Design and Implementation of PROSYT: a Distributed Process Support System

Gianpaolo Cugola and Carlo Ghezzi
{cugola, ghezzi}@elet.polimi.it
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da Vinci 32
20133 Milano (Italy).

Abstract

The paper describes the design and implementation of a process support system (PROSYT), which is intended to provide guidance in performing business processes and cooperation among people over a local or geographically distributed network. In particular, it can be used as a Process-centered Software Engineering Environment (PSEE) to support distributed software development.

Our main purpose is to describe how complex applications of this kind can be developed systematically. In particular, how the requirements of high flexibility, reconfigurability, scalability, and efficiency demanded by these applications can be met through appropriate design choices.

1 Introduction

During the last decade, software development has quickly become one of the most complex engineering activities carried out by humans. Process-centered Software Engineering Environments (PSEEs) [2, 20] aim at supporting such activities by providing a language to formalize the process that have to be carried out (usually called "PDL", an acronym for "Process Description Language"), and an environment capable of interpreting a process model to guide people during the process and to automate the more repetitive activities. In this paper we describe the design and implementation of PROSYT, a PSEE especially conceived to support distributed processes.

Since software development processes do not differ radically from generic business processes, PROSYT has been conceived to support any kind of business process that requires people to cooperate according to a well known process by taking advantage of the services provided by a network of computers. As a consequence, we usually refer to PROSYT as a Process Support System (PSS) to underline its generality with respect to the kind of business process carried out.

A second consequence of this choice was the need of providing mechanism to increase PROSYT flexibility in executing the process model. Business processes (and particularly software processes) can be hardly modeled in advance with the required precision. It is often the case that some situation arise during the process that had not been anticipated into the model. As a consequence, a PSS has to be flexible enough to support their users even in presence of such unexpected situations.

Last consequence of the previously mentioned choice was the need of supporting distributed processes that involve several kind of people ranging from software engineers that spend most of their time sat in front of the same computer to *nomadic users* who connect to the network from arbitrary locations using notebook or PDAs and who are not permanently connected. In this paper we focus our attention to this last aspect and in particular to the consequences of this choice on the design and architecture of PROSYT[1].

Traditionally, distributed applications like PSSs are based on the client-server approach. Clients use some form of remote procedure call (RPC) to request a service to a server which is known to provide that service. Examples of middleware that adopt this approach are CORBA [3] and RMI [18]. The resulting software architecture is characterized by a tight coupling between the object that requests a service (i.e., the client) and the object that satisfies such request (i.e., the server). This approach reduces the possibility of reconfiguring the architecture of the application at run-time, and results in a limited scalability.

In developing PROSYT we adopted a completely different architecture based on two emerging technologies that

[1]The readers interested in the PROSYT aspects related to its ability of supporting people in presence of unexpected situations may consult[12] and [13].

offer the chance to overcome the limitation of client-server: mobile code and events.

In the last years a number of *mobile code languages* and libraries are becoming available to support the development of a new class of distributed applications composed of several components (often called *agents*) that are capable of moving from host to host in a local or wide area network. These applications are commonly called *mobile code applications* (MCAs) [27]. They provide an excellent support to nomadic users: as the users move and reconnect to the system, appropriate pieces of code can be moved to follow them. Moreover, they may be useful to reduce network traffic by moving communicating agents close to one another [10, 4].

As for architectural styles, an emerging style that is receiving increasing attention is based on the notion of *events*. The components of an *event-based architecture* cooperate by sending and receiving events, a particular form of messages. The sender delivers an event to an *event dispatcher*, which is in charge of distributing the event to all the components that have declared their interest in receiving it. Thus, the event dispatcher allows the sources and the recipients of an event to be fully decoupled.

PROSYT takes benefit of a combination of these two technologies to support highly dynamic, distributed processes. In particular, it is based on JEDI, an infrastructure, which integrates an event-based layer with support to code mobility. In this paper we describe our experience in developing PROSYT and draw an initial assessment of the benefits and drawbacks of the combined use of mobile code and event-based coordination frameworks to develop distributed PSSs.

The paper is organized as follow: Section 2 describes the requirements of PROSYT. Section 3 gives a brief description of JEDI. Section 4 describes how JEDI was used to implement PROSYT and shows the benefits and drawbacks of mobile code and event-based systems in implementing a PSS. Finally, Section 5 describes related work and Section 6 draws some conclusions and shows some directions for future research.

2 An overview of PROSYT

To satisfy the requirements of a modern PSS, PROSYT adopts innovative approaches in the areas of process modeling, process enactment, and system architecture.

- As for process modeling, the PROSYT PDL (called *PLAN*: the Prosyt LANguage) adopts an *artifact-based* approach. Each artifact produced during the process is an instance of some *artifact type*, which describes its internal structure and behavior. Each artifact type is characterized by a set of *attributes* whose values define the internal state of its instances, a set of *exported operations* that may be invoked by the users upon the artifact type's instances, and a set of *automatic operations* that are automatically executed when certain events happen (like invoking an exported operation on another artifact) and are used to automate the process and to react to changes in the state of the tools controlled by the environment.

