Wolf – an Eclipse Plug-in for WADE

G. Caire, M. Porta, E. Quarantotto, G. Sacchi
Telecom Italia
Via Reiss Romoli 274
10148 Torino - Italy

Preliminary Version

ABSTRACT

This paper describes WOLF, a development environment for WADE-based applications. WADE is a software platform, based on JADE, a popular Open Source framework, for the development of distributed applications based on the agent oriented paradigm and exploiting the workflow metaphor to define system logics.

The main advantage of the workflow metaphor is the expressiveness of the workflow itself, since it can be easily understood both by programmers and domain experts.

The main feature of WOLF is the support for the graphical definition of workflows. Besides that, it also helps developers in setting up an Eclipse project for developing and managing WADE-based applications. WOLF is an Eclipse Plug-in and, as a consequence, allows WADE developers to exploit the full power of the Eclipse IDE.

The paper focuses on the graphical definition of workflows and shows how WADE can take advantage of WOLF in defining the system logics by means of workflows.

Categories and Subject Descriptors

I.2.11 {**Artificial Intelligence**]: Distributed Artificial Intelligence - *Multiagent systems*; C.2.4 {**Computer Communication Systems**]: Distributed systems; D.2.11 [**Software Engineering**]: Software architecture

General Terms

Management, Performance, Languages.

Keywords

Software Agent, workflow, JADE, Open Source, XPDL, OSS, Telecommunication network, Scalability, Flexibility.

1. INTRODUCTION

Following the workflow metaphor, a process can be defined in terms of the activities to be executed, the relations between them, used to specify the execution flow, and conditions of starting and termination. Other additional information such as the software tools to be invoked in the activities and the required inputs and expected outputs can be found in a workflow process definition.

The main advantage of using the workflow metaphor is that a workflow can be represented in a graphical form, easily understandable by domain experts as well as by programmers. Domain experts can then validate the system logics and, in some cases, they could even contribute to the actual development of the system without the need of programming skills.

Another important characteristic of workflow representation is that the steps that compose the process are explicitly identified. That facilitates the implementation and the use of automatic mechanisms to trace the execution of a workflow for system monitoring and problem investigation purposes. Additionally, automatic or semi-automatic rollback procedures can be activated in case of unexpected fault, when processes have to be executed within the scope of a transaction. .

Finally, because of their expressiveness workflows can be considered self-documented and the development team is no longer required to keep documentation aligned each time design choices must be revisited.

At the moment the workflow representation is mainly used in BPM environments to represent business processes. WADE (Workflow and Agent Development Environment) is a domain independent platform, built on top of JADE [1], a popular open source middleware for the development of distributed applications based on the agent-oriented paradigm. In [8] a detailed description of WADE is provided toghether with the presentation of two mission critical applications, developed by Telecom Italia in WADE.

WADE tries to bring the workflow approach from the business process level to the level of system internal logics. Even if it could be used for that purpose too, WADE does not target high level orchestration of services provided by different systems or the representation of business processes, but the implementation of the internal behaviour of each single system.

Several attempts to support graphical development of system internal logics have been tried and several descriptive formalism, such as XPDL, BPEL, WS-BPEL [4], [5], have been defined. However, if on the one hand they provide a clear and intuitive representation of the process execution flow, on the other hand they are not suitable to specify all the details involved in the implementation of a piece of the business logic of a given software system. A usual programming language such as Java is definitely more powerful and flexible to deal with data management and transformations, computations and other low level auxiliary operations that can be needed when specifying the business logic of the system under development.

Taking into account the above considerations, the approach followed by WADE is to provide a workflow view on top of a normal Java class. That is a workflow is implemented as a Java class with a well defined structure (detailed in section 2.2). Moreover instead of providing a single powerful workflow engine, WADE gives each JADE agent the possibility of executing tasks defined according to the workflow metaphor enabling delegation of sub-tasks to other agents. In this way a set of agents can cooperatively carry out a complex task.

