Building Product Populations with Software Components

Ommering, Robbert Christiaan van

Chapter 8

Horizontal Communication

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

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

8.1 Introduction

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

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

But they also have some typical properties of their own:

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

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

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

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

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

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

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

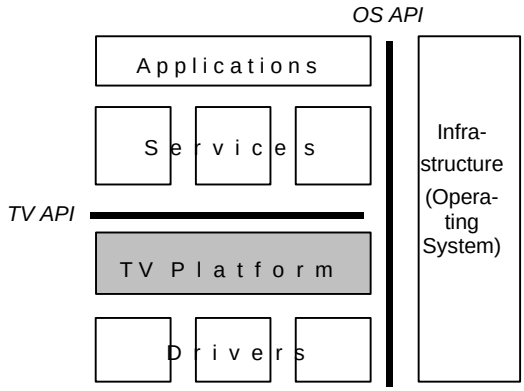

Figure 55. The software architecture in a television

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

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

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

8.2 Component Technology and Architecture

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

8.2.1 Koala

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

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

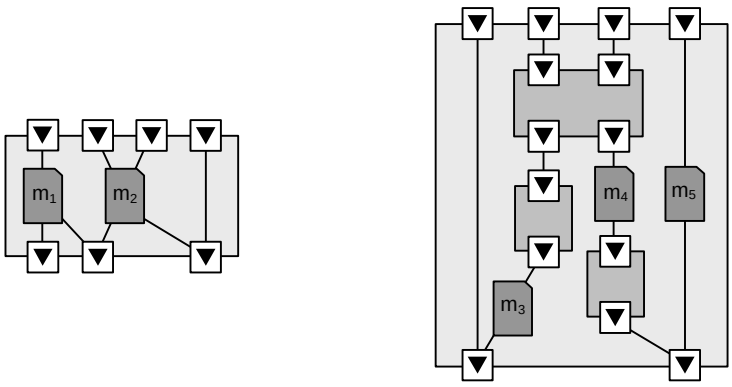

Figure 56. A Koala basic and compound component

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

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

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

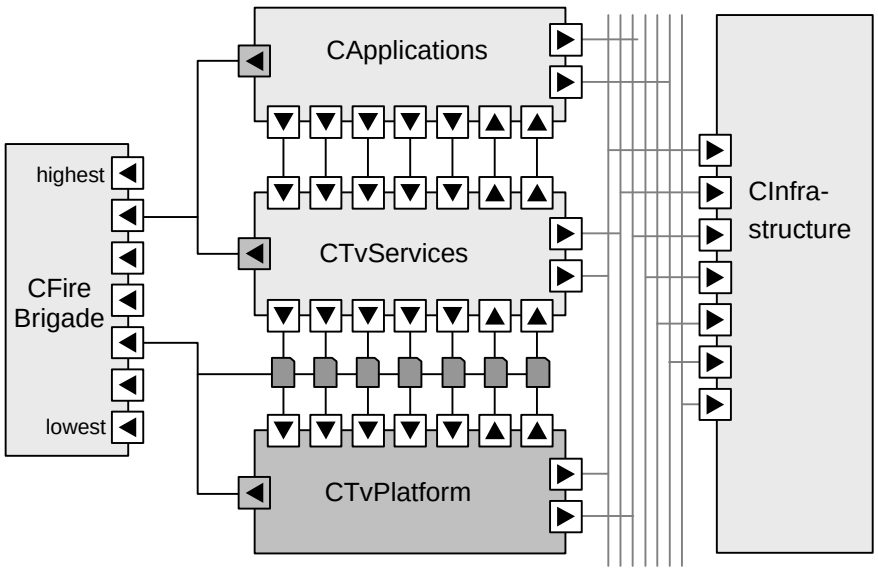

Figure 57. TV architecture in Koala

8.2.2 Modular Architecture

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

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

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

8.2.3 Execution Architecture

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

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

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

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

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

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

8.3 The Control Problem

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

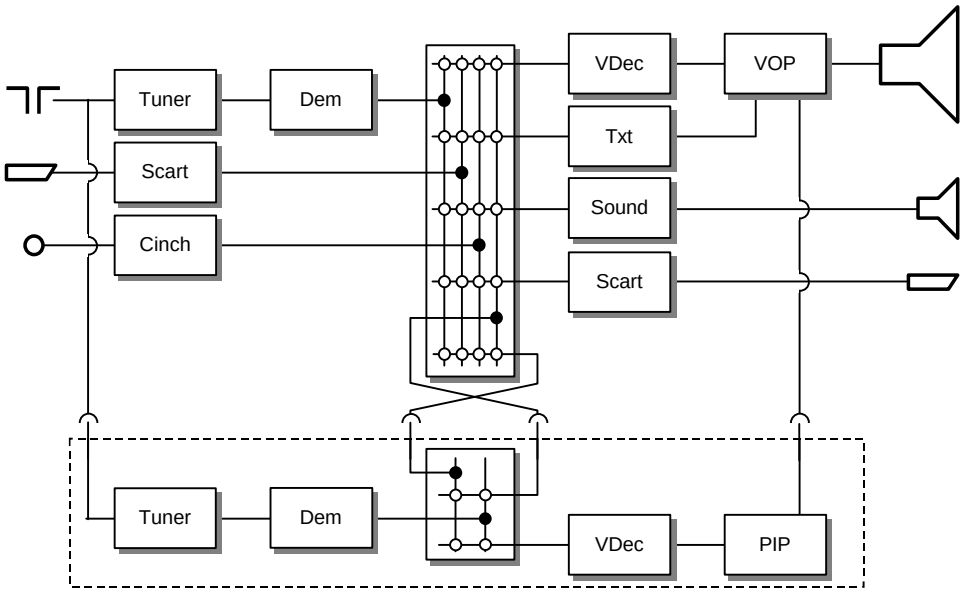

Figure 58. A simplified television

8.3.1 A Simplified TV

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

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

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

