Chapter 10
A Reference Architecture for Consumer Electronics Products and its Application in Requirements Engineering

Tim Trew, Goetz Botterweck and Bashar Nuseibeh

Abstract  Consumer electronics (CE) products must be appealing to customers, have features that distinguish them in the market and be priced competitively. However, this must be achieved with limited hardware resources, so requirements engineers and architects must work together to specify an attractive product within these constraints. This requires an architectural description from early in development. The creation of this description is hampered by the lack of consensus on high-level architectural concepts for the CE domain and the rate at which novel features are added to products, so that old descriptions cannot simply be reused.

This chapter describes both the development of a reference architecture that addresses these problems and the process by which the requirements and architecture are refined together. The reference architecture is independent of specific functionality and is designed to be readily adopted. The architecture is informed by information mined from previous developments and organised to be reusable in different contexts. The interplay between the roles of requirements engineer and architect, mediated through the reference architecture, is described and illustrated with an example of integrating a new feature into a mobile phone.

10.1 Introduction

Consumer electronics (CE) products, such as TVs, smart phones and in-car entertainment, must be appealing to customers, have features that distinguish them in the market and be priced competitively. Despite falling hardware costs, the resources available for their implementation, such as processing power, memory capacity and speed, and dedicated hardware elements, are still limited. These constraints may restrict the features that can be offered, reduce their capabilities or...
limit their concurrent availability. Requirements engineers and architects must work together to specify an attractive product within these constraints, which requires an architectural description from the beginning of development.

Given the rate at which novel features are added to product categories such as mobile phones, requirements engineers cannot only rely on past experience. Instead, they have to reason from first principles about how a new feature might be used, how it might interfere with other features, whether implementations developed for other product categories would be acceptable and how requirements may have to be adapted for the feature to be integrated into the overall product at acceptable cost and risk.

This reasoning is hampered by the lack of consensus on high-level architectural concepts for the CE domain. In contrast, the information processing domain has widely-recognized industry standards and concepts, such as transactions and the transparencies supported by distributed processing middleware [1], which are rarely relevant for embedded software. As an example of the differences, a transaction, a core concept in information processing, is rarely used in the lower levels of embedded software. This is because any software action that changes the state of the hardware may be immediately observable by end-users and cannot be rolled-back unnoticed.

In this chapter, we describe a reference architecture for CE products, which facilitates the creation of concrete architectures for specific products, and show how this reference architecture can be used by requirements engineers to ensure that products are both attractive for consumers and commercially viable. The reference architecture was developed within Philips Electronics and NXP Semiconductors, Philips’ former semiconductor division. Philips/NXP developed both software-intensive CE products and the semiconductor devices that are critical elements of their hardware. These semiconductor devices are associated with considerable amounts of software (over one million lines of code for an advanced TV), and are sold on the open market.

The reference architecture addresses the limited consensus on concepts in this domain, and avoids the need for architects to become familiar with many abstract concepts before it can be used. This is by proposing recommended design solutions for each element in the architectural structure, each followed by a process for reviewing the design decisions in the light of the specific product requirements. The content of the reference architecture is based on the experience of many product developments, and the granularity of its structure is determined by the architectural choices that must be made. Since architects must map new features onto the reference structure, they are confronted with architectural choices and their consequences for the requirements from the outset. We present a process for the concurrent refinement of requirements and architecture.

In Section 10.2, we describe the architectural concerns of requirements engineers, with examples of requirements and architectural choices that should be refined together and the types of architectural information required in each case.
This includes the requirements for both complete products and individual COTS components, which may have to be integrated into a variety of architectures.

Section 10.3 discusses how reference architectures and other forms of architectural knowledge have been used in software development and considers how they can address a broad application domain, independent of specific functionality.

Section 10.4 describes the scope of the domain of CE products and identifies some of the characteristics and requirements of the domain that distinguish it from others. Section 10.5 describes how, given the limited consensus on high-level concepts relevant to the architecture of embedded software, appropriate information was mined from earlier developments. It then describes how the information was organised into our reference architecture. Finally, Section 10.6 describes a process in which the requirements engineer and architect uses the reference architecture to refine the requirements and gives an example of its use in the integration of a novel feature into a mobile phone.

10.2 Architectural concerns of requirements engineers

Nuseibeh’s “Twin Peaks” model describes both a concurrent process for requirements engineering and architecting and the relationship between requirements, architecture and design artefacts [2]. While this provides an overall framework, on its own it is not concrete enough to provide guidance for the development of CE products. Our reference architecture aims to pre-integrate elements of design and architecture, so that, when developing the architecture for a new product, it will both provide guidance on the decisions that must be made and give insight into the refinement of the requirements.

As a first step in the development of a reference architecture, we consider the types of architectural information that are relevant when establishing the requirements for a CE product. Specifying these products requires a careful balance between functionality and product cost, while meeting the constraints of performance, quality and power consumption. The company that is first to introduce a new feature, at the price-point acceptable for the mainstream range of a product category, can achieve substantial sales.

10.2.1 Aligning requirements and resources

*Balance between functionality and price* – Achieving the balance between functionality and selling price requires early insight into alternatives for how a feature might be implemented, and the hardware consequences for each of the options. Many CE products have real-time requirements, both firm performance requirements for signal processing, (e.g., audio/video processing or software-
defined radio), and soft ones to ensure that the product appears responsive to the user. If it is only found late in development that these requirements cannot be met, then features may have to be dropped or downgraded, undermining the value proposition for the product. As reported by Ran, by the mid-1980s embedded products had already reached a level of complexity where it was no longer possible to reason about their performance without architectural models that characterise their behaviour [3]. Therefore, there should be an architectural model from the very outset as the basis for refining requirements and architecture together, to specify a product that makes the best use of its resources.

Visibility and resolution of resource conflicts – The requirements engineer must be able to ensure that the resource management policies that resolve conflicts between features result in a consistent style of user interaction. CE products often have several features active concurrently, which not only impacts performance, but can also result in contention for non-sharable resources and thereby feature interaction [4]. Although resource management policies might be considered to be a purely architectural issue, they can affect the behaviour at the user interface. Therefore, the requirements engineer must be able to understand both the nature of the resource conflicts, to be able to anticipate that feature interaction could occur, and the options for their resolution.

