10 Linking Programs to Architectures: An Object-Oriented Hierarchical Software Model Based on Boxes

Jan Schäfer, Markus Reitz, Jean-Marie Gaillourdet, and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany

10.1 Introduction

Modeling software systems has several purposes. The model provides a communication means between developers, a backbone to specify and check properties of the system, and a structure to organize, explain, and develop the implementation of the system. The focus of our approach is to address these purposes for hierarchically structured, object-oriented software systems. The hierarchical structure refers to the component instances at runtime: a *runtime component* may consist of a dynamically changing number of objects and other runtime components. Our modeling technique builds on and extends the concepts of class-based object-oriented languages. Runtime components are created by instantiating *box classes*. The modeling technique provides ports to tame object references and aliasing and to decouple components from their environment. It supports *dynamic linkage*, i.e. ports can be connected and disconnected at runtime. The used concurrency model is based on the join calculus.

Hierarchical software models allow to structure a system into components of different sizes where large components are recursively built from smaller ones. Dynamic linkage enables modeling of client–server behavior and open systems. For such systems, it is more difficult to separate the typical architectural level that deals with components and connectors and the realization level that describes the internals of components. The reasons are:

1. The architectural description on one layer of the hierarchy is the implementation of the component one layer up in the hierarchy.
2. Dynamic linkage is tighter intertwined with realization aspects.

Our approach to achieve a deeper integration of the architectural and component realization is by integrating them based on a common semantical framework supporting executable models. The development of our modeling technique is still in progress. CoCoME provided us with an interesting case study to evaluate our approach and to compare it to other techniques.

10.1.1 Goals and Scope of the Component Model

The goal of our approach is to fill the gap between high-level description techniques and the implementation of component-based systems. We use a two-layer

A. Rausch et al. (Eds.): Common Component Modeling Example, LNCS 5153, pp. 238–266, 2008.

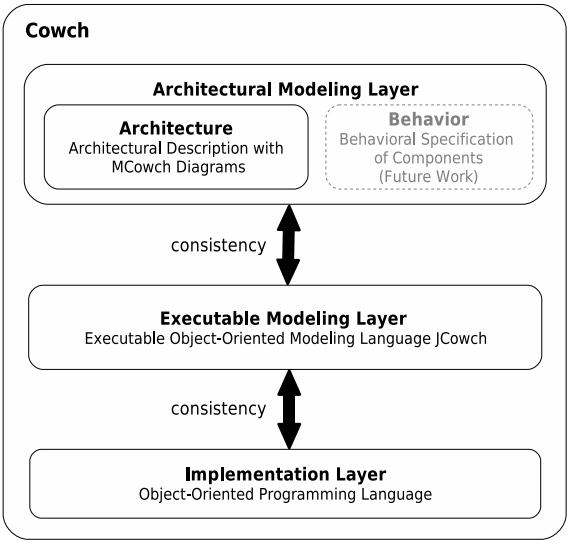

Fig. 1. The Cowch approach

approach with an executable modeling layer just above the implementation to describe the behavior of a system, and a graphical description language on top of that layer to describe the runtime architecture of the executable layer. Our component model is closely related to object-oriented programming languages allowing us to clearly identify components on the code-level. This is the cornerstone to achieve consistency between architectural descriptions and implementation.

10.1.2 Modeled Cutout of CoCoME

We used our graphical architectural description technique to give a hierarchical overview to the whole CoCoME system. Our focus are the structural aspects of the system at runtime. The behavior of the system can be fully described by our executable model. However, we only show this exemplarily in this paper, as our executable model is closely related to the implementation level it is straightforward to create that model for CoCoME. Although we are working on the behavioral specification of object-oriented components, we are not able to formulate functional properties yet. Non-functional properties are not in the focus of our approach, but could be integrated in principle.

10.1.3 Benefit of the Modeling

Our modeling technique has the benefit that it is close to the underlying implementation. This allows to identify components on the code-level, providing a link between high-level descriptions and implementation. This link makes it easier to ensure consistency between these layers, preventing erosion of architectures. As our model is hierarchical, it allows to concisely describe the architecture

of an implementation in a top-down fashion. Our system describes the runtime structure of an object-oriented system, allowing to capture the aliasing of components, which is an important feature distinguishing our approach from others that only describe the class and package-level of systems. In addition, we provide a high-level concurrency mechanism based on the join calculus which allows to express concurrency in terms of asynchronous messages and synchronization by join patterns.

10.1.4 Effort and Lessons Learned

We needed one person month to understand any detail of the system to be modeled. This includes understanding the delivered reference documentation as well as the delivered reference implementation. As our executable model is close to the underlying implementation it was not difficult to find an executable model for CoCoME. Most time went into drawing the graphical architectural model, as we do not have productive tools for this task, yet. Altogether it took us approximately one person month for creating our models. An additional person month was needed to write this paper. So the CoCoME contest took us about three person months in total.

We have been able to successfully model the CoCoME system with the Cowch approach. The resulting MCowch diagrams provide high-level and concise views to the CoCoME system. The CoCoME contest gave us valuable feedback to improve our modeling technique.

Related Work. Architectural Description Languages (ADLs) [1] serve as high-level structuring techniques for software systems. Examples include Darwin [2], Wright [3], Rapide [4], and UniCon [5]. These languages allow the formal description of software architectures as well as the specification and verification of architectural and behavioral properties on the architectural level. However, ADLs are loosely coupled to implementation languages, making it difficult to check consistency of architectural description and the actual implementation. Without consistency, however, all guarantees on the architectural level are lost on the implementation level.

A different approach is to extend programming languages by architectural elements. Examples for these approaches are ArchJava [6], ComponentJ [7], and ACOEL [8]. The advantage of these approaches it that there is no gap between architecture and implementation. The disadvantage is that they do not provide high-level views on the structure of a system. In addition, these approaches do not introduce new concurrency constructs, making it difficult to apply them to distributed systems.

Our approach falls into both mentioned categories. We define an executable modeling language which is closely related to the underlying implementation language. This language contains architectural elements which define the runtime structure of programs within the programming language. In addition, we define a graphical description language to describe the architecture of a system. The semantics of that language is closely related to the executable modeling language

so consistency checking is easier, when compared to implementation-independent ADLs.

Overview. The remainder of this paper is structured as follows. In the next section we give a general overview and introduction to our approach. Section 10.2.1 explains the executable modeling layer. The parts of the architectural model layer used in this paper are described in Section 10.2.2. Our model of the CoCoME system is given in Section 10.3. We shortly discuss analysis techniques for our approach in Section 10.4. Current tool support is sketched in Section 10.5. Finally, we give a summary in Section 10.6.

10.2 Component Model

Our COWCH[1] provides three layers of abstraction: the implementation layer, the executable modeling layer, and the architectural modeling layer (cf. Figure 1). On the implementation layer, the system is described using ordinary programming languages. In our current setting, we use Java as the language of the implementation layer.

