it quite unusual at first. For that reason we had to build a prototype that simulates the protocol and shows how all commands traverse the signal chain. It turned out that Visual Basic was a convenient means to build this prototype. As an extra, we built a wiring protocol into the simulation that allows us to graphically arrange controls (representing devices) on a form (representing the product) and from that derive the signal flow topology automatically.

8.8.2 Generalizing our Findings

Presenting a case as the one in this paper is always insightful as far as realism and complexity is concerned: we have actually implemented this protocol and are using it in all of our up-market TVs. A remaining question is how well this approach generalizes to other domains. To answer that, three different aspects must be distinguished in our paper.

We consider the overall approach of communicating meta information along signal paths an architectural style. It resembles the pipe and filter style, and it embodies several design patterns. It can be used for any software controlling signal flow, both for hardware and for software streaming. Note that the communication between components is more directed than with shared data space approaches, such as black boards or tuple spaces. One generalization may be that we create a distributed control system where communication between control components follows the topology of the controlled (hardware) components in the domain. Perhaps similar techniques are useful in for instance telephone switching or factory control.

Within our domain, numerous variations are still possible for the protocol. We devised one protocol for controlling analogue up-market TVs, but we are sure that in other subdomains revised protocols are necessary. Such protocols are not discussed here, but hopefully the protocol described may serve as an example.

Finally, the implementation of our protocol with direct function calls in two directions is also quite unusual. Although it looks complicated at first, we have had little trouble in making it actually work, and we feel that the increase of efficiency outweighs the added complexity. Most of the related approaches that we could find use more traditional forms of message passing.

8.8.3 Acknowledgements

I would like to thank Erik Kaashoek for initial support in the ideas presented in this paper, Elmar Beuzenberg and Klaas Brink for being the first critical recipients, Jeff Magee and Jeff Kramer for helping me to formalize part of the protocol, Gerben Soepenberg for analyzing and innovating the implementation, and Gerben Soepenberg, Chritiene Aarts en Jan Bosch for reviewing this document.

Chapter 9

Validation and Future Work

9.1 Introduction

In Chapter 1, we explained how software in consumer products grows according to Moore's law, and how most - if not all - manufacturers of such products will eventually face the following challenges:

- How can we maintain a high *quality* of the software?
- How can we handle the *diversity* in a product family?
- How can we maintain a short *time to market*?

Additionally, for manufacturers that produce more than one product family:

- How can we *combine* features from different families into single products?

We indicated three important fields of research that can contribute to this:

- Software *architecture*;
- Software *components* and component based software engineering;
- Software *product lines*.

This led us to formulate the following research questions:

- Can *software architectures* be made more *explicit*, and does this increase the quality of products?
- Can *component technology* help to build product *families* and *populations*, and what design patterns are needed for that?
- Can component technology be applied in *resource-constrained products*, and what consequences does this have on the technology?
- Can this all be made into a business success, and what impact does this have on development *process* and *organization*?

This chapter answers these research questions. It also provides pointers and ideas for additional research in the future.

9.2 Explicit Software Architectures

The first question was: can *software architectures* be made more *explicit*, and does this increase the quality of products? We shall answer this question in steps.

Koala can indeed describe the architecture of software for televisions at a level of abstraction that is sufficiently high, yet with enough detail to allow for discussions of the architecture based on the architectural description alone. Especially the graphical syntax is a useful instrument in this. We base this conclusion upon the following observations:

- Many of our architects have Koala diagrams of their part of the architecture lying on their desk, for quick reference.
- We have performed a number of meaningful architectural reviews using only Koala diagrams.
- When asked whether we (the community of architects) should stop making Koala diagrams, *all* architects answered 'no', even though the diagrams were made by hand using Visio, without a tool-supported relation to the textual Koala descriptions [56].
- When shown on conferences and in workshops, Koala diagrams inevitably trigger admiration [67][80]. A Koala diagram was used in the keynote speech of the chief technology officer of Philips [37].
- Other communities in Philips are keen on using the Koala notation (although they also ask why we do not use the UML notation).

Koala is sufficiently general as a language so that it does not have to be bypassed:

- We performed a number of tests to check whether developers *bypass* the Koala component and interface mechanisms, and found hardly any.

This means that it is valid to base architectural discussions on Koala descriptions, since by definition they accurately represent the architecture of the implementation.

Now the important question, does Koala help to build software with a higher quality, and with a shorter lead-time?

- The first product built with Koala was not built in a shorter time than previous products (still two years), and did not have fewer problems (5000).

This shows at least that Koala is not a panacea, and that people are still people. Two important elements may have influenced this result:

- We started development with a relative inexperienced team, so learning-in explains part of the lead-time. On the other hand, we developed the software for stable hardware, whereas usually, the hardware specification changes during software development.

- The factory could only use our software for new products at distinct points in time, synchronized with their yearly program. Especially testing prereleases of our software was not possible due to other engagements of the test team.

We shall say more about the business success of our approach in the next section. In general, the use of Koala improves at least the *perceived* quality of the software architecture:

- Koala diagrams provide a unique way to navigate through the software.
- We studied the architecture of a number of other software stacks in Philips, and found that not having an architecture description language leads to many more undesired – or even unknown – connections between components.

This means that in systems built with Koala, much less architectural verification is necessary! This does not imply that *no* architectural verification is needed:

- Although systems built with Koala tend to have much better structure, it is still necessary to verify whether developers use the language in the 'correct' way [48].

9.3 Families and Populations

The second question was: can *component technology* help to build product *families* and *populations*, and what design patterns are needed for that?

We can safely say that the use of Koala in Philips to build a family of televisions is a business success:

- *All* Philips mid-range and high-end televisions now have software inside that has been built using the approach described in this thesis.
- Philips Consumer Electronics can produce a sufficient diversity of products; software is *not* on the critical path. Around a dozen different product teams produce several ROM images per year, each image capable of controlling several different products (estimated total products >50).

Even more importantly:

- The architecture shows *no signs of degeneration* yet (after 6 years). Previous architectures became unmanageable after 3-5 years.

An important reason for this is the split of the software into subsystem packages, each with its own independent evolution cycle. This split-up is not *forced* by Koala, but it is at least *enabled* by the mechanisms provided by Koala, such as interfaces as first class citizens and explicit provides and requires interfaces.

Can the approach be used to build product *populations* rather than 'just' families?

- We have shown on several occasions that software built with Koala can be easily combined with software from other domains to build ‘combi’ products such as an analogue TV with built-in set-top box and hard disk recorder.
- We have shown it to be relatively easy to ‘Koalatize’ software originally not written with Koala, and integrate that with Koala software.
- (Anecdote) When hearing that software written with Koala could be easily split into its components and then integrated in another Philips stack, while the other stack could not be split-up at all, management in Philips *almost* decided to make the other stack the corporate standard, seeing that that would give the easiest evolution path for the total set of software…

However, building product *populations* with Koala has not happened yet, and this is mainly due to non-technical issues:

- Our original aim, to integrate VCR software with TV software, has never happened because the development and production of video recorders is now outsourced by Philips.
- Integrating TV and set-top box software has not happened as Philips stopped making set-top boxes. The net effect of this integration: the creation of digital TVs, *is* now happening (see below).
- Integrating TV and DVD software has not happened yet as DVDs were produced by a different organization in Philips, and up to now just building a DVD module into a TV was more cost effective. A more full integration of TV and DVD software *is* happening now (see below).

Summarized, there are no technical reasons whatsoever why our approach would not be suitable for creating product populations, but non-technical issues (mainly organizational) turned out to very difficult to overcome. We still have hope that it will happen; our Semiconductors division is now very keen on delivering a single set of software with its chips providing (at least) analogue and digital television and DVD functionality. To achieve this, three software stacks are currently being aligned, and much of the approach described in this thesis is being used.

Finally, what are the technical mechanisms that make developing families and populations a success? Beyond any doubt, the notion of explicit *requires interfaces* that can be bound by third parties comes first:

- The notion of requires interfaces makes components dependent on *services*, rather than on *implementations* of these services. This allows components to be bound to different implementations in different products.
- Although it is still possible to make a component depend on a specific other component, by containing that component as sub-component, experience

shows that developers tend to delay this decision to a later point in (design) time, thus making the component more reusable.

- The fact that every context dependency must be made explicit is a reason for developers and their architects to critically examine every dependency for its necessity, thus resulting in generally cleaner designs.
- Requires interfaces also make testing components much easier, by running the component in an environment where all requires interfaces have been stubbed.

Not many other component models support requires interfaces as primary element of their model. Darwin supports requires interfaces [52] in the same way as Koala. COM has connection points [17], but these are mainly used for notification. SOFA [90], IEC 61131-3 [38], PECOS [114] and CORBA's component model version 3.0 all support requires interfaces to some extent, but routing every dependency on the context through requires interfaces is unique to Koala.

The second major mechanism is that of *diversity interfaces*:

- Diversity interfaces make explicit in what ways a component can be tuned into a product. Note that classical C program often depend on undocumented environment variables passed to the compiler on the command line. With Koala, every such dependency is made explicit.
- The use of expressions to calculate the diversity parameters of a component in terms of the diversity parameters of the compound component solves the problem that in many software stacks, either low-level components depend on product specific flags, or at the product-level, low-level details must still be defined.

The third mechanism is the use of switches and the calculation of 'reachability':

- Switches can be used to implement variant components, but actually provide a much more general control of the list of contained components and their bindings.
- 'Reachability' is the mechanism used to remove unreachable components before compilation. It allows defining more reusable components without the additional use of resources in products where only part of the component is used.

It has been argued a disadvantage that the *containment* of components in Koala is at most a *closed* variation point [12]. This means that it is possible to *select* a sub-component still at a late stage in design, but it is not possible to add a new variant without changing the Koala descriptions (in other words, there are no component plug-ins). This is true; the advantage of our approach is that the designer of a compound component can actually – at least in principle – test the component with all allowed selections of subcomponents.

9.4 Resource Constraints

The third question was: can component technology be applied in *resource-constrained products*, and what consequences does this have on the technology?

When we proposed component technology as a means to handle diversity, there was a fierce debate on whether the computing facilities in a television would be powerful enough to cope with the additional overhead induced by the technology. By inventing Koala, we showed that component technology could be used without *any* additional overhead. This effectively stopped *any* debate on the applicability of component technology in televisions, although it replaced these with a discussion on the use of proprietary tools.

Within the television department, we were able to show that the investment in tools would be easily compensated with the advantage of handling diversity properly. In other departments, we could not make this clear, until we decided not to compare Koala to Microsoft's COM, but rather to their own proprietary build environments. In one case we could compare the 23,000 lines of code of the Koala compiler to over 25,000 lines of code of build scripts, and explain that the use of Koala would actually mean less maintenance. Only recently has this convinced other groups to start using a Koala-based technology.

Comparing Koala to COM again, we see that a call of one component to another in Koala has no overhead whatsoever as compared to C programming, whereas in COM:

- There is a triple dereference run-time overhead, which was significant on a 16-bit microcontroller with a 32-bit address space, and which is still significant on a modern computer architecture that relies heavily on caching.
- There is a code size overhead in these triple dereferences, but also in setting up VTables for interfaces and interface tables for classes. This could easily add up to hundreds of kilobytes if using the same granularity of components.

One solution to the latter would be to only use VTables at larger grain size and use classical programming techniques within such components. We have seen other groups define larger components for similar reasons, and judge that this results in a severe loss of diversity handling capabilities. A better solution would be to use the Koala binding *within* such components. Actually, a design goal for Koala was to abstract from the binding implementation technology, and to use static binding (with #define) and dynamic binding (e.g. with VTables) where appropriate. Up to now, we only implemented this idea with classical link-time binding.

Comparing Koala to C programming, we see that the use of Koala is as efficient as the use of plain C, and even more efficient when building layered systems where some functions in a layer just pass the control to lower-level functions. In C

programs, this must be programmed as a function, whereas in Koala this can be shortcut.

There is one additional advantage of the static binding techniques used in Koala, and that is that after generating a product, classical source code browsers can be used to navigate through the software. Also, static analysis techniques can be used to determine the correctness of (certain aspects of) the product. In systems relying heavily on dynamic binding, this virtually becomes impossible.

Our expectation was, in 1996, that in 5 years time, computing power in a TV would have increased such that use of VTables would be no problem anymore. Now, in 2004, we still observe that many development groups in Philips rely on static binding, and introduction of a system-wide VTable technique is still out of the question. This strengthens our belief in the usefulness of Koala. Others report similar experiences [61].

9.5 Process and Organization

The fourth question was: can this all be made into a business success, and what impact does this have on development *process* and *organization*?

We have shown in section 9.3 that Koala is currently being used to build *all* software for all mid-range and high-end televisions of Philips, and that software is not on the critical path. The realization of this had some strong consequences on the process and the organization of software development. The most prominent are the following:

- An organization was set-up with asset teams and product teams. The asset teams developed packages of components *for* reuse, the product teams developed products *with* reuse of these components.
- Asset teams are not in full control of their roadmap, but rather develop on demand of multiple ‘customers’ (the product teams).
- Instead of making the asset teams part of a platform organization that is separate from the product organization, asset and product teams are together part of one development program.
- Development is controlled by a hierarchy of architects and project leaders. Global architects rule over asset and product architects; the latter are peers and not in a hierarchical relation. The program manager rules over asset and product project leaders; again, the latter are peers and not in a hierarchical relation.

In the development process, it took quite some effort to change from a traditional single-product way of working to a product-line way of working:

- Asset teams had to learn that they could not use classical project planning any more. Instead, they had to serve multiple customers and deliver different results to customers at different moment in time.
- This had the most profound effect on quality officers, whose sole task it was to ensure that the teams followed the department's standards in software development (which were classical and single-product oriented).

The most important lesson that we learned in our work and in related work within Philips [67] was the importance of having a carrier product almost right from the beginning:

- Without a carrier, asset teams tend to fall into a 'genericity' trap, and spend a long time in creating a platform that is too generic and heavyweight.
- The first carrier product should be carefully chosen. It should be complex enough to represent the product family, it should receive a sufficient amount of management attention to ensure visibility of the product line approach, and yet the company should not critically depend upon the success of the project.
- Inevitably, when using a carrier, asset teams tend to build software that is too product specific at certain places. Architects should guard this.

Some other issues that we encountered:

- In turned out to be hard to have developers, architects, project leaders and development managers talk about the assets (subsystems in our technology) using the *same* identifier. The Teletext services subsystem was called *txtsvc*, *txsvc*, *txsvcs*, *Txt Services*, *Teletext Services*, and many more variants. While this may seem a trivial issue, in our experience it is an indication that the mental image of these people is different. This can have repercussions on many other aspects of development.
- Setting up a web site per team where a team publishes the current status of their code and documentation on a daily basis, and where they publish their releases, turned out to be invaluable, especially when compared with other development in Philips where this has not been standardized.

9.6 Other Evidence

In our introduction, we mentioned the transferability of the method as an essential element. We have found that Koala can be easily transferred – developers have no problem learning it:

- All developers follow a 3-day course in Koala and the MG-R architecture, and in general have no problems with the language afterwards (there are a

few misuses of the language that are difficult to prevent, such as the *inline* construct).

- Most developers have no problems whatsoever in applying the language in their work, and do not question the usefulness, with the exception of a few very experienced C 'hackers' (who usually have a single-product view).
- Even groups at different sites have had no problem adopting the language. In all honesty, it should be said that teaching the language to other companies has not proven fruitful yet, but this can also be due to other causes.

The Koala mechanisms for coping with diversity turned out to be easy to transfer:

- In other software systems in Philips, we find a great variety of diversity mechanisms without any standardization. In software based on Koala, most developers select the right diversity mechanism as a 'second nature', without inventing new mechanisms 'on the fly'.
- In the previous software architecture for televisions, diversity management was the largest problem identified. In the current architecture, diversity has completely disappeared from the list of problems.