An example of this is the muting of TV audio, which is used by several features, such as the user mute, automatic muting while changing channels or installing TV channels, and the child lock, in which the screen is blanked and the sound muted when a programme’s age rating is too high [5]. Since these features can be active concurrently, feature interaction will result, and it must be possible to articulate policies across the features. These policies should be directly traceable to the architecture to ensure that they are correctly implemented.

Consequently, it must be possible to map features in the requirements specification onto elements of the software architecture to ascertain which features can co-exist and for the software architecture to be able to represent the different resource management policies for resolving resource conflicts.

**10.2.2 Architectural compatibility of software from external suppliers**

Major CE companies used to develop all their software in-house in order to have complete control over its requirements and to be able to fully optimize the implementation for their hardware architectures. However, this became uneconomic as the number of product features increased and they now purchase software for non-differentiating features. While features available on the open market are usually governed by standards, these standards largely focus on the interface between the product and its environment and rarely address the APIs
between the implementation of this feature and the remainder of the product. For instance, the standards for Conditional Access systems for controlling viewing in pay-TV systems specify how a TV signal is encrypted and the interface to the user’s smart card, but not the particular functions that should be called to activate decryption in the TV.

Consequently, each supplier develops their own API, with the expectation that it can be integrated into the architecture of any of their customers’ products, but with little knowledge of how the architectures may differ between customers. Although integration problems can arise from many sources, a significant class of problems result from the dynamic behaviour of the software, particularly since multiple threads often have to be used to satisfy performance requirements. Different companies may adopt different policies for aspects such as scheduling, synchronization, communication, error handling and resource management. Failure of the supplier and integrator to achieve a mutual understanding of their policies can lead to complex integration problems. A second source of problems is when the functionality of components from different suppliers overlaps, so that the components do not readily co-exist. The requirements engineer of the component supplier should be aware of these potential problems, which will be described in more detail shortly. Conversely, the requirements engineer of the product integrator should be aware that such mismatches can occur and that these might be too risky to resolve if there is a tight deadline for delivery.

The reference architecture should allow the compatibility between a component and the remainder of the product to be assessed at an early stage. Ruling out a COTS component on these grounds, despite having an attractive feature list, allows the requirements engineer to focus on less risky alternatives. This may require abandoning low-priority requirements that are only supported by that component. To be able to detect incompatibilities, the options for the behaviour of the software must be explicit in the reference architecture, while being described independently of the structure of the software. This independence is required because the lack of API standardisation results in suppliers using different structural decompositions for the same functionality. We therefore capture alternative behavioural policies as architectural texture, which Ran describes as the “recurring microstructure” of the architecture [3] and which van der Linden characterizes as “the collection of common development rules for realising the system” [10]. Kruchten’s architectural mechanisms for persistency and communication [11] are concrete implementations of behavioural policies. The identifica-

---

1 There have been many industry standardization initiatives for particular product categories for interfaces below the application layer, such as LiMo for mobile phones [6], the MPEG Multimedia (M3W) for audio/video platforms [7] and OpenMAX for media processing libraries [8]. However, to date, none has been widely-adopted in the market. Contributors to this lack of adoption are the high degree of technical and market innovation in the CE domain and the unstable structure of the industry, which is in a transition away from vertically-integrated CE companies [9].
tion of the alternative policies to include in our reference architecture and the structuring of its architectural texture are described in Section 10.5.

These issues are described in greater detail in the following paragraphs:

Policies for error handling and resource management – From a requirements perspective, policies for error handling and resource management can affect the behaviour observed by the end-user and, hence, must be assessed for their acceptability. For example, the vendor of a TV electronic programme guide may have a policy of displaying their logo when the guide is activated, but the overall product may support the restart of a specific feature if it fails. This restart would aim to restore the feature to as close to its previous state as possible, and with minimal disturbance to the user. However, this recovery would be compromised if the guide also displays the logo during this restart.

Degree of control by supplied components – Another source of incompatibility with supplied components is the scope of their functionality and the degree of control that they expect to have over the platform. Multi-function CE devices may integrate best-of-breed functionality from several suppliers. Problems can occur if the required interfaces of a component are too low, so that the component encapsulates the control of the hardware resources it requires, or if the provided interfaces are too high level.

The first case can cause two types of problems: either it is not possible for this feature to execute concurrently with another that should share the same resource, or it is not possible to achieve a smooth transition between features that require exclusive access to the resource. The required interfaces of these components should always be high enough that it possible to insert a resource management mechanism below them. However, new features are often originally conceived for products in which they would always have exclusive access. Then provisioning for an additional layer might have appeared to be an unnecessary overhead and an additional source of complexity. It may only be years later, when the product is extended with functionality from another category, that the problem emerges.

As an example of restrictions on concurrent execution, consider the potential conflicts between interactive services and video recording in a TV. Both features must be able to both monitor broadcast data continuously and to select new stations. Both features must be active continuously and must be able to share the resources. However, they may not have been designed with that in mind. All terrestrial digital TVs in the UK have supported interactive services from the outset. However, it was only a decade later that digital video recording was integrated into TVs. In planning the extension of the TV to include the recording functionality, it may have been thought that it is only necessary to add the new feature, whereas it may also have been necessary to acquire a new interactive TV engine from a different source and to develop a resource manager. If the TV is scheduled for launch within a tight window, the additional risk associated in this change may result in the introduction of the recording feature being deferred.

Even if features are not to be active concurrently, excessively low-level required interfaces can impair the end-user experience. For instance, as Wi-Fi home
networks became common, stand-alone adapters were developed to allow consumers to browse for content on their PCs or the Internet and then to decode the video streams, which could then be fed to a conventional TV. When this functionality was later integrated within the TV, it was desirable to reuse the same software components. However, previously these had exclusive control of the video decoders, but in the new context this control has to be handed over to the conventional broadcast TV receiver. If the Wi-Fi browser does not have provision for this, it may be necessary to reinitialize the component whenever the feature is selected, causing it to lose internal state information and taking an excessive time. The requirements engineer must be aware of such consequences of reusing a proven component in this new context to be able to decide whether the resulting behaviour will be acceptable for the end-user.