To describe executable models, we develop an object-oriented modeling language called JCOWCH. JCOWCH provides support for hierarchical heap structuring and encapsulation, ports to make component interfaces and component linkage more explicit, and a high-level concurrency mechanisms based on the join calculus [9]. The sequential core of JCOWCH is based on Java's statements and object-oriented features. An implementation is consistent with a JCOWCH model if it has the same behavior at the interfaces of runtime components (see [10] for a formal treatment of component behavior). In the future we plan to specify the behavior of components on the architectural layer. We also plan to develop (semi-)automatic methods to verify consistency between executable modeling and implementation layer. Another goal is to generate efficient and consistent implementations from executable models.

The architectural modeling layer provides a high-level view on the system which only consists of the hierarchy of component instances and their communication structure. This layer abstracts from the creation and linkage phase of components and describes dynamic restructuring behavior of a system only by underspecification (in the future this layer will as well support behavioral specification of component properties). The architectural model is consistent with an executable model if the structure and component connections of the system during runtime correspond to the structures and connections described by the architectural models. Again, consistency can be achieved by checking techniques and by generating the structural aspects of the executable model from the architectural model.

Software components have to support composition by providing contractually specified interfaces and explicit dependency descriptions [11, p. 41]. As a prerequisite for this, the COWCH approach provides the notion of runtime components

[1] Component-oriented Development with Channels.

with well-defined boundaries and ports as communication points to its surrounding environment. Ingoing ports represent functionality usable by other entities within the environment whereas outgoing ports define functionality needed for proper functioning of the component. Communication with other entities in a component's environment is routed through its ports. Components which create other runtime components are responsible for establishing the connections between them.

10.2.1 The Executable Modeling Layer

In this section we describe the executable modeling language JCowch. The sequential part of JCowch is based on Java's statements and object-oriented features. We do not explain these standard features here and refer to the Java language specification [12]. Here we explain the central constructs to structure the heap, to control component interfaces, and to handle concurrency. We first describe the core elements of the language, and then show convenience extensions.

JCowch: Core Language

Boxes. A runtime component or component instance in our approach is called a *box*. A box consists of an *owner object* and a set of other objects. A box is created together with its owner by instantiating the class of its owner. Boxes are tree-structured, that is, a box b can have *inner* boxes. We distinguish two kinds of classes, normal classes and box classes (annotated by the keyword `box`). The instantiation of a normal class creates an object in the *current box*, that is, in the box of the current this-object. The instantiation of a box class creates a new inner box of the current box together with its owner. For simplicity, we do not support the direct creation of objects outside the current box. Such nonlocal creations can only be done by using a method. Note that this is similar to a distributed setting with remote method invocation. Currently, we do not support the transfer of objects from one box to another one.

Our approach only uses structural aspects of ownership (similar to [13]). It uses the box boundary to distinguishes between local method calls and external calls, i.e. calls on ports or objects of other boxes. We use the term *component* for the pair consisting of a box class and its *code base* where the code base of a class is defined as the smallest set containing all classes and interfaces transitively used by the class.

Boxes allow to clearly separate objects into groups which semantically belong together. This clear separation can then be used to describe the behavior of such runtime components in a representation-independent way [10]. In Cowch, boxes are the foundation for interfaces and the architectural structure. Boxes are used to control object references crossing box boundaries. Only the box owner is allowed to be used directly from the outside. All other internal objects can only be accessed through ports. Currently, this restriction is not enforced by the type system of JCowch, so the programmer must manually ensure this discipline.

Work to incorporate ownership type systems into JCowch are under way (see [14] for first steps).

Channels and Ports. A formally well-understood way to structure communication is to use channels [15, 16]. A channel is a point-to-point connection between two partners. In our approach, these partners are ports or objects. Channels are *anonymous*, i.e. two partners connected via a channel communicate without "knowing" each other. This supports a loose-coupling of communication partners. It also allows transparent *exogenous coordination* [16], i.e. communication partners can be transparently changed at runtime. The anonymity of channels allows to specify component behavior in an abstract way. A channel is also *private*, as communication cannot accidentally or intentionally be interfered by third parties [16]. This also means that it is always clear which entities can communicate with each other by looking at the channel structure. This gives strong invariants which support system verification.

In JCowch we use channels to express box communication. *Peer* boxes, i.e. boxes with the same surrounding box, can only communicate via channels. In addition, the surrounding box is allowed to directly call methods on inner boxes by using the reference returned when an inner box was created. Communication links crossing more than one layer in the box hierarchy can only be established by forwarding. Thus the surrounding box controls all communication to and from inner boxes.

The notion of a port defines a component's relationship to its surrounding environment. *Ingoing ports* represent functionality usable by other entities within the environment whereas *outgoing ports* define functionality needed for proper functioning of the component.

Ingoing Ports. An ingoing port in JCowch is a member of a class and is declared by the keyword **inport**. A simple ingoing port declaration may look as follows:

```
box class A {
  inport J p {
    void m() { }
  }
}
```

Class A has a single ingoing port p of type J. J is an interface with a single method m(). The ingoing port in this example provides a trivial, empty implementation of interface J. An inport can be used by connecting it to another port (see below) or by direct use from the surrounding box. If a is variable of type A referencing an inner box, method m() of port p can be invoked by a.p.m(). It is also possible to treat ports similar to objects, to assign them to local variables, and to use them like ordinary object references, e.g. J b = a.p.

A class having an ingoing port p of a type J is responsible for providing an implementation for p. This implementation can be provided directly as part of the port declaration (see above) or indirectly by connecting the ingoing port to an inner object or port (connections are explained below). In that case the ingoing port is declared without a body.

Outgoing Ports. Outgoing ports are declared by the keyword **outport**. Like ingoing ports they have a type and a name, but they do not provide an implementation. From a class's perspective, an outgoing port is an interaction point with the environment. The implementation of outgoing ports must be provided by the environment, which is complementary to ingoing ports. A simple declaration of an outgoing port may look as follows.

```
box class B {
  outport J q;
  void n() {
    q.m();
  }
}
```

Class B declares an outgoing port q of type J. Like ingoing ports, outgoing ports can be used like regular object references. The difference is that the actual implementation of an outgoing port is determined at runtime depending on the channels established by the environment.

Connect Statements. To provide an implementation for an outgoing port, the environment has to create a channel from that port to another port or object. A channel in JCowch is created by a **connect** statement with the following syntax:

```
connect <source> to <sink>
```

The source parameter has to be a reference to a port, the sink parameter can be either an object reference or a port reference. After the **connect** statement has been executed, all method invocations on the source are forwarded to the sink. As connect statements must not break type-safety, the type of the sink expression has to be a subtype of the source expression. Calling a method on an unconnected outgoing port results in a runtime exception being thrown.

The following example reuses the classes A and B. The outgoing port q of an instance of A is connected to the ingoing port q of an instance of B. As port b.q is connected to port a.p and method n invokes m on port q, the effect of b.n() is essentially the same as calling a.p.m() directly.

```
box class C {
  A a = new A();
  B b = new B();
  void k() {
    connect b.q to a.p;
    b.n();
    a.p.m(); // same effect
  }
}
```

Deleting a previously created channel is performed by the **disconnect** statement, having the following syntax:

```
disconnect <source> from <sink>
```

If no channel exists from the given source to the given sink, the statement has no effect.