Setting up new teams to follow the same processes showed no further difficulties:

- Over the last 6 years, many teams have joined the community and adopted the processes. This never caused serious problems, although every team had to go through a period of a number of weeks of understanding and accepting the set of solutions.

The software development group that uses the approach described in this thesis has undergone almost a dozen assessments, many by external consultants. All of these list possible improvements (of course), but none of these have concluded that the approach is fundamentally wrong, nor that the underlying component technology is the wrong choice. Some of the assessments list our work as one of the prominent examples of product line approaches in Philips [66].

9.7 Koala Design Patterns

Since its inception in 1996, the Koala language has hardly changed. Our *use* of the language, on the other hand, has significantly improved over the years, which we feel demonstrates the general applicability of the language. This section contains a number of ideas that we have developed over the years.

Component can be parameterized using diversity interfaces. Syntactically, and also semantically, these are just requires interfaces, but thanks to function binding and expression evaluation in the language, this provides a full parameter mechanism. What we did not provide is a mechanism to specify *default values for parameters*. We discovered that we could specify default values *in* the language by declaring

diversity interfaces optional and adding a switch to choose the default values if the interface is not connected. One disadvantage was that you have to specify *all* values if you want to change *one*. The pattern in Figure 78 solves this. The pattern is quite verbose, so there is still some room for improvement here.

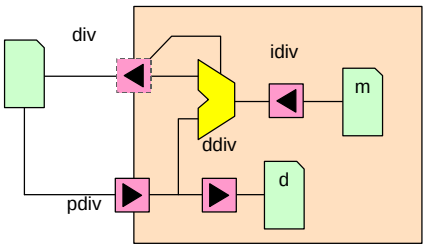

Figure 78. This pattern assigns default values to diversity parameters.

In COM, the functionality of a component can be extended by adding a second interface while keeping the original. This works in Koala too, but it is also possible to change the type of an interface. To guard compatibility, Koala supports interface sub-typing. We discovered that *sub-typing could also reduce dependencies between packages*. Suppose that C1 is part of package P1 and that it provides interface I1 defined by P1. C2 requires this interface and is part of package P2. This makes P2 dependent on P1. If P2 *copies* I1 and renames it to I2, then the dependency is removed but composition will still work. We found this particularly useful to decouple *versions* of packages.

Diversity and other requires interfaces cannot be extended without breaking the compatibility. It is of course possible to *add* another interface, but sometimes it is more convenient just to extend the interface. By having a version parameter in the interface, the provider can indicate which version of the interface is implemented, and the requiring component can program default values for parameters that were not yet defined in that version. This feature is called *versioned interfaces*. It only works at compile-time.

One of the research challenges in component-based software engineering is to calculate system properties in terms of properties of the components. We were able to predict the code size of a system in terms of the code sizes of the constituting components [30]. This is non-trivial, since the size of components may depend on the settings of the diversity parameters. Important for this thesis is that we were able to calculate code size to a high precision within the current Koala framework, i.e. without extending the language.

We can extend this principle to build *self-configuring systems* using the reflection mechanisms built into Koala. Components can specify certain resources that they need in the Koala language in a provides interface, e.g. the number of semaphores created by the component. At the top-level, these requirements can be added and

fed to the diversity interface of the component encapsulating the real-time kernel. This way, the system itself will ensure that there are sufficient resources.

Brown, Spence and Kilpatrick propose an architectural description language called ADLARS [19], in the style of Koala, in which they model features of a component as oval interfaces. In their formalism, they can link features to the subcomponents that implement them. We believe that features can be modeled *inside* Koala as it is now, for instance in the form of Boolean parameters in provided (if a component implements them) or required (if a component needs them) interfaces. This allows for interaction between features and diversity parameters, e.g. components can be fine-tuned dependent on provided features, or features can be provided dependent on settings of diversity parameters. A more detailed discussion of this is outside the scope of this thesis.

Asikainen, Soininen and Männistö describe an approach for modeling configurable software product families based on Koala [4], where they extend the language with constructs to describe a *family* of products instead of a single product. They also add *constraints* to specify the valid products. We believe that Koala is already able to describe families rather than individual products, using conditional expressions, the switch statement and 'reachability'. Moreover, constraints have recently been added to the Koala language using assert statements expressed in terms of diversity parameters.

Our last example deploys Koala as planning tool. When developing components for use in multiple products, it becomes necessary to check whether subsequent releases of the component fit correctly in (releases of) the products. See [104] for a description of this problem. By adding a diversity interface to each component with a parameter to specify the date, and making provides interfaces optionally present as function of that date, *one* Koala description can describe the evolution over time of a component. This enables the product can to check whether its sub components are mutually consistent at distinct points of time. Although we do not use this in our development group yet, a feasibility study has shown the applicability.

9.8 The Future of Koala

In the previous section, we discussed how the *current* version of Koala can be used to implement a variety of patterns that help to manage the quality, diversity and road maps of the products. We also developed some ideas for which we *do* need to extend the language, and list a number of these in this section.

Software in a television is executed by multiple threads in parallel. Threads are usually shared between components, and can be allocated to components at a late point in design time [80]. By default, components have no built-in synchronization mechanisms, so when constructing a system, proper synchronization facilities have to be added where needed. For this, it must be clearly specified which interfaces can run on arbitrary threads, and which must run on specific threads. We added this

information to Koala descriptions and were thus able to prove the correctness of products with respect to synchronization [82].

In our development organization, many of the components are not stable but rather evolve over time, providing new interfaces and making old interfaces obsolete. This introduces two risks: the use in products of interfaces of which the status is still pending, and the continued use of interfaces that have been declared obsolete. By making the status explicit in the Koala language, we can calculate the maturity of the product (much like the use of *deprecated* in Java and *obsolete* in .Net).

In [80] we explained how we use *packages* to manage the 'design in the large'. Packages can be made formal in the Koala language, and a strict usage relation between packages can be specified. By making the package concept recursive, we can both manage the usage relation at higher levels (as proposed in [96] and [27]), and add substructure to the large packages that we currently have.

The current Koala comes with a set of coding conventions: the programmer should include a generated header file and specify all dependencies in Koala, and also use certain naming conventions for implementing or using functions in interfaces. It is easy to encapsulate code not written with these conventions in Koala components, but it would be better if Koala descriptions could just be added to existing code. For this we invented *round interfaces*, corresponding to the inclusion of classical (C) header files, and *round modules*, performing their own includes. This step is necessary to introduce Koala into a larger part of our organization and thus finally realize our dream of product populations. The result of this can unfortunately not be part of this thesis.

9.9 Conclusion

We started this thesis with the software creation problems that manufacturers of consumer electronics products face. They must create a larger diversity of products of a higher complexity in a shorter time while maintaining high quality. Explicit software architecture and reuse of software components in a well-organized product line approach are the key solution elements. To realize this, a number of hurdles have to be taken, especially if the ambition is to create *populations* of products, rather than 'just' families.

We showed how architectures can be made explicit, but if you do this 'after the fact' then you can measure discrepancies between architecture and implementation but you will not be able to do much about them. If instead you use an architectural description language in forward mode, you will get cleaner designs that are also better documented. Most ADLs use components, but you need mechanisms that support independent deployment to build product populations. Composition is then a better way to deal with diversity than 'just' variation. Making this all work is not a technical problem alone; you also need to adapt the development process and organization. One example is traditional configuration management, which you can

use to manage versions and temporary variants, but which is not the best solution for managing permanent variation. Finally, there are also technical hurdles to take in making software compositional: our horizontal communication protocol is an example of this.

We thoroughly enjoyed the work that we described in this thesis and that spread over two centuries, and are at the same time sure that the work is by no means completed. We invite all readers to improve on and add to our approach wherever possible.

Appendix A The Koala Language

This appendix defines the Koala language as of August 2003. Features that have been removed are still defined in this appendix but are clearly marked obsolete. This is not an introduction into Koala: familiarity with the language is assumed. Note that the Koala compiler may impose additional limitations on the use of the language.

Section A.1 explains the basic concepts of Koala. Section A.2 defines the lexical syntax. Section A.3 defines the syntax and well formedness of interface definitions, section A.4 of component definitions, and section A.5 of data type definitions. Section A.6 provides naming conventions, while section A.7 lists a non-optimizing code generation scheme. Section A.8 ends the appendix with concluding remarks.

For a proper interpretation of the syntax diagrams, the following rules hold:

- Every line segment is reachable.
- A line segment with an arrow can only be traversed in that direction.
- A vertical line segment can be traversed in either upward or downward direction.
- If a horizontal line segment can be traversed in rightward direction, it is forbidden to traverse it leftward direction.

A.1 Concepts

The Koala language is based upon the following concepts:

- An *interface* is a small set of semantically related elements, which can be functions, parameters or constants. An *interface definition* describes the syntax and semantics of these elements, and is a unit of specification. An *interface instance* provides access to the functionality of a component, or it specifies the use of (external) functionality by a component.
- A *component* is a set of files in a single directory. A *component definition* describes the provided and required interfaces and the internal structure of the component, and is a unit of reuse. A *component instance* is a component as compiled and linked into a running system. A *basic component* is a component without subcomponents. A *configuration* is a component without interfaces on the border.
- A *module* is a unit of code; put differently, there is no code outside of modules. A module is contained in a component. Note that the module is *not* a unit of reuse, but the containing component *is*. The module implements and uses interface instances connected to the module. The implementation can be chosen per interface element: in C or in Koala expressions.

- A *cable* – or *interface binding* – connects an interface to another interface, an interface to a module, or a module to an interface.
- A *switch* connects a list of interfaces to one of a specified number of lists of other interfaces, depending on the evaluation of an expression. Switches are used to implement compile-time and run-time binding.
- A *fiber* – or *function binding* – implements a single function in a module as a Koala expression that can be partially evaluated by the Koala compiler, thus providing room for compile-time optimizations.
- The *repository* is the place where all component, interface and data type definitions are stored.

Koala targets C as implementation language.

A.2 Lexical Syntax

The lexical syntax of the Koala component and interface definition language determines how to split a stream of characters into a sequence of tokens. Unless specified otherwise, the longest match is made.

- A non-empty sequence of space (ASCII code 32), tab (9), carriage-return (13) and line feed (10) characters is treated as *white space*. White space separates tokens but is further ignored during parsing.
- There are two forms of *comment*: between /* and */ brackets, and following // up to the first new line (line feed on most platforms) character. Comment is treated as white space and is further ignored during parsing. Bracketed comments may not be nested.
- An *identifier* is a sequence of letters (‘a’..‘z’, ‘A’..‘Z’), digits (‘0’..‘9’) and the underscore character (‘_’), starting with a letter. Keywords as defined below are not valid identifiers. Note that Koala is case sensitive.
- The following words are Koala *keywords*: **addressable**, ***apply***, **component**, **connects**, **const**, **contains**, **except**, **false**, **file**, **group**, ***handle***, **in**, **inline**, ***instance***, **interface**, **koala**, **legacy**, ***library***, **module**, ***multi***, ***no_lib***, **no_prefix**, **null**, **on**, **optional**, **otherwise**, **out**, **prefix**, **present**, **provides**, **requires**, ***single***, **specials**, **switch**, **true**, **type**, **uses**, **using**, **void**, **within**. The keywords that have been made *italic* are obsolete.
- The following single character symbols are valid Koala *operators*: ‘**,**’, ‘**?**’, ‘|’, ‘^’, ‘**&**’, ‘**<**’, ‘**>**’, ‘**+**’, ‘**-**’, ‘*****’, ‘/’, ‘**%**’, ‘**!**’, ‘**~**’. The following double character symbols are valid Koala *operators*: ‘||’, ‘**&&**’, ‘**==**’, ‘**!=**’, ‘**<=**’, ‘**>=**’, ‘**<<**’, ‘**>>**’. In any sequence of these characters, the longest match is preferred.

- Additional *symbols* recognized by Koala are: '**(**', '**)**', '**{**', '**}**', '**;**', '**=**', '**.**' and '**:**'.
- A *number* is a non-empty sequence of digits ('0'.. '9', see below how to represent hexadecimal numbers), optionally prefixed with '0x' or '0X'. Numbers may be post fixed with 'u' or 'U' (for *unsigned*) or with 'l' or 'L' (for *long*) or both[13]. The notation is assumed to be:
  - *Decimal* if the number starts with '1'..'9'.
  - *Octal* if the number starts with '0'. Only digits '0'..'7' are allowed[14].
  - *Hexadecimal* if the number starts with '0x' or '0X'. For digits > 9, the letters 'a'..'f' or 'A'..'F' can be used.
- A (normal) *string* is a sequence of characters enclosed in double quotes (""). The double quote character itself cannot be included in a string. The back slash character ('\') is taken literally. An embedded new line is not allowed.
- A *code block string* is a sequence of characters enclosed in '@' signs. Embedded new lines are allowed, nor are '@' sign characters. Leading and trailing blank lines in the code block string are ignored, as is any leading and trailing white space of each line. Code block strings can be used wherever normal strings are used.

Data type definitions are C header files enriched with Koala annotations. There are three types of Koala annotations in these files:

```
/** koala type  <identifier> **/
/** koala group <identifier> **/
/** koala using <identifier> **/
```

Remarks:

- A sequence of space and/or tab characters can be used to separate the tokens in these annotations.
- Any other comment that starts with /** *koala* will result in an error.

Data type definitions are discussed in more detail in section A.5.

A.3 The Interface Definition Language

Below, we provide syntax diagrams and well-formedness rules for the interface definition language (IDL). In the diagrams, rounded rectangles denote keywords, symbols or operators; plain rectangles denote non-terminals, identifiers or strings.

---

[13] The Koala compiler performs all integer calculations with 32 bit signed integers. Unsigned and long integer constants are therefore cast to a normal C '*int*'.

[14] Koala currently itemizes '09' as octal '0' followed by decimal '9'.

A.3.1 Interface Definition

An *Interface Definition* (see Figure 79) defines a single interface type.

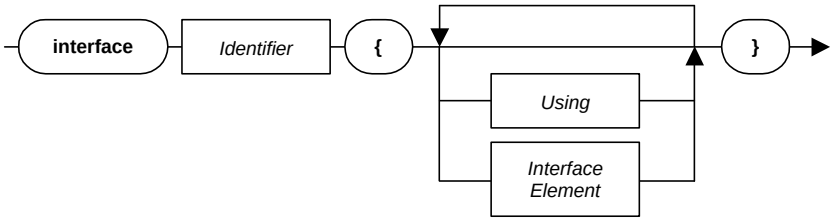

Figure 79. Interface Definition

The following rules apply:

- The identifier that follows **interface** defines the name of the interface type.
- This name must be globally unique: it may not be equal to the name of any other interface type, component type or data type (group) in the repository.

A.3.2 Using

The *Using* clause (see Figure 80) makes additional data types accessible.

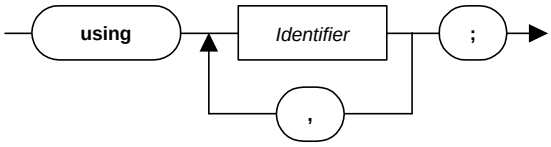

Figure 80. Using

The following rules apply:

- Each identifier listed in the using clause must represent a data type (see section A.5.1) or data type group (see section A.5.2) that is defined in the repository.
- The using clause is obsolete – there is no immediate use for it. All data types occurring in function prototypes are automatically made accessible, and components can specify using clauses themselves (see section A.4.4).

A.3.3 Interface Element

An *Interface Element* (see Figure 81) specifies a function, an interface parameter or an interface constant.