Having provided interfaces at too high a level can compromise the consistency of the user interface. The supplier of a component of a resource-intensive feature has to ensure that it can operate reliably with the available hardware resources, e.g. memory capacity, processing power or interconnect bandwidth, and possibly with minimal power dissipation. This is most easily achieved with resource managers that are not only aware of the current global state of the component, but also of the desired future state of the component so that state transitions can be planned in ways that avoid transient exhaustion of resources. For instance, in a product with multi-stream audio/video processing, the semiconductor supplier may wish to have complete control of the processing of these streams and of the transitions between different stream configurations. This can be achieved by raising the level of the provided interface, so that the client only makes a single declarative request for a new configuration, much in the style of SOA. This enables the supplier to both provide a component that can be fully-validated, independent of the behaviour of the customer’s software, and allows the supplier to innovate by evolving their hardware/software tradeoffs without affecting their customers’ code. These properties of dependability and evolvability are important non-functional attributes for component supplier, but they can lead to two problems for the product integrator. Firstly, in this example, the integrator may be reluctant to reveal the stream configurations that it plans to use and, secondly, the supplier’s state transition strategy may differ from that used in other features, resulting in inconsistent overall product behaviour.

An architectural model is required that allows this tension to be discussed without either party exposing critical intellectual property (IP), possibly providing the motivation for the parties to enter a closer commercial partnership where requirements can be discussed more freely. Therefore, the architecture should represent the responsibilities of the components, while being independent of their specific functionality.

This section has identified some situations in which requirements and architectural choices should be refined together, both (1) to achieve a satisfactory balance between functionality and product cost and (2) to ensure that resource management policies result in a consistent user interface. It has also addressed the
selection of COTS components, both by identifying policies that have some influence on the behaviour observed by the end-user, and by rapidly screening components for architectural compatibility, so that unsuitable components can be disregarded at an early stage. In each case, the types of architectural information required has been highlighted, including identifying the resources used by any feature and the options for managing resource conflict, and representing the scope of different COTS components, in terms of the levels of their provided and required interfaces. This must be done with a reference architecture which is abstracted from the concrete product line architecture, both to allow these decisions to be made at an early stage in development, before a refined architecture is available, and to protect the IP of the parties.

Having described the support that a reference architecture should provide the requirements engineer, the remainder of the chapter described how such an architecture was developed for the CE domain and illustrates how it can be used in practice. As a first step in this, the next section reviews how industry develops and uses reference architectures in general.

### 10.3 Reference architectures in software development

Before describing how our reference architecture was developed and how it can be applied, we will introduce the form and use of reference architectures in some more mature application domains and what lessons can be learnt for the development of our architecture.

The role of reference architectures in software development is well-established; the Rational Unified Process uses them to capture elements of existing architectures, which have been proven in particular contexts, for reuse in subsequent developments [11,12]. Reference architectures can exist at many levels of abstraction and can take many forms, depending on the context in which they are to be applied. The Open Group Application Framework (TOGAF) introduces the *architecture continuum* to describe the degree of abstraction a reference architecture has from an organisation-specific architecture [13]. TOGAF describes the characteristics of potential architectures in this continuum, ranging from *Foundation Architectures* to *Organization-Specific Architectures* and provides a *Technical Reference Model* (TRM) as an example Foundation Architecture. The TRM is a taxonomy of applications, services and service qualities. The service taxonomy is specific to information processing applications. For our purposes, we require a model that is less specific to an application domain, since the definition of services changes rapidly as the functionality supported by a product category evolves. We also require a model that provides more technical guidance, while being independent of the functionality being supported.

The OASIS reference architecture for service-oriented architecture (SOA) [14] is an example of such an architecture, since it captures the information that is
important for a successful SOA, independent of its functionality. In this case, the overall structure of the SOA relates to the structure of the business, so that there is a straightforward mapping from the requirements to the structure of the software architecture. This mapping is more complex in an embedded system, with aspects of a particular feature being implemented at different layers in the system, e.g. based on the need to support variability of requirements and hardware and to separate operations with different temporal granularity [15]. Therefore, in contrast to the SOA reference architecture, our reference architecture for CE products should provide guidance on structuring the software.

Eeles and Cripps’ classification of architectural assets [16] uses axes of granularity and level of articulation (or implementation). At a fine grain, they identify Architectural Styles, Architectural Patterns, Design Patterns and Idioms, which are at a suitable level of articulation for our purposes. However, at a coarser grain, their Reference Model is more domain-specific, being comparable to the TOGAF TRM. We seek a reference architecture that can aggregate the fine grain architectural information and provide guidance on its use, while still being independent of the application domain.

POSA4 [17] presents an extensive pattern language that addresses these aims. This provided inspiration for some aspects of our reference architecture but it does not provide sufficient guidance for determining the overall structure of an embedded system for two reasons. Firstly, rather than giving specific guidance, it raises a set of general questions about the behaviour of an application, the variability that it must support and its life expectancy and then describes the characteristics of the architectural styles and patterns, relying on the insights of the architects, who must be familiar with a wide range of concepts before the language can be applied. Secondly, developing the structure of embedded software is particularly challenging because it is usually a hybrid of architectural styles. For example, in the structure in Fig. 10.5, the software is largely structured as layers, but the operating system may be orthogonal to these, being accessible by all layers. Moreover, different architectural styles may be used in different layers, e.g. the media/signal processing may employ pipes and filters and the user applications may use model-view-controller.

POSA4 addresses the problem of how to interpret the general questions in the context of a specific application by preceding its pattern language with an extensive example of the development of a warehouse management system. This approach of using a running example is also used by Moore et al. in their B2B e-commerce reference architecture [18]. Here, the reader can draw parallels between these examples and their own applications by using the widely-accepted concepts of the information processing domain. This approach is less effective for embedded software because of the limited consensus on higher-level concepts.

Considering how better support might be given to architects, Kruchten states that “architecture encompasses significant decisions” about the software [11], therefore we might expect that the reference architecture will have made some decisions, which are applicable throughout its scope, and identify decisions topics
that have to be addressed for the current system. In their model of architectural knowledge de Boer et al. state, “decision making is viewed as proposing and ranking Alternatives, and selecting the alternative that has the highest rank … based on multiple criteria (i.e. Concerns)” [19]. The reference architecture should provide guidance for making such decisions.

