Journal of Universal Computer Science, vol. 14, no. 13 (2008), 2158-2181
submitted: 31/10/07, accepted: 27/6/08, appeared: 1/7/08 © *J.UCS*

An Adaptation Logic Framework for Java-based Component Systems

Enrico Oliva, Antonio Natali, Alessandro Ricci, Mirko Viroli
(Alma Mater Studiorum–Università di Bologna, Cesena, Italy
{enrico.oliva,antonio.natali,a.ricci,mirko.viroli}@unibo.it)

Abstract: This paper describes a Java-based framework for developing component-based software systems supporting adaptation with logic laws and considering component interactions as a first-class aspect.

On the one side, the framework makes it possible to specify the logic of interaction at the component-level, in terms of input and output interfaces, the events generated and observed by a component, and related information about the management of the control flow. On the other side, it is possible to specify the logic of interaction at the inter-component level, providing a modelling and linguistic support for designing and (dynamically) programming the glue among the components, enabling general forms of adaptation, observation and construction of the interaction space.

As a result, the framework supports the adaptation of components at different levels: from interoperability among heterogeneous and unknown components, to the support for dynamic introduction, removal and update of components, to general coordination patterns, such as workflow.

The framework uses first-order logic as the reference computational model for describing and defining the logic of interaction: the modalities adopted by components to interact, the adaptation laws gluing the components and the interaction events occurring in the system are expressed as facts and rules. They compose the (evolving) logic theories describing and defining the interaction at the system level, and can be observed and controlled at runtime to allow dynamic re-configurability.

Key Words: software adaptation, software component, logic programming
Category: D.2.11, D.2.2, D.1.6, D.1.5

1 Introduction

Nowadays component-based technologies and frameworks (often referred to as *componentware*) can be considered mainstream approaches for designing and developing complex software systems [Szyperski *et al.*, 2002]. Examples of most used frameworks include EJB (Enterprise Java beans) as part of the J2EE architecture, CCM (CORBA Component Model) as part of CORBA middleware, and DCOM/COM+ [Szyperski *et al.*, 2002]. Also some service-oriented frameworks, such as OSGi [OSGi, 1999] and .NET, can be considered essentially as component-based frameworks, where components are called services.

Generally speaking, existing mainstream approaches are all essentially based on a sort of "*LEGO*-like" vision of software systems: the focus is on the notion of component as a basic brick to compose systems, both at design and runtime. The

composition is made possible essentially by explicitly declaring the interfaces that a component *provides* for exploiting its services and *requires* for being able to realise its services. Interfaces act as the formal description of the dependencies which connect together the components — as the joints for (LEGO) bricks. Accordingly, this leads software engineers to reason on application design and development in terms of structural composition of entities.

Actually, such an approach can be considered quite weak when dealing with the engineering of modern software systems, where component interactions and related dynamics are essential elements. Current mainstream approaches do not provide first-class support for specifying and managing interactions among components: most of the support concerns solving static dependencies where components are (dynamically) introduced or removed from the system. Back to the LEGO-metaphor, it is not sufficient to have bricks which are composed and linked together for asserting that the overall brick construction works from a dynamic point of view: some kind of dynamics and interaction can lead the overall system to break down, even if the bricks are (statically) connected in a right way.

In this work we present a framework for supporting component-based systems on top of object-oriented mainstream technologies such as Java, which provides a first-class support for representing, enacting and controlling the interactions inside the system. The approach does not consider the individual component as the center of the design and development of a component-based systems: this role is instead played by the *the logic of interaction*, which glues components together, according to a notion of interaction richer than the one that can be specified e.g. with standard object-oriented interfaces. In particular, the framework makes it possible to characterise the logic of interaction at two different levels: at the component level, specifying the interactive capabilities of individual components; and at the system level, specifying the laws that define and govern interactions which do not concern a specific component of the system, but characterise the overall ensemble of the components together—in a way similar e.g. to [Murillo *et al.*, 1999]. In particular, through the logic of interaction it is possible to express the adaptation behaviour among the components: that is, the laws specify some adaptation "policies". Observing interactions and properly re-acting to them is what closes the feedback loop of control, enabling the adaptation behaviour.

In the overall, the framework makes it possible to design and develop component-systems adopting mainstream technologies — including other component-based framework such as OSGi and EJB — but providing a support for managing interactions at a higher level of abstraction, focussing on the logic of interaction. As a main application case for this framework, we show that it can support component adaptation as defined e.g. in [Bracciali *et al.*, 2005];

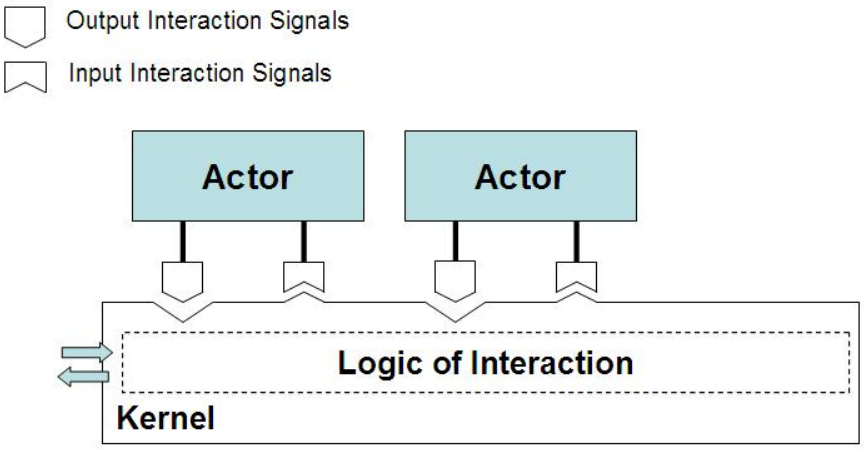


Figure 1: Architectural view of components and kernel

namely, given a theory of adapation computing the proper glue process that can adapt the behavioural interface of existing components, our framework provides a simple means to implement such adaptation using an expressive logic-based specification.

The remainder of this article is organised as follows: Section 2 presents the principles grounding the framework, Section 3 describes how the framework is realised on top of the Java platform, Section 4 focuses on a concrete instance of kernel based on first-order logic, Section 5 exemplifies the approach describing how adaptation policies can be supported, and finally Section 6 discusses related works.