8.3.2 Typical Control Tasks

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

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

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

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

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

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

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

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

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

8.3.3 The Problem Statement

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

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

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

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

8.3.4 Traditional Solutions

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

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

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

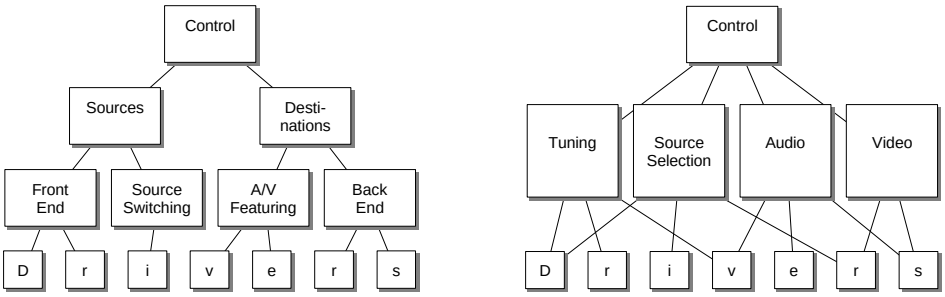

Figure 59. Traditional solutions to the control problem

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

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

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

8.3.5 The Proposed Solution

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

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

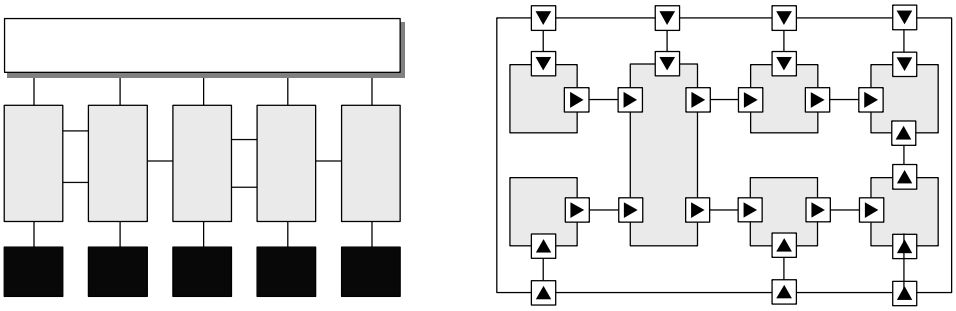

Figure 60. Proposed solution: horizontal communication

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

8.4 Horizontal Communication

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

8.4.1 Synchronous Drop Request and Restore

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

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

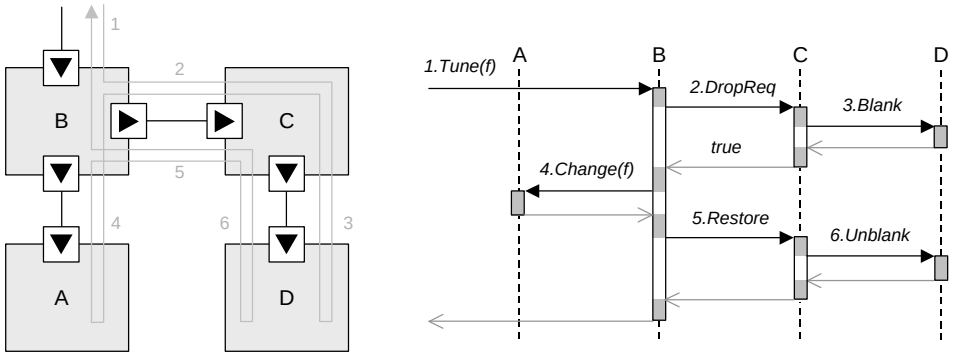

Figure 61. The synchronous drop/restore protocol

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

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

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

8.4.2 Asynchronous Drop Request and Restore

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

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

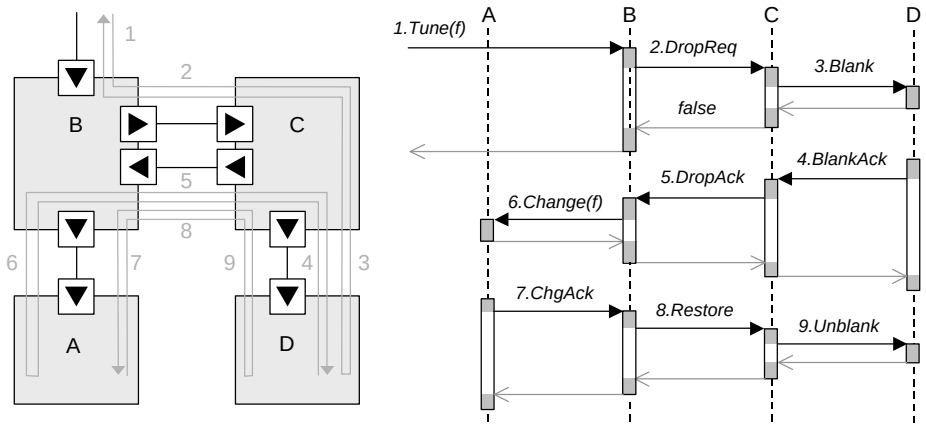

Figure 62. Asynchronous drop request and asynchronous restore

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

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

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

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