Concurrency. Modern software systems need distributed and local concurrency. A modeling language has to be both flexible with respect to the concurrency patterns it supports and restrictive to achieve controlled concurrent behavior. To approach these goals, JCowch combines the concurrency model of the join calculus [9] with the structuring and interfacing techniques of boxes. Concurrency models based on the join calculus have already been integrated into OCaml (called JoCaml [17]), $C^{\#}$ (called C^{ω}[18]), and Java (called Join Java [19]). Similar to these realization, JCowch does not support the explicit creation of threads. Instead, it distinguishes between synchronous and asynchronous methods.

Synchronous methods are syntactically and semantically similar to methods in object-oriented programming languages like Java or $C^{\#}$. The caller waits until the called method terminates, receives its return value if any, and continues execution. Asynchronous methods are denoted by the keyword **async** in place of a return type. They have no return value. Calling an asynchronous method causes it to be executed concurrently to the caller without blocking the caller. Calls to asynchronous methods implicitly spawn a new thread. Synchronization of threads is expressed by *chords*.

Chords. Chords, or *join patterns*, provide a powerful and elegant synchronization mechanism which groups methods that wait for each other and share one method body. That is, chords can be considered as a generalized construct for method declaration. Syntactically, a chord is a method with multiple method signatures separated by &. In the body, the arguments of all method signatures are accessible. There may be at most one synchronous method per chord. The body of a chord is executed when all methods it includes have been called, i.e. calls to methods which are part of a chord are enqueued until the chord is executed.

A method header may appear in several chords. It is dynamically decided to which chord it contributes, depending on which chord is executable first. These unbounded method queues extend the state space of objects. Chords must not appear in interfaces, whereas the methods they consist of are declared in interfaces.

The code in Fig. 2 shows an unbounded asynchronous buffer for strings. An arbitrary number of calls of method *put* is stored in conjunction with the corresponding arguments and removed when appropriate *get* method calls arrive.

JCowch: Convenient Extensions to the Core

The language elements described so far are the core of JCowch. We now introduce some additional concepts which can be seen as convenience constructs to make the life easier for developers in practice.

Optional Ports. We said above that the environment has to provide an implementation for an outgoing port. Invoking a method on an unconnected outgoing

```
interface StringBuffer {
  async put(String s);
  String get();
}
class StringBufferImpl implements StringBuffer {
  String get() & put(String s) { return s; }
}
```

Fig. 2. An unbounded buffer

port results in a runtime exception. However, it is often the case that a component has outgoing ports which it does not require to work properly. For example, a logger port for debugging purposes is in general not needed by a component. It is just a port to inform the environment of some events happening inside the class. These kinds of messages are typically called *oneway messages* [20]. For this reason we introduce *optional* outgoing ports in contrast to *required* outgoing ports. Outgoing ports are by default required and can be declared to be optional by using the optional keyword. An unconnected optional port acts as a *Null Object* [21]. If a method is called on such a port, the method call is silently ignored. For simplicity, we only allow calls to **void** and **async** methods on optional ports.

Nested Ports. So far ports have been flat in the sense that they only contained methods. However, to be able to hierarchically structure interfaces of larger components, it is essential to have *nested ports*. To enable port nesting, interfaces have to be extended to allow the declaration of ports. A simple interface J with a single port declaration of type K may look as follows:

```
interface J {
  port K k;
}
```

The direction of a nested port is left open in the interface. Thus such interfaces can be used for ingoing as well as outgoing ports. The direction of the nested ports is always equal to the direction of its surrounding port. For example, a class can have a port of type J, which in turn has nested port K. If that port is an ingoing port, the class has to provide an implementation for the port itself and the nested port. Assuming an interface K containing a single method m(), this looks as follows.

```
class A {
  inport J p {
    inport K k {
      void m() { }
    }
  }
}
```

Method Channels. In many cases interface-based wiring of ports and objects is the right choice. However, there are at least three situations in which interface-based connections are inconvenient.

1. If two ports with different interfaces should be connected.
2. If only single methods, but not all methods of a port are of interest.
3. If a class only needs one or two outgoing methods with no fitting interface.

The first situation requires the creation of an adapter class that has a port of each interface and which forwards each method of the first interface to the appropriate method of the second interface. The second situation requires the implementation of methods with empty bodies. In the third case the introduction of an artificial interface is needed. Because of these reasons we allow the wiring of single methods.

Method Connections. To be able to connect single methods, we extend the **connect** statement to take method targets as sources and sinks. A method target consists of two parts, the first part is an expression which defines the target object or port of the method, and the second part defines the method which should be used. The second part can be either a simple method name or a full method signature. The latter is needed if the target method is overloaded.

To maintain type-safety the parameter and result types of the sink method must match the parameter and result types of the source method. The matching rules are equal to the overriding rules in regular Java, i.e. the parameter types can be contra-variant and the result types can be co-variant. However, method names are irrelevant when connecting methods, only the parameter and return types are important.

Outgoing methods. To avoid artificial interfaces when only single methods are of interest, we allow the declaration of single *outgoing methods* without ports. Outgoing methods are declared by the keyword **out** and have no body. They can be seen as anonymous ports with a single method. Like outgoing ports, outgoing methods can be declared to be optional by the optional keyword. A simple outgoing method declaration may look as follows.

```
class A {
  out void m();
}
```

A Larger Example

Figure 3 illustrates the use of the discussed concepts. Box class A's only functionality is to forward a call of p.m to q.m after having increased the passed integer parameter by one. Method m of box class B of port in is implemented by calling the outgoing printInt method with the passed integer value being used as parameter. The constructor of box class C creates two inner boxes, one of box

class A and one of box class B, and forwards ingoing port in to port p of the inner box A using the **connect** statement. The q port of box A is connected to port in of box B by the **connect** statement. Finally, the outgoing method printInt of box B is forwarded to the print method of the printer port.

Box class C represents a composing entity which creates and wires instances of box class A and B. We defined a new box component with a distinguished behavior. A call of in.m results in increasing the passed integer value by one and then printing the new value to the output stream.

```
interface Print {
  void print(int i);
  ...
}

interface J {
  void m(int i);
}

box class A {
  outport J q;
  inport J p {
    void m(int i) {
      q.m(i+1)
    }
  }
}

box class B {
  inport J in {
    void m(int i) { printInt(i); }
  }
  out void printInt(int i);
}

box class C {
  private A a = new A();
  private B b = new B();
  outport Print printer;
  inport J in;
  C() {
    connect in to a.p;
    connect a.q to b.in;
    connect b.printInt(int) to printer.print(int);
  }
}
```

Fig. 3. Code example which shows the usage of port and connections

10.2.2 The Architectural Modeling Layer

The architectural modeling layer consists of a graphical language – MCowch – depicting the structure and communication topology of the modeled systems. A single MCowch diagram describes the internal structure and the communication endpoints of one component of the JCowch implementation. The described component is denoted in the upper left corner of the diagram with its full qualified package name. The boundary of a diagram must show all nonprivate methods and ports of the declared box class. Thus an MCowch diagram always shows the complete external interface of a component.

