The Koala Component Model for Consumer Electronics Software

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

Rob van Ommering

Frank van der Linden
Philips Research Laboratories

Jeff Kramer
Jeff Magee
Imperial College, London

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

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

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

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

Today's dynamic CE market makes it impossible to wait two years between the conception and introduction of a new product. Instead we must create new products by extending and rearranging elements of existing products. The highly competitive market also requires us to keep prices low, using computing hardware with severely constrained capabilities.

THE CHALLENGE

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

Why software components?

Software reuse lets us apply the same software in different products, which saves product-development effort. Software reuse has been a goal for some time.[1] The classical approach of defining libraries can be highly successful in limited domains, such as scientific and graphical libraries. However, while stimulating low-level code reuse, libraries do not help much in managing the similarities and differences in the structure of applications.

Developers devised *object-oriented frameworks* to create multiple applications that share structure and code.[2] The framework provides a skeleton that they can specialize in different ways. This approach makes application development faster, as long as the applications share similar structures. But changing the structure significantly is difficult because it is embedded in the framework. Also, a strong and often undocumented dependency exists between compo-

nents and the framework because of implementation inheritance.

Component-based approaches let engineers construct multiple configurations with variation in both structure and content.[3,4] A software component is an encapsulated piece of software with an explicit interface to its environment, designed in such a way that we can use it in many different configurations. Classical examples in desktop application software are the button, tree view, and Web browser. Well-known component models are COM/ActiveX, JavaBeans, and CORBA.

Why an explicit architecture?

Many component models are used with one or more programming languages—such as Visual Basic and Java—to construct configurations out of sets of components. Such an approach has one disadvantage: difficulty in visualizing and therefore managing the structure of the configurations. You can use visual tools to design the structure and even generate skeleton code from it, but keeping such designs consistent with the actual code often proves difficult. Although we can use round-trip engineering techniques to extract the design information from the actual code, wouldn't it be better to make the structure explicit in the first place?

With an architectural description language (ADL), you can make an explicit description of a configuration's structure in terms of its components.[5] This description makes both the diversity of the product family and the complexity of the individual products visible; thus, it serves as a valuable tool for software architects.

The perfect marriage?

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

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

Figure 1 illustrates the need for *requires* interfaces. In Figure 1a, if component A needs access to component B1, it would traditionally import B1, but this puts knowledge of B1 inside A, and therefore A cannot
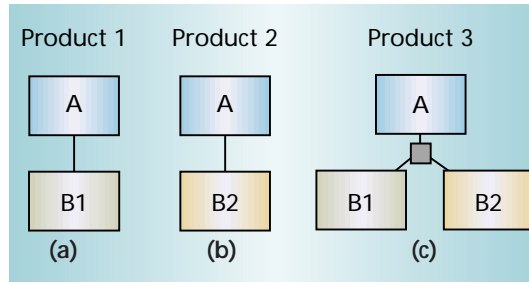

Figure 1. The use of requires *and* provides *interfaces takes the binding knowledge out of the components.*

combine with B2, shown in Figure 1b. One solution would be to let A import an abstract component B and have the configuration management system choose between B1 and B2. But this would not allow us to create product 3, shown in Figure 1c, where A is bound to either B1 or B2 depending on some condition to be determined at runtime.

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

Darwin,[6,7] although originally designed for distributed systems, provides most of what we need from an ADL: an explicit hierarchical structure, components with *provides* and *requires* interfaces, and bindings. However, it did require modification to support