2 The Framework: Vision

The framework introduces two first-class abstractions to represent the components and the environment where they are immersed, so as to better support both the micro (component) and macro (system) levels: *actors*[1] and *kernels* (see Figure 1).

Actors play the role of components of a system, as the basic unit of deployment, embedding some kind of business logic. They are meant to execute some kind of task such as the provision of a service, triggered by the reception of

[1] In spite of the name, the notion of *actor* is not directly linked to the actor abstraction as introduced by Carl Hewitt, but it rather refers to a component capable of interacting, as explained in the following

some form of stimuli. As components, actors can be introduced and removed dynamically into / from the kernel.

Kernels explicitly represent component environments, providing actors with specific services for supporting their interaction. A system then is composed by a kernel and a dynamic set of actors, linked and connected through the same kernel. To some extent, the kernel abstraction is similar to the notion of *container* as found in current component frameworks, extended toward the idea of configurable and programmable coordination medium [Denti *et al.*, 1997]. As happens for the actors, also kernels can be dynamically extended and replaced.

2.1 Interaction Signals and Interaction Primitives

From the interaction viewpoint, actors can be conceived as normal objects with the capability of generating and perceiving *interaction signals*. In particular, they provide their service by reacting to the reception of some interaction signals, and trigger the execution of services by generating signals.

Interaction signals are the basic bricks of the vocabulary of interaction, and are used to define the logic of interaction characterising the component-system. In this framework such a notion is represented in the simplest way, as a couple *(n, v)*, where n identifies the name of the signal and v the information content. Each component is characterised by the set of interaction signals that it can eventually generate (output interface) and the set of interaction signals that it can receive (input interface) during its life. Such sets must be explicitly defined for each component and are declared / published in the environment when the component is introduced in the system. So, interaction signals are meant to specify a form of interaction among actors minimising the (static) dependencies among them: the components do not interact directly with other components directly knowing their references and invoking methods, but indirectly generating and perceiving shared set of signals; components are indirectly connected so as to minimise dependencies, which is a key point for enabling adaptation behaviour based over the kernel abstraction.

Concerning the output interface, the kernel provides actors with a basic set of *interaction primitives*, which actors can use to generate interaction signals. Such primitives are actually important for characterising some basic aspect of the interaction *semantics*, in particular the attitude or intention of the act and what is expected from that act. Currently the basic set accounts for three primitives:

– `notify` - This is used to emit an interaction signal to make some kind of information related to the state or the behaviour of the component observable to all the components that are interested in it — namely, those that declared the signal in the input interface. The actor emitter is not interested in receiving any kind of information as a result of the operation.

- `inform` - This is used to emit an interaction signal to inform its environment of some information, in order to trigger some kind of activity or to answer to a request received in the past. The primitive succeeds if (and when) the information has been completely delivered into the environment (all the interested components have been informed), otherwise an *InformException* is generated; note no reply is actually expected from an `inform`.

- `invoke` - This is used to emit an interaction signal to execute a service and receive the corresponding result. The primitive works then as a traditional RPC or method invocation, looking for one available component (non-deterministically), with the result provided as the return parameter of the call. The primitive generates an exception `InvokeException` if the service cannot be delivered.

It is worth noting that all the primitives are meant to generate a signal without specifying the target actor: the component or set of components that will receive the signal depend on the specific logic of interaction defined for the system and enacted by the kernel, as shown in next subsection.

Generally speaking the set of primitives defines and constraints the expressiveness of the interaction support provided by a kernel. The objective here is to factorise the interaction needs that are most frequently found when building component-based systems, abstracting away from how interaction takes place (e.g. either through message passing or shared memory, either local or distributed) and which technology is used (RMI, Web Services, Plain Java Objects, CORBA, and the like), focussing exclusively on the logics of the interaction. Accordingly, the same primitives — with the same semantics — could be supported at the deployment stage by different kinds of kernel, adopting different kind of implementation strategies and technologies, depending on the computational and hardware environment.

Dually to the set of interaction primitives to emit signals, each actor must provide an interface with a `doAction` operation, which is used — from the environment point of view — to obtain the services that the actor is able to deliver. In particular, `doAction` specifies the behaviour of the actor reacting to the reception of any interaction signal that the actor declared among its input signals. The operation can directly return some result — representing the return value of the service invoked, and can generate a `DoActionException` to represent some kind of runtime error related to the execution of the service, for instance a (semantic) violation of the contract due to wrong arguments. The execution of the service can result also in the generation of output signals (using `notify` / `inform` / `invoke` primitives), for instance for notifying some kind of event or for executing some other services.

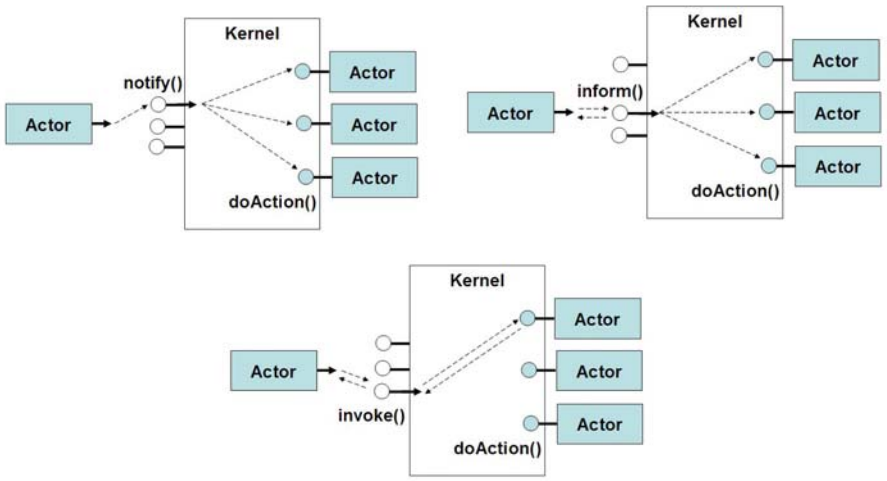


Figure 2: Kernel default behaviour for *notify*, *inform* and *invoke* primitives

2.2 Kernel mediation and Interaction Laws

The role of the kernel is then to act as the glue which enables, mediates and controls the generation of output interaction signals of some components which can become input interaction signals for other components. In other words, from a logical point of view, the kernel plays the role of an *adapter*, factorising services for managing component dependencies and dynamic interactions.