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

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

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

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

8.4.3 Asynchronous Drop Request / Synchronous Restore

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

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

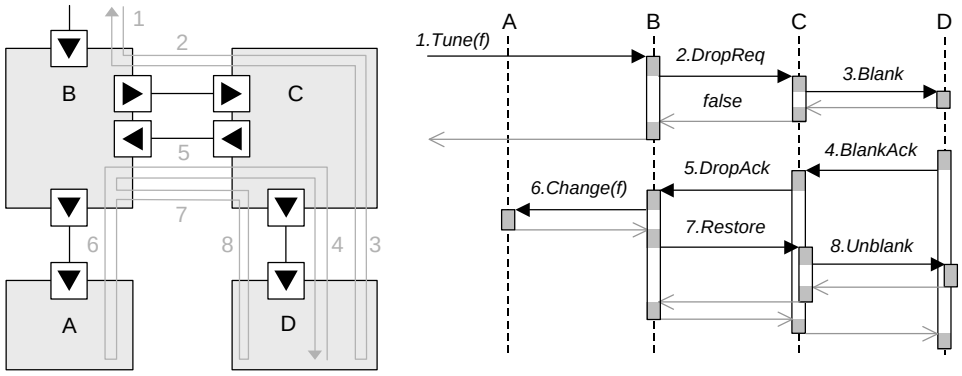

Figure 63. Asynchronous drop request and synchronous restore

8.4.4 The Fork

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

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

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

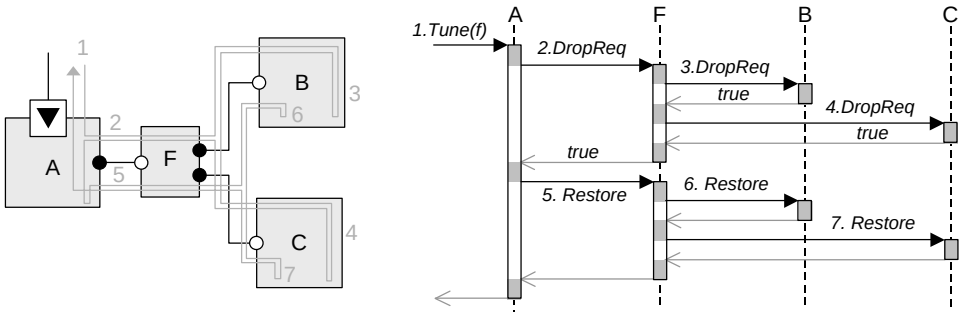

Figure 64. A fork with synchronous drop requests and restores

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

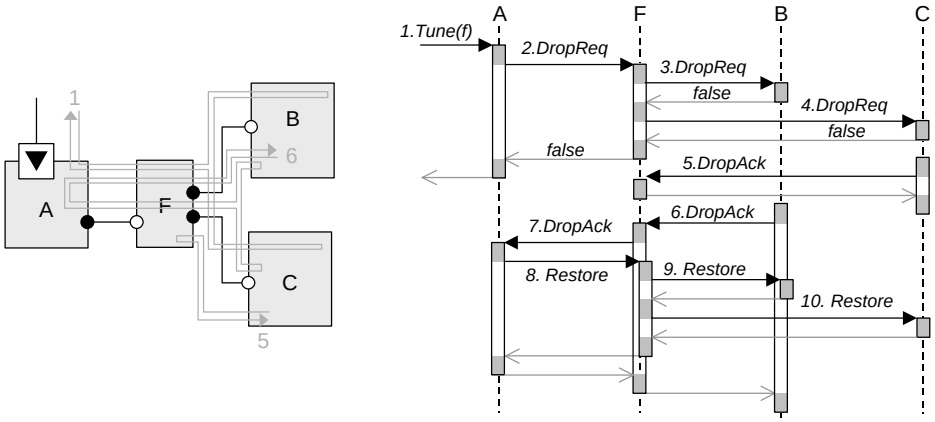

Figure 65. A fork with asynchronous drop requests and synchronous restores

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

8.4.5 The Switch

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

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

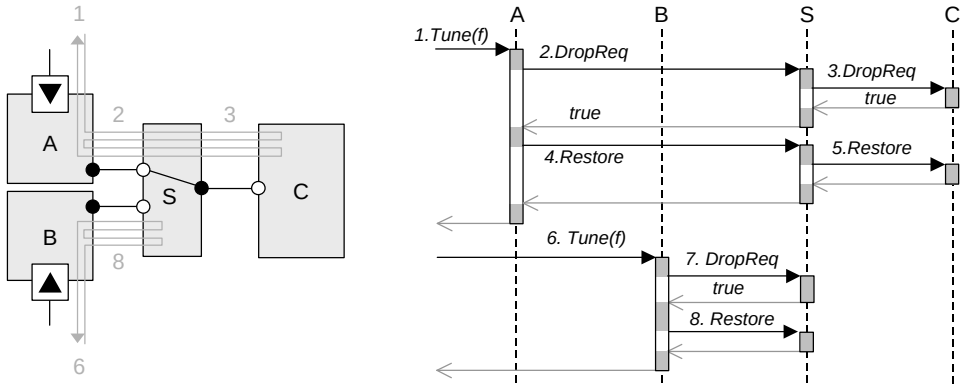

Figure 66. A switch forwarding the drop request protocol

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

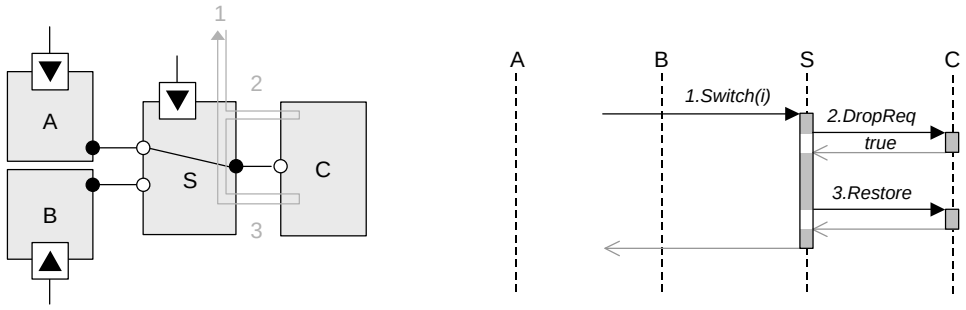

Figure 67. A switch issuing the drop request protocol

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

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

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

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

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

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

8.4.6 The Matrix

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

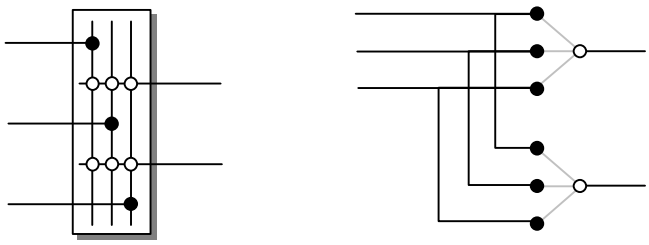

Figure 68. A matrix modeled as forks plus switches

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

8.4.7 Source Selection

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

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

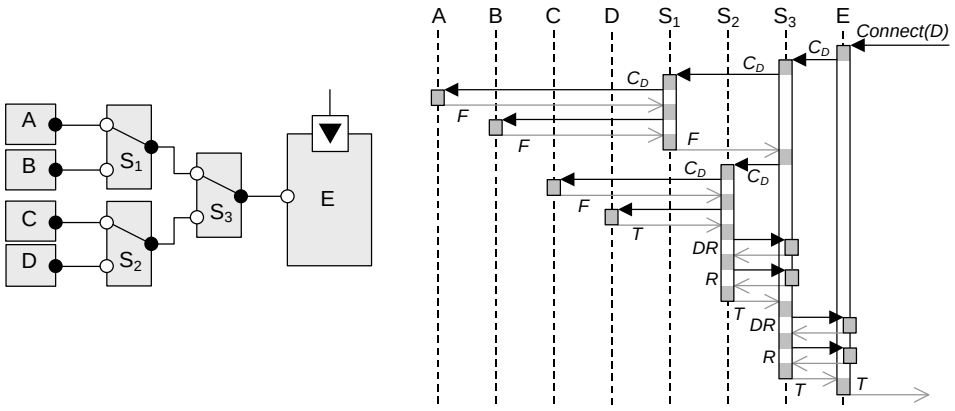

Figure 69. A more complicated switch topology

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

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

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

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

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

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

