Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 154 (2006) 43–61

www.elsevier.com/locate/entcs

A Framework for Engineering Interactions in Java-based Component Systems

Antonio Natali, Enrico Oliva, Alessandro Ricci, Mirko Viroli

DEIS, Alma Mater Studiorum, Università di Bologna,
via Venezia 52, I-47023 Cesena, Italy
{anatali,eoliva}@deis.unibo.it
{a.ricci,mirko.viroli}@unibo.it

Abstract

This paper describes a Java-based framework for the development of component-based software systems supporting the specification of the logic of component interactions as a first-class aspect. Java is used as the reference development language.
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 observation, control and construction of the interaction space.
As a result, the framework supports the coordination 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 adopts first-order logic as the reference computational model for describing and defining the logic of interaction: the modalities adopted by components to interact, the coordination 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.

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 [13]. 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

doi:10.1016/j.entcs.2005.12.032

DCOM/COM+ and its future incarnations on the .NET platform [13]. Also some service-oriented frameworks, such as OSGi [11], 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 and do their job properly.

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 the overall, the framework makes it possible to design and develop component-systems adopting mainstream technologies — including other

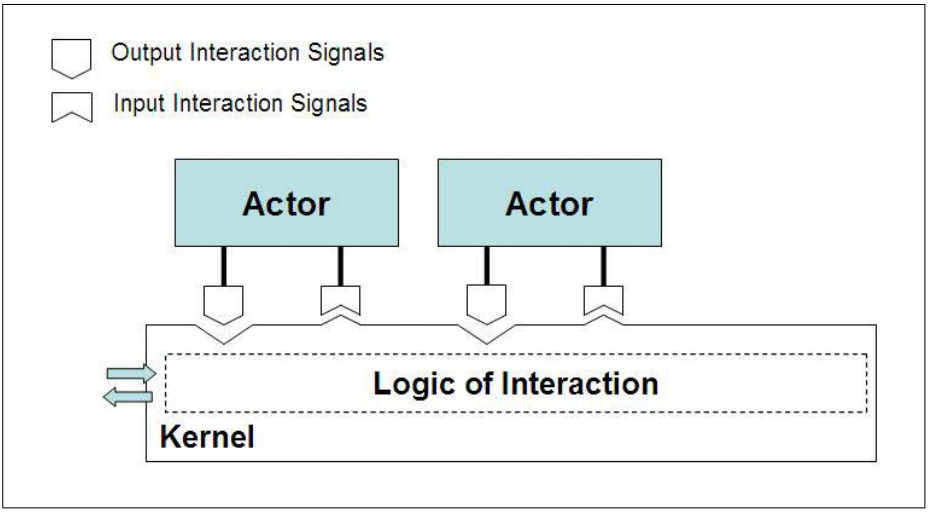

Fig. 1. Architectural view of components and kernel

component-based framework such as OSGi and EJB — but providing a support for managing interactions at an higher level of abstraction, focussing on the logic of interaction.

The remainder of this article is organised as follows: Section 2 presents the principles on which the framework is based, 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 a simple component-based system built on the framework, and finally Section 6 concludes the paper.

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

[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

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 [4]. 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 which can be 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.

For what concerns 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* - used to emit an interaction signal to make some kind of information related to the state or the behaviour of the component observable to other components. The actor emitter is not interested in receiving any kind of information as a result of the operation;
- *inform* - 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, otherwise a *InformException* is generated;

- *invoke* - 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, 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 the primitives defines and constrains 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, 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.