A key element in this approach is WOLF (WOrkflow LiFe cycle management environment), the graphical development

environment for WADE based applications. Using Wolf, developers are free to adopt the workflow based development model, when they think it is suitable and they can switch between the workflow view and the code view, according to what they consider more appropriate.

As its name suggests, WOLF is not only a tool to graphically create workflows for the WADE platform, but a complete environment to manage the whole life cycle of workflows: from the development to the testing up to the deployment. WOLF is an Eclipse (www.eclipse.org) plug-in and as a consequence allows WADE developers to exploit the full power of the Eclipse IDE plus additional WADE-specific features.

The paper is structured as follows: in chapter 2 an overview of the WADE Platform is given and the internal micro workflow engine is presented. Chapter 3 describes the main functionalities provided by WOLF, both to support the graphical definition of workflows and to help developers in setting up and managing WADE-based applications. In chapter 4 a description of the WOLF architecture is provided and finally in chapter 5, we try to draw some conclusions.

2. WADE OVERVIEW

This chapter describes the main components of a WADE Platform: an overview of its architecture is given and, moreover, a description of the micro workflow engine is provided.

2.1 The WADE Platform

WADE (Workflow and Agent Development Environment) is a domain independent platform, built on top of JADE [1], an open source middleware for the development of distributed applications based on the agent-oriented paradigm. The distribution of JADE includes a runtime environment, a library of classes that programmers can use to develop their application and some graphical tools for administration and monitoring purposes.

Each running instance of the JADE runtime environment is called **Container** and a set of containers is called **Platform**. In a JADE Platform a single special **Main Container** must always be active and the other containers register with it at startup.

One or more application agents can be started into a Container. The actual job of an **Agent** is to perform some tasks assigned to it. In JADE, a "**Behavior**" represents a task to be performed by an Agent and it is implemented as an object of a class that extends the class `Behaviour` of the JADE. An Agent to perform its tasks may need to communicate with other Agents in the Platform. JADE provides the agents with the ability to communicate. The communication model adopted is the "Asynchronous Message Passing" and the format of the messages is the ACL (Agent Communication Language) defined by FIPA [2].

WADE adds to JADE the support to the workflow execution and a few mechanisms to manage the complexity of the distribution, in terms of administration and fault tolerance. In order to do that, the following specific components have been added:

- **BootDaemon processes**: there is a bootDaemon process for each host in the platform and it is in charge of the Containers activation in its local host.
- **Configuration Agent (CFA):** the configuration agent always runs in the Main Container and is responsible for interacting with the boot daemons and controlling the application life cycle.
- **Controller Agents (CA):** there is a controller agent for each container in the platform and they are responsible for supervising activities in the local container and for all the fault tolerance mechanisms provided by WADE.
- **Workflow Engine Agents (WEA):** Workflow Engine Agents embeds an instance of the micro workflow engine, described in section 2.2, and therefore they are able to execute workflows.

Generally, both Workflow Engine Agents and JADE agents, running usual behaviours, can be found in a WADE application. How many performer agents to use and in which containers to deploy them depend on the application requirements. It is even possible not to use performer agents at all. In these cases, the application takes advantage just of the administration and fault tolerance features of WADE.

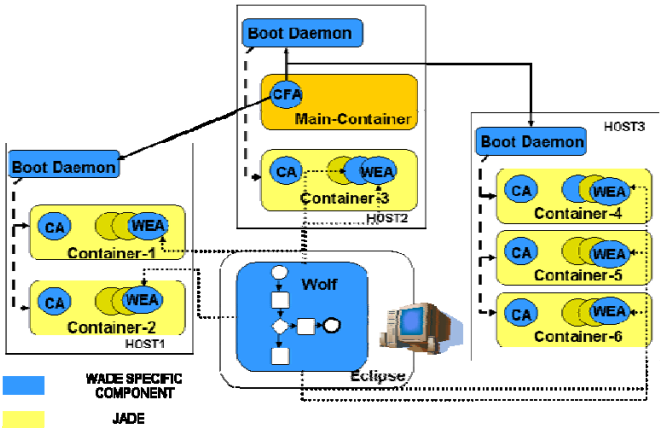

Figure 1 – a WADE platform

Figure 1 shows the topology of a WADE-based application. WADE specific components are highlighted in blue. As mentioned, in a WADE-based application WOLF supports the graphical definition of workflows, the setup of the Eclipse project to develop the application and the management of the application running remotely or locally.