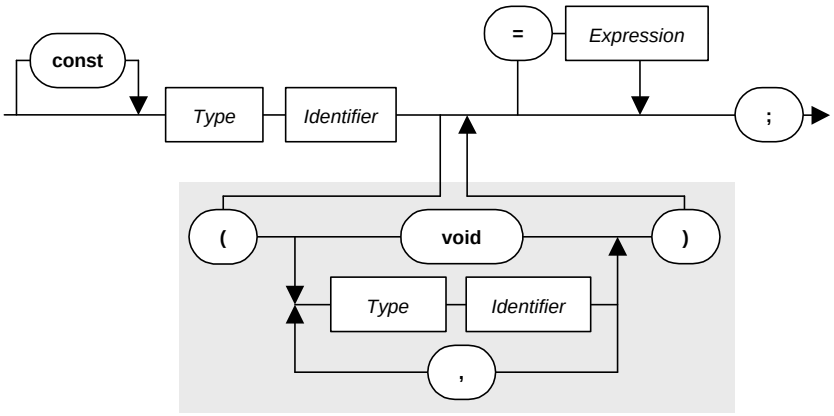

Figure 81. Interface Element

The following rules apply:

- If **const** is specified, then Koala will generate an error at an instance of this interface type if it cannot guarantee at compile-time that the element has a constant value.
- The *type* of the interface element should be valid (see section A.3.4).
- The identifier specifies the *name* of the interface element.
- This name should be unique for the interface definition.
- This name should also not be equal to the name of any data type or data type group in the repository.
- **Definition:** a *function* is an interface element that has a parameter list (shaded in Figure 81).
- The parameter list may be empty, which must be specified with '**(void)'**.
- If the parameter list is non-empty, then each parameter must have a valid type (see section A.3.4).
- Each parameter must have a name (the identifier) that is unique for the list of parameters, and that is not the name of a data type or data type group defined in the repository.
- **Definition:** an *interface parameter* is an interface element without a parameter list (shaded in Figure 81), and without an expression assigned to it.
- **Definition:** an *interface constant* is an interface element without a parameter list, but with an expression assigned to it.

- For an interface constant, it must be possible to calculate the assigned value by considering the interface definition alone.
- The expression may refer to other interface elements; in such cases the name of the element must be prefixed with the name of the interface type.
- It is not allowed to use an in-line expression (section A.4.21) in an interface definition.

A.3.4 Type

A *Type* (see Figure 82) specifies the data type of an interface element or of one of its parameters.

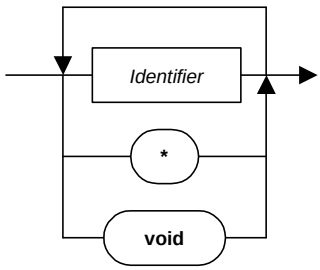

Figure 82. Type

The following rules and remarks apply:

- The sequence of identifiers, **void** and '*' should be a valid C type (such as 'long int' and 'void *')[15].
- Remark: type constructors such as *struct*, *union* and *array* cannot be used.
- Koala treats each identifier (and also **void**) as a separate data type that should be defined in the repository (see section A.5.1), with the exception of 'char' and 'int', which are predefined in Koala.

A.3.5 Interface Sub-typing

An interface type IA is said to be a *subset* of interface type IB if the following conditions are met:

- Every element of IA also occurs in IB, with the same name, the same type and – if present – the same parameter names[16] and types. C sub-typing is not permitted here, so 'char' and 'int' are different types.

[15] The current Koala compiler allows at most 2 identifiers. Also, it allows at most one '*', which may follow either (but not both) of the identifiers.

- If an element of IB is assigned an expression in the interface definition, then either the corresponding element of IA must have an expression that results in the same value[17], or it must not have an expression assigned to it at all.
- If an element in IA is declared **const**, then the corresponding element in IB must be constant, i.e. it must either be declared const as well, or it must have be assigned an expression that Koala can evaluate to a constant.

In formal type theory, IB is said to be a *sub-type* of IA.

A.4 The Component Definition Language

Below, we provide syntax diagrams and well-formedness rules for the component definition language (CDL). In the diagrams, rounded rectangles denote keywords, symbols or operators; plain rectangles denote non-terminals, identifiers or strings.

A.4.1 Component Definition

A *Component Definition* (see Figure 83) defines a single component type.

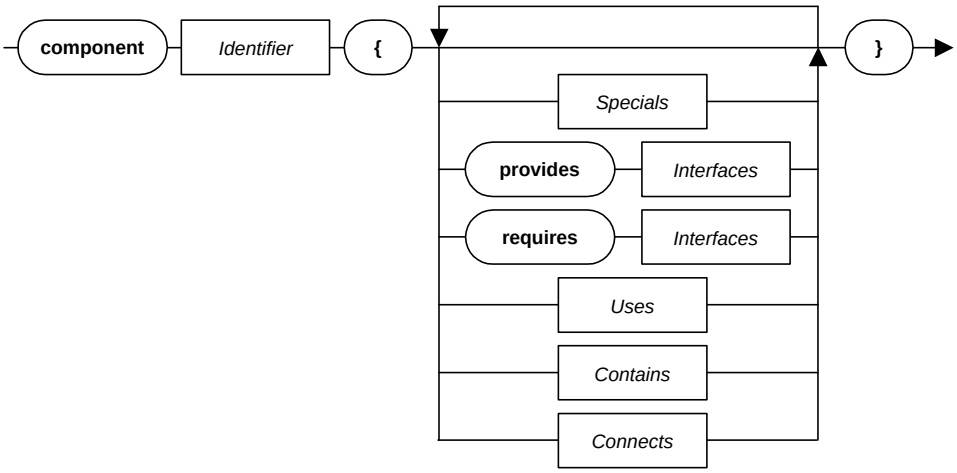

Figure 83. Component Definition

The following remarks apply:

- The identifier that follows the keyword **component** specifies the name of the component type.

---

[16] Note that the subtype relation is *implicit* in Koala. Parameter names are included in the subtype relation as an additional measure to prevent accidental subtyping.

[17] The Koala compiler requires that the expressions in the previous case be structurally equal.

- This name must be globally unique: it may not be equal to the name of any other component type, interface type or data type (group) in the repository.
- The **specials** clause declares some properties of the component (see section A.4.2).
- Interfaces declared after the keyword **provides** are called *provides interfaces*. They provide functionality to the environment of the component.
- Interfaces declared after the keyword **requires** are called *requires interfaces*. Through these the component requires functionality of its environment.
- The **uses** clause makes additional data types (and groups) accessible to all modules of the component (see section A.4.4).
- The **contains** clause declares the modules, subcomponents and internal interfaces of the component (see section A.4.5).
- The **connects** clause specifies the connections between interfaces and modules (see section A.4.8).
- Although the language allows the clauses mentioned above to be specified in any order, the order as specified in Figure 83 is recommended.

A.4.2 Specials

The *Specials* clause (see Figure 84) defines properties of the component.

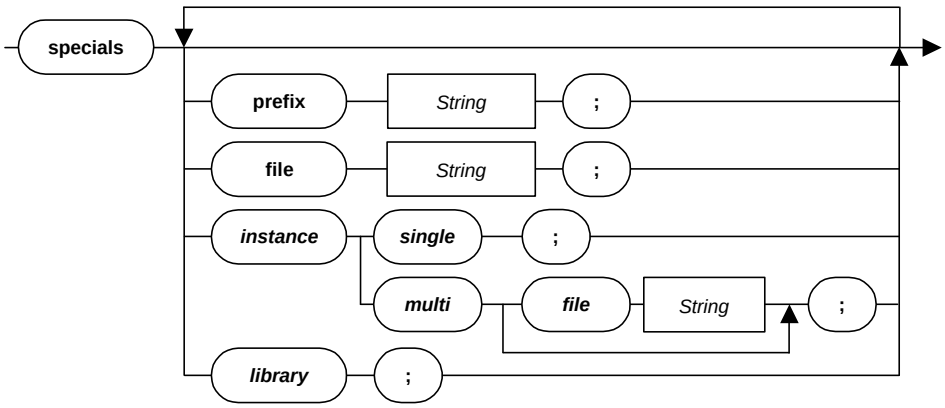

Figure 84. Specials

The following rules apply:

- The string that follows **prefix** specifies the *component prefix* (also known as the component *short name*) as used in the code generation.
- The component prefix must start with a letter and consist of a sequence of letters and/or digits (i.e., it may not be empty and should not contain '_').

- The component prefix must be globally unique: it may not clash with the prefix of any other component in the repository.
- If not specified explicitly, the prefix is assumed to be equal to the component type name (also known as the component *long name*).
- The string following **file** names the C file generated for the component if applicable.
- This string may not be empty, and it must include the extension (‘.c’).
- If not specified explicitly, the component prefix preceded by ‘_’ and followed by ‘.c’ is used as name for the output file.
- The **instance** clause specifies whether the implementation of the component is prepared for multiple instantiation (MI). The default is single instantiation.
- For an MI component, the string following **multi file** specifies the name of the file where Koala generates the instance data.
- This file name may not be empty, and must include the ‘.c’ file type.
- The Koala compiler no longer supports multiple instantiation. The instance clause is documented here for historic reasons only.
- The **library** clause specifies that a library is to be generated for this component.
- The Koala compiler no longer supports the library clause. The library clause is documented here for historic reasons only.
- The **prefix**, **file**, **instance** and **library** clauses may appear in any order, but each clause may appear only once.

A.4.3 Interfaces

A list of *Interfaces* (see Figure 85) declares provided, required or internal interface instances.

The following rules apply:

- The first identifier specifies the *type* of the interface.
- This interface type must be defined in the repository.
- The second identifier names the *instance* of the interface.
- The instance name must be unique within the component definition: it may not clash with the name of any other interface instance, subcomponent or module.

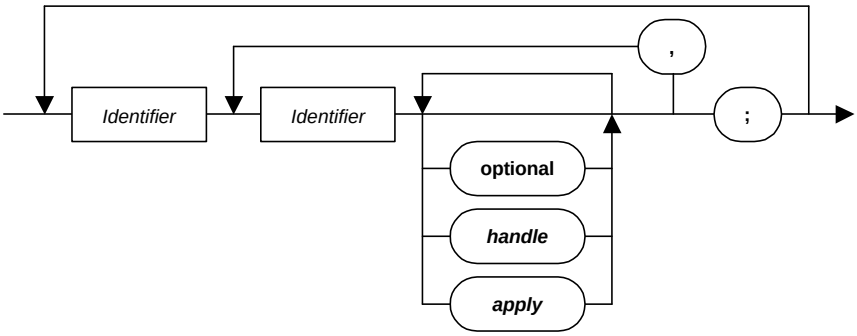

Figure 85. Interfaces

- An interface instance can be declared to be **optional**. If an interface is not optional, it is said to be mandatory (which is the default).
- An interface instance can be declared to be an **apply** interface, and/or it can be declared to have a **handle**. Both features are currently not supported by the Koala compiler.
- **optional**, **handle** and **apply** only relate to the interface of which the instance name directly precedes the specifiers.
- **optional**, **handle** and **apply** may occur only once, but can occur in any order.
- **handle** and **apply** are obsolete.

A.4.4 Uses

The *Uses* clause (see Figure 86) provides all modules in the component definition with access to the data types and data type groups specified.

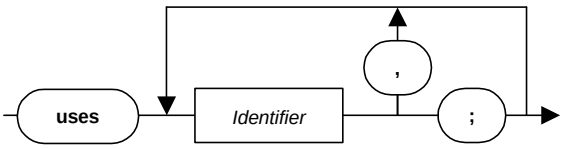

Figure 86. Uses

The following rules apply:

- Each identifier must refer to a data type or data type group that is defined in the repository.
- Note that modules have automatic access to all data types occurring in interfaces that they implement or use.

A.4.5 Contains

The *Contains* section (see Figure 87) specifies the 'parts-list' of the component, i.e. it lists the modules, the subcomponents and internal interfaces of the component.

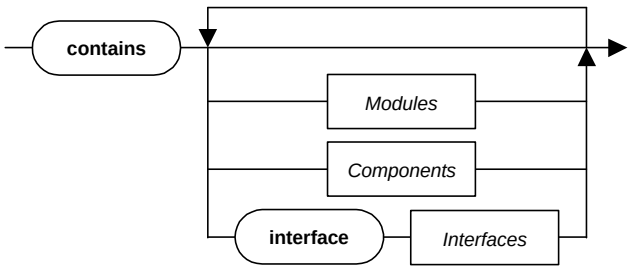

Figure 87. Contains

The following remarks apply:

- Modules contained in a component definition are the sole property of the component; reuse of the module outside of the component is not permitted.
- Components contained in a component definition are encapsulated instances of reusable component types. The instance can only be accessed through the component, but the reusable component type can have other instances in other components. However, a component can only be instantiated once in a single product, unless the component is capable of multi instancing.
- Interfaces declared after the keyword **interface** are called *private* or *internal interfaces.*

A.4.6 Modules

A *Modules* list (see Figure 88) declares 'glue' (also known as 'code') modules.

The following rules apply:

- The identifier that follows the keyword **module** defines the name of the module.
- This name must be unique within the component definition: it may not clash with the name of any other module, subcomponent or interface instance.
- The string following **file** specifies the header file to be generated for this module.
- This string may not be empty, and should include the file extension ('.h').
- The default file name is an underscore, followed by the prefix of the component, an underscore, the name of the module and '.h' (e.g. '_c_m.h').

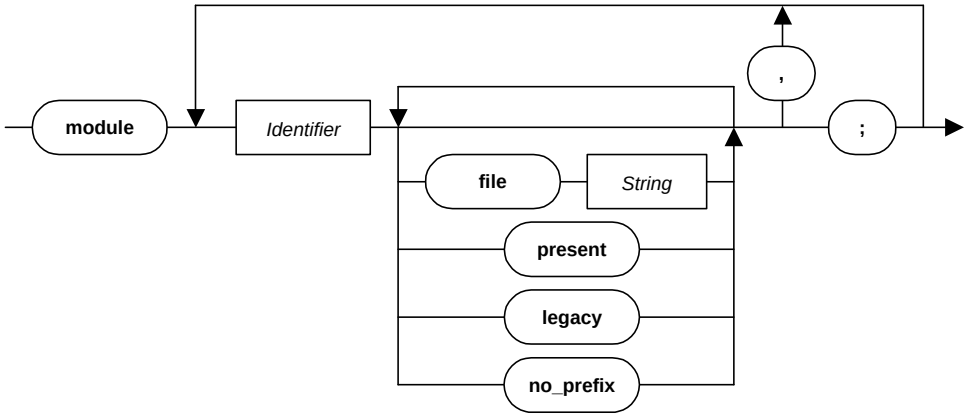

Figure 88. Modules

- The keyword **present** specifies that this module *must* be included when compiling a configuration that contains this component. By default, modules are only included when they are *reachable*.[18]
- The keyword **legacy** overrules the naming conventions in the generated header file, as explained in section A.7.
- The keyword **no_prefix** has the same effect as **legacy**.
- The keywords **file**, **present** and **legacy/no_prefix** only relate to the module of which the instance name immediately precedes the keywords.
- The keywords **file**, **present** and **legacy/no_prefix** can be specified in any order, but each may occur at most once.

A.4.7 Components

A *Components* list (see Figure 89) declares subcomponents of a component.

The following rules apply:

- The first identifier specifies the *type* of the subcomponent.
- A component type with this name must be defined in the repository.
- The second identifier names the subcomponent *instance*.
- The instance name must be unique within the component definition: it may not clash with the name of any other subcomponent, interface, or module.

[18] Reachability is not discussed in this document.