Reed provides an example of a reference architecture for information processing, using an N-tier architecture and identifying the decision topics for each tier [12]. Here the decision criteria can be described concisely and unambiguously since they are based on widely-understood concepts and established technology standards. While this example is a valuable illustration of the role of reference architectures in supporting the creation of a wide variety of applications, many more decisions are required to cover the whole information processing domain.

A more extensive example is Zimmermann et al.’s reusable architectural decision model for developing a SOA, containing 300 decisions [20]. To guide the architect through the decisions, the decision model is structured by an extension of IBM’s 7-layer SOMA model for SOA development [21]. However, for this guidance to be effective, and for the consequences of the decisions to be fully appreciated, the architects should already be familiar with the concepts in SOMA, otherwise the initial effort required to adopt it will inhibit the reference architecture’s deployment. This consensus is lacking in the CE domain, as highlighted by the problems of enforcing several hundred architectural rules for a single concrete CE architecture, developed across multiple sites, reported by Clerc et al. [22]. The adoption of a reference architecture in Zimmermann’s form would be even more challenging, given the broader scope of the domain and the lack of an initial structure in which to position the decisions. Therefore, while architects claim that they do not want to be unduly constrained, and following early trials with a structure comparable to Zimmermann’s, we concluded that our reference architecture had to be more prescriptive. Therefore, rather than beginning with a sequence of decision topics from which the architect would develop their architecture, it begins by proposing a design, followed by the decision topics, with alternatives and design rationale, that should be considered where the architects believe the recommended design to be inappropriate.

Many of the decisions relate to the satisfaction of non-functional requirements (NFRs), comparable to TOGAF’s service qualities. We have extended Gross and Yu’s approach to guiding the selection of design patterns, given the product’s NFRs [23], which is itself based on Chung et al.’s NFR Framework [24].

Muller and Hole report on the views of architects, developing embedded software in several industries, on the role of reference architectures and how they can be developed [25]. They show how reference architectures are informed by feedback from the field, in terms of both new stakeholder requirements that would be satisfied if the architecture could be changed, and the constraints that should be imposed on new architectures to avoid problems that have occurred in practice. They note that one of the main objectives for developing a reference architecture might be to ensure that integration effort is low and, indeed, this was the starting
point for developing our architecture. However, before describing how the architecture was developed, we will first scope the domain of CE products and identify the viewpoints that the reference architecture should contain.

10.4 Required scope and viewpoints of the reference architecture

When developing the reference architecture, we have to address of the scope of the CE domain to be covered by the architecture and the identification of appropriate viewpoints. These can be considered in relation to the business aims that motivated the development of the architecture, which go beyond the needs of the requirements engineer. Indeed, the search for the form and content of the reference architecture was driven by the desire to avoid integration problems. The overall set of business aims were as follows:

- To enable requirements engineers to ensure that the product makes the best use of its resources and to ensure that resource management policies result in a consistent user interface, as introduced in Section 10.2.1. This is particularly important the first time that a feature is incorporated into a product category.
- To support requirements engineers in the selection of software components from external suppliers, as introduced in Section 10.2.2.
- To support software component suppliers in establishing the requirements for components to be supplied to CE manufacturers, as introduced in Section 10.1.
- To enable architects to exchange best practices across different product categories, having different concrete architectures. This is particularly to avoid problems during component integration and to improve maintainability.
- To support internal standardization to facilitate reuse as the requirements of different product categories converge.

Note that these aims do not include aspects, such as hardware-software co-design, where specialised analytical models, such as Synchronous Data Flow [26], specific to the nature of the processing, are used to optimise the system architecture. While such optimisation is critical to the success of the product, it normally addresses only a small proportion of the code. The overall software architecture must ensure that the remainder of the software does not compromise the performance of these critical elements.

In selecting the application domain to be addressed by the reference architecture, we have taken the broad domain of CE products, rather than developing separate reference architectures for each product category, such as TVs and mobile phones. This is for several reasons. During the requirements phase, we need to be able to handle the expansion in the scope of functionality supported by a product category, whether with an entirely novel feature or a feature that was originally developed for another category. By abstracting from specific functionality we are able to provide support for feature combinations that had not been
promote reuse between product categories. Without a reference architecture, the differences in requirements and business models can obscure their commonalities. A broader reference architecture will be exposed to a wider range of design choices, which makes it more effective when incorporating COTS components, and will be more satisfactory for architects to use since it cannot be overly prescriptive. Finally, the effort of developing a broadly scoped architecture can be recouped over more development projects.

Fig. 10.1 Context diagram for the domain of CE products.

The scope of the CE application domain is characterised in two ways:
1. An abstract context diagram for a generic CE product, shown in Fig. 10.1. This informal diagram is annotated with examples of actors and protocols.
2. A list of the general capabilities of these products, namely:
   o The reception, generation, storage, processing and rendering of audio, video and graphics.
   o Interaction with data services.
   o Communication through wired and wireless connections.
   o Interaction with peripheral devices.
   o Interaction with a user, either supporting the physical user interface or, if only a co-processor is being developed, its control interface to a host processor.
The following are the general non-functional requirements and constraints of the products in this domain:

Requirements: The products must meet firm and soft real-time performance constraints. Their user interfaces must be responsive, even when actions are inherently time-consuming. Most actions should be interruptible, with the system transitioning smoothly to respond to the new command. Actions can be triggered by both user commands and spontaneous changes in a product’s environment. Several features may be active concurrently. Products are usually members of a product line, whose members may address different geographic regions or ranges of features.

Constraints: The products have limited resources, such as application-specific hardware and processing power and memory. They have limited user interfaces, in which feature interaction must be carefully managed, rather than providing virtualized interfaces using a windowing system. Because any change to the state of the hardware may be directly observable by end-users, product-level requirements may impose constraints on the exact sequence in which the sub-steps of an action are made. For instance, when changing the channel of a TV, some manufacturers favour only displaying the picture when all its parameters are known, whereas others display it at the earliest opportunity and then adapt its presentation as its characteristics, such as its aspect ratio, are detected. While the post-conditions are the same in both cases, the user experience is quite different.