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

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

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

8.4.8 Properties

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

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

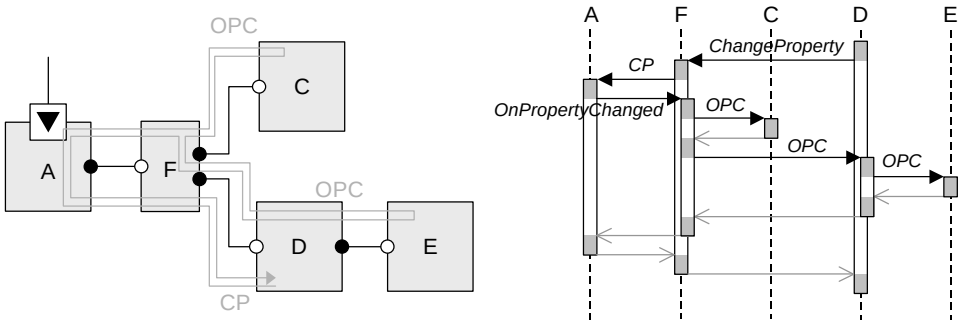

Figure 70. Communicating properties in a branched signal path

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

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

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

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

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

8.4.9 Capabilities

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

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

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

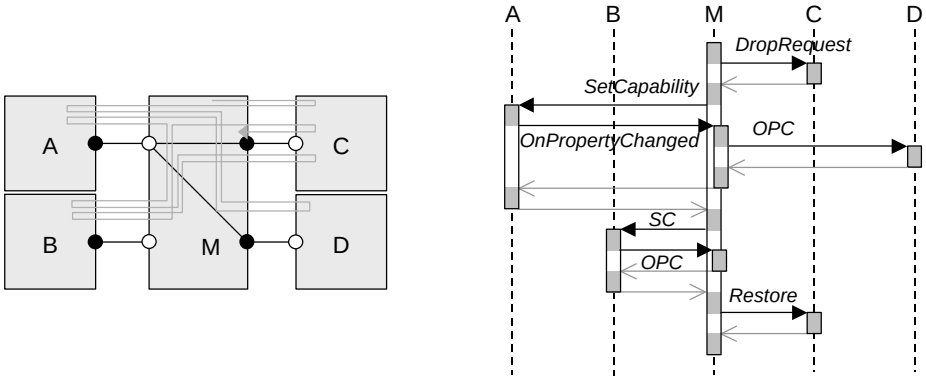

Figure 71. Changing capabilities on the signal path

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

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

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

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

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

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

8.5 Introducing the Protocol

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

8.5.1 Simulating the Protocol in Visual Basic

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

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

The VB controls and forms provide a *view* on these models. Figure 72 shows an example configuration consisting of a single tuner (named Tuner1) and a single output component (named Hop1), connected by a wire. The tuner control shows the values of its two state variables, while the Hop control simulates an image. This

image contains a number that represents which source is currently connected to the Hop (in the simple configuration of Figure 72, this cannot change).

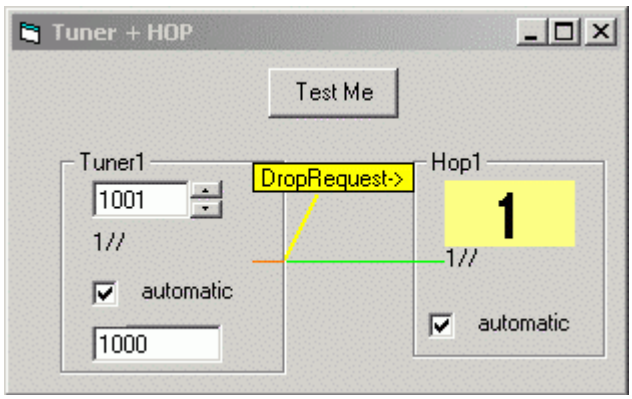

Figure 72. An example simulation

With buttons and check boxes we can simulate the actions that the application can perform. We can for instance change the desired frequency of the tuner (which has just been set to 1001 in Figure 72). We can also interactively change the behavior of the simulated components. The 'automatic' check boxes in Figure 72 can for instance be used to make the simulated devices handle drop requests and restores synchronously (checked) or asynchronously (unchecked).