As it will be explained in next chapters, WOLF tries to keep the advantages of workflows representation, without renouncing the power and the flexibility of usual programming languages. As a consequence it is a key element in the challenge to bring the workflow approach from the business process level to the system logics.

2.2 The Micro Workflow Engine

In this section, the micro Workflow Engine embedded in Workflow Engine Agents is described. This engine gives Workflow Engine Agents the ability to execute workflows and is therefore one of the core components in the whole WADE platform.

2.2.1 The meta-model for process representation

As mentioned the approach followed by WADE is to provide a workflow view on top of a normal Java class. As a consequence, a workflow is implemented as a Java class and no standard

workflow definition language is used. However, Wolf adopts the workflow meta-model defined in the XPDL [2], standard specified by the Workflow Management Consortium. The XPDL meta-model has been chosen, because the XPDL language has been conceived as interchange formalism between different systems. WADE supports the import of XPDL files and the adoption of this meta-model facilitates these operations. Moreover, the XPDL meta-model is based on a Finite State Machine computational model that is the same model supported by the WADE agents.

In the XPDL meta-model a process is represented as a workflow, consisting of one or more **activities** that can be thought as tasks to be executed. In a workflow, the execution entry point is defined, specifying the first activity to be performed; this activity is called **Start Activity.** On the other hand, a workflow must have one or more termination points, named **Final Activities.**

The execution flow is defined by means of **transitions**. A transition is an oriented connection between two activities and may have a condition associated. Regular or exception transitions can be defined. Exception Transitions allow specifying branches that are taken only when an Exception is raised in the source activity. Excluding the final ones, each activity may have one or more outgoing transitions. When the execution of an activity is terminated, the conditions associated to its outgoing transitions are evaluated. As soon as a condition is verified the corresponding transition is activated and the execution flow proceeds towards the destination activity.
Normally a process execution uses some internal data, for instance, to pass intermediate results between activities and/or for evaluation of conditional expressions. In the XPDL meta-model internal data are modeled by **Data Fields**.

A process can have one or more inputs to be provided and one or more outputs expected at the end of its execution. Inputs and outputs of a process can be formalized in the XPDL meta-model by means of the workflow **Formal Parameters**.

The XPDL meta-model defines some predefined types of activity. The most important ones are:

- **Tool Activity:** a tool Activity is an activity that is implemented by means of the invocation of one or more software tools, named Applications.
- **Subflow Activity:** a sublflow Activity is an activity whose execution corresponds to the invocation of another workflow. Formal/actual parameters are used the exchange information between the calling and the called process. A distinguishing characteristic of the WADE workflow engine is the **Delegation mechanism** that allows a set of agents to cooperatively execute a complex process. More in details the agent executing the calling workflow can decide to delegate the subflow to another agent on the basis of conditions evaluated at runtime. Such conditions may be related for instance to the current load (thus supporting the implementation of a GRID-like system) or to specific abilities required to carry out a portion of the whole process. The delegation mechanism is implemented by means of a an extension of the fipa-contract-net protocol ([3]) where the agent executing the calling workflow acts as initiator while the agent executing the subflow acts as responder. This protocol allows managing, when required, a process carried out by a set of cooperating agents as a single transaction.
- **Route Activity:** a route Activity is an activity which performs no work processing, but simply supports routing decisions among its incoming and out coming transitions.

Finally, WADE introduces a new type of Activity, named **CodeActivity**. In a code activity the operation are specified directly by a piece of Java code embedded in the workflow process definition.