The default behaviour of a kernel is to enable interactions based on the name of interaction signals that actors declared to generate or to perceive. In particular (see 2 for a graphical description):

- a signal generated by an actor with a `notify` causes the execution of the `doAction` operation — with the interaction signal as a parameter — of *all* the actors that listed the signal among the input ones. The emitter actor is not interested in knowing any information about the effects of the invocation, so the kernel could e.g. to do its best in order to realise the call as asynchronously as possible;
- a signal generated with an `inform` causes the execution of a `doAction` on *all* the actors that listed the signal among the input ones. The primitive succeeds if the kernel is able to deliver the signal to everyone, i.e. to execute the `doAction` on all the actors, in spite of the possible generation of a `DoActionException` by each actor.
- a signal generated with an `invoke` causes the execution of `doAction` on *one*

actor chosen (in principle non-deterministically) among all the actors that listed the signal among the input ones. The return value of the `invoke` primitive is directly the result provided by the `doAction` operation. In particular, the kernel provides an adaptation behaviour finding an actor that executes the action without failures. So, if the execution of the operation on the chosen actor fails (with the generation of a `DoActionException`), another actor is to be selected from the remaining ones and the operation `doAction` is to be executed again on it. If no actor is found providing the services without exceptions, then the `invoke` fails by generating an `InvokeException`.

Besides these basic interaction primitives, the kernel can actually be extended to provides services for defining *interaction laws* in order to directly support some basic patterns of interaction, beyond the basic gluing behaviour. These laws can be specified during the (re-)configuration stage of the system, which can take place anytime during the execution of the applications. The patterns currently supported in the framework for adaptation are actually some of the most frequently used ones in mainstream component-based systems, such as Enterprise Java Beans, but working here at a higher level of abstraction:

- *event-listening* – the kernel provides a support for allowing a dynamic set of listener / reacting actors to observe a specific interaction signal generated by a specific emitter actor;

- *interaction-vetoing* – the kernel provides a support for realising vetoed interactions, i.e. interactions which actually take place only if no registered actor issues a veto. More precisely, the kernel service makes it possible to specify that a specific input signal for a specific receiver actor could be vetoed by a certain vetoer actor; dynamically, an interaction signal directed to the receiver is actually dispatched to the component only if none actors specified as vetoers disagree.

More complex laws can be obtained by composing the specification of multiple simple reactions and veto rules. Others are currently investigated to realise more adaptation-oriented interaction patterns, enriching the basic support provided by the kernel. Examples include the ability to specify constraints such as the order in which listeners are to be informed, or atomicity/consistency as in transaction-like scenarios.

It is worth noting that enriching the description of interaction aspects with semantic information improves the support for the principle of *local development* of components, and — more generally — for engineering open and extensible systems. Components are typically designed and developed without an a-priori knowledge of the specific environments where they will be deployed to; the availability of information concerning the semantics of the interaction of a component

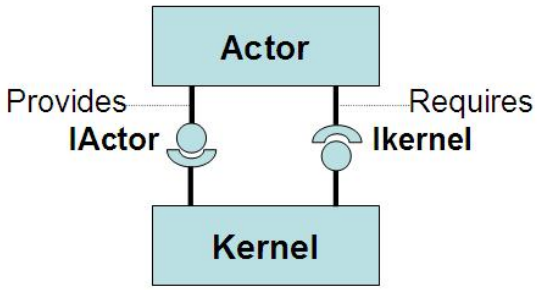


Figure 3: Architectural view of elements in the framework

— beyond the pure syntactic aspects — simplifies their integration and dynamic gluing by the kernel: for instance, this is achieved by applying some kind of adaptation rules to enable interoperability among components in spite of syntax and semantics mismatches among the interactions signals generated / perceived.

2.3 *Wired* and *In-The-Space* Interaction Modalities

The kernel realises its mediation role by injecting into the actors the logics necessary to realise interactions. In particular, this can take place according to two basic different modalities, called *in-the-space* and *wired*, which can basically be seen as different implementation approaches for the kernel. In the former, the kernel is actually a logical and *runtime* entity, shared and accessed any time a component is generating signals or is stimulated with signals; in this case the logics injected into the components simply provides for basic interaction acts towards the kernel. In the latter, all the peer-to-peer logics of interaction is *injected* in the components, without any runtime centralising entity. In other words, in the wired case the kernel is completely distributed and injected directly in the components; the component system at runtime becomes an interaction network, with actors playing the roles of the nodes, logically immersed in a shared environment, but actually wired in order to have direct, non-mediated interaction.

3 Specification and Implementation Issues

In this section we describe the main aspects of the current design of this framework, including specification of architecture and implementation details. In Figure 3, the elements that compose the framework are represented. On the one hand, an actor component should provide the interface `IActor` — namely by implementing it —, defining the operations that it makes available to the other

actors and to the kernel. These include the methods to configure the actor itself, as well as the method implementing the services realised by the actor, used to receive signals. On the other hand, the actor component should require the interface `IKernel` — namely, the kernel referenced by the actor should implement the `IKernel` interface. This interface includes the methods to register an actor to the kernel, to declare its input and output signals, and to invoke kernel interaction primitives (to emit signals). It should be noted that by this design choice we exclude the ability of legacy components to be used in our framework: the only solution to this problem is to wrap them into Java objects that properly provide `IActor` interface and require `IKernel` interface.

Table 1 shows a possible way to classify these operations. The `IActor` interface is used at configuration-time to inject the kernel into the actor instance, and by the kernel at interaction-time to invoke services. Dually, the `IKernel` interface is used at configuration-time to register an actor and its signals, and by the actors at interaction-time to invoke interaction primitives.