Beside this easily derivable information, the diagram shows a conservative view to the runtime structure of its described component, which is done by the following architectural elements.

Architectural Elements. Within MCowch diagrams the following architectural elements can appear.

Inner boxes are indicated by rectangles.
Ports are depicted by the symbols ▷ for outgoing ports and ▶ for ingoing ports. Ports that have interfaces which only consist of asynchronous methods are represented by the symbols ▶ and ▷.
Methods are depicted by the symbols ⊐ for ingoing methods and ⊃ for outgoing methods. Asynchronous methods are depicted by ⊏ and ⊐, respectively.
Channels are indicated by lines with arrows. Thick lines represent port channels, thin lines represent method channels.

Figure 4 shows an MCowch diagram which is consistent to the JCowch example in Figure 3 on Page 248. It contains most of the architectural elements. The semantics of MCowch diagrams is explained in the next section.

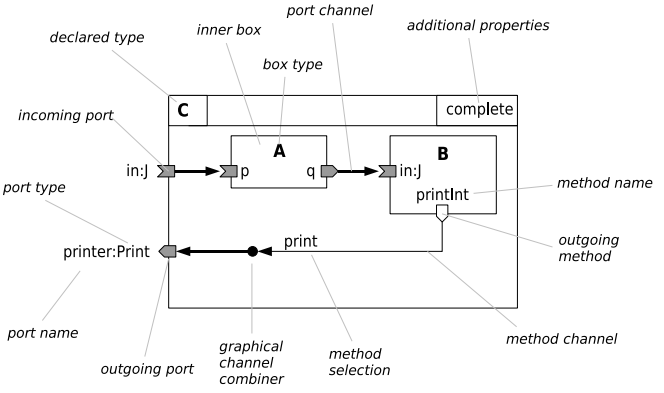

Fig. 4. Example of an MCowch diagram

Relation to the Executable Modeling Layer. An architectural model denotes the static aspects of a program in the executable modeling layer. The semantics of the model are described, rather than defined, by relating the architectural elements of the model to artefacts of the program.

Boxes. Every rectangle drawn in a MCowch diagram represents a box that is created by the defined component at runtime. The exact number is determined by the given multiplicity with a default of one. If the given multiplicity is constant it means that during the whole lifetime of the component instance the stated number of boxes exist. This means that these boxes have to be created in the constructor of the component. A plus (+) used as multiplicity denotes one or more boxes, i.e. at least one box must exist during the whole lifetime of the component. A star (*) indicates an arbitrary number of boxes (including zero) being created during the box lifetime. Boxes can be drawn by dashed lines denoting a multiplicity of zero or one.

To be consistent to the underlying implementation, the MCowch diagram must show *all* possibly created boxes. That is, if a diagram is consistent to the

underlying implementation, the implementation cannot create boxes that do not appear in the diagram. In addition, all boxes appearing in a diagram can only communicate via the indicated channels.

Channels. The channels appearing in MCowch diagrams represent channels created by **connect** statements in the executable layer during runtime of the defined component. We distinguish *durable* channels and *optional* channels. A solid line indicates a durable channel and represents a channel that always exists at runtime when both connected boxes exist. Usually this means that the channel is directly created after both boxes are created, and it is never disconnected. A dotted line denotes an optional channel, expressing the fact that the channel *might exist* at runtime.

Connecting a line to a rectangle with a multiplicity greater than one implies that there exist a channel for each box of the box set. This allows to express 1 and N:1 connections. To express arbitrary N:M connections we use a special *unknown* connection indicated by a question mark which indicates that connections exist but leaves the exact wiring unspecified.

Helper Objects. An MCowch diagram only illustrates the inner box structure of a component, i.e. all instances of box classes and their connections. It does not show helper objects which are used by the component implementation, i.e. objects of regular non–box classes.

Complete Diagrams. An MCowch diagram can be specified to be *complete*. To be complete, the corresponding component implementation may not have any behavior other than specified by the diagram, it has a constant number of boxes and only durable channels. From a complete MCowch diagram it is always possible to generate its executable model.

Active and Reactive Components. An additional property of components is shown in MCowch diagrams which can be derived from the executable layer, namely whether a component is *active* or *reactive*. Reactive components are only able of reacting to incoming method invocations. This means that after the invoking thread has left the component, a reactive component cannot send any further messages on its outgoing methods or ports. On the other hand, an active component can send messages on its outgoing entities whether or not a method has been invoked on it. On the modeling layer, a component can be identified to be active if it either uses an active component or has an asynchronous method that appears in a chord without a synchronous method, as only these chords can create new threads. If a component has neither an outgoing method nor an outgoing port, the component is always reactive, as the environment cannot observe any activity.

MCowch diagrams of active components are declared to be active in the meta-bar, otherwise they are reactive. In addition, active boxes are drawn with a thicker border than reactive boxes when appearing as boxes in MCowch diagrams.

```
interface J {
  void m();
  void n();
}
```

```
box class B {
  inport J q;
  outport J p;
  // ...
}
```

```
box class C {
  inport J q;
  // ...
}
```

Fig. 5. An interface and two box classes needed by the following examples

```
box class A {
  inport J q;
  B b; C c;
  A() {
    b = new B();
    c = new C();
    connect this.q to b.q;
    connect b.p to c.q;
  }
}
```

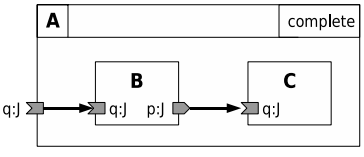

Fig. 6. An implementation of a box class and its corresponding MCowch diagram

```
box class D {
  inport J q;
  B b;
  D(boolean createB) {
    if (createB) {
      b = new B();
      connect b.p to c.q;
    }
  }
}
```

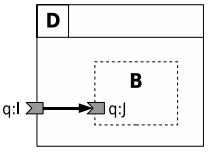

Fig. 7. An example of an optional inner box

Graphical Channel Combiner. To graphically express that a channel is only connected to a part of a port and not the whole port, we allow to explicitly name the connected part and write it next to the channel. This can be combined with the graphical channel combiner. For example in the MCowch diagram in Figure 4 the printInt method of the B box is connected to the print method of the printer port.