To connect an output of control *A* to an input of control *B* in our simulation, we insert an intermediate object *S* of type *Signal* (see Figure 73). This object is in fact owned (and created) by *A*, and assigned to a variable representing the input in *B* by the configuration C (the form). Note that in our real implementation, we do not have these signal objects.

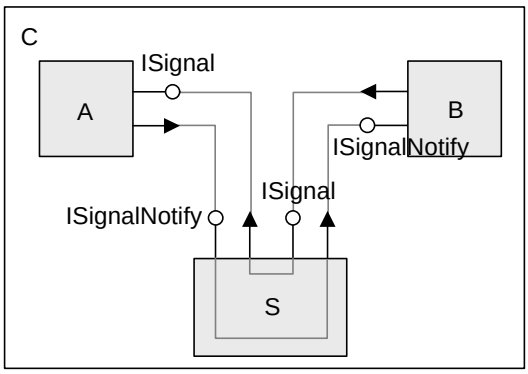

Figure 73. A signal object between two components

The signal object implements two interfaces, one of type *ISignal* and one of type *ISignalNotify*. The down-stream device uses the *ISignal* interface for up-stream communication. The signal object implements this interface by calling the

corresponding interface type in the up-stream device. Similarly, the up-stream device uses the *ISignalNotify* interface for down-stream communication. The signal object implements this interface by calling the corresponding interface in the down-stream device.

To make composition extremely simple, we use VB6's reflection capabilities. A control can query its container for its own position. If we create graphical representations for the inputs and outputs of a component, then the control can find the absolute coordinates of these. If we layout devices such that inputs are co-located with outputs, then we can let a control ask the form for the objects to be assigned to each of its inputs. The form passes the coordinates to each of the other controls, until it obtains the actual signal object involved. This allows us to let the controls wire themselves at initialization time. To make it even more convenient, we introduced a wire as an extra 'pass-through' device, as can be seen in Figure 72.

The next problem is how to visualize the different up- and down-stream calls. For that, we add another control to the form: a yellow flag as shown in Figure 72. Whenever a signal object processes a command, it sets the position of this flag to the location of the output that owns the signal object. The text of the flag is set to the command being processed. When the simulation is run, this results in a flag that moves across the form just as the flow of control is traversing the components. The protocol can be single stepped to study it more carefully.

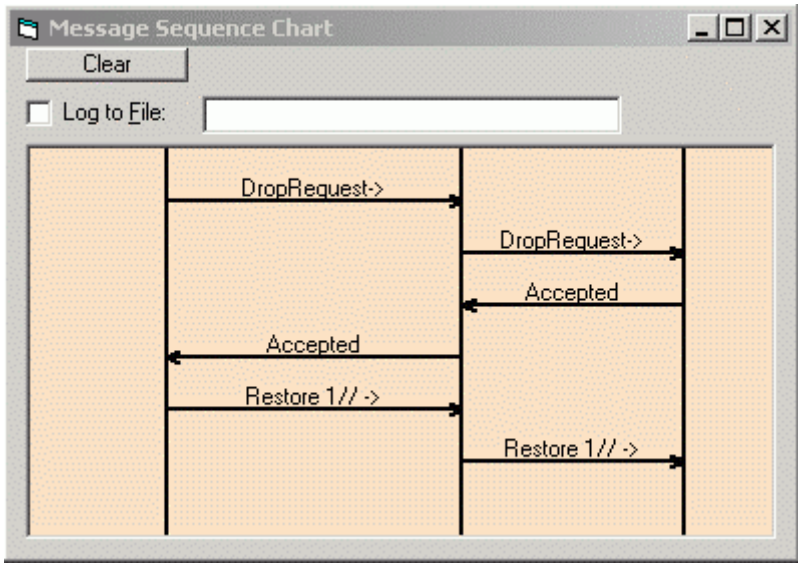

Figure 74. A message sequence chart

Another way to visualize the communication is to generate a message sequence diagram on the fly. To obtain this, we can also instrument the signal objects. First we assign a horizontal position to each control depending on the top and left coordinate of each control. Then we draw an arrow for each message that passes

each signal object, labeled with the name of the message. The result for a simple tune operation can be seen in Figure 74.

The prototype with the two techniques for visualization turned out to be sufficient to convince our designers that the protocol would work. In fact, when writing this paper a few years later, we still use the simulation to verify all examples that we have shown. Our simulation can be downloaded from [71].

8.5.2 Testing the Protocol

Running and verifying the protocol by hand is one way to gain confidence, but it is very difficult to walk through all possible scenarios by hand. We therefore added a random test generator to each form (activated by a button labeled 'Test Me', see Figure 72). This generator randomly selects an action that is allowed in the current state of the protocol, and calls it. Assert statements in the code then verify whether the protocol behaves as expected (using for instance the traffic light paradigm of section 8.4.2). The generator repeats this in a continuous loop until cancelled by the user.

One could argue that the quality of this test depends on the quality of the assert statements in the code, and that is true. We used this test to quickly find a number of errors in our initial implementation. After correcting them, the test now runs for hours without halting. While this gives a fair amount of confidence that the protocol behaves as it should, it still does not provide us with certainty.