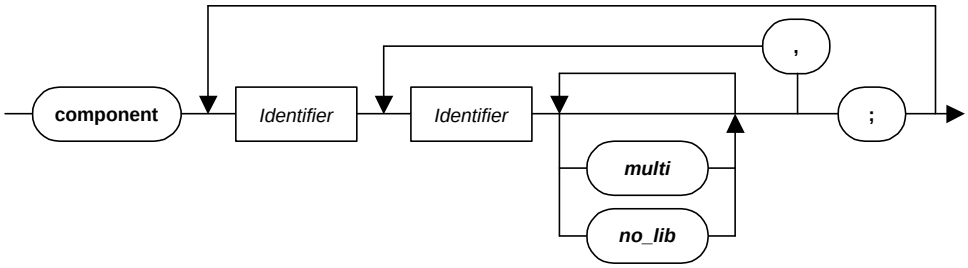

Figure 89. Components

- A preferred choice for the component instance name is the short name of the corresponding component definition.
- The keyword **multi** specifies that this subcomponent can be instantiated dynamically. Dynamic instantiation has never been implemented in Koala.
- If any of the subcomponents supports only single instantiation, then the component itself may support only single instantiation.
- The Koala compiler no longer supports multiple instantiation. The **multi** clause is documented here for historic reasons only.
- The keyword **no_lib** specifies that this subcomponent should not be treated as library, and thus overrules a possible **library** keyword in the definition of that subcomponent.
- The Koala compiler no longer supports libraries. The **library** clause is documented here for historic reasons only.
- The keywords **multi** and **no_lib** relate only to the subcomponent of which the instance name immediately precedes the keyword(s).
- The keywords **multi** and **no_lib** can be specified in any order, but each keyword may occur at most once.

A.4.8 Connects

A *Connects* section (see Figure 90) specifies the ‘net-list’ of the component.

See sections A.4.9 for cable, A.4.11 for switch, and A.4.13 for within clauses.

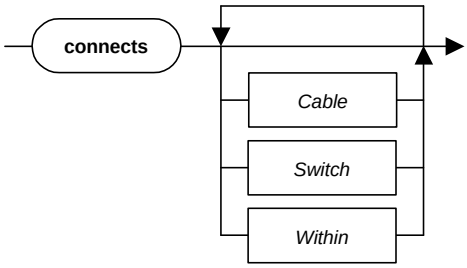

Figure 90. Connects

A.4.9 Cable

A *Cable* (see Figure 91) connects an interface to an interface, an interface to a module, or a module to an interface.

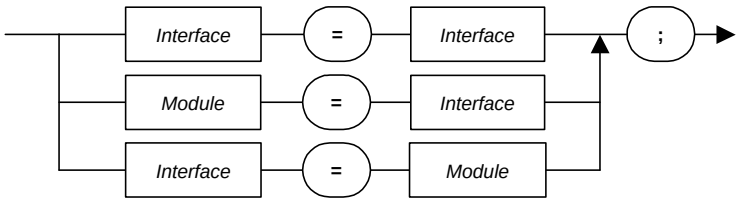

Figure 91. Cable

The following rules apply:

- *Interface* must refer to a valid interface instance in the component definition (see section A.4.10).
- **Definition:** the left-hand side interface (if present) is bound at its *tip* and is called the *tip interface* of this cable. The right-hand side interface (if present) is bound at its *base*, and is called the *base interface* of this cable.
- An interface instance can only be bound once at its tip, but can be bound zero or more times at its base.
- Provides interfaces of the component and requires interfaces of the subcomponents can only be bound at the tip in this component definition, requires interfaces of the component and provides interfaces of the subcomponents only at the base. Internal interfaces can be bound at both sides.
- If an interface is connected to an interface, then the type of the tip interface must be a subset of the type of the base interface (as defined in section A.3).

- A mandatory tip interface may only be connected to an optional base interface if Koala can determine at compile-time that the optional interface is present.
- If a module is connected to a base interface, then the module gets access to any function or data type (group) defined in that interface.
- If a module is connected to an *optional* base interface, then the module must check the *iPresent* function of that interface before using any function or parameter of the interface.
- Interface constants of optional interfaces can be used without check.
- If a mandatory interface is connected with its tip to a module, then this module must implement every function and every parameter of the interface (in C or in a Koala within clause, see section A.4.13).
- If an optional interface is connected with its tip to a module, then the module must implement the *iPresent* function of that interface (in C or in Koala). It only needs to implement the other functions of the interface if *iPresent* is not *false*.

A.4.10 Interface

An *Interface* (see Figure 92) refers to an interface instance of a component or of a subcomponent.

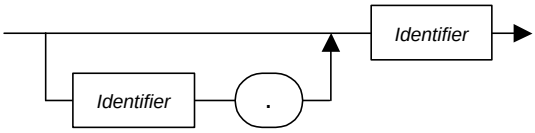

Figure 92. Interface

The following rules apply:

- If present, the first (optional) identifier names the subcomponent that provides or requires the interface.
- This subcomponent must be defined in the component (see section A.4.7).
- The second identifier names the actual interface instance.
- This interface must be defined on the subcomponent if present (as provides or requires interface) or in the component (as provides, requires or private interface).

A.4.11 Switch

A *Switch* (see Figure 93) allows conditional binding between sets of interfaces.

The following rules apply:

- The expression that follows the keyword **switch** may only refer to interfaces that are in scope: i.e. it may only use elements of provides, requires or private interfaces of the component, or of provides or requires interfaces of any of the subcomponents.

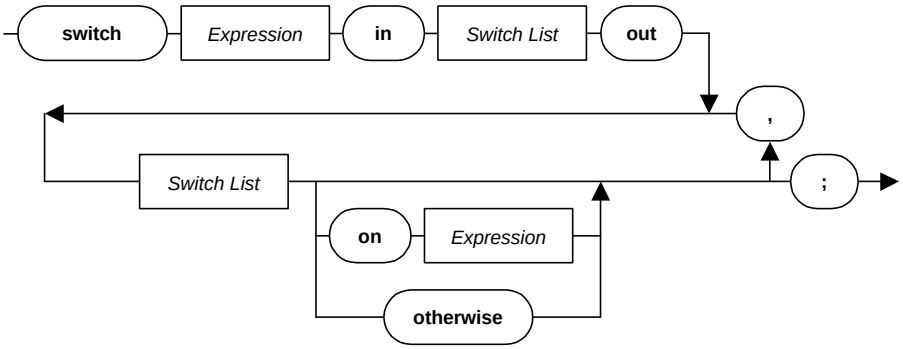

Figure 93. Switch

- The switch expression should evaluate to a numeric value. Note that Boolean expressions evaluate to 0 (**false**) or 1 (**true**) in Koala (see section A.4.16).
- The switch expression may not be a comma expression (see section A.4.18), i.e. have a comma as top level operator.
- All switch lists in a switch expression should be of equal length.
- Each interface of the **in** clause should match the corresponding (in position) interface of each **out** clause as if they were connected by a cable (see section A.4.9), unless Koala can decide at compile time that the connection will not be made.
- The switch connects the tip of each interface of the **in** clause to the base of each interface in the corresponding position of the selected switch list in the **out** clause.
- An **out** switch list is selected if its value corresponds to the value of the switch expression, or if it is specified as **otherwise** and no other **out** list has a value that matches the expression.
- The **otherwise** clause may only be attached to the last switch list in the **out** clause.
- An **out** switch list that has no **on** clause and no **otherwise** clause is called an implicit list. The first implicit list is assigned the value 0. Subsequent implicit lists are assigned the value of the previous implicit list plus 1. Note that explicit values (of **on** clauses) are not taken into account!

A.4.12 Switch List

A *Switch List* (see Figure 94) is a list of references to interface instances.

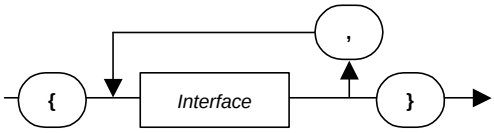

Figure 94. Switch List

The following rules apply:

- Each *Interface* must refer to a valid interface instance (see A.4.10).

A.4.13 Within

A *Within* clause (see Figure 95) allows to implement functions in Koala.

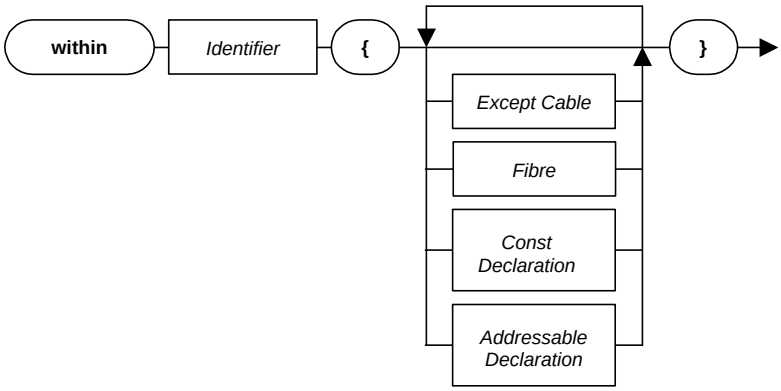

Figure 95. Within

The following rules apply:

- The identifier that follows the keyword **within** must refer to a module declared in the component as described in section A.4.6.
- The module declaration must precede the within clause for that module.[19]
- For each module, there must be at most one *within* clause.
- Modules for which no within clause exist are assumed to be completely implemented in C.

[19] The current Koala compiler does not enforce this rule, but future versions may.

- Modules for which a within clause exists may have any number of functions or parameters defined in Koala. Functions or parameters not defined in the within clause are assumed to be implemented in C.

A.4.14 Except Cable

An *Except Cable* (see Figure 96) implements an interface in terms of another interface with the exception of a specified set of functions or parameters.

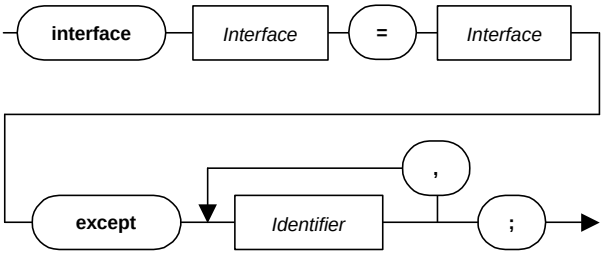

Figure 96. Except Cable

The following rules apply:

- An *Except Cable* is a *Cable* so all rules that apply to cables also apply here.
- The left hand side *Interface* must be bound with the tip to the module.
- The right hand side *Interface* must be bound with the base to the module.
- At most one *Except Cable* may be defined for a particular interface bound with the tip to the module.
- The *Identifiers* specify the functions that are excluded from this binding. These functions must be implemented separately in the within clause or in C.

A.4.15 Fibre

A *Fibre* (see Figure 97) allows to bind a function to a Koala expression.

The following rules apply:

- The *Interface* must be bound with its tip to the module.
- The *Identifier* refers to an element of the interface. This element must exist in the corresponding interface definition.
- An interface element can only be implemented once. If it is not implemented in a within clause, it must be implemented in C.
- For an interface parameter, the parameter clause (shaded in Figure 97) must be absent.

- For a function, the parameter clause must have the same length as the parameter list of the function in the corresponding interface type.

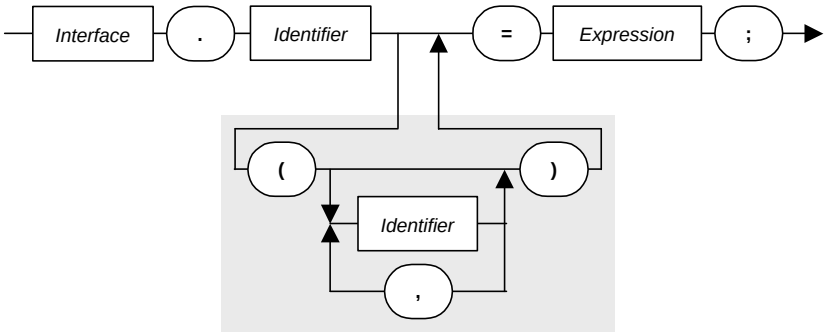

Figure 97. Fibre

- For a function, the parameter clause may also be absent if the expression consists of a reference to a single function with the same parameter list but specified without parameters (a so called *direct function binding*).
- The parameters in the parameter list must have distinct names. However, these names need not be the same as the parameter names in the interface definition.
- The expression may contain references to interfaces if they are connected to the module (both tip and base are allowed).
- The expression may contain identifiers if they refer to the parameters in the parameter clause.

A.4.16 Const Declaration

The *Const Declaration* clause (see Figure 98) allows to specify that certain functions or parameters of interfaces must be constant, i.e. Koala must be able to calculate the value.

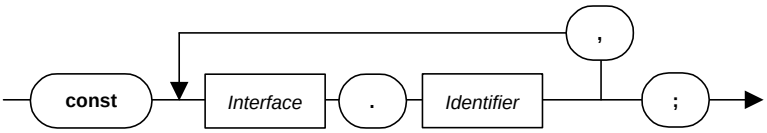

Figure 98. Const Declaration

The following rules apply:

- The *identifier* refers to a function or parameter of the interface. This function or parameter must exist in the corresponding interface definition.

A.4.17 Addressable Declaration

The *Addressable Declaration* clause (see Figure 99) allows to specify that certain functions of an interface must be addressable, i.e. it must be possible in the module to take the address of the function.

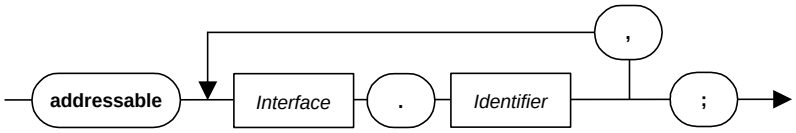

Figure 99. Addressable Declaration

The following remarks apply:

- The *Interface* must be bound with its base to the module.
- The *identifier* refers to a function or parameter of the interface. This function or parameter must exist in the corresponding interface definition.
- Koala makes sure that the interface element is indeed *addressable*. If necessary, Koala generates a function for this purpose.

A.4.18 Expression

An *Expression* (see Figure 100) calculates a value given an operator and one or more operand expressions.

The following rules apply:

- All operators have the same meaning as in the C language.
- All operators have the same priority as in the C language. Figure 100 can be read to show these priorities from low (top) to high (bottom). Operators that are grouped together in Figure 100 share the same priority.
- All operators have the same associativity as in the C language. For most of the operators, $\alpha \circ \beta \circ \gamma$ is parsed as $\alpha \circ (\beta \circ \gamma)$. Use parentheses when in doubt!
- As in C, Boolean expressions (such as '!=' and '<') result in 0 for **false** and 1 for **true.**
- The *const* operator is not the same as const in C. It returns **true** if Koala can determine the value of the operand expression at compile-time, and **false** otherwise.
- Although the Koala compiler recognizes 'U' and 'L' suffices on numbers, *all* arithmetic is currently performed using signed integers of 32 bits.