Examples. To better understand the relation between the modeling and the architectural layer we give some simple examples shown in Figures 6, 7, 8, and 9. On the left side we show the implementation in JCowch and on the right side we show an MCowch diagram which is consistent to that. We use the interface and the box classes shown in Figure 5 for this purpose.

```
box class E {
  C c;
  B[] b;
  E(int numB) {
    c = new C();
    b = new B[numB];
    for (int i=0; i <numB; i++) {
      b[i] = new B();
      connect b[i].p to c.q;
    }
  }
}
```

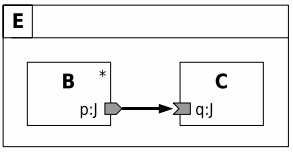

Fig. 8. An example of an arbitrary multiplicity

```
box class F {
  outport J q;
  B b;
  F(boolean createChannel) {
    b = new B();
    if (createChannel) {
      connect b.p to this.q;
    }
  }
}
```

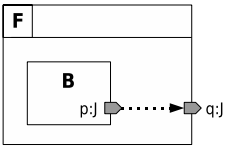

Fig. 9. An example of an optional channel

10.3 Modeling the CoCoME

We now present our model of the CoCoME system. Figure 10 shows a high-level view on the CoCoME system specified by an MCowch diagram. An instance of the CoCoME component consists of a TradingSystem box and a Bank box. In addition, the outgoing bank port (depicted by ▻) of the TradingSystem box is connected to the Bank box. Notice that the rectangle of the TradingSystem box is drawn with a thicker line than the Bank box rectangle, which indicates that the TradingSystem is active and the Bank is reactive. As the CoCoME component has no further functionality the diagram is specified to be complete.

10.3.1 The Bank Component

The Bank component is used to validate credit cards and to debit money from bank accounts. As it is an external component, not modeled by CoCoME, we only give its interface which consists of two ingoing methods (cf. Figure 11).

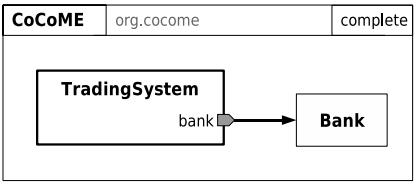

Fig. 10. The top level view on CoCoME

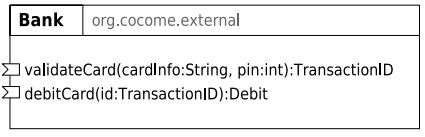

Fig. 11. The Bank component

10.3.2 The TradingSystem Component

To go deeper into the CoCoME system we take a look at the TradingSystem component (cf. Figure 12). The first difference to the CoCoME component is that the TradingSystem component has a port declaration. In this case the port has name bank and is of type Bank. The symbol ▷ indicates that it is an outgoing port. This port declaration is consistent to the port usage in the previous diagram, where the bank port is connected to a Bank instance. The TradingSystem component uses three different components, namely EnterpiseClient, StoreModel, and Database. The number of StoreModel and EnterpriseClient boxes is left unspecified, indicated by a star (*), but there exists exactly one Database box per TradingSystem box as this is the default multiplicity. No matter how many EnterpriseClient or StoreModel boxes exist, the diagram specifies that the db port of each of them is connected to the ingoing (◁) jdbc port of the Database box (the bullet (•) graphically summarizes these two parallel running lines). In addition, the bank port of each existing Store box is connected to the bank port of the TradingSystem box.

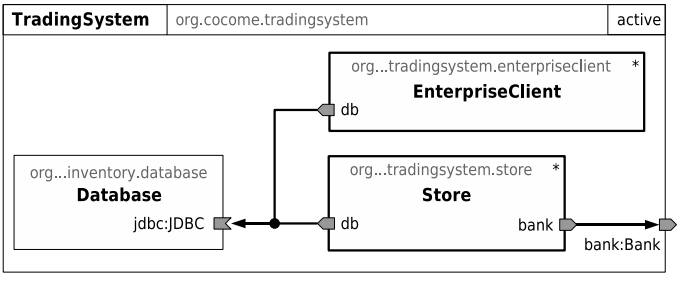

Fig. 12. The TradingSystem component

10.3.3 The EnterpriseClient Component

The EnterpriseClient component is shown in Figure 13. It offers the enterprise manager the possibility to look at the statistical data of the enterprise by using the graphical user interface of the Reporting box. To access the database the EnterpriseClient box uses Application and Data boxes and forwards the outgoing port of the Data box to its own outgoing db port of type JDBC.

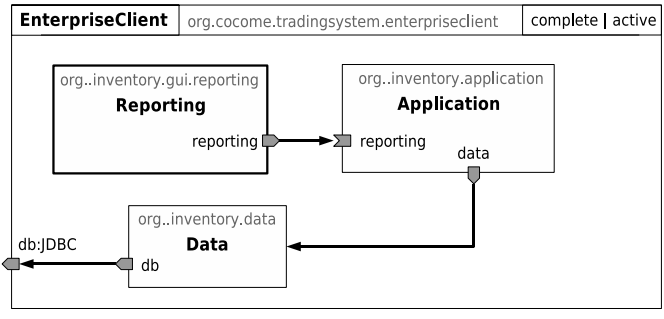

Fig. 13. The EnterpriseClient component

10.3.4 The TradingSystem::Store Component

Figure 14 shows the Store component, which represent a single store. Such a component does not exist in the reference description of CoCoME. We believe that such a component is useful and thus we give a model here. A Store box consists of a single CashDeskLine box, an Application and a Data box to abstract from the JDBC interface, and an unspecified number of Inventory::GUI boxes to allow the stock manager and the store manager to access the database. In the CoCoME reference implementation the CashDeskLine and the Application communicates via an event bus as well as via RMI. We model the bus communication by an asynchronous method channel, the RMI communication is modeled by synchronous port channels.

10.3.5 The CashDeskLine Component

The CashDeskLine component is shown in Figure 15. An instance of a CashDeskLine consists of several CashDesk boxes and a single Coordinator box. The Coordinator box manages the express checkout, wheras the CashDesk boxes represent the cash desks. Like before, we model the bus communication by asynchronous method channels. In addition, the outgoing bank and inventory ports, as well as the accountSale method of each CashDesk box are forwarded to the corresponding ports and method of the CashDeskLine component.

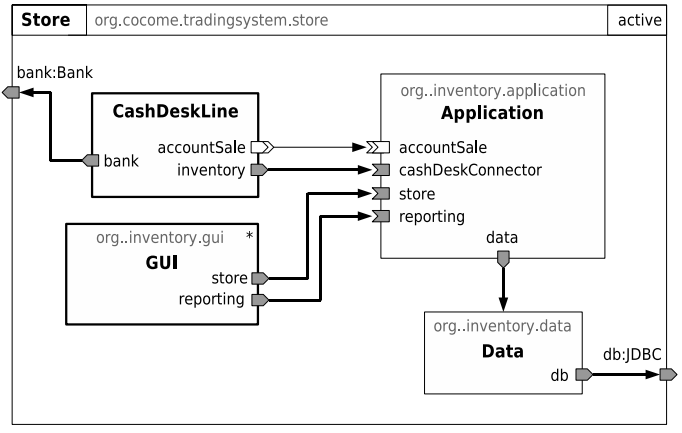

Fig. 14. The Store component

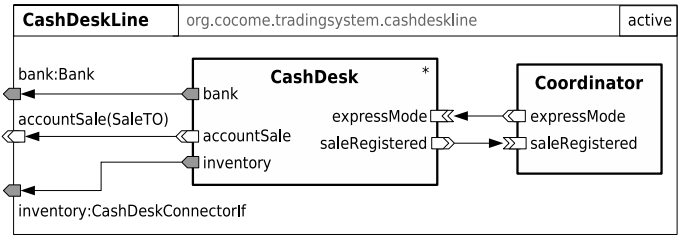

Fig. 15. The CashDeskLine component

10.3.6 The Coordinator Component

Figure 16 shows the Coordinator component, which is responsible for managing express checkouts. For this reason it has an inner SaleStatistics box which collects information about sales. When a call to the saleRegistered is made, it informs an inner SaleStatistics box about the new sale and asks it whether an express mode is needed. If this is the case the outgoing method expressMode is invoked with the previously passed cashdesk string. The MCowch diagram is shown in Figure 16. What might not be obvious is that the Coordinator component is active. To clearify this we give its JCowch model in Figure 17. As the Coordinator component must access the iternal SaleStatistics box in a sequential way, it has to sequentialize its incoming asynchronous saleRegistered messages. For this reason it has an internal loop which sequentially handles these messages, which is started by invoking the asynchronous waitForMessage method in the constructor. The loop sequentially calls the synchronous handleNextMessage method. As this method is in a chord together with the saleRegistered method, it is only executed when that method has been invoked. This realization ensures a sequential access