2.2.2 Workflow representation in WADE (Java centric approach)

This section describes the Java Code that actually defines a workflow. As mentioned a workflow is implemented by a Java class that extends directly or indirectly the WADE class `WorkflowBehaviour`. The `WorkflowBehaviour` class provides a set of APIs, consistent with the XPDL meta-model described in the previous section, which can be used by developers to implement their own workflows. The mapping between the meta-model objects and the workflow implementation is shown in the example depicted in Figure 2.

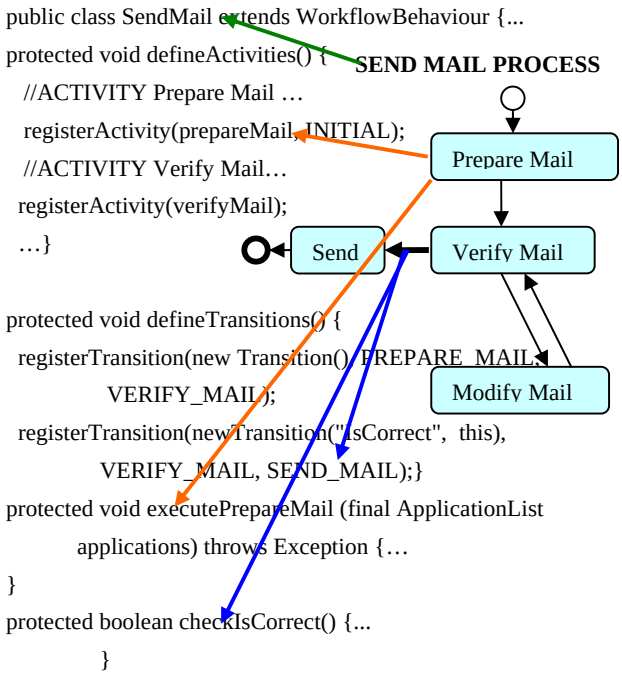

Figure 2 – Mapping between meta – model objects and workflow implementation

The methods `registerActivity()` and `registerTransition()` must be used respectively to add an activity and a transition. When the `registerActivity()` method is called, its first parameter is an instance of the activity to be added. The following types of activities, corresponding to the activities described in the meta-model, can be created:

- `ToolExecutionBehaviour`
- `SubflowDelegationBehaviour`
- `RouteActivityBehaviour`
- `CodeExecutionBehaviour`.

The actual tasks to be performed by an activity (no matter of its type) are specified in a `void` method of the workflow class; this method must have the same name of the activity, preceeded by the prefix "execute" (`execute<ActivityName>`). The workflow engine is in charge of invoking that method when the activity is visited.
In the same way, the `registerTransition()` method takes a transition instance as parameter. For transitions with conditions, the `boolean` expression to be evaluated by the workflow engine is specified in a boolean method that has same name of the condition, preceeded by the prefix "check" (`check<ConditionName>`).

As mentioned, many types of activity are available and the class `WorkflowBehaviour` provides a set of APIs to manage this activities. For example, to add an Application to a Tool Activity the method "`addApplication()`" must be used in the workflow definition. For more details about workflow coding, see the WADE tutorial [7].

An important feature of the object-oriented programming languages is the possibility to reuse the code by means of **inheritance mechanisms**. In order to bring the programming languages power also to the workflow representation, it has been chosen to provide the workflow with the inheritance mechanism too. Therefore it is possible to define a new workflow extending an old one, and then adding/removing activities and transitions. The overriding of methods associated to the activities and to the conditions of transitions is also permitted.

3. AN ENVIROMENT TO DEVELOP WADE BASED APPLICATIONS

WOLF is the graphical development environment for WADE-based applications and is implemented as an Eclipse plugin. It allows the management of the whole life cycle of workflows, using a single development environment and provides support for:

1. graphically editing of workflows;
2. controlling a local or remote WADE-based application;
3. deploying and executing workflows in the controlled application.

This chapter describes how to manage WADE-based projects and how to edit workflows with the WOLF Editor.