8.5.3 Formal Verification of the Protocol

We have started with a formal verification of the protocol, using the Labeled Transition System Analyser (LTSA) [53]. Figure 75 shows a simplified version of the protocol with only the drop request ('dr'), drop acknowledge ('da') and restore ('re'). In the figure, '.t' stands for returning true, '.f' for returning false, and '.r' for a void return. We have used this technique to verify simple properties, such as that the protocol will indeed ensure that the screen is blanked for a configuration with a tuner and an output component.

```
HORCOM = ( dr -> ( dr.t -> re -> re.r -> HORCOM
                 | dr.f -> da -> ( re -> re.r -> da.r -> HORCOM
                                 | da.r -> re -> re.r -> HORCOM
                                 )
                 )
         ).
```

Figure 75. Part of HorCom specified in LTS

While the use of LTSA was extremely helpful to increase our insight into the protocol, the full discussion of the use of LTSA to model HorCom falls outside the scope of this paper. Besides, we have only succeeded in modeling a small part of the protocol; modeling the full protocol will take quite some effort. We invite

readers of this paper to help us in this analysis, or to come up with alternative ways of describing and/or verifying the protocol. One such an alternative technique was used by Uchitel *et al.* [107], who generated a model of part of our protocol from a set of scenarios.

8.6 Experiences

In this section some of our experiences with the protocol are listed.

8.6.1 Managing Complexity and Diversity

We have implemented the protocol in the software architecture that is currently being used for all of our analogue up-market televisions, and that is starting to be used for our mid-range TVs as well. The protocol consists of a single pair of interfaces, one for up-stream and one for down-stream communication, with six respectively four functions. Of these ten functions, three handle drop/restore, two source selection, three properties and two capabilities. Over 60 components implement these interfaces, with in total 700 signal inputs and outputs. The protocol currently handles 30 signal properties and associated capabilities (this is small enough to implement a set as a 32-bit word).

With a single set of components, we are able to create TV platform software for 15 different topologies of the hardware. Using Koala's partial evaluation mechanism, we create ROM images for five groups of products; the rest of the diversity is handled at run-time. Note that the decision which products to group into a single image is largely determined by the logistics of the factory, not by the software architecture! We would like to give some examples of different television topologies, but the complexity of those are such that they cannot be included in this paper. Hopefully, Figure 58 gives the reader at least some impression of the complexity.

Although we have no exact metrics on the performance of the protocol, we have experienced no major performance problems. The platform control typically consumes only 25% of the CPU cycles. Although the load approaches 100%, the duration of a channel change is still not determined by software, but rather by hardware circuits catching up with the new signals.

During a single channel change, we observed 370 calls of functions in our protocol with a maximum nesting of 11, which is roughly the maximum length of the signal path of a particular platform. Remember that Koala can optimize certain function bindings. We do need sufficient stack space for function call nesting, since also within a component functions may call each other. We have calculated that in certain situations the nesting may be up to 50 levels deep. But in all fairness it should be said that traditional designs (as depicted in Figure 59) also tend to have deep call chains due to the many levels of management.

A typical experience before introducing the protocol was that solving a bug in the TV platform software resulted in the introduction of more than one new bug – no person could oversee all of the consequences anymore. This has been solved now: control aspects have become much more local. However, there is also a downside to this. We have in fact replaced a classical top-down control system with what is essentially a distributed control system. Such systems are known to be harder to understand and debug if something is wrong.

8.6.2 Direct Function Calls

The reader may wonder why we use direct function calls instead of sending messages. In our systems, the typical overhead of sending a message is 100 µs. Strangely enough, this overhead does not reduce significantly when faster processors are used. Such processors typically have more registers to save, and although the speed of processors doubles every two years, the speed of memory increases much more slowly. To handle the latter, caches are often introduced, but context switches often result in cache misses. To put this in perspective: we have measured over 350 function calls during a single channel change in a TV.

There is another way of looking at our protocol. Traditionally, we draw the signal flow from left to right in our diagrams, and the control flow from top to bottom. But we could also draw the signal flow from top to bottom and align it with the control flow: a screen needs an output processor, which needs a decoder, which needs a source selector, which needs a demodulator, which needs a tuner, which needs an antenna (see Figure 58). In that sense it is perfectly logical that a down-stream (read: higher) component asks an up-stream (read: lower) component to select a source. Similarly, when an up-stream component obtains new information about the signal, it should notify the down-stream components about this. This is why we call our interfaces *ISignal* and *ISignalNotify* in Figure 73.

This explains why we use direct function calls up-stream. Most software architectures, however, decouple notifications (our down-stream calls) by using a kind of message passing. We could do have done that as well, but we have found that using direct function calls here is not harmful, provided that one uses some simple rules. In our protocol, the *Restore* can be nested in a *DropAcknowledge*, the *DropRequest* can be nested in a *Connect*, and an *OnPropertyChanged* can be nested in a *Connect*, a *ChangeProperty* or a *SetCapability*. Other forms of nesting are not allowed, nor is nesting within nesting.

8.6.3 Using Templates

Although the protocol is fairly simple, it is by no means trivial. If a dozen developers or more have to implement the protocol, chances are high that at least one of them will make a mistake somewhere. This may bring down the entire

application – a chain is as strong as its weakest link! For that reason, we sought to decouple the protocol implementation from the components.

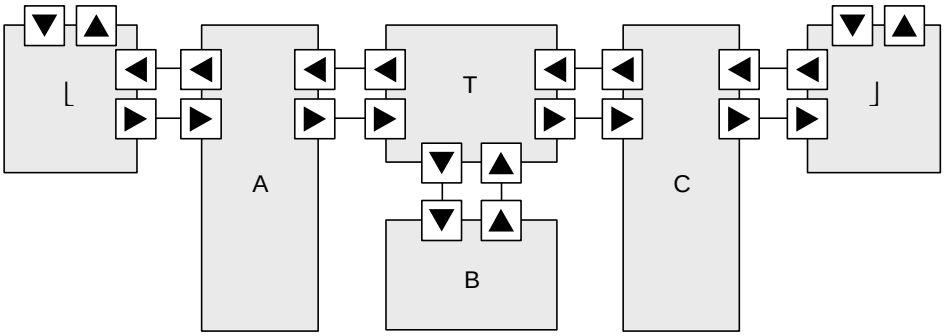

Figure 76. Separating HorCom from components

This turns out to be quite possible. We shall not discuss it in detail here, but essentially we create reusable T-pieces (component *T* in Figure 76) that can be inserted in the signal chain and placed on top of a more traditional component (*B*). The vertical interfaces of *T* are more easily understandable for most software engineers. Moreover, the developer of *B* need not worry about parts of the protocol not relevant for *B*, such as for instance source selection and capabilities.