Since our reference architecture is used to create the software architecture description for a specific product line, its viewpoints are aligned with those of the existing architecture descriptions. Philips and NXP Semiconductors used Obbink et al.’s Component-Oriented Platform Architecting (COPA) method [27], which addresses the development of a product family from the perspectives of business, architecture, process and organisation (BAPO), as elaborated by van der Linden et al. [28]. COPA’s architecture perspective has five viewpoints: the purely commercial viewpoints of customer and application, the purely technical viewpoints of conceptual and realization, and a shared functional viewpoint that represents the traditional product line requirements specification [29].

To support requirements engineering it might appear to be best to focus on the commercial viewpoints, which characterize business value of the software and the context in which it will be used. For instance, the COPA application viewpoint is comparable to the TOGAF Industry Architecture [13]. However, we see that apparently similar products, such as TVs sold in the retail market and set-top boxes supplied to cable TV operators, have quite different business models and value propositions. Similarly, the business model of a supplier of components into these markets will be very different from that of a product integrator, which will usually translate into differences in the technical viewpoints to support a greater degree of variability. Consequently, the commercial viewpoints are usually
specific to a business unit, which can develop views that are more specific than is appropriate for a reference architecture covering a broad domain.

In NXP Semiconductors, the technical viewpoints are documented according to a “Software Architecture Design” (SAD) template, described and evaluated by van Dinther et al. [30]. This template is used in several companies and is an extension of Kruchten’s “4+1” model [31]. The template uses three viewpoints, conceptual, logical and physical, which approximate to COPA’s functional, conceptual and realization viewpoints. For clarity, we will use COPA’s terminology in the remainder of this chapter. The SAD template informally describes the information that should be included in each viewpoint, each of which is divided into static and dynamic parts.

The functional view in an instantiated SAD is application-specific, containing the requirements and variability model for the concrete product line. In contrast, the corresponding view in our reference architecture is primarily intended to orientate new users. It is limited to the illustration of its scope, shown in Fig 10.1, together with an elaboration of the capabilities, requirements and constraints listed at the beginning of this section. The generic elements of the realization viewpoint relate to the rules for the directory structure and permitted dependencies, together with coding standards, which are addressed in the company’s reuse standards and will not be elaborated further.

The primary focus of the reference architecture is COPA’s conceptual viewpoint. Here, we require a structural model that is abstracted from concrete architectures and from particular applications, while still being sufficiently specific to address issues of resource requirements and component scoping, introduced in Section 10.2. The texture of the reference architecture must identify the decision topics that must be addressed with regard to behaviour. In practice, the granularity of the structural model also had to be fine enough to express the alternative policies in the architectural texture, e.g. to be able to express that alternative policies allocate responsibilities to different components.

As noted by Muller and Hole, the reference architecture can be informed by proven concepts and known problems in existing architectures [25]. Given that it was not known at the outset what information the reference architecture should contain, nor how it should be structured, this approach was taken to develop insight incrementally. This was first by mining reusable architectural information from previous developments and then by structuring this information into a reference architecture that can be used from the beginning of a new development, as will be described in the following section.

10.5 Developing a reference architecture for the CE domain

Given that few concepts from information processing can be applied to embedded software, generally-recognised concepts for embedded software are only found at
the level of state machines and the services offered by real-time operating systems. These concepts are too low-level for the early stages of architectural development. Furthermore, the software architectures of concrete embedded products are highly influenced by the associated hardware architecture and the diversity that this introduces obscures the commonalities across product categories that could form a reference architecture to support early decision-making.

The main challenges in the development of our reference architecture were ascertaining what information should be documented and how it should be structured. For instance, while the inclusion of a structural model in a reference architecture is uncontroversial, what should it contain? The definition of “architecture” in IEEE 1471 includes "the fundamental organization of a system embodied in its components, their relationships to each other …” [32] but what should be the semantics of a component in a model abstracted from any specific product?

It was even less certain a priori what decision topics and other information should be included in the architectural texture. However, Kruchten states that one of the purposes of the architectural description is to “be able to understand how the system works” and to “be able to work on one piece of the system” [11]. Consequently, one way of identifying the necessary information is through the study of the root causes of failures that occurred during integration and to record and abstract those that resulted from insufficient architectural information. Furthermore, architectures should support evolution and a similar approach can be taken with components that have poor maintainability.

Our reference architecture was therefore developed in two phases. Firstly the problems encountered during the integration and maintenance of existing products were studied to obtain guidelines and checklists that could be used to review new concrete architectures. Secondly, the understanding gained here was used in the construction of a reference architecture that would support the creation of concrete architectures, providing support from the earliest phase of the development. The information flow is illustrated in Fig. 10.2.

Fig. 10.2 Information flow in the development of the reference architecture.
10.5.1 Mining information from earlier developments

As noted by Jackson, “the lessons that can be learned from engineering failures are most effective when they are highly specific” [33]. Therefore, when setting expectations for what will be achieved from the study of previous projects, we expect much more than merely reiterating that decisions should be recorded for each of the aspects identified in COPA, e.g. initialization, termination, fault handling [27]. However, it is challenging to gain insights on developments incorporating COTS components, given the limited information that is usually provided and the reluctance of suppliers to discuss problems in detail. Therefore, in searching for decision topics to include in the architectural texture, we first exploited the experience of multi-site development within a single company. Here, while architectural decisions must be explicit because of the limited communication between the sites, we were not hampered by IP issues. It was only after developing an understanding of the architectural issues that affect integration that the study was broadened to include developments incorporating third-party software and issues encountered during subsequent maintenance.

We began with a study of 900 failures that occurred during the initial integration of the sub-systems for a product-line of TVs, developed across multiple sites [34]. These sub-systems contained no legacy code, were developed according to a new architecture, and had not yet accreted changes as the result of evolution. Furthermore, all versions of code and documents, together with comments in the problem tracking system, were available. As such, these failures were an ideal candidate for identifying policies that should have been defined at an architectural level. Many of these related to component communication and synchronization.