Operations	`IActor`		`IKernel`
	`IActorSpecification`	`IActionBase`	
Configuration time	Injecting and configuring the kernel		Registering and declaring signals to the kernel
Interaction time		Requesting execution of actions/services	Invoking the interaction primitives

Table 1: Interfaces structure

3.1 IActor

In the actual incarnation of our framework an actor is expressed as a Java class, which has to implement the standard interface `IActor`:

```
interface IActor extends IActionBase, IActorSpecification {}
```

This interface simply extends `IActionBase` and `IActorSpecification`, respectively describing interaction-time and configuration-time functionality. The former simply provides method `doAction` — which has the semantics described in previous section — used to execute a service realised by the actor. In particular, this is invoked by the kernel as a response of a request coming from another actor, achieving both the execution of a service and the return of a result. One

such invocation can also fail for a number of reasons — wrong arguments, failures in accessing back-end services, and so on — in which case the execution throws an exception.

```
interface IActionBase {
  Object doAction(String actionName,Object arg)
            throws DoActionException;
}
```

The argument `actionName` represents the name of the service requested, the argument `arg` the input information provided for describing details of the requested service; the output result is given type `Object` for generality.

The interface `IActorSpecification` provides all the operations used at configuration-time, by which the presence of the actor in the system can be configured.

```
public interface IActorSpecification {
  public String getName( );
  public String [] getInputSignals();
  public String [] getOutputSignalsInform();
  public String [] getOutputSignalsNotify();
  public String [] getOutputSignalsInvoke();
  public void setKernel(IKernel kernel);
  public IKernel getKernel( );
  public boolean isActive();
}
```

The method `getName` returns the name of the Actor — unique in the running application. The method `getInputSignals` is used by the kernel to retrieve all the input signals that the actor is able to process, namely, the `actionName` it is willing to accept by a `doAction`. The methods `getOutputSignalsNotify/Inform/Invoke` return the output signals that the actor can generate, namely the list of `actionName` for the services it can request — either through a `notify`, `inform`, or `invoke`. The methods `setKernel` and `getKernel` store and retrieve the reference to the kernel where the actor is connected to: hence, our framework injects the referenced kernel into the referencing actor through the setter method. In this way, the kernel itself gets aware of the existence of components, and can perform the proper checks like name uniqueness. In the current version of the framework the actors implement other interfaces that allow to inject in the actor also some basic support of the JavaBeans component framework.

3.2 IKernel

`IKernel` is the standard interface which any kernel has to implement, providing those methods that each actor has access to — in order to either interact with others or to register its input and output signals.

```
interface IKernel {
  void notify(IActor emitter,String signalName,Object args);
  void inform(IActor emitter,String signalName,Object args)
        throws InformException;
  Object invoke(IActor emitter,String signalName,Object args)
        throws InvokeException;

  void declareNode(String name, Class clazz, Object obj);
  void declareInputSignals(IActor receiver, String[] signals);
  void declareOutputSignalsNotify(IActor emitter, String[] signals);
  void declareOutputSignalsInform(IActor emitter, String[] signals);
  void declareOutputSignalsInvoke(IActor emitter, String[] signals);
}
```

As an actor can invoke a service in three different styles, this interface provides the three corresponding methods `notify`, `inform`, and `invoke`. Method `notify` is used to send a signal to interested actors without actually caring about any reply result or either any acknowledgment, hence it throws no exception. Method `inform` is used to send a signal to interested actors: no result is returned, but the end of the operation means that `all` the interested actors processed the signal. Finally, method `invoke` is used to request a service to one agent that can execute it, correspondingly receiving a reply. In all cases, the kernel has the burden to retrieve actors (one or more) able to execute a service with the specified name, invoke their `doAction` name, and properly providing acknowledgment/reply to the emitting actor.

The other methods are used by the actor to register information about its interface — in the component-based acceptation of the term. Methods `declareNode`, `declareInputSignal` and `declareOutputSignals`, respectively register the presence of the actor in the system, its input signals (the services it realises), and its output signals (the services it invokes on other actors).

3.3 Interaction Laws

Other than providing a basic interaction support, conceptually linking input and output signals and guaranteeing the three different semantics of service requests, a kernel can be implemented so as to support interaction laws. These are used in all those cases where a more advanced adaptation ability is to be charged upon the kernel. As explained in previous section, examples of such laws include those supporting event-listening and vetoer semantics.

Each such law is associated with a proper interface that the kernel class has to implement. This interface provides the method (or methods) used to configure the interaction law, thus extending the underlying semantics of subsequent calls to methods `IKernel.notify`, `IKernel.inform`, and `IKernel.invoke`. This mechanism is thus used to change the default semantics of a kernel, where signals are associated to output signals solely based on the matching of their names.

For the event-listener interaction law, we have for instance the interface:

```
public interface IReactInteraction {
  public void reactInteraction(
    IActor reactor, IActor emitter, String signalName );
}
```

Method `reactInteraction` is to be implemented to realise the pattern publish-subscribe: this is used to register the `reactor` to receive invocations of the signal with name `signalName` executed by the actor `emitter` — namely `reactor` will observe actions `signalName` of the `emitter`. One such law constrains the space of interaction and limit the notify method of an actor for a determined signal and only to the actor indicated in `reactInteraction`. Actually `reactInteraction` can be exploited in the framework also for supporting the wired modality as described in previous section. In particular, by calling a set of `reactInteraction(O,E,S)` we fix the set of specific observers {`O`} that can observe the signal `S` emitted by `E`. By doing so, at configuration time the kernel (in the `reactInteraction`) can inject in the emitter actor a support for sending the signal directly to the specified observers, without the mediation of the kernel itself.

Similarly, the vetoing functionality is supported by interface:

```
public interface IVetoInteraction {
  public void vetoInteraction(
    IActor vetoer, IActor receiver, String signalName );
}
```

By calling method `vetoInteraction`, the kernel is configured so that actor `vetoer` can negatively reply to an output signal `signalName` produced by actor `receiver`.

These laws are just a subset of those a kernel can implement: further laws can be realised by adding new interfaces.

4 A Logic-based Kernel

While developing our framework, we experimented various implementations for the kernel, providing different ways to represent and manage interactions based on different kinds of lower-level technologies.

Among the others, we found the logic programming paradigm quite useful. The corresponding kernel, called *Logic Kernel*, adopts first-order logic for describing and enacting the logic of interaction, including both the interaction capabilities of individual actors, and the adaptation laws which define how the interactions are globally managed. In other words, the kernel handles as logic theories both the configuration of the system — actors immersed in the environment, their set of input / output interaction signals, and the laws governing

interactions — and the interaction events that dynamically occur. The mediation and adaptation activities of this kernel are then realised by exploiting a logic engine (based on Prolog), properly handling the occurring interactions based on the interaction laws and the actors configuration.