The T-piece allows us also to insert traditionally programmed (read: legacy) control components into our signal path. We also wrote an L-piece (and its mirror image, both shown in Figure 76) that allows us to convert (chains of) components communicating horizontally to a more traditional vertical API.

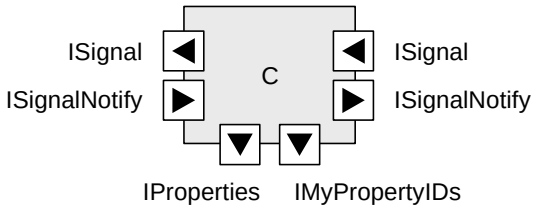

Figure 77. Making properties bindable

If all of the horizontal communication implementation is localized T- and L-pieces, then it is easy to change these to experiment with different strategies. We tried for instance to replace the change property strategy, where each property change traverses all components in the signal path, by a subscription mechanism. This greatly reduces the ‘call storms’ that occur whenever a property changes, but it replaces these with other ‘call storms’ whenever the topology changes. Although we certainly saw the advantages, we decided not to implement subscription in our second version of the protocol.

As a final remark on the reusable T-pieces, they also allow us to implement generic tracing facilities, in the same way as that we extended the signal objects in our simulation (Figure 73). Without T-pieces, tracing is still possible by adding glue modules or small components in the Koala bindings of inputs to outputs, but it is more cumbersome.

8.6.4 Binding Properties

The reader may remark that although we have localized measurement and use of properties, still global knowledge of properties seems to be embedded in the components. Furthermore, it seems difficult to get an overview of which component deploys which property. Most components only measure or need access to a small number of properties. How can we make that explicit, and sufficiently decouple components from the global list of properties? To achieve this, we let components require *two* additional interfaces to handle properties (see Figure 77). One is of type *IProperties* and contains general functions for dealing with properties, capabilities and sets of these. The other is of a type specific for the component (*IMyPropertyIDs* in Figure 77), and contains the IDs of the properties required by this component. We can bind this subset to an interface providing all IDs of all properties in the system, using the interface subtype capability of Koala. We can also rename certain properties in the process, using Koala's function binding. This provides us with much greater flexibility with respect to the measurement and use of properties.

8.6.5 Three Types of Reuse

The horizontal communication protocol has helped us significantly in managing the diversity of a product family of up-market televisions. Software components were coupled one to one to hardware components, and could be reused whenever the hardware was reused. Then came a new generation of mid-range TVs, where all hardware was different. Our managers were disappointed that we could not directly reuse the up-market TV software. We have several answers to this.

For our first answer, recall Figure 55. We split the software into three parts, one depending on the computing hardware, one on the domain (TV) hardware, and one hardware-independent. We envisage the majority of reuse to happen in the third part (services and applications), and we expect that part to grow significantly in the near future. Unfortunately, it is the TV platform that is the most complicated to build currently (as explained in section 8.1), and that part had to be rebuilt for the new generation of mid-range TVs. The fact that the computing hardware was also different did not help much either.

Still, the *interfaces* of the TV platform and infrastructure could be kept the same, and although this does not show up in measurements of reused lines of code, it certainly has a large positive impact on development time. Unfortunately, in our

development much lead-time is lost due to changes of or uncertainties in hardware specifications, so it is difficult to make the gain obtained by reusing interfaces explicit.

While we see the first type of reuse as reuse of services and applications, the second type of reuse is the reuse of software coupled to hardware as enabled by our horizontal communication protocol. So as long as hardware is being reused, the software can be reused. In practice, there is much hardware reuse in *space* (different products) but less in *time*. Although new functions shift over time from high-end TVs via mid-range to low-end TVs, their concrete implementation does not remain the same, due to miniaturization and other cost-saving measurements.

This brings us to a desired third type of reuse: of components within the TV platform. We already gave an example of such components in Figure 76, where we introduced reusable T- and L-pieces. Many other components can be made reusable too, such as power managers, channel tables and certain control algorithms. While we highly appreciate such reuse, the global structure of our TV platform software remains a mirror image of the (product specific) hardware architecture. The third type of reuse is visible only at deeper levels in this hierarchy.

8.7 Related Work

We believe our approach to be quite unique, but we can name corresponding work that has at least influenced our thoughts.

8.7.1 Unix Make

The first influencing factor is already quite old and maybe unexpected in this context. The Unix ‘make’ facility allows developers to specify the compilation of their software by just listing how a file depends on other files and how it should be compiled. The ‘make’ tool collects all build rules and executes them in the right order. Our protocol also allows to specify signal drops, source selection, property and capability handling locally, while still executing these in the right order. Note that in ‘make’, missing a single dependency in one of the rules may result in an erroneous rebuild, just as in our protocol an error in one of the components may cause the entire application to fail.

8.7.2 OO Design Patterns

Our protocol resembles certain object-oriented design patterns [31]. It corresponds most closely to the *Chain of Responsibility* pattern, where a request is decoupled from the object that actually implements it by passing it along a chain of objects until one handles it. In our case, the request is sent through a chain of signal processing components. All parts of the protocol exhibit this behavior: the drop/restore, the source selection, the property change and the capability management. In fact, to bring this one step further, we even contemplated letting

'normal' commands such as *Tune* or *VolumeUp* traverse the signal chain. This would make it easy to change the frequency of the tuner connected to the main stream – in the current situation at least some knowledge about the signal path is still required.