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,

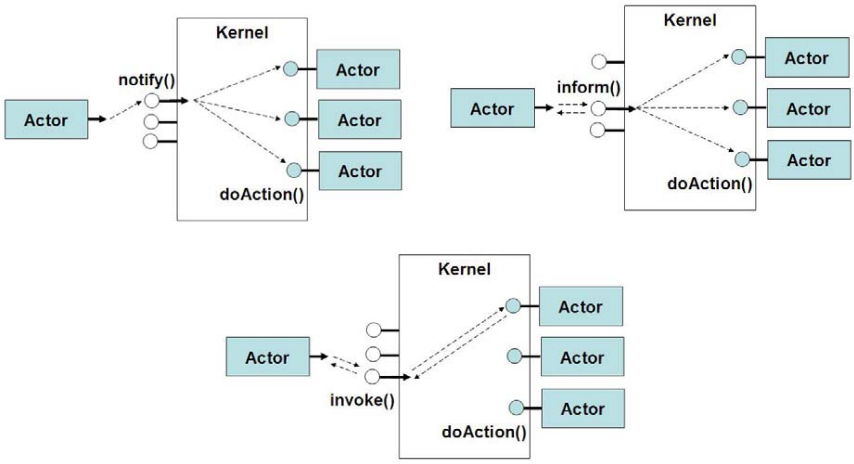

Fig. 2. Kernel default behaviour for *notify*, *inform* and *invoke* primitives

from a logical point of view, the kernel plays the role of a *interaction artifact*, 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 Fig. 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 *invoke* causes the execution of *doAction* on *one* actor chosen 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 is meant to provide the best effort to find an actor executing 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 — e.g. non-deterministically — 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 a *InvokeException*.
- 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.

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 are actually some of the most frequently used ones in mainstream component-based systems, such as Enterprise Java Beans, but working here at an 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 coordination-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 semantics 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 — beyond the pure syntactic aspects — simplifies their integration and dynamic gluing by the kernel: for instance, this is achieved by applying some kind of coordination rules to enable interoperability among components in spite of syntax and semantics mismatches among the interactions signals generated / perceived.

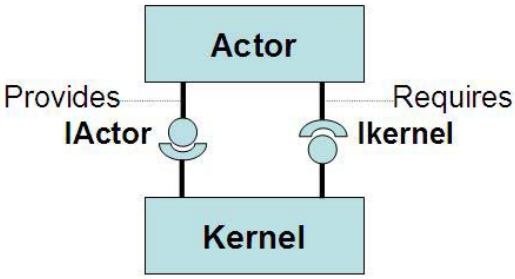

Fig. 3. Architectural view of elements in the framework

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.

The relationship between kernel and actor is realised exploiting the inversion of control (IoC) pattern[2]: some configuration code is in charge of injecting a reference to the kernel inside the actor, so that the latter can directly access its environment without being responsible for retrieving it. 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

[2] The IoC pattern is now becoming a standard approach for developing containers in component-based systems (`http://www.devx.com/Java/Article/27583`)

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 through the IoC pattern 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).

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`
Configuration time	Injecting 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 — is used to execute a service realised by the actor. In particular, this is invoked by 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 [] getOutputSignalsNotify();
  public String [] getOutputSignalsInvoke()
  public String [] getOutputSignalsInform()
  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/Invoke/Inform` return the output signals that the actor can generate, namely the list of `actionName` for the services it can request — either through a `notify`, `invoke`, or `inform`. The methods `setKernel` and `getKernel` store and retrieve the reference to the kernel where the actor is connected to — according to the IoC pattern. 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 declareOutputSignalsInvoke(IActor emitter, String[] signals);
  void declareOutputSignalsInform(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 coordination 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.invoke`, `IKernel.inform`, and `IKernel.notify`. 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 realize 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 coordination 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 coordination 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 runtime 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 runtime. The implementation of this class is based on the `tuProlog` open source project we developed [5] (`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 realize the methods provided by the `IKernel` interface. The methods supporting configuration simply cause a term — 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/Invoke/Inform` — These three methods are used to register the output `notify/invoke/inform` 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 facts `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 be-

tween actors are inserted in the knowledge base dynamically. A method `trace` in class `LogicKernel` writes in the knowledge base a term 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 `invoke`, `inform`, or `notify` is invoked on the kernel, a corresponding prolog predicate `invokeInTheSpace/4`, `informInTheSpace/3`, and `notifyInTheSpace/3` is called, which is in charge of allowing the proper actors to perceive the signal, supporting the precise semantics of each of the three primitive.