4.1 Implementation

This kernel is realised following the "in the space" modality, namely, as a run-time abstraction where interaction signals are reified and properly managed. It is implemented through a class `LogicKernel` implementing interface `IKernel` — namely, a component providing the `IKernel` interface. Moreover, it also implements the interface `IContextLocal` that provides functionality to load, save and execute a logic theory, configuring and modifying the kernel at run-time. The implementation of this class is based on the `tuProlog` open source project we developed [Denti *et al.*, 2005] (`http://tuprolog.sourceforge.net`). This is a lightweight Prolog engine and API written in Java which provides smooth integration of Prolog and Java programming, allowing to either represent and invoke Prolog goals from Java, as well as calling Java libraries within Prolog theories.

```
public interface IContextLocal {
  public alice.tuprolog.Prolog getPrologEngine();
  public void register( String term, Object obj );
  public boolean loadTheory( String absPath );
  public boolean saveTheory( String absPath );
  public String standardQuery( String queryS );
  public String query( String queryS );
  public String nextSolution( );
  public alice.tuprolog.SolveInfo solve( String queryS );
}
```

Basically, this interface provides a wrapper to the API of tuProlog, with methods to handle basic Prolog primitives to load and save theories, execute queries and retrieve solutions, and so on.

By exploiting these functions, the `LogicKernel` has to realise the methods provided by the `IKernel` interface. The methods supporting configuration simply cause a fact — also called here a tuple — containing information on the arguments to be reified in the knowledge base as follows:

`declareNode` — This method is used to register an actor in the kernel; an invocation is represented by the tuple `node(NodeName,Class)`.

`declareInputSignals` — This method is used to register the input signals an actor is interested in receiving; an invocation is represented by a tuple `reacts(Reactor,ActionName)` for each signal specified in the input array.

`declareOutputSignalsNotify/Inform/Invoke` — These three methods are used to register the output `notify/inform/invoke` signals an actor may receive; an invocation is represented by a tuple `declaresNotify(Emitter,ActionName)`, `declaresInvoke(Emitter,ActionName)`, or `declaresInform(Emitter,ActionName)`, for each signal specified in the input array.

These tuples are then actually seen as Prolog predicates `reacts/2`, `declaresNotify/2`, `declareInvoke/2`, `declareInform/2` and `node/2`, inserted dynamically in the knowledge based at configuration time.

Other than configuration details, also the occurrence of interactions between actors are inserted in the knowledge base dynamically. A method `trace` in class `LogicKernel` writes in the knowledge base a fact of the kind `out(Emitter, ActionName, Arg)`, where `Emitter` is the agent responsible for the interaction, `ActionName` is the signal name, and `Arg` is the signal argument.

When a method `notify`, `inform`, or `invoke` is called on the kernel, a corresponding prolog predicate `notifyInTheSpace/3`, `informInTheSpace/3`, and `invokeInTheSpace/4` is called, which is in charge of allowing the proper actors to perceive the signal, supporting the precise semantics of each of the three primitives.

The implementation of predicate `invokeInTheSpace/4` is as follows:

```
invokeInTheSpace(Emitter,ActionName,Arg,Res):-
  reacts(Reactor,ActionName),
  node(Reactor,Class),
  declaresInvoke(Emitter,ActionName),
  Reactor <- doAction(ActionName,Cmd) returns Res,
  !.
```

While the first three arguments are as usual, the last is an output, providing invocation result. The predicate orderly *(i)* retrieves a `Reactor` willing to accept the signal, *(ii)* checks whether it is registered as a node, *(iii)* checks whether the emitter declared the output signal, and finally *(iv)* invokes method `doAction`, returning result `Res`. Note that in tuProlog, binary infix predicate `<-` is used to invoke the method specified on the right-side over the Java object identified by the reference specified on the left-side — with the optional final part `returns` specifying the result. If such an invocation fails for some reason, predicate `<-` fails: for the backtracking semantics of Prolog this causes predicate `reacts` to find another solution, namely another `Reactor`. If the invocation is instead successful, the cut predicate `!` completes the execution. In the end, this preserves the semantics of `invoke` primitive: the kernel will keep looking for one (and precisely one) actor that successfully executes the service requested.

The implementation of predicate `notifyInTheSpace/3` is as follows:

```
notifyInTheSpace(Emitter,ActionName,Arg):-
  reacts(Reactor,ActionName),
  node(Reactor,Class),
  declaresNotify(Emitter,ActionName),
  Reactor <- doAction(ActionName,Arg),
  fail.
notifyInTheSpace(Emitter,ActionName,Arg).
```

Differently from the previous case, this predicate does not provide replies, but simply returns when its task is over. As a proper actor is found and its `doAction` method is invoked, meta-predicate `fail` causes the Prolog engine to backtrack and find another actor by predicate `reacts`. When no more such actors exist, the second clause positively terminates the invocation. Note that if some invocation of `doAction` would fail, this does not interfere at all with the engine execution. This behaviour preserves the semantics of `notify` primitive: the kernel should find all actors interested in the notification — the emitting actor being not interested about some registered actor not perceiving the notification.

Finally, the implementation of predicate `informInTheSpace/3` is as follows:

```
informInTheSpace(Emitter,ActionName,Arg):-
  assert(proceed(Emitter,ActionName)),
  reacts(Reactor,ActionName),
  proceed(Emitter,ActionName),
  node(Reactor,Class),
  declaresInform(Emitter,ActionName),
  retract(proceed(Emitter,ActionName)),
  Reactor <- doAction(ActionName,Arg),
  assert(proceed(Emitter,ActionName)),
  fail.
informInTheSpace(Emitter,ActionName,Arg):-
  proceed(Emitter,ActionName),
  retract(proceed(Emitter,ActionName)).
```

This is similar to predicate `notifyInTheSpace`. The main difference is that a fact `proceed` is reified in the space at the beginning and is dropped if some `doAction` fails. As it is dropped the execution terminates negatively, otherwise when all actors have been informed without exceptions the execution returns positively. This behaviour preserves the semantics of `inform` primitive: the kernel should find all actors interested in being informed — the emitting actor being interested in whether all registered actors correctly perceived the signal.

5 Application to Software Adaptation

In this section we provide two examples cases for our framework. The first example is introductory, and shows how a simple connection between components can be realised using our framework. The second example aims at showing how a formal model of adaptation can be supported by the framework, by translating an adaptation process into a proper logic-based specification for the kernel.