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

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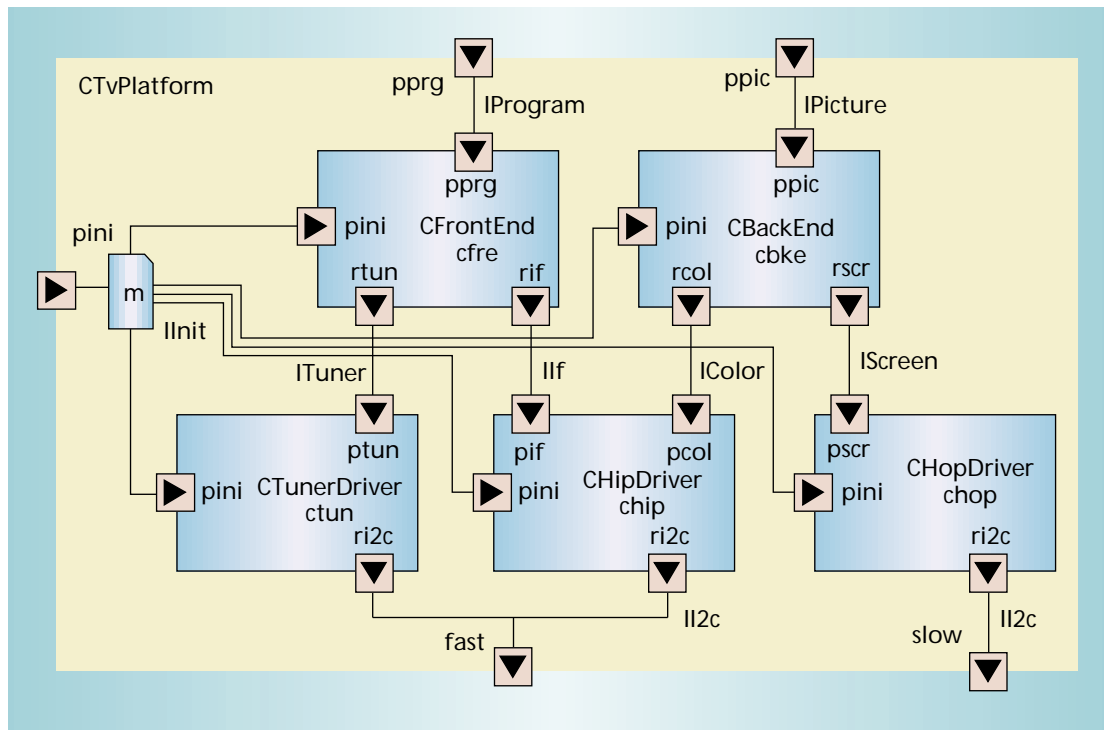
THE KOALA MODEL

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

Components

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

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

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

Figure 2 graphically represents a TV software platform that contains these drivers and some extra components. We deliberately designed Koala's graphical notation to make components look like IC chips and configurations look like electronic circuits. Interfaces are represented as pins of the chip; the triangles designate the direction of function calls. The configuration in Figure 2 binds the tuner and HIP driver to a fast I2C service and binds the HOP driver to a slow I2C service.

Interface definitions

We define an interface using a simple interface definition language (IDL), in which we list the function prototypes in C syntax. For instance, this is the `ITuner` interface definition:

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

`ITuner` is an example of a specific interface type, which will be provided or required by only a few different components. The `IInit` interface, also present in Figure 2, exemplifies a more generic interface: It contains functions for initializing a component, and most components will provide this interface.

Component descriptions

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

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

Each interface is labeled with two names. The long name—for example, `ITuner`—is the *interface type name.* This globally unique name refers to a particular description in our interface repository. The other name—for example, `ptun`—is a local name to refer to the particular interface *instance.* This convention allows us to have two interfaces on the border of a component with the same interface type—for

instance, a volume control for the speakers and one for the headphones—as long as the instance names are different.

Configurations

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

Compound components

A typical component may contain 10 interfaces, and a typical configuration contains tens of components. Hence, it is not convenient to define system configurations directly in terms of basic components. Therefore, as in Darwin, we introduce *compound components*. Figure 2 shows an example, the TV platform. Here is an incomplete definition:

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

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

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

Modules

In Figure 2, each subcomponent provides an initialization interface that must be called when initializing the compound component. We cannot just connect all initialization interfaces to that of the compound component; that violates our binding rule (What would be the order of calling?). We could define a new component to perform the initialization, but
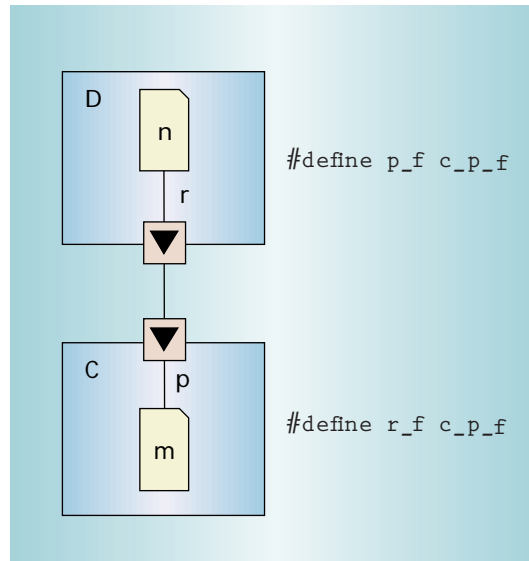

Figure 3. Implementation of static binding. Module m implements a function p_f, while module n refers to the function as r_f. Macros do the rest.

this nonreusable component would pollute our component repository.