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

```
invokeInTheSpace( Emitter, ActionName, Arg, Res ):-
  reacts(Reactor,ActionName),
  node(Reactor,Class),
  declares(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 it declared the corresponding input signal, and finally *(iv)* invokes method `doAction` on it, 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),
  declaresNotify(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.

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 4 shows the `Inspector` tool of the framework, used to display all the relevant information about state and evolution of the logic kernel. In particular, this tool can inspect the current kernel configuration (button `selfDescribe`), the interactions occurred and reified as `out` tuples (button `showInteractions`), the logic theory governing interaction laws — namely, the coordination behaviour in the system

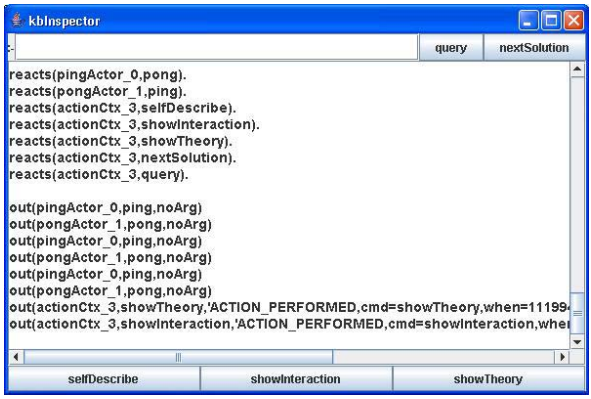

Fig. 4. Inspector tool

— (button `showTheory`). The inspector tool can be used to debug application and to 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 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.

5 A Simple Example: 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 coordinated by a simple rule. The source code of the Java classes implementing this example is reported in Fig. 5. 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 coordination 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

```
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",3);
    veto.setKernel(kernel); //injection of the kernel

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

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

Fig. 5. Code for the Ping-Pong Example

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.

6 Related Work and Conclusion

Several models and architectures for component specification and component composition can be found in literature, both in the context of coordination models and languages [12,8] and in the context of software architectures [6], including ADL (architectural description language) approaches such as [9,7,10]. Among the others, two recent and notable examples are respectively Reo [1] and Rainbow [3]. Differently from these approaches, 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 coordination as a main engineering dimension, and the introduction of first-class abstractions (media) for their specification and management (exogenous coordination).

A distinguishing point of the framework with respect to current approaches in literature is the investigation of first-order logic for specifying and representing the logic of interaction, including the interaction contracts of the components (actors), the interaction / coordination laws gluing the components and the interaction events actually happening at runtime. A somewhat similar approach working on mainstream technologies is given by Composition-Filters — exploited in the context of Aspect-Oriented Programming — 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 [2].

Several research lines will be explored in future works. Among the others: the enhancement of the basic set of interaction / coordination laws directly supported by the kernel; the exploitation of the logic-based kernel for the engineering of self-adapting and self-healing systems; the definition of a formal

model for the framework in order to specify and understand more rigorously the behaviour of component-based system built on top it.

References

[1] Arbab, F., *Abstract behavior types: a foundation model for components and their composition*, Science of Computer Programming **55** (2005), pp. 3–52.

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

[3] Cheng, S.-W., A.-C. Huang, D. Garlan, B. Schmerl and P. Steenkiste, *Rainbow: Architecture-based self adaptation with reusable infrastructure*, IEEE Computer **37** (2004), pp. 3–52.

[4] Denti, E., A. Natali and A. Omicini, *Programmable coordination media*, in: D. Garlan and D. Le Métayer, editors, *Coordination Languages and Models – Proceedings of the 2nd International Conference (COORDINATION'97)*, LNCS **1282** (1997), pp. 274–288.

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

[6] Garlan, D., *Software architecture: a roadmap*, in: A. Finkelstein, editor, *The Future of Software Engineering*, ACM Press, 2000 .

[7] Garlan, D., R. T. Monroe and D. Wile, *Acme: an architecture description interchange language.*, in: *CASCON*, 1997, p. 7.

[8] Gelernter, D. and N. Carriero, *Coordination languages and their significance*, Communication of the ACM **35** (1992), pp. 96–107.

[9] Luckham, D. C. and J. Vera, *An event-based architecture definition language*, IEEE Transactions on Software Engineering **21** (1995), pp. 717–734.

[10] Magee, J., N. Dulay, S. Eisenbach and J. Kramer, *Specifying distributed software architectures.*, in: *ESEC* (1995), pp. 137–153.

[11] *Osgi service platform*, `http://www.osgi.org` (1999).

[12] Papadopoulos, G. A. and F. Arbab, *Coordination models and languages*, Advances in Computers **46** (1998), pp. 329–400.

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