![Fig. 10.3 Framework for reasoning about policies for communication and synchronisation.](image)

This study did not merely result in a catalogue of the problems encountered and the design patterns that would have avoided them. Such a catalogue would have
been difficult to reuse by different architects in the context of another development. Instead, a framework was created that not only identified decision topics for the observed problems, but also identified new problems decision topics that could occur in other contexts. This framework is illustrated in Fig. 10.3.

The key to achieving this generalisation was the identification, for each architecture-related problem, the property (specific intent) that had been violated and the general nature of the interaction between the components (interaction context). Examples of interaction contexts are synchronous function calls and asynchronously-communicating state machines, which are straightforward for architects to recognise within any architecture. The generic intents are an abstraction of the specific intents, formulated in a way that they can be reinterpreted in different interaction contexts, thereby anticipating new decision topics. Table 10.1 shows examples of intents and their specializations.

Table 10.1. Example intents and their specializations for specific interaction contexts.

<table>
<thead>
<tr>
<th>Interaction Contexts</th>
<th>Intents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>Variables must be initialized before they are used</td>
</tr>
<tr>
<td></td>
<td>Designs should be insensitive to the order of completion of unconstrained activities</td>
</tr>
<tr>
<td>Notification Handling</td>
<td>Variables that will be read by a notification handler must be set before the handler executes.</td>
</tr>
<tr>
<td></td>
<td>The notifications of a specific event should be generated and delivered in the same order.</td>
</tr>
<tr>
<td>Power-up</td>
<td>Avoid cyclic dependencies between sub-systems during initialization.</td>
</tr>
</tbody>
</table>

For each intent, several alternative policies might be identified that can guarantee its satisfaction. The implementation of each policy is documented in terms of a design pattern. In some cases, the choice of policy is arbitrary but it must be consistent throughout the architecture. Inconsistencies may arise when incorporating third-party components.

As an example, the problem of notification handlers reading uninitialized variables, listed in Table 10.1, can arise when a server completes an action on one thread before the function call that requested that action, made on another thread, returns to its client. The reference architecture identifies three different policies that can satisfy this intent. However, these are mutually incompatible, so a global choice must be made.

More often, the choice of policy will be guided by the NFRs for the product. Here, Gross and Yu’s approach to selecting design patterns is used to illustrate the relative merits of the alternatives in relation to the NFRs [23]. We extend their notation by adding the intents to their softgoal interdependency graphs, together with the argumentation of how these are satisfied by each of the design patterns.

It may be that different choices will be made at different layers in the architecture. For instance, in addressing the problem of ensuring the correct ordering of notifications, listed in Table 10.1, there is a trade-off between performance and configurability. Rather than making a single decision for the entire product, a policy that favours performance may be selected for the lower levels of the
software and a policy giving greater configurability may be used at higher levels. The reference architecture is a vehicle for the requirements engineer to understand how the efficiency of different parts of the software contributes towards the overall product performance.

Fig. 10.3 shows how, for a concrete product line, the choice of policy would be recorded in the architecture. In contrast, the reference architecture would contain a decision topic with the design alternatives.

Having developed this understanding of integration issues and established a framework for structuring decision topics, several analyses were undertaken of the integration of software from external suppliers [35]. Since this software had been developed with no knowledge of the architecture of the products in which it was to be integrated, these studies revealed a much larger range of policy mismatches, e.g. in relation to resource management and error handling. These mismatches were included as new alternatives in the reference architecture.

Another important class of mismatches related to the scoping of the functionality supported by components, which caused some of the problems introduced in Section 10.2.2, for which further examples will be given in Section 10.6. These mismatches gave insight into the granularity required of the reference architecture’s structural model to be able to compare the scopes of components.

Finally, as the product line architectures were subjected to adaptive maintenance, further studies were undertaken of components having low maintainability. Although the principles of object-oriented design, as articulated by Martin [36], would have addressed these problems, these are less easy to apply in the C programming language, which dominates embedded software development. Furthermore, the flexibility that these principles support is often at the expense of performance, so guidance is required on their application. Here, it is crucial for architects and requirements engineers to have a shared roadmap to ensure that components are structured to support anticipated changes in requirements.

10.5.2 The organisation of the reference architecture

As introduced in Section 10.4, the reference architecture is primarily intended to support the creation of COPA’s conceptual viewpoint for the concrete product line [27]. Its purpose is to facilitate communication between its stakeholders: the requirements engineer, the software architect, the project manager and the test architect. It must support the mapping of requirements onto a structural model so that the resource usage of each requirement can be ascertained. It must define concepts, such as different styles of resource management, which facilitate reasoning about the product.

The principal organisation of the conceptual viewpoint is provided by the architectural structure and texture, as introduced in Section 10.4 and shown in Fig. 10.4. The architecture is documented as dynamic web pages, which link the
elements in the structure to pages that guide the user through the decision topics relevant for that element and its interaction with its immediate neighbours. These decision topics are underpinned by the orthogonal texture axis, which addresses the topics in greater depth but in a more abstract context, enabling consistent choices to be made throughout the product.

Additionally, given the variation in the conceptual views of existing architectures, a new user requires some orientation. This is provided through the scope of the reference architecture, which is an abstracted form of the COPA functional viewpoint [27]. It contains the context diagram, already shown in Fig. 10.1, which includes the mapping of examples of concrete actors and protocols, used in existing products, to the abstract representations used in the remainder of the architecture. It also lists the general capabilities of CE products, already described in Section 10.4, together with their typical dynamic behaviour at a product level and typical non-functional requirements. This assists the architect in interpreting the concepts of the framework in the context of their specific product.

The top levels of the architectural structure and texture will now be described in more detail. This will be followed by a description how they are linked through the lower levels of the reference architecture, such as through the recommended designs and decision topics.

**Architectural Texture**

![Diagram](Fig. 10.4 Primary organisation of the reference architecture.)
**Structure:** Normally a structural model shows a partition of the software that can be traced to concrete system elements [3]. However, as discussed in Section 10.3, our reference architecture is more generic. As shown in Fig. 10.4, the top level of the structure is the primary entry point to the architectural guidance, so its abstraction level must be high enough to be broadly applicable, yet concrete enough to be usable in practice. To achieve this, we have adopted Wirfs-Brock and McKeen’s responsibility-driven design (RDD) [37]. In RDD the role of a class is defined as “a set of related responsibilities”, which may be abstracted from their specific functionality. The structural model, illustrated in Fig. 10.5, identifies the roles that are normally present in a CE product. Following the RDD approach, the roles are annotated with their purpose and responsibilities. In use, RDD provides the method for mapping the requirements for a specific product onto the responsibilities that defines these roles. This mapping is assisted by annotating many of the roles with examples from concrete architectures that would be familiar to all architects in the company.