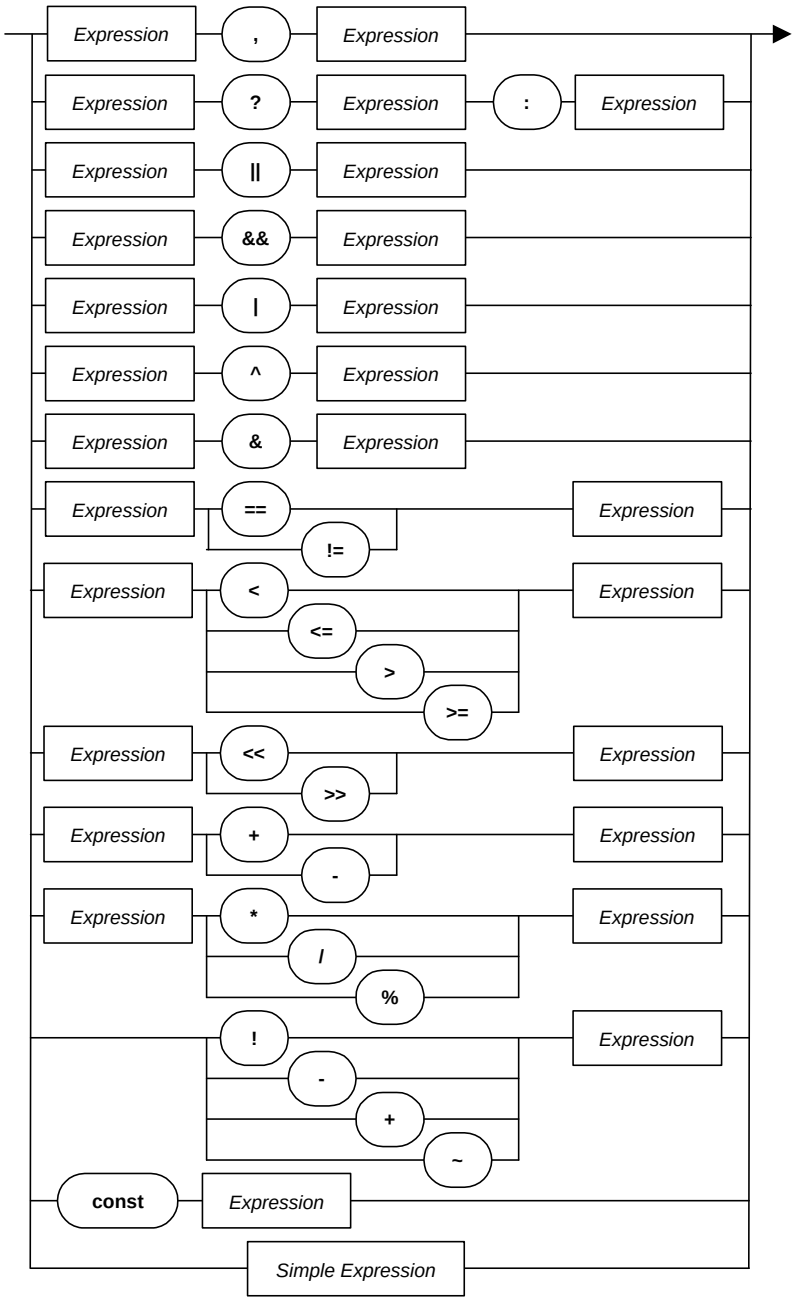

Figure 100. Expression

- **Definition:** An expression that contains no inline expression is called *native*. An expression that does contain an inline expression is called *mixed* or *non-native*.

A.4.19 Simple Expression

The syntax for a *Simple Expression* is shown in Figure 101.

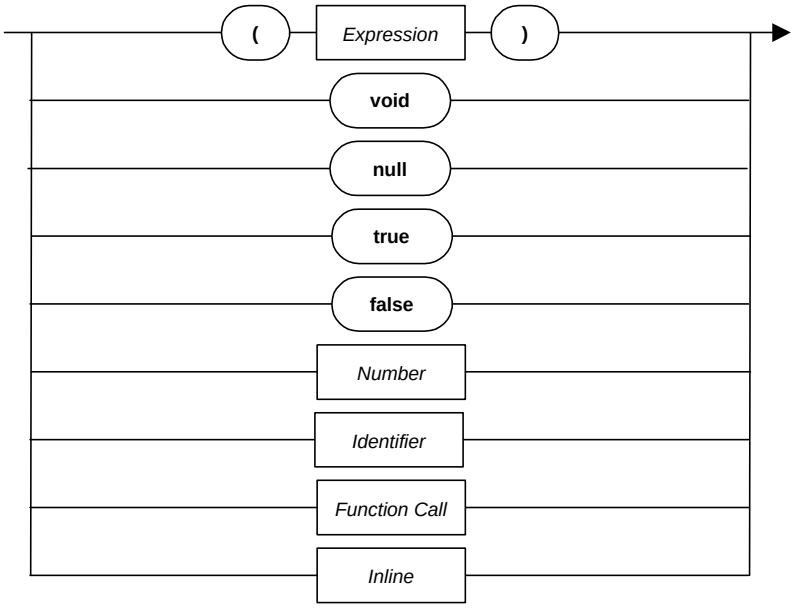

Figure 101. Simple Expression

The following rules apply:

- Parentheses can be used to change the order of evaluation.
- The keyword **void** denotes an empty expression. It can be used as body of void functions.
- The keyword **null** denotes an empty expression. It can be used as body of functions returning a (non-void) result.
- The value of **true** is 1. The value of **false** is 0.
- A *number* stands for itself.
- An *identifier* can only be used if the expression occurs in the right hand side of a function binding (see section A.4.15) or in the expression assigned to an function in an interface definition (see section A.3.3). The identifier must then refer to a formal parameter of the function.

A.4.20 Function Call

A *Function Call* (see Figure 102) invokes an element of an interface.

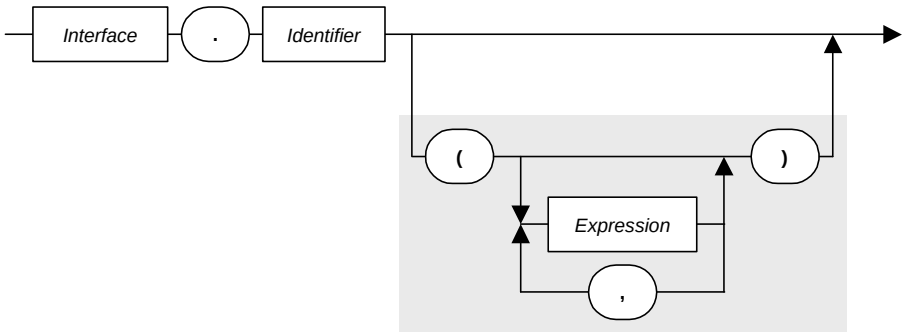

Figure 102. Function Call

Remarks:

- *Interface* specifies the interface instance or definition of which the element is called.
- If the function call is part of an expression in an *interface* definition, then it can only refer to members of the same interface, and the interface must consist of the name of the interface definition only.
- If the function call is part of an expression in a *component* definition, then the interface must be valid as defined in section A.4.10, and it must be in scope as defined below.
- If the expression is part of a *within* clause (section A.4.13), the interface must be connected to the corresponding module (see section A.4.9) with the tip or with the base.
- If the expression is part of a *switch* (section A.4.11), then any interface of the component or of its subcomponents can be used.
- The *Identifier* specifies the element of the interface to be invoked.
- This element must be defined in the corresponding interface type.
- A parameter list (shaded in Figure 102) is not allowed for the invocation of an interface parameter or interface constant.
- A parameter list is mandatory for the invocation of a function in an interface.
- In that case, there must be one expression for each formal parameter.
- In a direct function binding (see section A.4.15), the parameter list must be omitted for a function ‘call’.

A.4.21 Inline Expression

An *inline* expression (see Figure 103) allows to embed literal C texts in Koala.

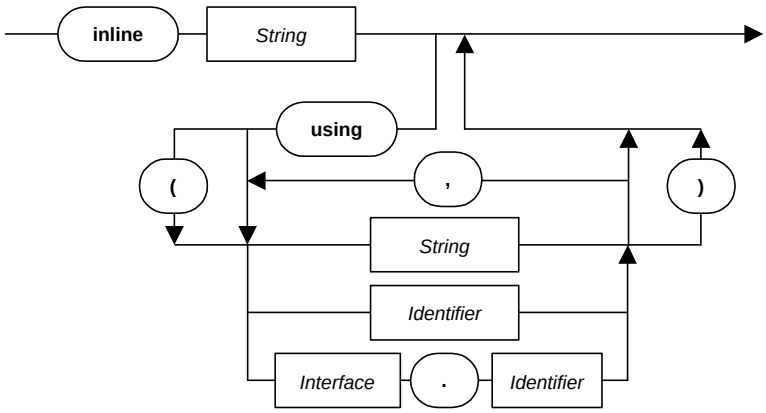

Figure 103. Inline Expression

The following rules apply:

- The string following the keyword **inline** is the piece of code that is inserted when the function or parameter is invoked. Any context that this string needs should be specified in the **using** clause.
- The **using** clause contains a list of elements that provide context to the string mentioned above.
- Parentheses in the using clause are optional but must either both be absent or both be present.
- A *string* in a using clause is literally included in generated header files. It typically contains the declaration of an external variable or function.
- An *identifier* in a using clause specifies a data type or data type group that is required to parse and compile the inline string. This data type (group) must be defined in the repository.
- The *interface* may only refer to an interface that is connected to the module if the inline expression is part of a fiber (section A.4.15).
- An inline expression in a switch (section A.4.11) may refer to any interface of the component or any provides or requires interface of its subcomponents.
- An inline expression in an interface definition (section A.3.3) may only refer to that interface definition itself, using the interface type name.
- The *identifier* following an interface names the interface element being used in the inline string. Koala will make this element accessible.

A.5 The Data Type Definition Language

Koala does not contain a specific data type definition language. Instead, data types are specified in C header files in the traditional way. To inform Koala which data types are defined in which header files, three pragmas are defined.

A.5.1 Koala Type

A *Koala Type* clause (see Figure 104) declares that the header file contains a C definition for a data type.

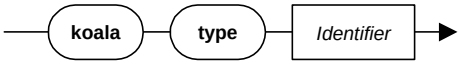

Figure 104. Koala Type

The following remarks apply:

- The *Identifier* specifies the name of the data type being defined in this file. This name can be used in **using** or **uses** clauses (see sections A.3.2, A.4.4, A.4.21 and A.5.3).
- This name must be globally unique: it may not be equal to the name of any other data type, data type group, component type or interface type in the repository.

A.5.2 Koala Group

A *Koala Group* clause (see Figure 105) declares that the header file defines a group of data types.

Figure 105. Koala Group

Remarks:

- The *Identifier* specifies the name for the data type group. This name can be used in **using** or **uses** clauses (see sections A.3.2, A.4.4, A.4.21 and A.5.3).
- This name may not be equal to the name of any other data type, data type group, component type or interface type in the repository.

A.5.3 Koala Using

A *Koala Using* clause (see Figure 106) specifies that the data type definitions use data types defined in other header files.

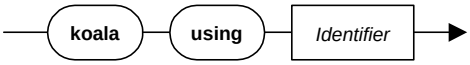

Figure 106. Koala Using

Remarks:

- The *Identifier* specifies the name of the data type or data type group being used in the header file.
- This identifier must refer to a data type or data type group defined somewhere in the repository.
- It is allowed to have multiple using clauses for the same data type or group in a file.

A.6 Naming Conventions

In this section, we describe naming conventions that are recommended (*should*, *may*) or required (*shall*, *must*).

A.6.1 Interface Definitions

The following conventions apply:

- An interface definition should have a name that is WordCased and prefixed with a capital 'I'. Example: *ITvSearchTuner*.
- An interface definition may specify (in comment) a prefix that can be used as preferred interface instance name. Example: *tun*.
- An interface definition must be stored in a file with extension '.id'.
- The name of this file should be equal the name of the interface definition.
- Storing multiple interface definitions in a single file is allowed though not preferred.
- Interface definitions can be stored anywhere in the repository.[20]

A.6.2 Component Definitions

The following conventions apply:

- A component definition should have a name that is WordCased and prefixed with a capital 'C'. Example: *CTvSearchTuner*.

[20] MG-R poses extra restrictions on the location of Koala definitions in the directory structure.

- A component definition should specify a prefix (section A.4.2) that is lower case, does not contain any '_' characters, starts with a letter and is preferably 5-7 characters long. Example: *tvstun*.
- A component definition must be stored in a file with extension '.cd'.
- The name of this file should be equal to the long name of the component.
- Storing multiple component definitions in a single file is allowed but strongly discouraged.
- A component definition must be stored in a directory. All source files (.c and .h) found in that directory are considered part of the component.
- The name of this directory should be equal to the component short name.
- If multiple component definitions are stored in the same directory (not recommended!), then if any of the components is present in the product, all files in that source directory will be compiled (but only once).

A.6.3 Data Type Definitions

The following conventions apply:

- Data type definitions are C definitions augmented with Koala pragmas (see section A.2).
- Data type definitions reside in normal header files, but these files should have extension '.dd' rather than '.h'.
- Data type definitions may be stored anywhere in the repository.

A.6.4 C and Header Files

The following conventions apply:

- Each C and header file should have a globally unique name, preferably one that starts with the component prefix followed by a '_' character.
- For each module that is not completely implemented in a within clause (see section A.4.13) there should be a C file with as name the component prefix followed by an '_' followed by the module name.[21]
- This C file must implement all functions of all interfaces connected with the tip to the module that have not been specified in a within clause.

[21] Actually, a module specifies a header file to be generated by Koala. This header file can be included in any C file of that component. But we strongly recommend to have one hand-written C file for each module.

- The C file must include the header file generated for the module. It may include other header files but only if they are located in the same directory.
- The C file may use any function, parameter or constant of any interface connected with the base or tip to the module.

A.6.5 Implementing Functions or Parameters

The following conventions must be used when implementing functions or parameters in C:

- A function named ‘f’ of a provides interface ‘p’ or private interface ‘i’ of the component, where ‘p’ or ‘i’ is connected with the tip to the module, must be implemented as a C function named ‘p_f’ respectively ‘i_f’.
- A function named ‘f’ of a requires interface ‘r’ of a sub-component with instance name ‘s’, where ‘r’ is connected with the tip to the module, must be implemented as a C function named ‘s_r_f’.
- A static function may have an arbitrary name.
- A function with external linkage that is not implementing an interface function should have a name that starts with the component prefix followed by a single ‘_’.
- A parameter can be implemented as a function, using the same naming conventions as described above, or (preferably) in a Koala inline expression.

A.6.6 Using Functions, Parameters, Constants and Types

The following conventions apply when using functions, parameters or constants of interfaces connected to a module:

- A function, parameter or constant named ‘f’ of a provides, requires or internal interface named ‘i’ of the component containing the module, where ‘i’ is connected with the base or tip to the module, may be invoked as ‘i_f’.
- A function, parameter or constant named ‘f’ of a provides or requires interface named ‘i’ of a sub-component with instance name ‘s’, where ‘i’ is connected with the base or tip to the module, may be used as ‘s_i_f’.
- Before calling any function or parameter in an optional interface, the caller should check whether the *iPresent* function of that interface yields **true**.
- Unless otherwise specified, once *iPresent* has returned true for a particular interface, it should continue to return true for that interface during the life time of the system.
- Constants of an optional interface can be used without checking *iPresent* first.

- Sometimes, a module assumes an interface element to be constant, e.g. when using it in an initialization expression of a static variable, as size of a static array, or in a case of a switch statement. If the element has not been declared constant in the interface definition, then this must be explicitly specified with a const declaration in the within clause for the module (see section A.4.16). Use of a const declaration in a within clause is not recommended!
- A module can inspect whether functions or parameters that have not been specified as constant still are constant in specific situations. The preprocessor macro <f>_CONSTANT is defined only if <f> evaluates to a constant.
- If the caller wants to take the address of a function of an interface, this must be explicitly specified with an addressable declaration (see section A.4.17) in the within clause for the module.
- A module can inspect whether an interface 'i' that is connected to it with the tip is actually connected at the outside. The preprocessor macro i_CONNECTED provides this information.[22]
- A module gets access to the definitions of all types mentioned in the definitions of all interfaces connected to the module, plus all types mentioned in the uses clause of the component that contains the module, plus all types mentioned in using clauses of inline expressions in function bindings for that module.
- A module can use the PREFIX macro to obtain the short name of the component.

A.7 Const-Free Semantics

In this section, we define the semantics of the Koala language by providing a simple non-optimizing code generation scheme (also known as Koala0). The following restrictions hold:

- We implement single instantiation only (see section A.4.2).
- We do not do any (partial) evaluation of Koala expressions. Every expression is turned directly into either preprocessor or C code.
- We ignore the keyword **const** in Koala (as introduced in sections A.3.3, A.4.16 and A.4.18), which means that we assume that interface elements other than interface constants are never used at locations where C assumes constants.

[22] Although in principle CONNECTED should be equivalent to *iPresent*, Koala currently uses a much simpler algorithm to establish whether interfaces are connected.

- We generate code for every component in the repository independently, i.e. without looking into other components (but with using the type information of provides and requires interfaces of subcomponents).
- At occasions, we may make more definitions visible than the language describes.

The resulting code for a component can be compiled independently from other components. An application can be built by linking the top-level component and all components recursively included.

