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.

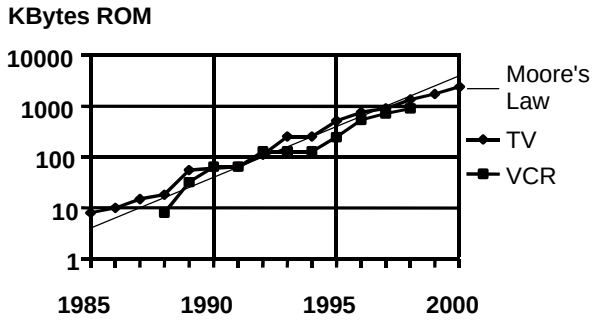

Figure 40. Growth of embedded software in CE products

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.

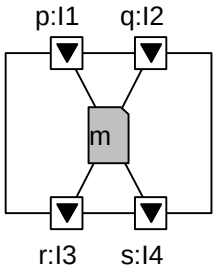

Figure 41. A Koala component

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.

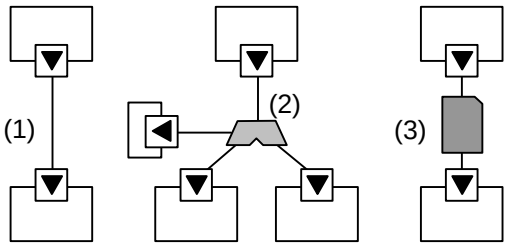

Figure 42. Koala's forms of binding

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.

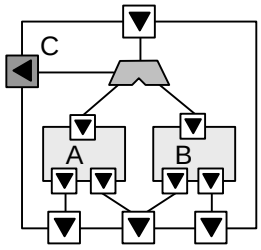

Figure 43. A compound Koala component

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.

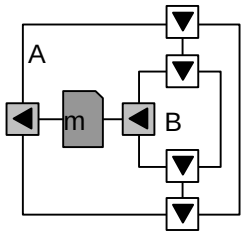

Figure 44. Components with diversity parameters

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.

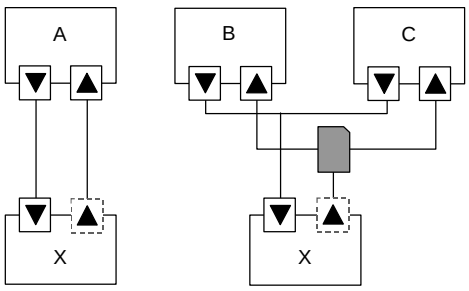

Figure 45. Notification Interfaces

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.

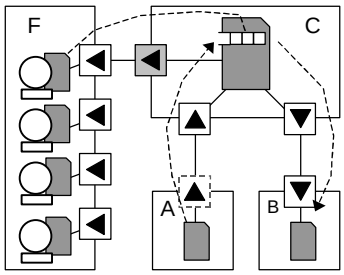

Figure 46. Decoupling notifications

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.

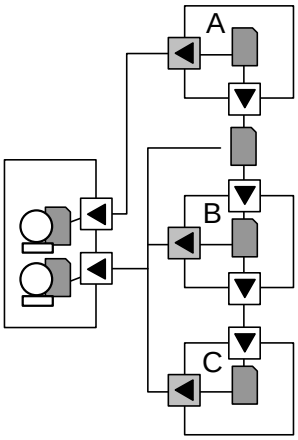

Figure 47. Synchronizing components.

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.

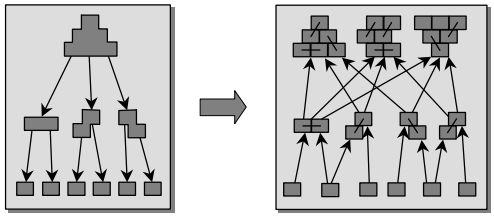

Figure 48. From decomposition to composition

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.

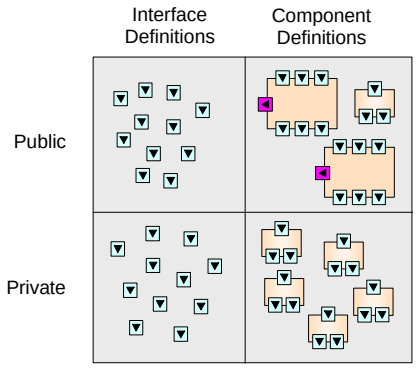

Figure 49. A package

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.

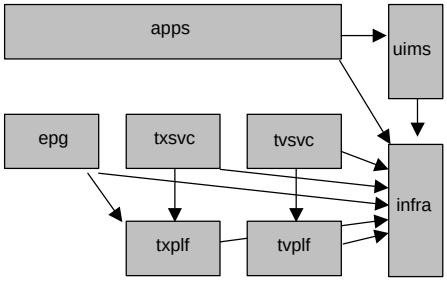

Figure 50. Strong dependencies between packages

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.

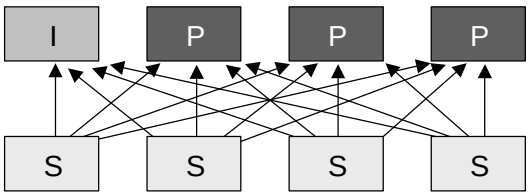

Figure 51. Subsystem deliveries to products

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.

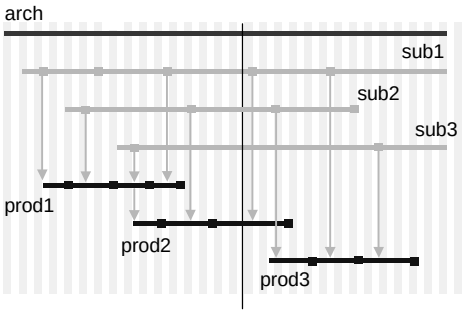

Figure 52. Roadmapping subsystems and products

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.

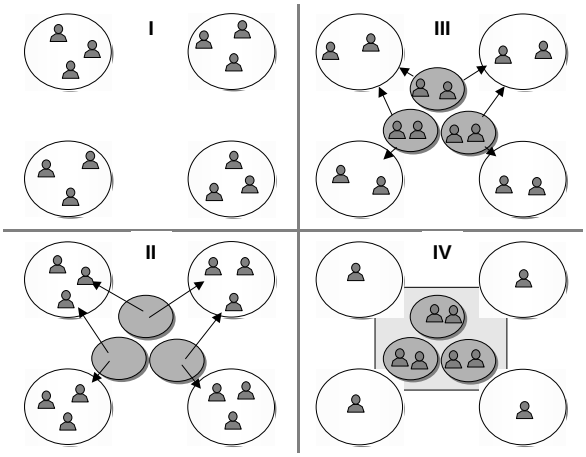

Figure 53. Types of organization

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.

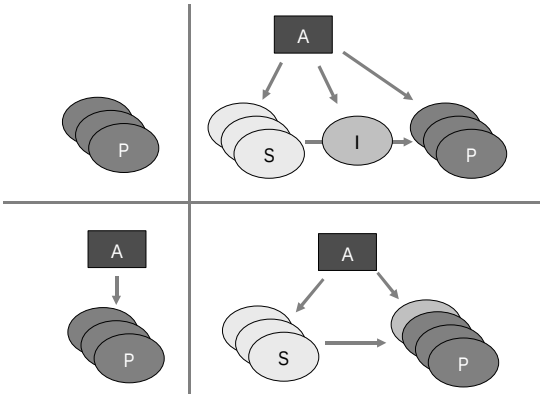

Figure 54. Strategies for organizing reuse

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.