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

Implementation

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

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

For each module, Koala generates a header file with renaming macros, as shown in Figure 3. A function f in interface p, implemented in module m of component C, is given the *logical* name p_f. Koala chooses a *physical* name c_p_f, where c is a globally unique prefix associated with C. To map logical to physical, Koala generates the following macro in the header file for m:

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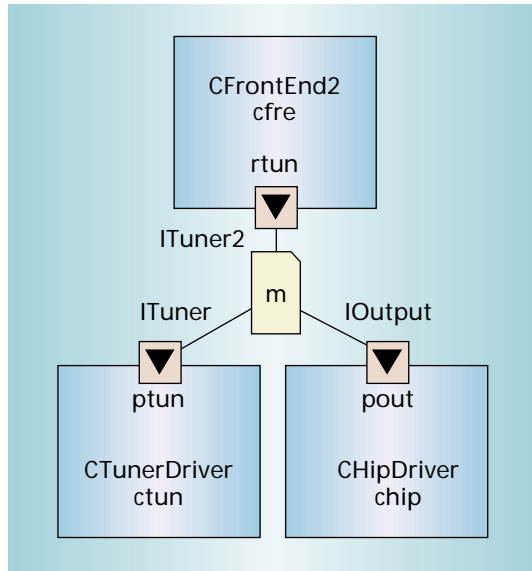
Similarly, a module n in component D refers to a function f in the *requires* interface r by its logical name, r_f. Koala calculates the binding and generates the appropriate renaming macro—in our case:

```
#define r_f c_p_f
```

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

HANDLING DIVERSITY

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

Interface compatibility

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

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

Function binding

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

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

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

Figure 4 shows how Koala performs function binding. Koala function binding can implement module m as follows:

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

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

Partial evaluation

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

Diversity interfaces

To be reusable, components should not contain configuration-specific information. Moving all configu-

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

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

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

Figure 5 shows how to give a television front end a diversity interface. A parameter in the interface `div` of `CFrontEnd` could be a Boolean function `ChannelMode()`, indicating whether the component should operate in frequency or in channel mode. The function is implemented in a module m that belongs to the configuration.

Koala can implement `ChannelMode` as a C function, which makes the diversity parameter dynamic. It can also bind `ChannelMode` to an expression that can be calculated at configuration or compile time. Koala will then assign the result value—for example, `true`—to the function so that the C compiler can, for instance, remove the `else` part of if statements referring to the parameter, resulting in less code. Koala will generate an extra macro of the form `#define div_ChannelMode_CONSTANT 1` that can be used to conditionally exclude certain pieces of code that are unreachable given this diversity parameter setting.

Diversity spreadsheets

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

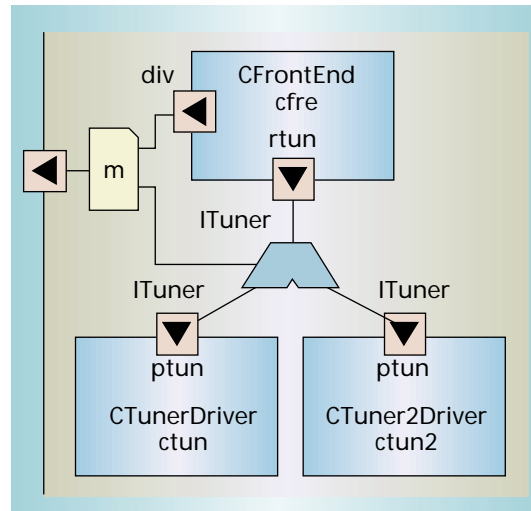The spreadsheet approach allows for even more ele-

gant diversity calculations. Consider the use of real-time kernel (RTK) threads. Each component will create zero or more of these threads. Some RTKs require the thread descriptor blocks to be allocated statically. If we let each component provide an interface that contains the number of threads required, we can use Koala to add all the numbers at compile time and bind the result to the diversity parameter of the RTK.