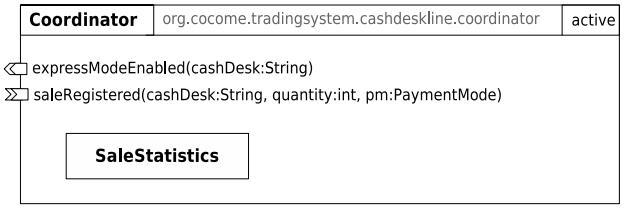

Fig. 16. The MCowch diagram of the Coordinator component

```
public box class Coordinator {
  private SaleStatistics saleStats;

  public out async expressModeEnabled(String cashDesk);
  public async saleRegistered(String cashDesk, int quantity, PaymentMode pm);

  public Coordinator() {
    saleStats = new SaleStatistics();
    allowAccess();
    waitForMessage();
  }

  private void handleNextMessage()
    & saleRegistered(String cashDesk, int quantity, PaymentMode pm);
  {
    saleStats.registerSale(quantity,pm);
    if (saleStats.isExpressModeNeeded()) {
       expressModeEnabled(cashDesk);
    }
  }

  private async waitForMessage() {
    while (true) {
      handleNextMessage();
    }
  }
}
```

Fig. 17. The JCowch model of the Coordinator component

to the internal SaleStatistics box. It also resembles the semantics of the event bus system which is used by the CoCoME reference implementation.

10.3.7 The SaleStatistics Component

Figure 18 shows the SaleStatistics component. It calculates the express mode necessity depending on the sale history. As it has no further inner boxes, the

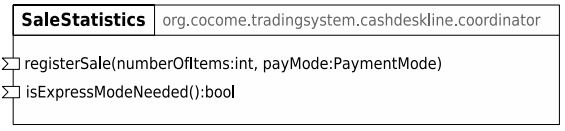

Fig. 18. The SaleStatistics component

MCOWCH diagram only shows its external interface consisting of two ingoing methods.

10.3.8 The CashDesk Component

The CashDesk component is the most complex component in the CoCoME system. We decided not to directly model the event bus. Instead we present a

```
public interface ScannerEvents {
  async productBarcodeScanned(Barcode b);
}

public interface ApplicationEvents {
  async runningTotalChanged(String productName, double productPrice,
                            double runningTotal);
  async changeAmountCalculated(double changeAmount);
  async saleSuccess();
  async productBarcodeNotValid(long barcode);
  async invalidCreditCard();
}

public interface CashBoxEvents {
  async saleStarted();
  async saleFinished();
  async paymentMode(PaymentMode paymentMode);
  async cashAmountEntered(KeyStroke keyStroke);
  async cashBoxClosed();
}

public interface CardReaderEvents {
  async pinEntered(int pin);
  async creditCardScanned(String cardInfo);
}

public interface ExpressMode {
  async expressModeEnabled(String cashBox);
  async expressModeDisabled();
}
```

Fig. 19. The different interfaces summarizing bus events

conceptional view on the CashDesk component which is behavioral equivalent to the actual implementation. For this reason we structured the different events of the bus that semantically belong together into several new interfaces, which can be seen in Figure 19. Figure 20 shows the CashDesk component. This architectural view shows the exact communication structure of a single cash desk.

10.3.9 The CardReaderController Component

The CardReaderController component abstracts from the card reader hardware (cf. Figure 21). To be able to react to express mode changes, it has an ingoing expressMode port. To send card reader events it has an outgoing events port.

10.3.10 The ScannerController Component

The ScannerController component handles events from the barcode scanner and emits them on its outgoing events port by sending productBarcodeScanned events (Figure 22). As we do not model the scanner hardware, it has no further inner boxes.

10.3.11 The LightDisplayController Component

The LightDisplayController reacts on incoming expressMode messages and switches the light display on or off. It has no further inner boxes (Figure 23).

10.3.12 The CashDesk::Application Component

The CashDesk::Application component is shown in Figure 24. It can be seen as the control center of the CashDesk component. It contains the application logic of the CashDesk component, and it handles the external communication with the Bank, the Inventory and the Coordinator components. As it has no further inner boxes the MCowch diagram only shows its interface.

10.3.13 The CashBoxController Component

The CashBoxController component is shown in Figure 25. It represents the cash box which is operated by the cashier. It has an outgoing port events of type CashBoxEvents as well as an outgoing method expressModeDisabled(), which represent the different kinds of keys which may be pressed by the cashier. The only ingoing method cashAmountCalculated(**double**) opens the cash box.

10.3.14 PrinterController

The PrinterController component is shown in Figure 26. It reacts to certain events from the Application and the CashBox component and prints useful information for the customer on a receipt.