3.1 Managing a WADE-based project

The first step to use WOLF in an Eclipse Java project is to mark it with the *WADE Nature*. The "nature" is an Eclipse concept used to characterize a project and to give it a set of additional functionalities. Once the *WADE Nature* has been added, all the features of the WOLF plugin become available to the project. These features are described hereafter.

XPDL workflow import

WOLF provides support to import XPDL files into the format adopted by WADE and described at paragraph 2.2.2.

Workflow editing, deployment and execution

Wolf provides support for:

- Workflow editing, described in paragraph 3.2;
- Workflow deploying on a running platform.
- Workflow execution on a given workflow engine agent;

Obviously a WADE platform must be active and running, to deploy and execute a workflow on it.

When activating the execution of a workflow the user is prompted with a dialog box by means of which he can specify among others:

- the workflow engine agent that will execute the workflow;
- the input parameters of the workflow;

Once the execution started he can monitor it by means of the *ExecuteWorkflow Editor*, which shows status information, output parameters, possible failure reasons and workflow tracing information.

Controlling/activating a WADE based application

In order to test newly created workflows (see paragraph 4.2, for details about the implementation) WOLF allows:

- starting an internal WADE platform: in this case the BootDaemon and MainContainer run inside Eclipse and both of them can be launched in normal or debug mode;
- starting an internal BootDaemon process belonging to an external platform (running outside Eclipse);

WOLF also provides support for connecting to an internal or external platform and to manage the application running on it. This can be done by means of the *Agent Tree View*. The *Agent Tree View* is a view which enriches the functionalities of the JADE Remote Management Agent (RMA) with WADE specific aspects.

In particular, two buttons are provided in the *Agent Tree View* toolbar to connect/disconnect to/from a running WADE platform. In case the Main Container process is launched internally, the *Agent Tree View* automatically connects to it.

Once connected, the buttons to control the application life cycle are enabled. In particular, by means of the "Import configuration" button, the user can import an application configuration. At this point the application can be started by means of the "startup" button. All containers and agents specified in the imported configuration will be automatically started and will appear in the Agent Tree view.

While the application is active, the user can save the containers and agents currently living in the platform ("Save configuration" button), causing the updating of the *Target Configuration* (the default configuration that is started if no configuration is imported). At any time, moreover, the user can export the Target Configuration in an xml file ("Export configuration" button).

By double-clicking on an agent in the Agent Tree view the user can inspect and, when relevant, modify its exported attributes by means of the *AgentDetails Editor.*

3.2 The Workflow Graphical Editor

The *Workflow Graphical Editor* (henceforth, simply Workflow Editor) offers a graphical view of a java class that represents a workflow. It can be opened by right-clicking on the Java class and selecting in the popup menu the item "Open with Workflow Editor".

WOLF lets developers switch between the graphical view provided by the Workflow Editor and the code view provided by Eclipse Java Editor. So, the *Workflow Editor* is strictly bound to the Java Editor in order to guarantee consistency between the two

views. Synchronization between the graphical and code view is maintained as follows:

- modifications made through the *Workflow Editor* are immediately visible in the code view;
- modifications made through the *Java Editor* become visible in the graphical view when clicking on the Refresh button in main toolbar of Eclipse.

When opened, the *Workflow Editor* provides developers with proper tools to modify the graphical view and switch from the graphical view to the code view. Figure 3 shows a snapshot of the Workflow Editor.

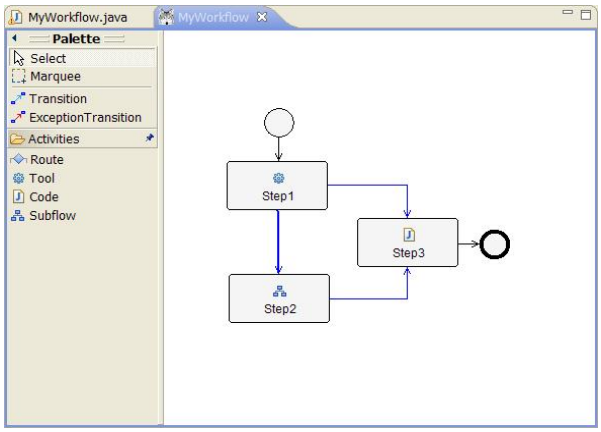

Figure 3 – A snapshot of the Workflow Editor

The developer uses a tool palette to insert activities and/or transitions in the workflow. The types of activity that can be created are those supported by WADE: Tool, Subflow, Route and Code. Regular or exception transitions can be created and in both cases the user can assign a condition to them.