A.7.1 Generated Files

For each component, a C file will be generated (see section A.4.2 for the name of this file) that contains:

- *#include* statements for all relevant data types (see section A.7.2).
- *#define* statements for all interface elements that have an expression assigned in the definitions of interfaces of the component or its subcomponents (see section A.7.5).
- Call-through functions for interface-interface bindings (see section A.7.6).
- Select functions for switches (see section A.7.7).

For each module, a header file will be generated (see section A.4.6) that contains:

- *#include* statements for all relevant data types (as explained in section A.7.2).
- *#define* statements for all interface elements with an expression assigned in the definitions of interfaces of the component or its subcomponents (see section A.7.5).
- Function definitions for all interface elements that are assigned an expression in a within clause (see section A.7.8).[23]
- *#define* statements and extern declarations for all other interface elements (see section A.7.9).
- A definition of *i_CONNECTED* for every interface ‘i’ connected with the tip to the module:

```
#define i_CONNECTED 1
```

[23] Note that the generated header file contains C code for functions implemented in within clauses! This is no problem as long as the header file is only included in one C file. An alternative is to generate the code in the component C file

A.7.2 Data Type Dependencies

The C file generated for the component includes those '.dd' files that contain data types or data type groups that occur:

- In interfaces of the component (provides, requires or internal) or of its subcomponents (provides or requires).
- In *uses* statements of the definitions of interfaces mentioned above.
- In *using* statements of expressions in the definitions of interfaces mentioned above.
- In *uses* statements of the component.
- In *using* statements of '.dd' files already included.

The module header file contains includes for those '.dd' files that contain data types or data type groups that occur:

- In interfaces of the component (provides, requires, internal) or of its subcomponents (provides, requires) that are bound to the module.
- In *uses* statements of the definitions of the interfaces mentioned above.
- In *using* statements of expressions in the definitions of the interfaces mentioned above.
- In *using* clauses of expressions in fibers in the within clause for the module.
- In *uses* statements of the component.
- In *using* statements of '.dd' files already included.

All includes are generated in the right order[24] and without any duplications[25].

A.7.3 Physical Names

A function 'f' of an interface instance 'i' of a component with prefix 'c' will become a C function with name 'c__i_f', regardless of the role of the interface (provides, requires, internal).

A parameter 'p' of an interface instance 'i' of a component with prefix 'c' will also become a C function with name 'c__i_p', regardless of the role of the interface. Note that 'p' is referred to *without* parentheses, both in C and in Koala!

---

[24] If A uses B then B is included before A.

[25] An easy way to avoid duplications is by surrounding each include with an *#ifdef/#endif* pair. Note that a simple (be it less restrictive) solution to generating the module header file is to copy the information written to the component C file (but using clauses of expressions in fibers must be added then).

Remarks:

- If a function or parameter is assigned an expression in the interface definition, then it will only have a logical name.
- Functions and parameters in provides and internal interfaces are defined in the context of the component that owns them. Functions and parameters of requires interfaces are defined in the context of the (compound) component that instantiates the component that owns them. Note that there can be more than one such component in the repository!

A.7.4 IPresent

For each mandatory interface 'i' (provides, requires or internal) in a component with prefix 'c', the following function is generated in the component C file[26]:

```
int c__i_iPresent(void)
{
    return 1;
}
```

For each optional provides or internal interface of a component or requires interface of a subcomponent that is *not* connected, the following function is generated in the component C file:

```
int c__i_iPresent(void)
{
    return 0;
}
```

Remarks:

- 'c' is the prefix of the component that owns interface 'i'.

In sections A.7.6, A.7.7, A.7.8 and A.7.9, optional interfaces are considered to have an extra function *iPresent*.

A.7.5 Interface Expressions

An interface *constant* 'c' of an interface 'i' of type 'I' is translated to a #define as follows:

```
#define i_c <expression>
#define i_c_CONSTANT <expression>[27]
```

Remarks:

---

[26] One could argue that *iPresent* for a mandatory requires interface of a component C should be implemented in the compound component that instantiates C.

[27] In the 'real' Koala compiler, the expression for 'i_c' may contain C type casts, while the expression for 'i_c_CONSTANT' may only have constructs that are known to the C preprocessor.

- The C <*expression*> is generated from the corresponding Koala expression using the scheme described in section A.7.10, with one exception:
- References to elements ‘x’ of the same interface are translated from ‘I.x’ to ‘i_x’, where ‘I’ is the type and ‘i’ the instance name of the interface.

An interface *function* ‘f’ of an interface ‘i’ of type ‘I’ is translated to a #define as follows:

```
#define i_f(<params>) <expression>
```

Remarks:

- The parameter list consists of parameter names only (no types).
- The expression is translated according to section A.7.10, with the same exception as mentioned above.
- There is no _CONSTANT macro being defined.

A.7.6 Interface-Interface Binding

For each function ‘f’ in the left hand side of an interface-interface binding, a C function will be generated in the component C file of the form:

```
<type> <c>__<lhs>_f ( <parameter declarations> )
{
    return <cc>_<rhs>_f( <parameters> );
}
```

Remarks:

- If the <*type*> of the function is void, then the *return* keyword is omitted.
- The *parameter declarations* may be empty.
- <*c*> is the prefix of the component that owns the interface <*lhs*>.
- <*cc*> is the prefix of the component that owns the interface <*rhs*>.

For each parameter ‘p’ in the left hand side of an interface-interface binding, a C function will be generated of the form:

```
<type> <c>__<lhs>_p ( void )
{
    return <cc>__<rhs>_p ( );
}
```

A.7.7 Switches

For each function ‘f’ in an interface ‘i’ that is input to a switch, a function will be generated in the C file of the component of the following form:

```
<type> <c>__i_f ( <parameter declarations> )
{
    int e = <switch expression>;
    if ( e == <value of first output choice>
        return <cc1>__<oc1>_f ( <parameters> );
    else ...
}
```

Remarks:

- *<c>* is the prefix of the component to which ‘i’ belongs. This can either be the compound component, or one of its subcomponents.
- The *parameter declarations* are copied from the interface definition.
- The *switch expression* is only evaluated once.
- For *void* functions, the *return* keyword is omitted.
- *<oc1>* is the interface to which this output choice is bound for the particular value of the expression.
- *<cc1>* is the component that owns *<oc1>*.
- The *parameters* contain the names of the formal parameters of the function.

For each parameter ‘p’ in an interface ‘i’ that is input to a switch, the generated code is similar:

```
<type> <c>__i_p(void)
{
    int e = <switch expression>;
    if ( e == <value of first output choice>
        return <cc1>__<oc1>_p ();
    else ...
}
```

A.7.8 Fibres

For a function ‘f’ of an interface ‘i’ that is implemented in a within clause in Koala, a function of the following form is generated:

```
#define i_f  c__i_f
<type> c__i_f(<param list>)
{
    return <expression>;
}
```

Remarks:

- ‘c’ is the prefix of the component that owns the interface ‘i’.
- *<type>* may be void, in which case the keyword *return* is omitted.
- The C <expression> is generated from the Koala expression following the translation scheme described in section A.7.10.

For a parameter ‘p’ of an interface ‘i’ that is implemented in a within clause in Koala, a function of the following form is generated:

```
#define i_p  c__i_p()
<type> c__i_p()
{
    return <expression>;
}
```

A.7.9 Interface-Module Binding

For each function ‘f’ of an interface ‘i’, where ‘i’ is connected to a module ‘m’ and ‘f’ is not assigned an expression in the interface definition of ‘i’ nor in a within clause for the module ‘m’, the following code is generated in the header file of module ‘m’:

```
#define i_f  c__i_f
extern <type> c__i_f(< parameter declarations>);
```

Remarks:

- ‘c’ is the prefix of the component that owns ‘i’.
- The function may have a void type.
- The parameter declarations are copied from the interface definition.
- If the interface ‘i’ is optional, and *iPresent* is not implemented in a within clause, then the code above is generated with *f = iPresent*.

For each parameter ‘p’ satisfying the same conditions, the following code is generated:

```
#define i_p  c__i_p()
extern <type> c__i_p(void);
```

If the interface ‘i’ is owned by a subcomponent with instance name ‘s’ and prefix ‘cc’, then the following code is generated:

```
#define s_i_f  cc__i_f
extern <type> cc__i_f(< parameter declarations>);
```

Similarly for a parameter:

```
#define s_i_p  cc__i_p()
extern <type> cc__i_p(void);
```

A.7.10 Expressions

Since the Koala expression language is derived from the C expression language, translation will be straightforward. We only discuss the non-trivial part here.

- *const* <*expression*> will be translated to *0* (for *false*).
- A *void* expression will be translated to *while (0) { }*.
- An *inline* expression will appear as is. All string constants in the *using* clause will be inserted in the generated code before the definition in which

the current expression appears. Function references are already declared in the context. Data type and data type group references of the using clause have been dealt with in section A.7.2.

- References to (other) elements of (other) interfaces will be converted from dot notation (i.f) to underscore notation (i_f).

A.8 Concluding Remarks

Reachability is part of the optimization facilities of the Koala compiler, and is not discussed in this document.

List of References

[1] Aesop, a Software Architecture Design Environment Generator, http://www.cs.cmu.edu/afs/cs/project/able/www/aesop.

[2] Pierre America, Jürgen Müller, Henk Obbink, Rob van Ommering, *COPA (Component Oriented Platform Architecting)*, http://www.extra.research.philips.com/SAE/COPA/, 2000.

[3] ARES, *Architectural Reasoning for Embedded Software*, ESPRIT Project 20477, http://www.cordis.lu/esprit/src/20477.htm, 1995-1999.

[4] Timo Asikainen, Timo Soininen, Tomi Männistö, *A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families*, Fifth International Workshop on Product Family Engineering, Sienna, Italy, November 4-6, 2003, LNCS 3014, Springer, ISBN 3-540-21941-2, p225-249, 2004.

[5] Felix Bachmann, Len Bass, *Introduction to the Architecture Based Design Method*, Tutorial at the First Software Product Line Conference (SPLC1), Denver, Colorado, USA, August 28-31, 2000.

[6] Robert Balzer, *An Architectural Infrastructure for Product Families*, Proceedings of the Second International ESPRIT ARES Workshop, LNCS 1429, Springer Verlag, Berlin Heidelberg, p158-160, 1998.

[7] Victor Basili, *The Role of Experimentation: Past, Present and Future*, ICSE-18 keynote slides, Berlin, March 27-29, 1996.

[8] Don Batory, Sean O'Malley, *The Design and Implementation of Hierarchical Software Systems with Reusable Components*, ACM Transactions on Software Engineering and Methodology, 1 no. 4, p355-398, October 1992.

[9] Joe Bauman, *The Perfect Architecture is non-optimal, Winning with Chaos*, Proceedings of the 4th international workshop on Product Family Engineering, Bilbao, Spain, October 2001, LNCS 2290, Springer Verlag, Heidelberg, p248-257, 2002.

[10] Marcello M. Bonsangue, Joost N. Kok, Maarten Boasson, Edwin de Jong, *A software architecture for distributed control systems and its transition system semantics*, Proceedings of SAC'98, the 1998 ACM Symposium on Applied Computing, Atlanta, Georgia, USA, February 27 - March 1, ISBN 0-89791-969-6, p159-168, 1998.

[11] Jan Bosch, *Organizing for Software Product Lines*, Proceedings of the 3rd international workshop on the development and evolution of software architectures of product families, Las Palmas, March 2000, LNCS 1951, Springer Verlag, Heidelberg, ISBN 3-540-41480-0, p117-134, 2000.

[12] Jan Bosch, *Design & Use of Software Architectures, Adopting and evolving a product-line approach*, ACM Press Books, Addison-Wesley, ISBN 0-201-67494-7, 2000.

[13] Remi Bourgonjon, *The Evolution of Embedded Software in Consumer Products*, International Conference on Engineering of Complex Computer Systems, (unpublished keynote address), Ft. Lauderdale, FL, 1995.

[14] Reinder J. Bril, André Postma, *A new architectural metric and its visualisation to support incremental re-architecting of large legacy systems*, 4th International Workshop on Software Architecture, Limerick, Ireland, June 4-5, 2000.

[15] Reinder Bril, René Krikhaar, and André Postma, *Embedding architectural support in industry*, International Conference on Software Maintenance, IEEE Computer Society, p348-357, 2003.

[16] Klaas Brink, *Interfacing Control and Software Engineering: a formal approach*, PhD thesis, Technical University, Delft, The Netherlands, ISBN 90-407-1456-8, June 24, 1997.

[17] Kraig Brockschmidt, *Inside OLE Second Edition*, Microsoft Press, ISBN 1-55615-843-2, 1995.

[18] Frederick P. Brooks Jr, *The Mythical Man-Month, Essays on Software Engineering*, Addison-Wesley Publishing Company, ISBN 0-201-00650-2, 1975.

[19] T. J. Brown, I. Spence, P. Kilpatrick, *A Relational Architecture Description Language for Software Families*, Fifth International Workshop on Product Family Engineering, Sienna, Italy, November 4-6, 2003, LNCS 3014, Springer, ISBN 3-540-21941-2, p282-295, 2004.

[20] E. Chikofsky and J. Cross. *Reverse Engineering and Design Recovery: A taxonomy*. IEEE Software, p13-17, January 1990.

[21] Paul Clements, Linda Northrop, *Software Product Lines, Practices and Patterns*, Addison-Wesley, ISBN 0-201-70332-7, 2002.

[22] Ivica Crnkovic, Magnus Larsson, *Building Reliable Component-Based Systems*, Artech House Publishers, ISBN 1-58053-327-2, 2002.

[23] Krzysztof Czarnecki, Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley Pub Co, ISBN 0-201-30977-7, 2000.

[24] Patrick Donohoe (Ed), *Proceedings of the First Software Product Line Conference (SPLC1)*, Denver, August 2000, The Kluwer International Series in Engineering and Computer Science, Volume 576, 2000.

[25] Mohamed Fayad and Doug Schmidt, *Object-Oriented Application Frameworks*, Communications of the ACM, 40 no 10, p32-85, October 1997.

[26] Loe Feijs, Rob van Ommering, *Architecture Visualisation and Analysis: Motivation and Example*, International Workshop on Development and Evolution of Software Architectures for Product Families Madrid, Spain, Nov 18-19, 1996.

[27] L. Feijs, R. Krikhaar, R. van Ommering. *A relational approach to Software Architecture Analysis*. Software Practice and Experience, 28(4), p371-400, April 1998.

[28] Loe M. G. Feijs, Roel de Jong, *3D Visualization of Software Architectures*, Communications of the ACM (CACM), Volume 41, Number 12, p72-78, December 1998.

[29] L.M.G. Feijs, R.C. van Ommering. *Relation Partition Algebra - mathematical aspects of uses and part-of relations* -. Science of Computer Programming, 33, p163-212, 1999.

[30] Alexandre Fioukov, Evgeni Eskenazi, Dieter Hammer,Michel Chaudron, *Evaluation of Static Properties for Component-Based Architectures*, Component-based Software Engineering Track, Euromicro Conference 2002, p33-39, 2002.

[31] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN 0-201-63361-2, 1995.

[32] David Garlan and Dewayne Perry, *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering, 21 (4), p269-274, April 1995.

[33] David Garlan, Robert Allen, John Ockerbloom, *Architectural Mismatch, or: Why it's hard to build systems out of existing parts*, ICSE 95, Seattle, Washington USA, p179-185, 1995.

[34] Gelernter, D., and Carriero, N., *Coordination Languages and Their Significance*, CACM, 32(2), February, p97-107, 1992.

[35] Richard C. Holt, *Structural Manipulations of Software Architecture using Tarski Relational Algebra*, Proceedings of fifth Working Conference of Reverse Engineering, WCRE'98, IEEE Computer Society, p210-219, 1998.

[36] H. James Hoover, Tony Olekshy, Garry Froehlich and Paul Sorenson, *Developing Engineered Product Support Application*, Proceedings of the 1st Software Product Line Conference, Denver, Colorado, USA, August 28-31, Kluwer, p451-476, 2000.