Switches

Koala can use diversity interfaces to handle the internal diversity of a component. But what about the structural diversity in the connections between components? Koala already provides for this: You can use function binding with conditional expressions to route the function calls to the appropriate components. Koala's partial evaluation mechanisms allow these connections to be turned into normal function calls if it can evaluate the condition at compile time. For us, this design pattern occurs so frequently we decided to make it a special Koala construct—the *switch*.

Figure 5 demonstrates the use of a switch. The front end connects to the first or second tuner driver depending on the switch's setting. An interface, which could be a diversity interface, controls the switch itself. If the switch setting is known at compile time, Koala's partial evaluation techniques will optimize the switch to a direct function call. Moreover, Koala removes unreachable components from the configuration automatically. These measures allow for late yet optimal component binding. Koala also permits multiple interfaces to be switched simultaneously and between more than two targets.

Optional interfaces

Our product family has a set of components that all provide a basic set of interfaces, but some of them

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

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

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

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

Connected interfaces

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

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

COPING WITH EVOLUTION

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

Interface repository

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

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

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

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

Component repository

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

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

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

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

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

Configuration management

The repositories are under the control of a standard configuration management system. This system manages the history of components and temporary branches—for example, where a developer repairs a

bug while another adds a feature. Koala handles all permanent diversity, either by diversity interfaces or by variants of components stored under a different name.

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

References

1. M.D. McIlroy, "Mass-Produced Software Components," *Software Engineering: Report on a Conference by the NATO Science Committee,* P. Naur and B. Randell, eds., NATO Scientific Affairs Division, Brussels, 1968, pp. 138-150.
2. M. Fayad and D. Schmidt, "Object-Oriented Application Frameworks," *Comm. ACM,* Oct. 1997, pp. 32-38.
3. D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Trans. Software Eng. and Methodology,* Oct. 1992, pp. 355-398.
4. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, Reading, Mass., 1997.
5. D. Garlan and D. Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Trans. Software Eng.,* Apr. 1995, pp. 269-274.
6. J. Magee et al., "Specifying Distributed Software Architectures," *Proc. ESEC '95,* Springer-Verlag, Berlin, 1995, pp. 137-153.
7. J. Magee, N. Dulay, and J. Kramer, "Regis: A Constructive Development Environment for Distributed Programs," *Distributed Systems Eng. J.,* Vol. 1, No. 5, 1994, pp. 304-312.

Rob van Ommering is a senior software architect at Philips Research Laboratories, Eindhoven, the Netherlands. His main interests are software architectures for resource-constrained systems, formal specification, and architecture verification and visualization. He received an MSc in physics from the Technical University Eindhoven. Contact him at Rob.van.Ommering@philips.com.

Frank van der Linden worked as a research scientist at Philips Research Laboratories, Eindhoven, the Netherlands, but now works for Phillips Medical Systems, Best, the Netherlands. His interests are in software architectures, with emphasis on family aspects of embedded systems. He has been active in research on parallel algorithms for proof checking, programming language semantics, and formal design methods. He earned an MSc and a PhD in mathematics from the University of Amsterdam, and is a member of the Dutch Mathematics Society. Contact him at Frank.van.der.Linden@philips.com.

Jeff Kramer is a professor and head of the Distributed Software Engineering Research Section in the Department of Computing at Imperial College, London. His research interests include requirement analysis techniques, design and behavior analysis methods, and software architectures, especially as applied to distributed software. He received a BSc in electrical engineering from Natal University and an MSc and a PhD in computing from Imperial College, London. Contact him at jk@doc.ic.ac.uk.

Jeff Magee is a professor in the Department of Computing at Imperial College, London. His research primarily concerns the software engineering of parallel and distributed systems, including design methods, operating systems, languages, and program support environments. He received a BSc in electrical engineering from Queens University Belfast and an MSc and PhD in computing science from Imperial College, London. Contact him at jnm@doc.ic.ac.uk.