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.

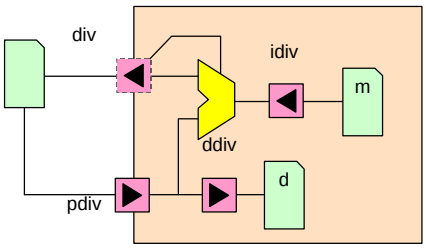

Figure 78. This pattern assigns default values to diversity parameters.

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.