[37] Ad Huiser, *When all pieces fit…*, Keynote speech at the Philips (internal) Software Conference, Veldhoven, The Netherlands, Feb 8, 2001.

[38] International Standard IEC 61131, Programmable controllers, 1992.

[39] IEEE *Recommended Practice for Architectural Description of Software Incentive Systems*, IEEE Standard 1471-2000, ISBN 0-7381-2519-9, 2000.

[40] IEEE Transactions on Software Engineering, *Special Issue on Software Architecture*, Vol. 21 Issue 4, April 1995.

[41] Ivar Jacobson, Martin Griss, Patrick Jonsson, *Software Reuse – Architecture, Process and Organization for Business Success*, Addison Wesley, New York, ISBN 0-201-92476-5, 1997.

[42] Java, *The source for Java Technology*, http://java.sun.com/.

[43] Hans Jonkers, *An Overview of the SPRINT Method*, Springer-Verlag, Lecture Notes in Computer Science LNCS 670, p403-427, 1993.

[44] Rick Kazman and S. Jeromy Carriere. *View Extraction and View Fusion in Architectural Understanding*. Proceedings of the Fifth International Conference on Software Reuse, ISBN 0-8186-8377-5, p290, 1998.

[45] Mark H. Klein et al., *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publishers, ISBN 0-7923-9361-9, 1993.

[46] René Krikhaar. *Software Architecture Reconstruction*. ISBN90-74445-44-6. Ph.D. Thesis University of Amsterdam, the Netherlands, 1999.

[47] Kruchten, P. *The 4+1 View Model of Architecture*, IEEE Software, Vol. 12 No. 6, p42-50, November 1995.

[48] Piërre van der Laar, *KoalaBEAR: model, visualize, and verify software architectures*, Philips internal (DoVo) presentation, Sep 11, 2003.

[49] Frank van der Linden (ed), *Development and Evolution of Software Architectures for Product Families*, Second International ARES Workshop, Las Palmas de Gran Canaria, Spain, Springer-Verlag, LNCS 1429, ISBN 3-540-64916-6, February 1998.

[50] Frank van der Linden, Jan Gerben Wijnstra, *Platform Engineering for the Medical Domain*, Proceedings of the 4th international workshop on Product Family Engineering, Bilbao, Spain, October 2001, ISBN 3-540-43659-6, p224-237, 2002.

[51] Jeff Magee, Naranker Dulay, Jeff Kramer, *Regis: A Constructive Development Environment for Distributed Programs*, Distributed Systems Engineering Journal, Vol 1 (5), Special Issue on Configurable Distributed Systems, p304-312, 1994.

[52] Jeff Magee, Naranker Dulay, Susan Eisenbach, Jeff Kramer, *Specifying Distributed Software Architectures*, Proc. ESEC'95, Wilhelm Schafer, Pere Botella (Eds.) Springer LNCS 989, ISBN 3-540-60406-5, p137-153, 1995.

[53] Jeff Magee, Jeff Kramer, *Concurrency – State Models and Java Programs*, John Wiley & Sons, ISBN 0-471-98710-7, March 1999.

[54] M.D. McIlroy, *Mass-Produced Software Components*, Software Engineering: Report on a Conference by the NATO Science Committee, P. Naur and B. Randell, eds., NATO Scientific Affairs Division, Brussels, p138-155, 1968.

[55] N. Medidovic, R. N. Tayler, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, 26(1), p70-93, January 2000.

[56] 9th MG-R Software Architecture Workshop, Philips internal, Eindhoven, November 19, 2001.

[57] Microsoft COM, http://www.microsoft.com/com/.

[58] Microsoft *DirectShow*, http://www.gdcl.co.uk/dshow.htm, part of DirectX.

[59] Microsoft, *Shell Programmers Guide*, http://msdn.microsoft.com/library/.

[60] Microsoft Visual Basic, http://msdn.microsoft.com/vbasic/, 2003.

[61] Anders Möller, Joakim Fröberg, Mikael Nolin, *Industral Requirements on Component Technologies for Embedded Systems*, to be published in the International Symposium on Component-based Software Engineering, Edinburgh, May 24-25, 2004.

[62] Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics, Vol. 38 No. 8, April 19, p114-117, 1965.

[63] G. Murphy, D. Notkin and K. Sullivan. *Software Reflexion Models: Bridging the Gap between Source and High-Level Models*. Proceedings Third ACM SigSoft Symposium on Foundations of Software Engineering, ACM New York, p18-28, 1995.

[64] National Instruments, *LabView*, http://www.natinst.com/labview/.

[65] Henk Obbink, personal communication, 1988.

[66] Henk Obbink, Herman Postema, Henk te Sligte, Platform-based product development: status experiences and lessons learned or; a nearly impossible necessity, NatLab Internal Report NL-REP 7101, Philips Research Eindhoven, 1999.

[67] Henk Obbink, Rob van Ommering, Jan Gerben Wijnstra and Pierre America, *Component Oriented Platform Architecting for Software Intensive Product Families*, in: Mehmet Aksit, Software Architecture and Component Technology, Kluwer Academic Publishers, p99–141, 2000.

[68] R.C. van Ommering, *Teddy User's Manual*, Technical Report 12NC-4322-2730176-1, Philips Research, 1993.

[69] Rob van Ommering, *The SPRINT Tutorial*, Technical Report NL-RWB-508-RE-94085, Philips Research, 1996.

[70] Rob van Ommering, *Koala, a Component Model for Consumer Electronics Product Software*, in Development and Evolution of Software Architectures for Product Families, Proceedings of the Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, February 1998, LNCS 1429, Springer Verlag, Berlin Heidelberg, p76-88, 1998.

[71] Rob van Ommering, http://www.extra.research.philips.com/SAE/koala/horcom/, 1999.

[72] Rob van Ommering, Frank van der Linden, Jeff Kramer, Jeff Magee, *The Koala Component Model for Consumer Electronics Software*, IEEE Computer, p78-85, March 2000. Also Chapter 3 of this thesis.

[73] Rob van Ommering, *Beyond Product Families: Building a Product Population?*, Proceedings of the 3rd international workshop on the development and evolution of software architectures of product families, Las Palmas, March 2000, LNCS 1951, Springer Verlag Heidelberg, ISBN 3-540-41480-0, p187-198.

[74] Rob van Ommering, *A Composable Software Architecture for Consumer Electronics Products*, XOOTIC Magazine, March 2000, Volume 7 number 3, also to be found at URL http://www.win.tue.nl/xootic/magazine/mar-2000/vanommering.pdf.

[75] Rob van Ommering, *Mechanisms for Handling Diversity in a Product Population*, Fourth International Software Architecture Workshop, Limerick, Ireland, June 4-5, 2000.

[76] Rob van Ommering, *Configuration Management in Component Based Product Populations*, 10th International Workshop on Software Configuration Management, May 14-15, Toronto, 2001, Canada, http://www.ics.uci.edu/~andre/scm10/, also published in LNCS 2649, p16-23. Also Chapter 6 of this thesis.

[77] Rob van Ommering, *Techniques for Independent Deployment to Build Product Populations*, WICSA 2001: The Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 28-31, 2001, p55-66. Also Chapter 4 of this thesis.

[78] Rob van Ommering, *Roadmapping a Product Population Architecture*, 4th International Workshop on Product Family Engineering, Bilbao, Spain, October 3-5, 2001, LNCS 2290, Springer Verlag Heidelberg, ISBN 3-540-43659-6, p51-63, 2002.

[79] Rob van Ommering, René Krikhaar, Loe Feijs, *Language for Formalizing, Visualizing and Verifying Software Architectures*, Computer Languages 27, p3-18, 2001. Also Chapter 2 of this thesis.

[80] Rob van Ommering, *Building Product Populations with Software Components*, International Conference on Software Engineering, Orlando, US, May 2002, ACM, p255-265. Also Chapter 7 of this thesis.

[81] Rob van Ommering, Jan Bosch, *Widening the Scope of Software Product Lines – from Variation to Composition*. Second Software Product Line Conference, San Diego, August 2002, LNCS 2379, ISBN 3-540-43985-4, p328-347, 2002. Also Chapter 5 of this thesis.

[82] Rob van Ommering, *Predicting Properties of Koala 'Assemblies'*, 2nd Workshop on the Predicable Assembly of Certifiable Components (PACC2), Software Engineering Institute, Pittsburgh, Jan 2003.

[83] Rob van Ommering, *Horizontal Communication: a Style to Compose Control Software*, Software: Practice and Experience, (33)12, p117-1150, 2003. Also Chapter 8 of this thesis.

[84] G. A. Papadopoulos and F. Arbab, *Coordination Models and Languages*, in Advances in Computers, Academic Press, August 1998, Vol. 46: The Engineering of Large Systems.

[85] David L. Parnas, *On the Criteria to Be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No. 12, p1053-1058, December 1972.

[86] Dewayne E. Perry, Alexander L. Wolf, *Foundations for the Study of Software Architecture*, Software Engineering Notes, Vol. 17 No. 4, p40-52, Oct 1992.

[87] Dewayne E. Perry et al., *Session Summaries on Product Line Development*, 10th International Software Process Workshop, Ventron FR, June 1996.

[88] Dewayne E. Perry, *Generic Architecture Descriptions for Product Lines*, Proceedings of the Second International ESPRIT ARES Workshop, LNCS 1429, Springer Verlag, Berlin Heidelberg, p51-56, 1998.

[89] Dewayne E. Perry, *Version Control in the Inscape Environment*, Proceedings of the 9th International Conference on Software Engineering, March 30 - April 2 1987, Monterey CA, ISBN 0-89791-216-0, p142-149, 1987.

[90] Frantisek Plasil, Dusan Balek, Radovan Janecek. *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*. Proceedings of ICCDS 98, May 4-6, 1998, Annapolis, Maryland, USA. IEEE CS Press, ISBN 0-8186-8451-8, 1998.

[91] Ben Pronk, *An Interface-Based Platform Approach*, Software Product Lines, Experience and Research Directions, Proceedings of the First Software Product Lines Conference (SPLC1), August 28-31, 2000, Denver, Ed. by Patrick Donohoe, Kluwer, p331-351, 2000.

[92] Jeff Prosise, *Programming Windows 95 with MFC*, Microsoft Press, ISBN 1-55615-902-1, 1996.

[93] Dennis Ritchie: *The Evolution of the Unix Time-Sharing System*, Language Design and Programming Methodology, Proceedings of a Symposium Held in Sydney, Australia, 10-11 September, 1979, Springer LNCS 79, ISBN 3-540-09745-7, p25-36, 1979.

[94] Dale Rogerson, *Inside COM, Microsoft's Component Object Model*, Microsoft Press, ISBN 1-57231-349-8, 1997.

[95] Jan Rooijmans, Hans Aerts, Michiel van Genuchten, *Software Quality in Consumer Electronics Products*, IEEE Software 13(1) , p55-64, 1996.

[96] R. W. Schwanke, R. Z. Altucher, M. A. Platoff, *Discovering, Visualizing and Controlling Software Structure*, ACM SIGSOFT Notes, Vol. 14 No. 3, p147-150, May 1989.

[97] Mary Shaw, *Larger Scale Systems Require Higher-Level Abstractions*, ACM SIGSOFT Notes, Vol. 14 No. 3, p143-146, May 1989.

[98] Mary Shaw, David Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice Hall, ISBN 0-13-182957-2, 1996.

[99] Jon Siegel, CORBA 3 Fundamentals and Programming, 2nd Edition, John Wiley & Sons; ISBN: 0-471-29518-3, 2000.

[100] Software Engineering Institute, Software Product Lines, http://www.sei.cmu.edu/plp/.

[101] Dilip Soni, Robert L. Nord, Christine Hofmeister, *Software Architecture in Industrial Applications*, Proceedings of the 17th International Conference on Software Engineering, 1995, p196-207.

[102] *The Stanford Rapide Project*, http://pavg.stanford.edu/rapide/.

[103] Clemens Szyperski, *Component Software, Beyond Object-Oriented Programming*, Addison-Wesley, Harlow, UK, ISBN 0-201-17888-5, 1998.

[104] Louis J. Taborda, The Release Matrix for Component-Based Software Architectures, CBSE7, International Symposium on Component-based Software Engineering, Edinburgh, May 2004.

[105] Peter Toft, Derek Coleman, Joni Ohta, *A Cooperative Model for Cross-Divisional Product Development for a Software Product Line*, Proceedings of the First Software Product Lines Conference (SPLC1), Denver, USA, August 28-31, p111-132, 2000.

[106] The TriMedia Streaming Software Architecture, Jan 2000, Philips Pub. No. 9397-750-06255, http://www.semiconductors.philips.com/acrobat/literature/9397/75006255.pdf.

[107] Uchitel S., Kramer J. and Magee J., *Synthesis of Behavioral Models from Scenarios*, IEEE Trans. on Software Eng., SE-29 (2), p99-115, Feb. 2003.

[108] VRML, Virtual Reality Modeling Language, http://www.w3.org/MarkUp/VRML/.

[109] Kurt Wallnau, Scott Hissam, Robert Seacord, *Building Systems from Commercial Components*, Addison-Wesley Pub Co; ISBN: 0-201-70064-6, 2002.

[110] Stephen Warshall, *A Theorem on Boolean Matrices*, Journal of the ACM, 9(1) p11-12, 1962.

[111] WEBOPEDIA, http://www.webopedia.com/TERM/U/upward_compatible.html.

[112] Jan Gerben Wijnstra, *Supporting Diversity with Component Frameworks as Architectural Elements*, Proceedings of the 22nd International Conference on Software Engineering, Limerick, June 4-11, 2000, p51-60.

[113] Tony Williams, *On Inheritance, What It Means and How To Use It*, Microsoft Internal Report, http://research.microsoft.com/comapps/docs/Inherit.doc, 1990.

[114] Michael Winter, Thomas Genssler, Alexander Christoph, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Peter Müller, Christian Stich, Bastiaan Schönhage, *Components for Embedded Software — The PECOS Approach*, Second International Workshop on Composition Languages, Málaga, Spain, June 11, 2002.

[115] *XML, The Extensible Markup Language*, http://www.w3.org/XML.

Brief Glossary of Terms

Architecture Description Language
Formal language for describing software architecture. See section 3.2.2.

Backward Compatibility
A component is backward compatible with an older version of itself if the new version can be used in all contexts where the old version can be used. See section 4.2.2.

Binding
The act of connecting two interfaces so that two implementations can work together. See section 3.3.4.

Cable
A binding between two interfaces. See section A.4.9.

Component
A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. See section 3.3.1.

Composition
Technique to handle diversity by identifying the points where software is the same for different products, and factoring those out as components. See section 5.2.5.

Compound Component
A component that contains and encapsulates (instances of) other components. See section 3.3.5.

Configuration Management
Classical technique to handle product variation and evolution. Works at compile-time only and is often architecture agnostic. See section 6.4.

Diversity
The variation between members of a product family or product population. See section 7.3.2.

Diversity Interface
Koala construct for specifying and handling diversity. See section 3.4.4.

Diversity Spreadsheet
Technique for specifying diversity parameters of subcomponents in terms of the diversity parameters of the compound component. See section 3.4.5.

Downward Compatibility
A client is downward compatible with a server if it also runs on an older version of the server. See section 4.2.2.

Evolution
The change of software artifacts over time. See section 3.5.

Fiber
See Function Binding.

Forward Architecting
The construction of an architecture followed by the systematic derivation of an implementation, so that architecture controls the implementation. See section 2.1.

Forward Compatibility
A component is forward compatible with a newer version of itself if the old version can be used in all contexts where the new version is used. See section 4.2.2.

Function Binding
An implementation in Koala of a function in an interface. See section 3.4.2.

Horizontal Communication
Technique for writing reusable control software by letting drivers communicate directly with each other, rather than through managers. See Chapter 8.