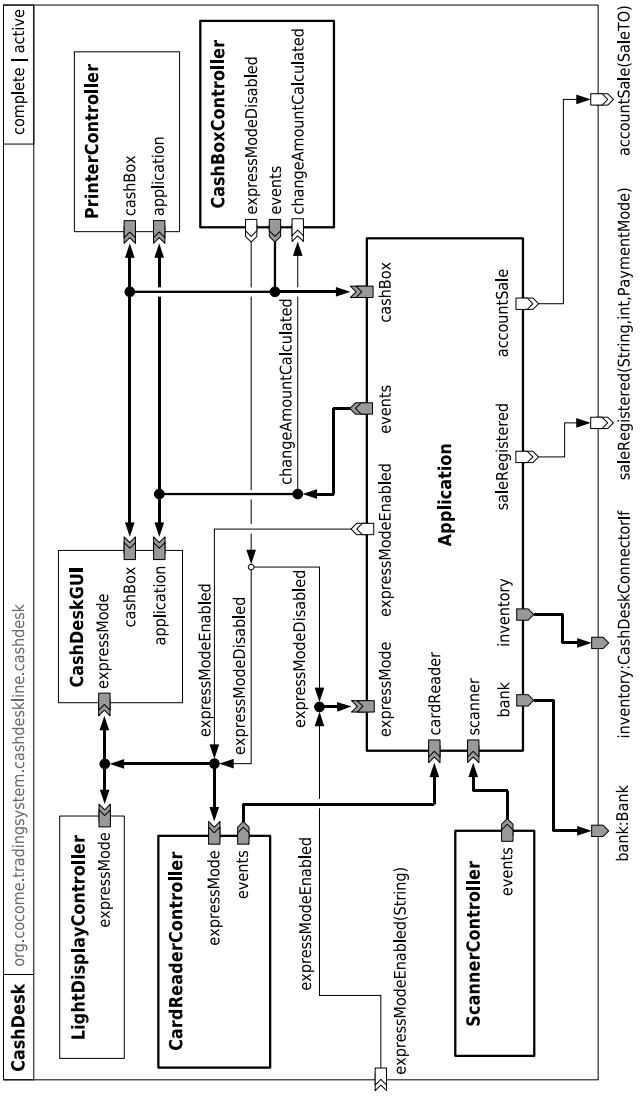

Fig. 20. The CashDesk component

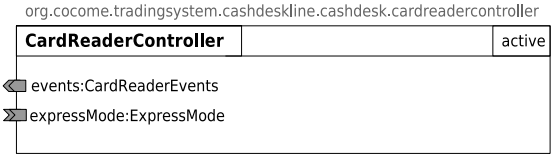

Fig. 21. The CardReaderController component

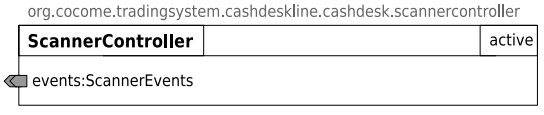

Fig. 22. The ScannerController component

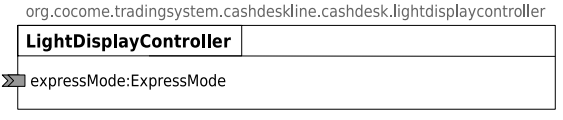

Fig. 23. The light display controller

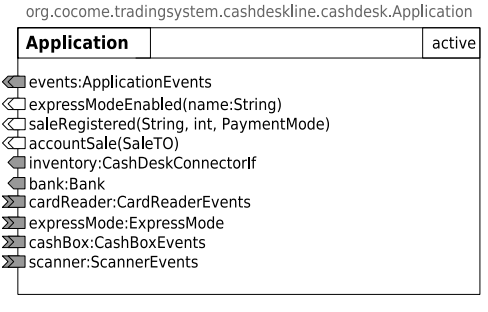

Fig. 24. The Application component

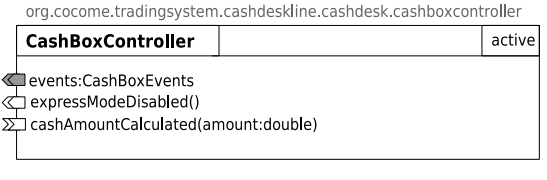

Fig. 25. The CashBoxController component

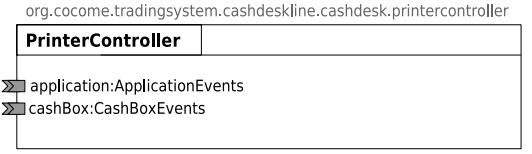

Fig. 26. The PrinterController component

10.3.15 CashDeskGUI

Figure 27 shows the CashDeskGUI component. Its purpose is to show useful information on a little display for the customer as well as the cashier. For this reason it has a set of ingoing ports for certain kinds of events.

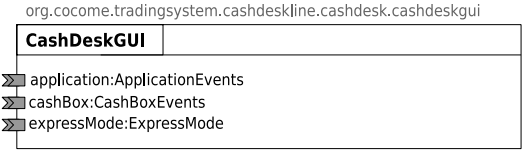

Fig. 27. The CashDeskGUI component

10.3.16 Inventory::Application

The Inventory::Application component is responsible for abstracting from the direct access to the Inventory::Data component, as well as handling accountSale messages. The runtime structure of the component is shown in Figure 28. It internally consists of three inner boxes, namely a Reporting, a Store box, and a ProductDispatcher box. The ingoing ports are forwarded to the Reporting and the Store box, respectively. In addition, the Application component has an ingoing accountSale method, which is handled internally and not shown by the MCowch diagram. The Store box uses the ProductDispatcher box to realize the product exchange among different stores (Use Case 8). All three inner boxes need access to the DataIf interface which is done by forwarding their corresponding ports to the outgoing data port.

10.3.17 Inventory::Data

The reference implementation of the Inventory::Data component does not fit well into our model. We found two problems that make this task difficult. The Data component of the reference implementation provides three interfaces namely PersistenceIf, StoreQueryIf, and EnterpriseQueryIf. Instances of these interfaces can be created by the DataIf interface. In order to use the StoreQueryIf and the EnterpriseQueryIf, however, a client must first get an instance of the PersistenceIf and then obtain a PersistenceContext by calling the getPersistenceContext() method

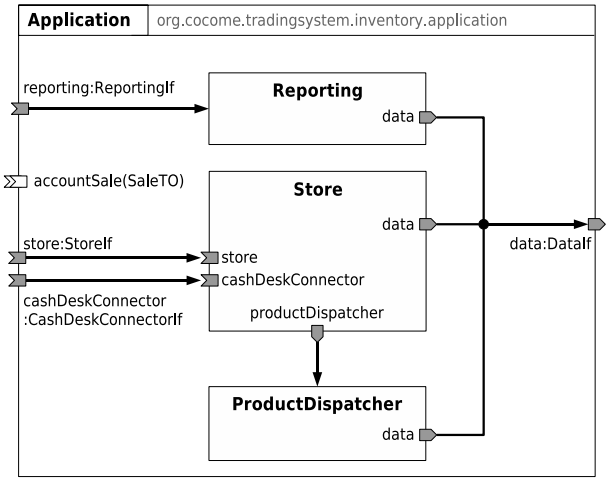

Fig. 28. The Inventory::Application component

on that interface. After that the client can use the other two interfaces, as all methods of these expect a PersistenceContext as an additional parameter.

This implementation does not follow an object-oriented programming style and thus does not fit well into our model. To give a more object-oriented model of the Data component, we changed the implementation in the following way. We removed the methods from the DataIf interface that allow to gain references to the EnterpriseQueryIf and the StoreQueryIf interfaces. We left PersistenceIf as it is, but we changed the PersistenceContext interface by adding two ports that allow access to the EnterpriseQueryIf and StoreQueryIf interfaces. We then could simplify the EnterpriseQueryIf and StoreQueryIf interfaces, as we were able to remove all PersistentContext parameters. The new interfaces are shown in Figure 29 The interesting aspect is that the COWCH approach led to a cleaner, more object-oriented design.

10.3.18 Missing Parts

Because of space reasons we did not present all parts of the CoCoME system. We left out the Inventory::GUI component as well as the architecture of the Inventory::Data component. We also mainly showed the architectural diagrams instead of the JCOWCH models, which we believe are more interesting.