A second pattern that we use is the well-known *Observer* pattern. Down-stream devices 'observe' in fact the signals as provided by up-stream devices, and they are notified by the latter whenever changes occur. The observer pattern has proved its usefulness at many an occasion, and this certainly includes our situation.

Our third party binding, where components do not know their direct neighbors but the *compound* component makes the connection, is a special case of the *Mediator* pattern. But where a mediator is usually seen as an object *between* the others, our compound component (usually) is not active anymore after it has made the connection. In that sense, it resembles more of a *Broker*, be it that a broker is usually passive and accessed by the components, whereas our compound component takes the initiative.

8.7.3 Dynamic Architectures

Our protocol enables the creation of a dynamic architecture: a software architecture that adapts itself to changing circumstances. To understand this, consider Figure 58 again, where the dashed rectangle represents a plug-in module with extra functionality. We could store the control software for this module in a ROM on the module itself. Then, when an end customer buys this extension module and plugs it into his television, the driver code could be downloaded and connected to the other code using some form of dynamic binding. Our protocol would then ensure proper operation of the module in the existing TV. Although we do not deploy this idea currently, we certainly consider it as an option for future products.

8.7.4 Blackboard Architectures

Boasson created a blackboard architecture for air traffic control and air defense systems [10]. This architecture exhibits a large degree of uncoupling, to the extent that individual sensors and actuators can be added to or removed from the system without breaking the system. It also supports having different sensors that provide the same information with different degrees of reliability, while control algorithms are able to select the most appropriate sensor for their use. This closely resembles our approach. Novel in our approach is that our components do not communicate through a general blackboard, but rather through specific communication channels that mirror the hardware signal flow. Also in implementation our protocol differs, as we use direct function calls to communicate between components.

8.7.5 Coordination Languages

A related field is that of coordination languages (see for instance [34]). The underlying claim there is that programs consist of *computation* and *coordination*, and that these could be separated into different languages. 'Normal' programming languages are largely computation languages; coordination languages on the other hand focus on the specification of and the interaction between components, such that they can be (dynamically) composed into systems.

A distinction is made in [84] between data-driven and control-driven coordination models. Data-driven models rely on some shared data space such as a black board or a tuple space. Control-driven models deploy ports or interfaces, which are connected by and channels or connectors. Our solution falls in the latter category, as we set-up a dedicated (though possibly dynamic) communication structure between components. Many examples of control-driven coordination languages are mentioned in [84], among which Darwin, the predecessor of Koala.

8.8 Concluding Remarks

This completes the description and discussion of our style and protocol. For the reader's convenience, we present a short summary of the paper in the next subsection, after which we try to generalize what we have learned. We end with some acknowledgements.

8.8.1 Summary

In a television, the low-level software that directly controls the hardware turns out to be the hardest to build. The software is particularly dependent upon the topology of the signal flow in hardware. This topology is different for different members of the product family, it is apt to frequent changes until late in development, and it also changes at run-time. The TV hardware itself allows for easy composition, even with parts of other CE products such as DVD players or hard disk recorders. So a natural question is: can we make the low-level control software as composable as the hardware?

We have realized this by letting software components that control particular hardware devices have input and output ports that mirror those in hardware, be it that the software exchanges meta information rather than video and audio signals. This allows us to build a control architecture whose hierarchy exactly follows the structure of the hardware: core cells build chips which populate layout cells which constitute printed circuit boards which are combined into products. As a consequence, software reuse follows hardware reuse, or put differently, if new hardware needs to be developed, we also have time to develop new software.

Instead of a traditional top-down control hierarchy (with bottom-up notifications), we now get a right-to-left (up-stream) hierarchy with left-to-right (down-stream)

notifications. For such a hierarchy we have devised a set of interfaces that allow us to handle the disappearance and recurrence of signals, the selection of sources, and the measurement and usage of properties of signals. The two major advantages of our approach are the automatic correctness of the order of operations in a single product, and the ease with which we can construct a variety of products from a given set of components.

Although the protocol may feel natural to some people, our experience is that most software developers find it quite unusual at first. For that reason we had to build a prototype that simulates the protocol and shows how all commands traverse the signal chain. It turned out that Visual Basic was a convenient means to build this prototype. As an extra, we built a wiring protocol into the simulation that allows us to graphically arrange controls (representing devices) on a form (representing the product) and from that derive the signal flow topology automatically.

8.8.2 Generalizing our Findings

Presenting a case as the one in this paper is always insightful as far as realism and complexity is concerned: we have actually implemented this protocol and are using it in all of our up-market TVs. A remaining question is how well this approach generalizes to other domains. To answer that, three different aspects must be distinguished in our paper.

We consider the overall approach of communicating meta information along signal paths an architectural style. It resembles the pipe and filter style, and it embodies several design patterns. It can be used for any software controlling signal flow, both for hardware and for software streaming. Note that the communication between components is more directed than with shared data space approaches, such as black boards or tuple spaces. One generalization may be that we create a distributed control system where communication between control components follows the topology of the controlled (hardware) components in the domain. Perhaps similar techniques are useful in for instance telephone switching or factory control.

Within our domain, numerous variations are still possible for the protocol. We devised one protocol for controlling analogue up-market TVs, but we are sure that in other subdomains revised protocols are necessary. Such protocols are not discussed here, but hopefully the protocol described may serve as an example.

Finally, the implementation of our protocol with direct function calls in two directions is also quite unusual. Although it looks complicated at first, we have had little trouble in making it actually work, and we feel that the increase of efficiency outweighs the added complexity. Most of the related approaches that we could find use more traditional forms of message passing.

8.8.3 Acknowledgements

I would like to thank Erik Kaashoek for initial support in the ideas presented in this paper, Elmar Beuzenberg and Klaas Brink for being the first critical recipients, Jeff Magee and Jeff Kramer for helping me to formalize part of the protocol, Gerben Soepenberg for analyzing and innovating the implementation, and Gerben Soepenberg, Chritiene Aarts en Jan Bosch for reviewing this document.