5.1 The Ping-Pong System

To give a flavour of framework classes and behaviour, here we consider a very simple system, referred to as *Ping-Pong*, made by two components which must be adapted by a simple rule. The source code of the Java classes implementing this example is reported in Figure 4. The components are represented by the classes `PingActor` and `PongActor`, referred here as respectively the ping actor and the pong actor. The behaviour of the components is very simple: they react to the reception of a specific input signal (`ping` for the ping actor, `pong` for the pong actor), and after doing their job (just sleeping in our implementation) they emit a specific output signal (`pong` for the ping actor and `ping` for the pong actor). Actors share the same interaction signals: the signal generated by an actor triggers the execution of the service by the other actor.

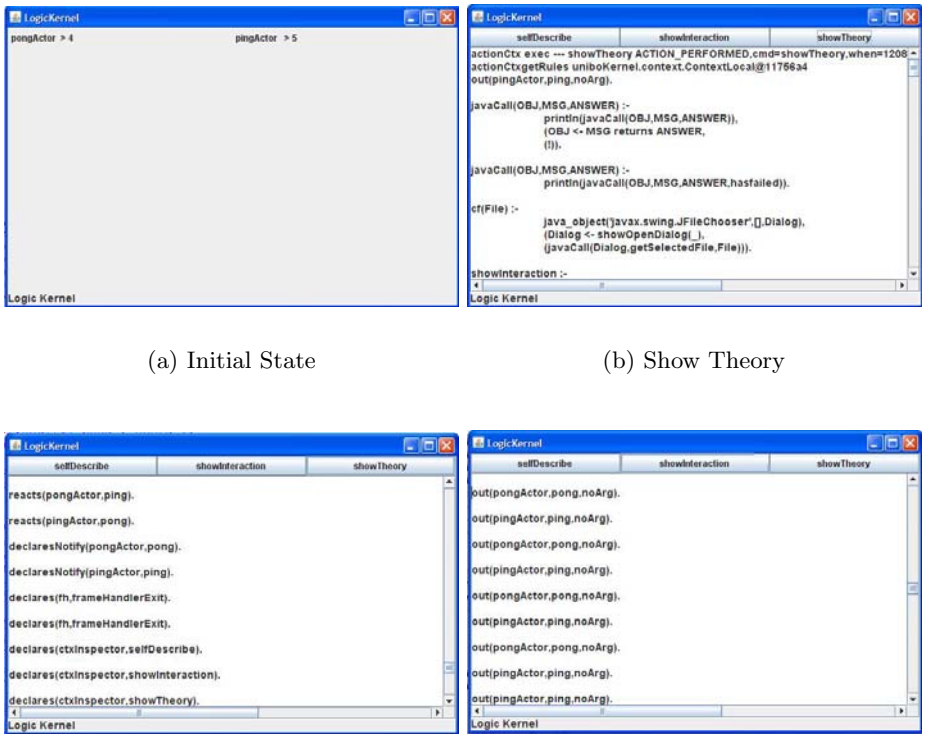
The simple adaptation rule that we want to realise accounts for stopping the interaction between the actors after N stages, i.e. after N generations of the `ping` - `pong` couple of signals. The rule must be specified and enforced without changing the behaviour of the individual actors. For this purpose, we define a vetoing interaction law, with a new actor acting as vetoer of the input signals notified to the ping actor. The vetoer essentially counts the number of times a `ping` signal is notified to the actor and gives its consensus for the delivery of the signal to the ping actor only if the number of signals is less than the N value.

Finally, in the main class the various parts of the system are created and configured, including the kernel, the actors and the vetoing interaction law making the `vetoActor` a vetoer for the input signal of `pingActor`. A `ping` signal is generated in order to trigger the activities of the components.

A main advantage of the logic kernel approach is that it allows for easily tracking the occurrence of interactions and their management, namely, the true run-time behaviour of the application. Figure 5 shows some screenshots of the `Inspector` tool of the framework, used to display all the relevant information about state and evolution of the logic kernel. The inspector tool can be used to debug application and to possibly modify the laws of interaction at run time, to see and experiment different system evolutions. So, typically the logic kernel is used in prototyping and debugging stages: when the logic of interaction has proven correct a more efficient version of the system can be obtained by

```
public class PingActor extends AbstractActor {
  public PingActor(String logo) {
    super(logo);
  }
  public void doAction(String actionName, Object args) throws DoActionException{
    try{
        Thread.sleep(1000);
        kernel.notify("ping","noArg");
    } catch(Exception ex){ throw new DoActionException(); }
  }
  public String[] getInputSignals() { return new String[] {"pong"}; }
  public String[] getOutputSignalsNotify() { return new String[] {"ping"}; }
}

public class PongActor extends AbstractActor {
  public PongActor(String logo) {
    super(logo);
  }
  public void doAction(String actionName, Object args) throws DoActionException{
    try{
      Thread.sleep(2000);
      kernel.notify("pong","noArg");
    } catch(Exception ex){ throw new DoActionException(); }
  }
  public String[] getInputSignals() { return new String[]{"ping"}; }
  public String[] getOutputSignalsNotify() { return new String[]{"pong"}; }
}

public class VetoActor extends AbstractActor {
  private int count;
  private int max;
  public VetoActor(String logo) {
    super(logo);
    count = 0;
  }
  public VetoActor(String logo, int max) {
    super(logo);
    count = 0;
    this.max = max;
  }
  public Object doAction(String actionName, Object args) throws DoActionException {
    return (count++ >= max);
  }
  public String[] getInputSignals() { return new String[]{"pong"}; }
  }
}

public class TestPingPong {
  public static void main (String [] args) {
    kernel = new LogicKernel();

    ping = new PingActor("pingActor" );
    ping.setKernel(kernel); //injection of the kernel
    pong = new PongActor("pongActor" );
    pong.setKernel(kernel); //injection of the kernel

    veto = new VetoActor ("vetoActor",4);
    veto.setKernel(kernel); //injection of the kernel

    kernel.vetoInteraction(veto,ping,ping.getInputSignals()[0]);

    kernel.notify("ping","noArgs");
  }
}
```

Figure 4: Code for the Ping-Pong Example



(a) Initial State (b) Show Theory

(c) Self Describe (d) Show Interaction

Figure 5: Inspector tool

wiring the interactions by means of the `reactInteraction` kernel primitive. In particular, in the Logic-kernel such a primitive wires the emitter and observers actors using the Java event-listener pattern. Figure 5(a) shows the initial screenshot of a system run, where the kernel shows a counter for the two actors that represents the number of signal emitted; the actors are stopped at fourth interaction by the vetoer. Three buttons are available in the inspector tool: button `showTheory` is used to visualise the internal Prolog theory for adaptation, as shown in Figure 5(b); button `selfDescribe` produces the actor configurations (input/output declared signals, and the like) such as `reacts(pongActor,ping)`, as shown in Figure 5(c) and button `showInteractions` shows the messages exchanged between the ping pong actors at execution time, by labeling them as `out(pongActor,pong,noArg)`, as shown in Figure 5(d).