Besides, acting directly on the graphical view, the user can move and resize each activity, and also modify the layout of the transitions by the insertion/deletion of routing points. These modifications are immediately reflected in the related java class.

The following functionalities are available for any activity and transition, by means of contextual menus:

- **Edit**: command to modify the properties of the selected element.
- **Go to Code**: command to switch to the Java Editor showing the method related to the activity or the transition condition;
- **Delete**: command to remove an element. If the element is an activity, all incoming and outcoming transition are automatically removed ;

Additionally, for Subflow Activity the OpenSubflow functionality is provided. Such functionality opens a new Workflow Editor for the workflow referenced by the activity.

As mentioned in paragraph 2.2.2, the Workflow Editor supports the workflow inheritance. The inheritance mechanism allows the extension of workflow classes only including the differences with the base class. These differences are

- Addition of new activities and/or transitions (`registerActivity, registerTransition`);
- Modification of activity order (`changeActivityOrder`)
- Removal of activity and/or transitions (`deregisterActivity`, `deregisterTransition`);
- Addition of formal parameters;
- Redefinition of operations performed in an activity by overriding the related activity method;
- Redefinition of a condition by overriding the related condition method;
- Overriding of types of activities and/or transitions, e.g. an activity of type Code that becomes of type Tool or a default transition that become a conditioned transition (`registerActivity, registerTransition`);
- Modifications of layout information.

Finally WOLF provides a Problems View window strictly related to the Workflow Editor where semantic problems encountered in the workflow definition are shown. For instance a non-initial activity with no incoming transitions will never be visited during the execution of the workflow and is therefore indicated in the Problems View as a warning.

4. WOLF ARCHITECTURE

As mentioned, WOLF has been implemented as a Plugin in the Eclipse Platform and it requires the availability of JDT (Java Development Tooling - included in the Eclipse Java IDE) and GEF Plugins (Graphical Editing Framework). This chapter describes the Plugin architectural elements.

4.1 Administration console architecture

The *AgentTree View*, the *ExecutionWorkflow Editor* and the *AgentDetails Editor*, previously described, are the main components of the administration console. Such components enable the communication with a local or remote WADE platform. As shown in Figure 3, this communication is implemented using the `JadeGateway class included in the JADE library` that provides a simple yet powerful way to make non-JADE applications (such as WOLF) interact with a JADE based multi agent system.

The `JadeGateway` maintains an internal JADE agent (`AgentTreeViewGatewayAgent`) that acts as entry point in the JADE based system. The activation and termination of this agent (and its underlying container) are completely managed by the `JadeGateway` class and developers do not need to care about them.

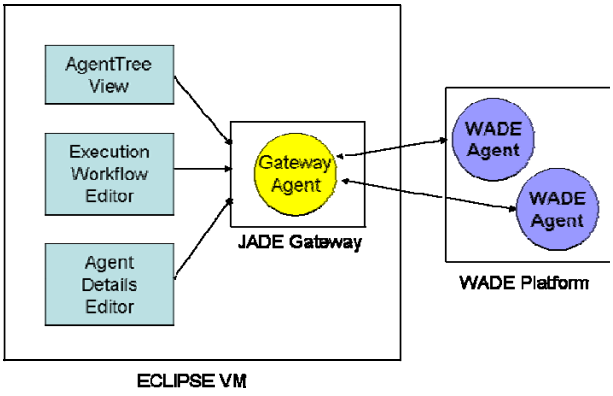

Figure 3 – Communication between the WOLF Plugin and a WADE Platform

Using the `JadeGateway`, the administration console asks the WADE Platform to perform operations such as the start-up/shut-down of the platform, the workflows execution or termination and so on.