Independent Deployment
The phenomenon that components are not designed for specific products, but rather have their own evolution in time. See section 4.2.1.

Interface
A small set of semantically related functions. See section 3.3.1.

Interface Compatibility
The required relation between two interfaces in a legal binding. See section 3.4.1.

Interface Definition
Description of the syntax and semantics of an interface. See section 3.3.2.

Interface Subtyping
See Interface Compatibility.

Mandatory Interface
An interface that must be connected at its tip. See section 3.4.7.

Module
A unit of code in Koala. See section 3.3.6.

Optional Interface
An interface that may but need not be connected at its tip. See section 3.4.7.

Part-of Relation
A fundamental relation in software architecture, a.k.a. as decomposition. See section 2.3.3.

Partial Evaluation
The conversion of a program to a more specialized program by filling in certain parameters. See section 3.4.3.

Portability
The use of a client with a different server. See section 4.6.

Product Family
A set of products with many commonalities and few differences. See section 7.2.2.

Product Population
A set of products with many commonalities but also with many differences. See section 7.2.4.

Product Line
A proactive and systematic approach for the development of software to create a variety of products. See section 5.1.

Provides Interface
An interface that provides access to functionality provided by a component. See section 3.3.1.

Relation Partition Algebra, or RPA
A mathematical framework to model software architecture and implementation, allowing to reason about each individually and both in their mutual relation. See section 2.3.

Requires Interface
An interface through which a component accesses functionality provided by its environment. See section 3.3.1.

Resource Constraints
The phenomenon that resources such as memory and processing time are not infinite. See section 7.2.1.

Reusability
The use of a server with a different client. See section 4.5.

Reverse Architecting
To infer the architecture from an implementation with the purpose to reason about or modify the implementation. See section 2.1.

Software Architecture
The specification of a design. See section 2.2.

Switch
A Koala construct for creating flexible (compile- or run-time) binding. See section 3.4.6.

Third-Party Binding
Construction where components A and B have no knowledge of each other but instead are bound by a third component C. See section 5.5.1.

Upward Compatibility
A client is upward compatible with a server if it also runs on a newer version of the server. See section 4.2.2.

Variation
Common technique to handle diversity, by identifying the points where software differs for different products. See section 5.2.5.

Summary of this Thesis

Today, many consumer products contain ‘small’ computers to control the product internally. Newer products will typically contain more powerful computers than their predecessors, following the same trend as personal and business computers, roughly doubling the capacity every two year according to Moore’s law.

The television was one of the first consumer products with a computer built-in. In 1978, a high-end Philips TV contained a microprocessor with a 1 kilobyte program memory. Ten years later, there were televisions with 64 kilobytes of memory, en another ten years later the size of memory exceeded 1 Megabyte. Other consumer products with built-in computer are video recorders, DVD players and recorders, phones (both fixed and mobile), photo and video cameras, microwave ovens, refrigerators, washing machines, vacuum cleaners, shavers, coffee machines, and the list is growing.

For companies such as Philips this provides the following challenges:

- The complexity of the software in an individual product is growing, and it is becoming increasingly difficult to maintain the desired high quality.
- A company does not sell one product, but rather a family of closely related products. The software should be designed such that as much as possible can be shared between members of the family.
- The market is becoming increasingly dynamic, which makes it important to shorten the time to market, more specifically the development time.
- Companies such as Philips do not produce just one family of products, but rather a number of families (TVs, CD/DVD players, sound amplifiers) which mutually share sets of features. Sharing software between these families reduces development costs even further, but more importantly, it enables the creation of combination products, such as a TV with built-in DVD player.

This thesis builds on three recent trends in science and technology: an increased attention for the *architecture* of software, the invention of techniques to build *software components*, and the systematic creation of *software product lines*. This thesis searches for answers to the following questions:

- Can we make the architecture of software explicit, and at the same time guarantee the consistency between architecture and implementation, so that we can reason about the implementation in terms of the architecture?
- Can we build families and ‘populations’ (read ‘family of families’) of products through the use of software components?
- Can we take advantage of modern techniques to create software components without making our systems more expensive or less optimal in performance?

- What are the consequences of this on the choice of the development process and the required development organization?

This thesis starts by introducing a mathematical technique to model software architectures. Use of this technique enables to check the internal consistency and the consistency with the implementation automatically with the use of a small collection of software tools. This technique, as developed by my colleagues and me has been applied successfully to a large variety of systems. An intrinsic disadvantage of the method is that inconsistencies are usually discovered late in the development process, and solving them is often not cost effective anymore.

Therefore, we introduce an architectural description language from which parts of the implementation can be generated, so that consistency is achieved automatically. At the same time, this language serves as component technology, enabling the creation of different products from the same set of components. The language is designed in such a way that the product's computing resources need not be more expensive, nor that the product is less efficient in performance. Some other contributions of our research are:

- The language supports *independent deployment*: a component need not be developed in a specific context, but can have its own evolution. This requires techniques to implement changes such that newer versions of the component can be used with older versions of other components or vice versa.
- The language emphasizes *composition*, while researchers in software product lines often stress *variation* as the important issue. We see composition as a 'wider' form of variation; the larger the scope of the family (or population) is, the more important composition becomes.
- The classical technique to build more than one products is *configuration management*. We find it important to incorporate management of variation in the architecture, instead of solving it independently.

The thesis describes how the language and method has been applied in Philips to the development of software for mid-range and high-end televisions. This has required an adaptation of the software development process and organization, which used to be optimized for the creation of a single product. We were helped in this by the fact that thinking in families was already starting or common-place in the development of electronics, mechanics and optics.

Finally, we discuss one (be it extended) example of a design pattern that enables composition in a part of the software that was previously considered complex and badly composable. This is by no means the last thing that can be said about composition, and we invite the readers to continue, improve and extend the work as described in this thesis.

Samenvatting

In veel consumentenproducten zijn tegenwoordig ‘kleine’ computers ingebouwd voor de interne besturing van het product. Net als in de wereld van de persoonlijke en zakelijke computer bevatten nieuwere producten krachtigere computers dan hun voorgangers. De capaciteit van de ingebouwde computers verdubbelt elke twee jaar, en volgt daarmee nauwgezet de Wet van Moore.

De televisie was een van de eerste consumentenproducten met een ingebouwde computer. In 1978 had een topmodel Philips TV een microprocessor met 1 kilobyte programmageheugen. Ruim tien jaar later telde een televisie 64 kilobyte aan geheugen, en na weer tien jaar meer dan een megabyte. Andere voorbeelden van consumentenproducten met ingebouwde computers zijn video recorders, DVD spelers en recorders, telefoons (vast en mobiel), auto’s, fotocamera’s, magnetrons, koelkasten, wasmachines, stofzuigers, scheerapparaten, koffiezetmachines, en de lijst is groeiende.

Voor bedrijven zoals Philips levert dit de volgende uitdagingen:

- De complexiteit van de programmatuur in een individueel product stijgt, en het wordt steeds moeilijker om de gewenste hoge kwaliteit te handhaven.
- Een bedrijf verkoopt niet één product, maar een familie van sterk verwante producten. Hierbij is het gewenst de programmatuur zo te ontwerpen dat zo veel mogelijk gedeeld kan worden tussen leden van de familie.
- De markt wordt steeds dynamischer, daarom is het belangrijk om de tijd van het ontwerpen en maken van een product steeds kleiner te maken.
- Tenslotte leveren bedrijven als Philips niet slechts één familie van producten (televisies), maar een aantal families (CD/DVD spelers, geluidsversterkers) die onderling weer gerelateerd zijn. Delen van programmatuur tussen deze families leidt tot een reductie van ontwikkelkosten, maar maakt ook combinatieproducten mogelijk, zoals een TV met ingebouwde DVD speler.

Dit proefschrift bouwt op drie stromingen in de wetenschap en technologie: een verhoogde aandacht voor de *architectuur* van computerprogramma’s, het ontstaan van technieken om *software componenten* te maken, en het systematisch opzetten van *productielijnen* voor het maken van programmatuur. Het probeert daarbij de volgende vragen te beantwoorden:

- Kunnen we de architectuur van computerprogramma’s expliciet maken, met garantie van consistentie tussen architectuur en implementatie, opdat we in architectuurtermen over de implementatie kunnen redeneren?
- Kunnen we families en ‘populaties’ van producten maken met behulp van software componenten (lees een ‘populatie’ als ‘een familie van families’)?

- Kunnen we profiteren van moderne technieken om software componenten te maken zonder dat onze systemen daardoor duurder of trager worden?
- Wat voor een consequenties heeft dit op het te volgen ontwikkelproces en de daarbij benodigde ontwikkelorganisatie?

Eerst beschrijft dit proefschrift een wiskundige techniek om software architecturen te modelleren. Hierbij is een automatische controle op interne consistentie en op consistentie met de implementatie mogelijk met behulp van een kleine verzameling gereedschappen. Deze techniek is door mij en diverse collega's ontwikkeld en succesvol toegepast op een verscheidenheid aan systemen. Intrinsiek nadeel van de methode is dat inconsistentie altijd achteraf gevonden wordt, en dan is het vaak te laat in het ontwikkelproces om de inconsistentie op te lossen.

Daarom introduceren we een taal waarin de architectuur beschreven kan worden en van waaruit delen van de implementatie gegenereerd kunnen worden zodat er per definitie consistentie is. Tevens dient deze taal als een component technologie waarmee het mogelijk is om vanuit dezelfde componenten verschillende producten te bouwen. De taal is zo ontworpen dat de producten niet duurder of trager gemaakt worden. We noemen nog enige andere contributies van ons werk:

- De taal ondersteunt *onafhankelijke ontwikkeling*: een component behoeft niet langer in een specifieke context ontwikkeld te worden, maar kan relatief zelfstandig evolueren. Dit vergt technieken om veranderingen zo te implementeren dat nieuwere versies van de component samen met oudere versies van andere componenten gebruikt kunnen worden.
- De taal benadrukt *compositie*, terwijl onderzoekers aan productielijnen vaak *variatie* benadrukken. Compositie kan gezien worden als een ruimere vorm van variatie, en wordt noodzakelijker naarmate het bereik van de familie(s) groter wordt.
- De klassieke techniek om meerdere producten te bouwen is *configuratie beheer*. Wij menen dat het verstandig is om variatiebeheer in de architectuur op te nemen in plaats van het apart op te lossen.

Vervolgens beschrijven we hoe we de taal en bijbehorende methodiek binnen Philips toepassen op de ontwikkeling van software voor middenklasse en topmodel televisies. Dit vergt een aanpassing van het software ontwikkelproces en van de ontwikkelorganisatie, die van oudsher geoptimaliseerd zijn voor het maken van één product. Hierbij zijn we wel geholpen doordat de elektronica, mechanica en optica afdelingen al langer in families denken.

We bespreken één (uitgebreid) voorbeeld van een ontwerp dat compositie mogelijk maakt in een deel van de programmatuur dat altijd als ingewikkeld en slecht componeerbaar werd beschouwd. Daarmee is het laatste woord over compositie nog lang niet gezegd, en wij nodigen de lezers dan ook uit om het werk dat in dit proefschrift beschreven is, toe te passen, te verbeteren en voort te zetten.

Acknowledgements

My last remaining task is to acknowledge all those people that have contributed to the work described in this thesis. This is an impossible task, given the many people that have helped to design, implement, apply, criticize, sponsor and evangelize the work. I am going to try anyway, and if your name is not listed, rest assured that my gratitude is not less than for those listed below.

Henk Obbink started our research in product families and stimulated me in writing some of it down in the form of a PhD thesis. Also, he was my cluster leader when the work started. Other managers that supported me were Hans van Antwerpen, Jean Gelissen, Jaap van der Heijden, Wouter-Jan Lippmann, Hugh Maaskant and Otto Voorman. And I should not forget our secretary, Anja van Dooren.

Remi Bourgonjon was the initial owner of our research work. Subsequent sponsors were Hans Aerts, Erik Kaashoek and Osman Khan. Joop van der Linden led the project that used Koala for the first time.

Jeff Kramer and Jeff Magee demonstrated Darwin in the ARES project, and this has strongly influenced the design of Koala. They also stimulated me to publish my work. Jeff Magee helped me to analyze the HorCom protocol with his LTSA.

The initial team that designed Koala consisted of Aad Droppert, Hans Jonkers, Gerard van Loon, Marc Loos and myself. Pivotal were comments by Robert Jagt and Maarten Pennings. Hans van Antwerpen and Maarten Pennings implemented the first Koala compiler. In subsequent years, the responsibility for the compiler was taken over by Chritiene Aarts, Bram Stappers and Rob Wieringa.

The MG-R architecture was designed by Hans van Antwerpen, Gerard van Loon and me. Initial subsystem architects were Chritiene Aarts, Klaas Brink, Maarten Pennings and Ganesh Thonse. John Coumans was our project leader.

More people became involved when the project was scaled-up. Among these were Wim Baekelandt, Elmar Beuzenberg, Kathleen De Sutter, Vinay Deshpande, René Geraets, Eric Heijkers, Padma Kulkarni, Dirk Lietaert, Geert Neuttiens, Louis Stroucken, Geert Vancompernolle, Mikaël Van Herpe, Jan Verbeke, Theo Vergoossen and Emiel Wijgerink. Important managerial roles were filled by Otto Voorman and Jan Rooijmans.

Within research, I learned a lot from cooperation in the *Composable Architectures* project, with Pierre America, Marcel Bijsterveld, Frank van der Linden, Jürgen Müller, Gerrit Muller, Henk Obbink, William van der Sterren and Jan Gerben Wijnstra.

Jan Bosch was instrumental in placing my work in the larger context needed to write this thesis. We first met in 1998, and first agreed on the topic of my thesis in

2000. That it took me so long to complete the thesis just shows how much I appreciate working with Jan.

Many other people provided valuable remarks, invited me to give presentations, or asked me to join program committees or otherwise help to organize workshops and conferences. Among these are Lenn Bass, Marcello Bonsangue, Joe Baumann, Paul Clements, Ivica Crnkovic, Eric Eide, Stefan Ferber, Dieter Hammer, André van der Hoek, James Ivers, Merijn de Jonge, Kyo Kang, Charles Krueger, Neil Maiden, Tomi Männistö, Nenad Medvidovic, Robert Nord, Linda Northrop, John Regehr, Heinz Schmidt, Judith Stafford, Kurt Wallnau, and last but not least, David Weiss.

There is life after Koala as described in this thesis. Evgeni Eskenazi, Robert Jagt, Piërre van de Laar, Milan van den Muyzenberg, Gerben Soepenberg and Ivan Vojsovic provided valuable input to extend Koala with various features. A major upgrade named Koala2 was recently designed by Chritiene Aarts, Bas Engel, Pieter Kunst, Milan van den Muyzenberg, Jacques Swillens, Jos Vergeer, Bernard van Vlimmeren, Rob Wieringa and me. Additional features are now being studied by Paul Hoogendijk, Piërre van de Laar, Felix Ogg and Jur Pauw. Reinder Haakma created an interesting variation on Koala, named Kangaroo. Maurits Rijk measured reuse in MG-R.

The work described in chapter 2 was done together with Reinder Bril, Loe Feijs, René Krikhaar and André Postma.

Hennie Alblas designed the cover of this thesis. The Koala diagram on the front page was created with KoalaViewer, a tool made by Chritiene Aarts, and it shows an early version of our TV platform using the HorCom protocol. The TV on the cover is an FL-1 that we bought around 1992, while I was sleeping, recovering from a concussion.

I am glad that my children Kim, Michael and Jamie can now finally read about what I have been doing all these years. Kim also helped me with the layout of this thesis. I thank my father and my late mother for laying the fundaments for this work. Last but not least, I want to thank my wife for all the support that she gave me during the years I have been working on this thesis, especially all the holidays that she arranged to give me time to write, to think and to talk to her about this subject. Had I written this last sentence myself, I couldn’t have put it any better.

Hapert, October 20th, 2004.