5.2 Supporting an adaptation model

In literature there exist several model-based approaches to provide protocol adaptation, e.g. [Bracciali *et al.*, 2005, Inverardi and Tivoli, 2003]. In [Bracciali *et al.*, 2005] a formal approach is presented to define an adaptation process for components whose signature and behaviour are represented in terms of an algebraic approach. The result of the adapter derivation process is a formal specification of the adapter behaviour which is guarantee to be deadlock-free. In this section, we show how this model can be put into practice, since the results of adapter derivation could be implemented easily in our logic kernel by specifying adaptation in a declarative style.

As concrete example of adapter implementation, we refer to the FTP transmission system presented in [Bracciali *et al.*, 2005] (Section 5). Briefly, the problem is to adapt the behaviour of a client and server during the exchange of files—for more details, the reader should refer to [Bracciali *et al.*, 2005]. The behaviour of the client is expressed by the sequence of output actions

```
behaviour: (login!(usr).pass!(pin).
            getfile!(file).logout!( ).0)
```

representing indication of username, password, a file to be received, and finally a logout signal. The behaviour of the server is instead defined by the process:

```
behaviour: (open?(ctl).user?(name,pwd,ctl).(
                put?(fn,ctl).close?(ctl).0
                + get?(fn,ctl).close?(ctl).0
                + close?(ctl).0
           ))
```

First, the server receives a session identifier `ctl`, then receives name and password of the client, then either a `put`/`get` action followed by a `close`, or directly a `close` action. In [Bracciali *et al.*, 2005] it is shown how an adaptor specification can be realised that provides mappings `MA` between sub-processes of client and server, and then a true adaptation process `AA`, as follows:

```
MA = { login!(usr), pass!(pin) <> open?(ctl),user?(usr,pin,ctl);
       getfile!(file) <> get?(file,ctl);
       logout!( ) <> close?(ctl); }

AA = login?(usr).pass?(pin).(ctl)open!(ctl).user!(usr,pin,ctl).
     getfile?(file).get!(file,ctl).
     logout?( ).close!(crtl).0
```

The adapter `AA` specification, first consumes `login` and `pass` actions of the client, and correspondingly executes `open` and `user` of the server—a new session identifier `ctl` is generated within the adapter. Then, `getFile` of the client is mapped

```
count(0).
generate(N):-retract(count(N)),N1 is N+1,assert(count(N1)).

invokeInTheSpace(Emitter,'login',Cmd):-
   assert(mem(Emitter,'login',Cmd)),!.

invokeInTheSpace(Emitter,'pass',Cmd):-
   retract(Emitter,'login',Cmd1),!.
   generate(N),
   assert(id(Emitter,N)),
   reacts(Reactor,'user'),
   node(Reactor,Class),
   declaresInvoke(Emitter,'pass'),
   Reactor <- doAction('user',[Cmd,Cmd1,id(N)]) returns Res,!.

invokeInTheSpace(Emitter,'getFile',Cmd):-
   id(Emitter,N),
   reacts(Reactor,'get'),
   node(Reactor,Class),
   declaresInvoke(Emitter,'getFile'),
   Reactor <- doAction('get',Cmd) returns Res,!.

invokeInTheSpace(Emitter,'logout',Cmd):-
   reacts(Reactor,'close'),
   node(Reactor,Class),
   declaresInvoke(Emitter,'logout'),
   Reactor <- doAction('close',Cmd) returns Res,!.
```

Figure 6: Logic Kernel FTP Transmission Adapter

to a `get` in the server. Finally, `logout` of the client is mapped to a `close` in the server.

To implement the adaptor `AA` with our kernel we exploit the `invoke` primitive, used to realise 1-to-1 connections: components rely on the `invoke` primitive to execute output actions, while input actions are realised by the kernel which automatically calls the `doAction` method of the receiving components. In Figure 6 a specification of the adapter for the kernel is shown, which is realised by simply turning `AA` process into a logic-based specification. First of all, a `generate` predicate is implemented that yields an increasing new number each time, modelling the generation of a new session identifier for the client-server interaction. Then, the specification provides a rule to manage each possible output action (`login`, `pass`, `getFile`, `logout`), by intercepting calls to the `invoke` primitive and properly managing them according to the adapter specification. Action `login` simply causes a `mem` fact to be asserted in the theory; this is receovered when later a `pass` action is executed, in which case a new identifier `N` is generated, a fact linking the client and the identifier is stored, the reference to the server is retrieved by `reacts` declaration of `user` signal, the check of actor registration to the kernel is made by `node` predicate, the control of `pass` signal consistency with the emitter declaration is made by `declareInform` predicate, the `doAction` method of reacting actor is called with a list of three parameters: user name, password and channel. Action `getFile` and `logout` respectively cause two server reactions

to the signal `get` and `close`.

The client and server components must declare to the kernel their reference and their input and output signals. Currently it is out of the purpose of our framework, but an interesting future work, to realise inside the logic kernel the formal algorithm to automatically create the adapter and its implementation out of the behavioural interface of components.

6 Related Works and Discussion

In Component-Based Software Engineering (CBSE), component adaptation is a problem that is receiving increasing attention in the last few years. The concept of adapter and software component adaptation in CBSE environment is well introduced and explained in [Becker *et al.*, 2006]. In our work, we provide a component system based on Java where a "logic kernel" is introduced as the mediator of interaction, and as the place where component adaptation can be implemented. In particular, our kernel can be see as an adapter, where the adaptation process is expressed in a declarative style.