The `AgentTreeViewGatewayAgent` is subscribed to JADE AMS to receive platform management events, such as creation/termination of agents and containers. In this way the *AgentTree View* is always updated. Moreover the `AgentTreeViewGatewayAgent` is subscribed to WADE platform events, in particular, to state change events (states can be down, starting, started …). Therefore the operations available in the *AgentTree View* can be enabled/disabled, according to the current platform status.

4.2 Workflows debug

The normal WADE BootDaemon forks a new JVM for each JADE container it is requested to activate. In order to support workflow debugging WOLF uses a modified version of the BootDaemon that activates containers in its own JVM that is the same JVM of Eclipse. As a consequence, in order to debug workflows it is sufficient to start the WOLF BootDaemon in debug mode and to submit the workflow to be debugged to an agent running in a container activated by the WOLF Boot Daemon.

4.3 Workflow Editor architecture

The *Workflow Editor* is based on the GEF plugin. GEF is a very powerful framework for visually creating and editing models. A model is the starting point for any GEF application; it represents the data to be displayed and edited by the graphical editor. GEF component is separated into two plugins:

- **Draw2d (org.eclipse.draw2d) -** the lightweight toolkit for painting and layout on an SWT Canvas;
- **GEF (org.eclipse.gef) - an interactive MVC framework built on top of Draw2d.**

At the heart of GEF there is the **Model-View-Controller** pattern.

Model

In our case, the model reflects the workflow meta-model previously described. Its main classes are `WorkflowProcess` (the activities and transitions container), `Activity` (the elementary work unit in a workflow) and `Transition` (the connection between activities). Instances of these classes form an in-memory representation of the workflow to be edited and are created parsing the workflow java file. A description of the parsing procedure is provided afterwards.

View

The Workflow model is displayed using figures from the Draw2D plugin.

Controller

The *controller* bridges the view and model. Each controller, or *EditPart* using the GEF terminology, is responsible for mapping the model to its view. Editparts contain also the *EditPolicies*, which generate the *Commands* for making changes to the model. Finally they also observe the model and update the view to reflect changes of the model state.

The Workflow Editor includes the following *EditPart* implementations: `WorkflowProcessEditPart`, `ActivityEditPart` and `TransitionEditPart`.

All parsing operations on the workflow java file are performed using the APIs provided by the JDT plugin. JDT is a full featured Java integrated development environment. For manipulating Java files JDT offers the *abstract syntax tree (AST)* APIs. The *Abstract Syntax Tree* maps plain Java source code in a tree form. This tree is more convenient and reliable to analyse and modify programmatically than text-based source.

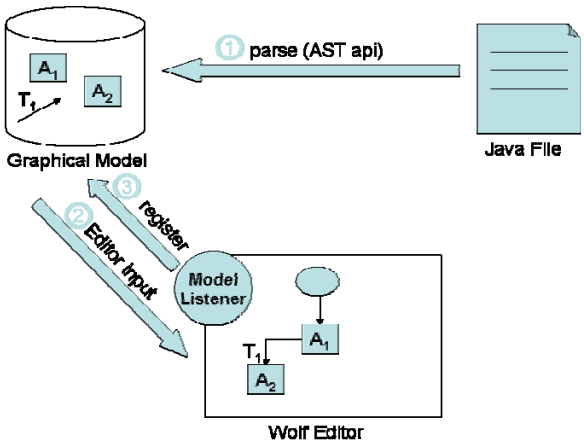

Figure 4 – Workflow Editor Initialization

After the model generation, the Workflow Editor uses it to create the EditParts associated to all activities and transitions included in the workflow process.

During the Workflow Editor initialization, it is also instantiates an object named `ModelListener` that is an observer of the graphical model. The `ModelListener` is notified of changes made by the user in the graphical editor and it is responsible for updating the java code.