The structural model, shown in Fig. 10.5, is sufficiently fine-grained to be able to compare the scopes of different components and to distinguish between different sets of decision topics identified in the studies in Section 10.5.1. For example, although the purposes of the three services roles shown in Fig. 10.5 are comparable, they have different behavioural characteristics and are therefore distinct in the model.

![Fig. 10.5 Structural model of the reference architecture with an example role description.](image-url)
if integration problems are to be avoided. This allocation is clarified in more detail in the recommended design linked to the role, an example of which is in Fig. 10.6, where the collaborations of the roles, the other element required by the RDD method, are made explicit. These relationships are not present in the top-level structural model since they can vary, depending on the decisions made. For instance, the invariant manager in Fig. 10.5 might be implemented as a single state machine or, as described by van Ommering, by protocol handlers integrated into each component in the next layer down [38].

![Diagram of recommended design for applications roles and their interfaces to services, highlighting one of the three domain-specific patterns used in this design.](image)

**Texture:** Our model largely follows the classification used in POSA4 [17], but with the contents adapted from the concerns of distributed systems to those of embedded software. It contains guidelines on both the structure of the software, e.g. for interface and component partitioning to support variability and evolution, and on the rules or decision topics for behaviour. Some examples of the categories of behavioural guidelines are:

**Synchronization:** This category includes the rules or decision topics identified in the studies of integration failures described in Section 10.5.1. Here the rules or decision topics are classified according to their interaction contexts and are therefore reusable throughout the architecture.
State behaviour: This extends the taxonomy of modal behaviour in POSA4 [17] to cover the much larger set of state-related patterns referenced in the architecture, providing both consistency in their description and a broader perspective on the options available to architects.

Resource management: The different classes of resources are an important concept to help the requirements engineer to understand how architectural choices affect feature interaction. The category identifies three different classes of resource management policies, namely synchronized, prioritized and virtual, and the issues that must be considered with each class. These definitions and arguments are widely-referenced throughout the architecture.

Having described the top level of the architecture, we will describe how the structural model is linked to more detailed recommendations and guidance on architectural decisions. Each of the roles is hyperlinked to a textual description of the most relevant NFRs, a recommended design approach and guidance on selecting design alternatives, as illustrated in Fig. 10.7.

![Figure 10.7 Example of steps in the guidance through architectural decisions.](image)

The recommended design approach is expressed in terms of a UML component model, in which the roles and responsibilities in the top-level structural model are expressed at a finer grain. Unlike the top-level structural model, component connectors now indicate the collaborations between components. Fig 10.6 shows
an example component model, including a tooltip describing the purpose and responsibilities of one of the components. More detailed information is provided through design patterns, documented in a conventional form as part of the orthogonal architectural texture. The model allows the user to display the different patterns in which the components collaborate, one of which is shown in Fig 10.6.

While the individual pattern descriptions are independent of where in the structure of the architecture they might be applied, Fig. 10.7 shows how the recommended design is also linked to a description of the concerns addressed by that design in its particular context. These concerns will include a re-interpretation of the intents, identified in Section 10.5.1, for the current role in the architecture.

This is followed by a diagram illustrating the process for reviewing the decision topics in the light of the particular product requirements or the characteristics of pre-existing components. A UML activity diagram is used, showing the tasks and resulting work products. Each task in this model is hyperlinked to decision topics, such as the alternative policies for handling notifications, introduced in Section 10.5.1. Each topic has a detailed discussion of the forces involved and examples of decisions taken in earlier product developments, obtained from the studies described in Section 10.5.1. Throughout the guidance, hyperlinks are made to definitions and discussions in the architectural texture, where the issues are described in a more general context. This both allows consistent decisions to be made throughout the product and reduces the amount of material that must be presented in the context of each of the individual roles in the architecture.

Finally, the design rationale for each decision topic is presented using our extension of Gross and Yu’s approach to selecting between design alternatives, based on their support for different NFRs [23]. As described in Section 10.5.1, we add the intent as a goal that must be satisfied by all design alternatives.

Early trials of the use of the architecture confirmed that the approach of beginning with a recommended design had a shallower learning curve compared with that of a pure pattern language, such as that in POSS44 [17], in which there are no default decisions. Such pattern languages require that the architect has a good initial grasp of many abstract concepts.

A general principle behind the use of web pages to document the reference architecture is that a user should be provided with the essence of recommendations in the first instance, but that it is easy to drill down and get more details when required. For example, Fig. 10.6 shows both tooltips and dynamic content, used for the overlay of different design patterns. The latter provides navigation to other pages through hyperlinks. Indeed, the ability to provide details on demand is the key to presenting a full design rationale in a compact and comprehensible form.

Having described the development and organisation of our architecture, the following section describes how it can be used during requirements engineering.
10.6 Usage of the reference architecture by requirements engineers

Our reference architecture both allows requirements and architectural decisions to be assessed together and prompts the requirements engineer to elicit how particular issues, e.g. resource management, should be handled. In this regard, the architecture also implicitly includes some of the **concerns** of Jackson’s Problem Frames [39], another element of the “Twin Peaks” approach. Fig 10.8 is an informal activity diagram that illustrates the role of the reference architecture in requirements-related activities.

![Fig. 10.8 Role of the reference architecture in requirements-related tasks. Tasks involving requirements engineers have a solid outline.](image-url)

---

**Fig. 10.8 Role of the reference architecture in requirements-related tasks. Tasks involving requirements engineers have a solid outline.**
The diagram includes three feedback loops from the architecture to the functional requirements specification:

A. Revise the requirements, having reviewed the architectural decisions that would be required to satisfy the related NFRs. This might discard requirements with a high technical risk.

B. Identifies cases where contention for resources restricts the concurrent availability of features. Where features cannot be active concurrently, new requirements may be added relating to the transition between those features.