Authors of [Becker *et al.*, 2006] also introduce a useful classification of component interface with the consequent taxonomy of component mismatches: technical, signatures, protocols and quality attributes. Our kernel could be considered as realisation of the pattern adapter for component systems, providing a solution to most cases of mismatches: signatures, protocols and quality attributes.

There are several ways to put adaptation theories in to practice: in [McKinley *et al.*, 2004b] and with more detail in [McKinley *et al.*, 2004a] a taxonomy of several compositional adaptation approaches is presented, such as Aspect Oriented Programming (AOP), Computational Reflection, Component-Based Design and Middleware.

AOP approach enables separation of crosscutting concerns, to be implemented outside the components of the system. The adaptation involves in particular the quality of service and the ability to dynamically replace one concern with another. In our framework we roughly follow the AOP approach in that the possibility to substitute the logic kernel specification on-the-fly is enabled: in this sense, the adaption policy can be seen as a crosscutting concern of the application. This feature could also be useful in order to provide the kernel with different technological support and enabling adaptation between different platforms, since e.g. client and server could be built in different technologies—one as a web server, the other as RMI object. A somewhat similar approach on mainstream technologies is given by Composition-Filters, where declarative rules are superimposed for intercepting, filtering, re-routing, and changing the message traffic among objects to support certain inter- and intra-class cross-cutting concerns [Bergmans and Aksit, 2001]. Differently from composition filter we provide

a framework where the filter adaptation is explicitly represented in our kernel instead of internally to each object and thus more easily modifiable.

Computational reflection enables a system to change its behaviour without compromising portability. An adaptative system uses reflection to observe and change its inner behaviour. In our framework we do not exploit reflection inside the components but implicitly in the logic kernel, since Prolog-based specifications are intrinsically meta-programmable: they could inspect themselves and change accordingly, due e.g. to unpredictable situations like failure of components.

From the system point of view dynamical refactoring features are provided by the mainstream approach of object-oriented design and recently component-based design. In a component based framework it is possible to reconfigure the application at run-time, for instance adding or removing components. In our framework Actors could be considered as components that are independent units with a common interface. The interaction and late binding between components is provided by the kernel introducing a level of indirection that enables the compositional adaptation.

The Middleware approach is based on a layer of services between operating system and user application. Recently, research on adaptative software focussed on this approach: in the survey by Sadjadi [Sadjadi, 2003] the role of middleware is discussed in the context of complex software architectures, and it is shown how software technologies (AOP, Computational Reflection, Design Pattern) are used to support adaptation. The existing middlewares can implement several adaptation types that could be classified in two main categories: static adaptation vs. dynamic adaptation. Our logic kernel is a dynamic middleware where it is possible at run time to tune and change component configuration and rules of adaptation. These features are realised by exploiting a declarative language to express adaptation and interpreting at runtime the adaptation rules. The main advantages of using a Prolog-based specification is that its declarative style well fits the typical rule-based semantics associated to interaction protocols, and hence, to adaptation processes. Moreover, its meta-programming capabilities and its logical character can allow those forms of introspection and intelligent reasoning that might evolve the framework towards a full featuring adaptive and self-healing middleware—though this is a subject of future research.

7 Conclusions

The framework presented in this paper takes as a reference context component-based technologies and frameworks that are currently used in the mainstream, in particular based on object-oriented languages such as Java. The objective is to inject in such contexts some of the principles and visions that typically

characterise most of the approaches found in the research, such as the focus on interaction and adaptation as a main engineering dimension, and the introduction of first-class abstractions (media) for their specification and management (exogenous coordination).

The framework is an investigation of first-order logic to specify and representing the logic of interaction and the interaction / adaptation laws gluing the components and the interaction events actually happening at runtime. The mediated form of interaction realised by the logic kernel enables a centralised form of adaptation instead of a more classical approach of an adapter for each components.

Several research lines will be explored in future works. Among the others, we plan to: enhance the basic set of interaction / adaptation laws directly supported by the kernel; exploit of the logic-based kernel for the engineering of self-healing systems; define a formal model for the framework in order to specify and understand more rigorously the behaviour of component-based system built on top it; and finally, apply the framework to real-world examples of component-based systems.

References

[Becker *et al.*, 2006] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. *Towards an Engineering Approach to Component Adaptation*. 2006.

[Bergmans and Aksit, 2001] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, oct 2001.

[Bracciali *et al.*, 2005] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74(1):45 – 54, 2005.

[Denti *et al.*, 1997] E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In David Garlan and Daniel Le Métayer, editors, *Coordination Languages and Models – Proceedings of the 2nd International Conference (COORDINATION'97)*, volume 1282 of *LNCS*, pages 274–288, Berlin (D), 1–3 September 1997. Springer-Verlag.

[Denti *et al.*, 2005] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 2005. In press. Available at http://dx.doi.org/10.1016/j.scico.2005.02.001.

[Inverardi and Tivoli, 2003] P. Inverardi and M. Tivoli. Deadlock-free software architectures for COM/DCOM applications. *J. Syst. Softw.*, 65(3):173 – 183, 2003.

[McKinley *et al.*, 2004a] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. A Taxonomy of Compositional Adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, Michigan, 2004.

[McKinley *et al.*, 2004b] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56 – 64, 2004.

[Murillo *et al.*, 1999] J. M. Murillo, J. Hernández Núñez, F. Sánchez, and L. A. Álvarez. Coordinated roles: Promoting re-usability of coordinated active objects using event notification protocols. In Paolo Ciancarini and Alexander L. Wolf, editors, *Coordination Languages and Models, Third International Conference, COORDINATION '99, Amsterdam, The Netherlands, April 26-28, 1999, Proceedings*, volume 1594 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 1999.

[OSGi, 1999] OSGi. Osgi service platform. `http://www.osgi.org`, 1999.

[Sadjadi, 2003] S. M. Sadjadi. A survey of adaptive middleware. Technical report, Software Engineering and Network Systems Laboratory, Department of Computer Science and EngineeringMichigan State University, 2003.

[Szyperski *et al.*, 2002] C. Szyperski, D. Gruntz, and S. Murer. *Components Software: Beyond Object-Oriented Programming.* Addison-Wesley, 2002.