When the user interacts with the editor, the *EditPolicies* generate a command that manipulates the model. Model changes are notified to both the *EditParts* and the `ModelListener`. The *EditParts* and the `ModelListener` refresh the view and modifies the java code respectively.

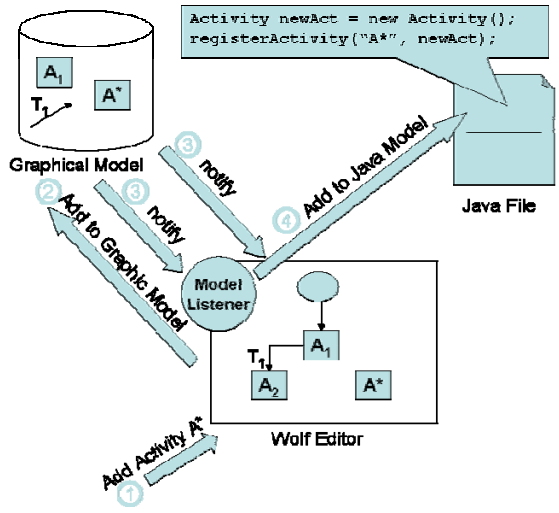

Figure 5 – Workflow editing

For example, as shown in Figure 5 , when the user adds an activity A* the `ActivityCreateCommand` is invoked. This command modifies the workflow process model object adding a new activity. A notification is sent to the `WorkflowProcessEditPart` and to the `ModelListener`. The former reacts by showing the activity. The latter reacts by modifying the AST tree in order to add new nodes that represent the behaviour instantiation and the `registerActivity()` method invocation.

When a workflow is displayed by the Workflow Editor also the Java editor is opened on the same workflow. To allow synchronization between the two editors the working copy mechanism is used. A working copy is an in-memory representation of the java file. Using a working copy permits to share it with multiple clients. In our case, it is used by the java editor and at the same time it is modified programmatically by the `ModelListener` so that changes appear immediately in the Java editor.

When the user modifies directly the workflow java class, the Workflow Editor is not automatically updated. Synchronization must be explicitly triggered by the user.

5. CONCLUSIONS

In this paper we described WOLF, the development environment for WADE-base applications. It has been implemented as a Plugin integrated in the Eclipse Platform. This choice allowed developing a complete environment to manage the whole life cycle of workflows. The integration with the Eclipse Platform allows also the exploitation of all features offered by the Eclipse Java IDE.

In WADE a workflow is represented as a java class and WOLF provides a graphical view of it, making available to the developers both the expressiveness of a visual representation and the power of usual programming languages. In this way WOLF represents a key element in the challenge to bring the workflow approach from the business process level to the level of system logics. Besides, since a workflow representation is extremely intuitive, it can be used as software documentation for domain experts' validation.

Moreover developers can completely control the Java class associated to a workflow, because no hidden code is added by Wolf.

Finally, because of their representation as java classes, workflows can be extended using the Object-oriented inheritance mechanism thus bringing strong advantages in terms of code re-use, maintainability and complexity reduction.

6. REFERENCES

[1] JADE - Java Agent Development framework. http://jade.tilab.com

[2] FIPA – Foundation for Intelligent Physical Agents http://www.fipa.org

[3] FIPA – The FIPA Contract Net interaction protocol

[4] XPDL XML Process Definition Language, http://www.wfmc.org/standards/xpdl.htm

[5] Shapiro, R. 2002. A comparison of XPDL, BPML and BPEL4WS (Rough Draft), Cape Vision

[6] BPMN Business Process Modeling Notation http://www.bpmn.org/

[7] G. Caire "WADE: An Open Source Platform for Workflows and Agents" http://jade.tilab.com/wade/doc/tutorial-aamas2008.zip

[8] Caire G., Gotta D., Banzi "WADE: A software platform to develop mission critical applications exploiting agents and workflows", M., Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008) – Industry and Applications Track, Berger, Burg, Nishiyama (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 29-36.