10.4 Analysis

The COWCH approach provides a hierarchical structuring of runtime components and their communication structure. The runtime structure is the cornerstone to enable a modular specification of the behavior of components. Modularity

```
public interface DataIf {
  port PersistenceIf persistenceManager;
}

public interface PersistenceIf {
  PersistenceContext getPersistanceContext();
}

public interface PersistenceContext {
  port EnterpriseQueryIf enterpriseQuery;
  port StoreQueryIf storeQuery;
  port TransactionContext transactionContext;
  void makePersistent(Object o);
  void close();
}

public interface EnterpriseQueryIf {
   TradingEnterprise queryEnterpriseById(long enterpriseId);
   long getMeanTimeToDelivery(ProductSupplier supplier,
                              TradingEnterprise enterprise);
}

public interface StoreQueryIf {
  Store queryStoreById(long storeId);
  Collection<Product> queryProducts(long storeId);
  Collection<StockItem> queryLowStockItems(long storeId);
  Collection<StockItem> queryAllStockItems(long storeId);
  ProductOrder queryOrderById(long orderId);
  StockItem queryStockItem(long storeId, long productbarcode);
  StockItem queryStockItemById(long stockId);
  Product queryProductById(long productId);
  Collection<StockItem> getStockItems(long storeId, long[] productIds);
}
```

Fig. 29. The new interfaces of the Inventory::Data component

is a key-property for the scalability of any analysis technique. So we see our approach as a basis to enable scalable analysis for object-oriented programs. Our approach is still at the beginning. We plan to develop behavioral specifications of components to be able to specify and verify functional properties.

While (manually) applying the COWCH approach to the CoCoME system we discovered some design flaws as well as errors. The main design flaw we found was the realization of the Data component. The COWCH approach forced us to find a different design. The result is cleaner and more object-oriented than the original design. Other errors we found were mainly inconsistencies between the reference implementation and the documentation. We found one "real" error in

the reference implementation. Due to the explicit communication structure of the MCowch models, we discovered that the reference implementation of the CardReaderController did not receive any messages. Thus the card reader was not disabled when the ExpressModeEnabledEnabled event was sent. Other errors have been posted to the CoCoME forum [22].

10.5 Tools

As our approach is still in an early phase, we cannot yet provide finished tools. However, we currently work on three different tools. First, we work on a library-based implementation of JCowch in Java called JCowchLib. It is based on Java's reflection mechanisms, thus it works with standard Java tools. Second, we work on a modification of the Eclipse Java Development Tools [23] called JCowchDT. This will lead to an integrated development environment for JCowch. And third, we work on a tool called CowchGM which is based on the Eclipse Modeling Framework [24]. The goal of this tool is to be able to create MCowch models. At a later stage we will merge CowchGM with JCowchDT to ensure the consistency of the architectural model and the underlying implementation. For certain kinds of models it will even be able to directly generate working code.

10.6 Summary

The Cowch approach is still at its beginning and under development. The CoCoME modeling contest gave us the opportunity to evaluate our approach on a larger example system. During the modeling of the system we gained valuable insights which will influence the further development of our approach.

10.6.1 Limitations

With the architectural description language we are only able to show the structure of object-oriented systems. This includes the component structure as well as the communication structure. Currently, we are not able to specify functional and nonfunctional properties.

10.6.2 Future Work

In this paper we described our architectural description techniques and applied it to the CoCoME system. Consistency checking between architectural, executable, and implementation layer is currently done manually without tool support. In addition, the semantics of the executable layer is only partially formalized. We have a formal model of the Box Model semantics which we explain elsewhere [10]. As our concurrency model is based on the join calculus [9] we have a solid formal basis for this part as well. We are currently working on a formal description of the port and channel constructs which is the third part of our Java extension.

The architectural modeling is the basis to allow modular behavioral specifications of components. For the future we plan to develop behavioral specifications techniques to describe the behavior of concurrent object-oriented components and to be able to specify and verify functional properties.

10.6.3 Response to the Jury Comments

The main criticism of the Jury is that the approach only provides weak abstraction mechanisms and that the modeling technique is too close to the code level.

Our approach considers three layers (cf. Figure 1). To keep consistency between these layers manageable, we assume certain semantic similarities for the languages used on these layers. In principle, the approach is open to use less coupled languages as long as the consistency criteria is expressible.

It is true that the level of abstraction of our approach is not as high as compared to UML. However, both modeling layers in Cowch support well-defined and helpful abstractions. The executable modeling layer allows us to abstract from communication technologies and distribution and provides a more abstract notion of concurrency and synchronization. The architectural modeling layer allows to abstract from method implementations, from the process of creating and initializing components, and from dynamic system reconfigurations.

Acknowledgments

We thank the reviewers Antonio Cansado, Eric Madelaine, and Ludovic Henrio for their helpful comments on previous versions of this paper.

References

[1] Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. 26(1), 70–93 (2000)

[2] Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Schäfer, W., Botella, P. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)

[3] Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)

[4] Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Softw. Eng. 21(4), 336–355 (1995)

[5] Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Trans. Softw. Eng. 21(4), 314–335 (1995)

[6] Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 187–197. ACM Press, New York (2002)

[7] Seco, J.C., Caires, L.: A basic model of typed components. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 108–128. Springer, Heidelberg (2000)
[8] Sreedhar, V.C.: Mixin'up components. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 198–207. ACM Press, New York (2002)
[9] Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1996), pp. 372–385. ACM Press, New York (1996)
[10] Poetzsch-Heffter, A., Schäfer, J.: A representation-independent behavioral semantics for object-oriented components. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468. Springer, Heidelberg (2007)
[11] Szyperski, C.: Component Software - Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Publishing Company Inc, Reading (2002)
[12] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification, 2nd edn. Addison-Wesley, Reading (2000)
[13] Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6) (2004)
[14] Poetzsch-Heffter, A., Geilmann, K., Schäfer, J.: Infering ownership types for encapsulated object-oriented program components. In: Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm. Springer, Heidelberg (to appear, 2007)
[15] Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, Cambridge (1999)
[16] Scholten, J.G., Arbab, F., de Boer, F.S., Bonsangue, M.M.: Mobile channels, implementation within and outside component. Electronical Notes in Theoretical Computer Science 66(4) (2002)
[17] Institut National de Recherche en Informatique et en Automatique: JoCaml (2007), `http://jocaml.inria.fr`
[18] Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 415–440. Springer, Heidelberg (2002)
[19] von Itzstein, G.S.: Introduction of High Level Concurrency Semantics in Object Oriented Languages. PhD thesis, University of South Australia (2005)
[20] Lea, D.: Concurrent Programming in Java, 2nd edn. Addison-Wesley, Reading (2000)
[21] Woolf, B.: Null object. In: Martin, R.C., Riehle, D., Ruschman, F. (eds.) Pattern Languages of Program Design, vol. 3, pp. 5–18. Addison-Wesley, Reading (1998)
[22] The CoCoME forum (2007), `http://naf.informatik.uni-kl.de/php/phpBB2/index.php`
[23] The Eclipse Foundation: Eclipse Java Development Tools (JDT) (2007), `http://www.eclipse.org/jdt/`
[24] The Eclipse Foundation: Eclipse Modeling Framework Project (EMF) (2007), `http://www.eclipse.org/modeling/emf/`