C. Where third-party components are to be used, low-priority requirements are removed if they are only supported by components that are architecturally incompatible with the remainder of the product.

The use of the reference architecture will be illustrated by an example of the integration of the PictBridge protocol into a mobile phone. This will show how feature interaction can be detected and resource management policies assessed.

PictBridge [40] is a standard that allows a user to select, crop and print photographs, using a camera connected directly to a printer, without requiring a PC. The standard only addresses the protocol between the camera and printer, and not the camera’s user interface or how the feature should be implemented in the camera.

Consider establishing the requirements the first time that this feature was integrated into a mobile phone, where it is to be implemented by a COTS component that has previously only be integrated in a conventional camera. The requirements engineer must:

- Determine a complete set of end-user requirements for the PictBridge feature.
- Identify potential feature interaction with the remainder of the phone’s features and identify how they can be resolved satisfactorily.

These aims are addressed, with reference to Fig. 10.8, with the following sequence of activities:

- **T1**: Map the functional requirements of the PictBridge feature onto the roles in the structural model. The component implementing the protocol is an example of a procedural service (see Fig. 10.5), which is one that executes a series of actions, normally running to completion. The PictBridge component will need access to the hardware drivers for the USB interface and the memory in which the photographs are stored. In addition, the feature will require a user interface.

- **T4**: Map the scope of candidate PictBridge COTS components onto the structural model.

  - Survey the COTS components that implement the PictBridge feature.
  - The scope of each promising candidate is identified from studying the features it supports and its interface specification. For instance, are its provided interfaces restricted to the PictBridge protocol, or does the component also provide some user interface functionality?
• **T7: Compare the scope of the candidate PictBridge COTS components with those of other features.** Will it be possible to maintain a consistent user interface across all features? If not then **T9: discard incompatible candidate components.** If these components also support some unique functionality that cannot otherwise be implemented, **T10: revise the feasible functional requirements.**

• **T2: Identify resource conflicts.** Identify the features that could be active concurrently and detect feature interaction. Since the user interface of most phones only permits one application to be selected at a time, we are primarily concerned with interference from features of the phone that are autonomous applications (see Fig. 10.5), i.e., features that make calls to the services without having been explicitly selected through the user interface. For a phone, these are incoming telephone calls and text messages. How should the product react when a call or message is received when the PictBridge feature is active?

  – What are the consequences for the user interface? PictBridge implementations on cameras normally retain control of the user interface while the photos are being printed. Would it be possible to continue printing in the background on a phone, so that it could continue to be used for other purposes? The architectural guidance for the user applications and their interface to the services includes a recommended design for managing the transfer of resources between applications, shown earlier in Fig. 10.6. Do the available components have the necessary synchronization functions to implement such design patterns?

  – Considering the lower levels of the structural model, can both features be active concurrently? Does the file system support concurrent access from multiple applications and are there sufficient memory and processor resources to support both features?

  – Based on this analysis, **T5: revise the requirements for concurrently active features.**

• **T8: Review the COTS components for mismatched policies.** The policies for how the feature should be activated and terminated should be consistent with those of other features. Many cameras activate the PictBridge feature only when the camera is switched on while connected to a printer, whereas a phone user would not expect to have to switch the phone off and on in the same way. Mismatches often occur in state behaviour when components developed for one category of product are integrated into a product of another category. Mismatches may also occur in the selection of design alternatives, such as those for handling notifications, introduced in Section 10.5.1. Such mismatches can be detected by architects during the later steps in Fig. 10.7 and may require acceptably complex glue code to integrate the component into the remainder of the system. Again, following this analysis, **T9: discard incompatible candidate components and, if necessary, T10: revise the feasible functional requirements** to remove those only supported by the discarded components.
A benefit of using the reference architecture, even when a concrete architecture already exists for the mobile phone, is that it supports the comparison of the scopes of COTS components implementing different features. This makes it easier to detect feature interaction and identify requirements for resource management.

### 10.7 Conclusions

Establishing the requirements for a CE product has many challenges, such as identifying user needs or desires for a diffuse market, identifying features that will differentiate a product from the competition, finding the right price/performance points and developing a simple and intuitive interaction style.

Much of the requirements specification for a CE product addresses the interaction between features, either because they are active concurrently or because they can be activated spontaneously by events in the environment. Even if all the software were to be bespoke, support is required to identify the sources of feature interaction, which arise both from resources that cannot be shared and from performance constraints. These problems are compounded when features are implemented by COTS components, which may initially have been developed for different product categories, having different overall requirements.

We have developed a reference architecture that covers a broad range of CE products. The breadth is required so that it can support the addition of novel features that were not anticipated at the time of the architecture’s creation, to enable the exchange of best practice between development groups and to promote reuse across product categories. The abstraction level is set high enough to cover this broad scope, while still being concrete enough to have clear relevance for product development. The architecture addresses the lack of consensus on architectural concepts in the CE product domain by proposing a structure of roles with recommended designs, while providing guidance on alternative design choices. This is based on architectural information mined from earlier multi-site and COTS-based developments.

Architectural texture provides consistency, with design guidelines that can be used throughout the architecture. This information would also facilitate the creation of variants of the architecture for other industries with similar technical characteristics, e.g. automotive engine management or medical image acquisition.

Because it provides an initial structure, our reference architecture is of benefit from the beginning of the requirements phase when identifying resource constraints or conflicts. For COTS-based developments, the architecture provides a framework for comparing the scope of functionality of COTS components, both to identify which features can be active concurrently and to ensure a consistent interaction style.

Our architecture was developed in the context of a CE manufacturer with a broad product portfolio. A company with a narrower range of products might be
tempted to move along TOGAF’s architecture continuum, representing more application-specific information. However, this would give little support for the integration of novel functionality.

The broad scope of our architecture is also valuable for COTS component suppliers, for whom it can be difficult to anticipate all the architectures used by potential customers. The design alternatives used in our reference architecture give an insight into what might be encountered. The architecture also provides a vehicle for detailed discussions with customers without either party exposing their IP, which will be of increasing value as the CE industry transitions away from vertically-integrated companies towards supply chains or ecosystems.

References