  Boolean expressions are used to express the constraints under which exported operations are allowed to start. Constraints are organized in different classes, depending on the type of condition they express. It is also possible to specify a set of artifact *invariants*, to characterize acceptable process states.

  To describe activities and invariants that refer to a collection of artifacts, PLAN provides the concepts of *repository* and *folder*. Each repository is an instance of some *repository type* and contains a set of *folders* organized in a tree structure. Each folder (instance of some *folder type*) is a container of artifacts and other folders. Attributes, states, exported operations, automatic operations, and invariants may be associated either with repository types or with folder types. Exported operations and invariants for folders and repositories may be used to describe business activities and constraints that refer to structured collections of artifacts.

  Finally, PLAN provides the concept of *project type*. Each PLAN business process is described as an instance of some project type. It is characterized by a statically defined set of repositories, by a set of *groups* (each user belongs to one or more groups), and by a set of exported operations, automatic operations, and invariants, which refer to the entire process.

- As for process enactment, to improve flexibility PROSYT users are not forced to satisfy the constraints stated in the process model. They can invoke operations even if the associated constraints are not satisfied. PROSYT keeps track of the results of these deviations and controls that the invariants are not violated as a result of such deviations.

  PROSYT allows process managers to specify a *deviation handling* and a *consistency checking policy*. Such policies state the level of enforcement adopted (i.e., the classes of constraints that can be violated during enactment) and the actions that have to be performed when invariants are violated as a result of a deviation, respectively. Both these policies may change at enactment-time, and may vary from user to user. As an example, some deviations may be allowed during some phases of the process while they may be disallowed during other, more critical, phases. Similarly, an expert user may be allowed to perform deviations that are forbidden to beginners.

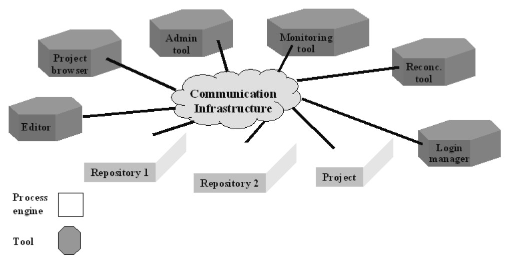

Figure 1. The logical architecture of PROSYT.

- As for system architecture, PROSYT adopts an event-based communication paradigm and takes benefit of code mobility to reduce network traffic and to support nomadic users.

2.1 The logical architecture of PROSYT

A typical PSS consists of a process engine, which interprets the process model and controls the execution of the tools used during the process, and a graphical interface used by the users to interact with the environment. Components interact in a client-server architecture where the process engine act as the server and the graphical front-end tools are the clients.

The structure of PROSYT is considerably more complex, since concurrency and distribution was exploited to achieve improved performance and flexibility. In particular, it exists a process engine for each repository, which is in charge of managing the entities (i.e., folders and artifacts) included in the repository and another engine that takes care of managing the instance of the project type that represent the currently executing business process. Using the PROSYT GUI, users can browse through the repositories and can access their contents by invoking the exported operations provided by the different entities. The process engine can also control the execution of the tools used to perform process specific tasks (like editors, office automation tools, and others), launching them and interacting with them.

In more details, the components that constitute the PROSYT environment are:

- A process engine for each repository and a process engine enacting the project type instance that represents the current business process.

- A *project browser* for each user, used to browse the entities (i.e., artifacts, folders, and repositories) that compose the enacting process, invoking the operations they exports.

- A *login manager* in charge of controlling users' login and logout. It manages all the information about the human agents that are involved in the process.

- The *administrative tool* used by the process manager to change the execution policies, and to add or remove users.

- The tools invoked by the system to perform process specific tasks like editors and compilers.

- A component in charge of monitoring execution by identifying the invocation of deviating actions (i.e., the *monitoring tool*) together with a tool that analyzes the result of this monitoring to support the reconciling activity (i.e., the *reconciling tool*).

Figure 1 shows the component involved in enacting a process model that includes two repositories. The figure describes a situation in which a single user is interacting with the PSS by using the project browser and an editor, while the process manager is using the reconciling tool to monitor occurred deviations, if any.

All the entities that compose the PROSYT environment must be able to communicate with one another. Moreover, the number of these entities and their location can change at enactment time. As an example, new artifacts may be created, existing artifacts may be moved from a repository to another (i.e., from a host to another), and new instances of tools may be launched. These characteristics, together with the fact that PLAN is intrinsically based on a notion of "event", motivate the choice of an event-based coordination paradigm among the entities that belong to the PROSYT environment. To implement mobility of artifacts and folders among different repositories mobile code technology was adopted.

3 The PROSYT run-time support

A number of frameworks have been (and are being) developed to provide an event-based middleware supporting distributed applications. When the PROSYT project started, however, these frameworks were not available. Thus we decided to develop our own framework, both to simplify the implementation of PROSYT and to start a research activity in the area of event-based architectures. The result of this research effort is JEDI, an event-based framework implemented in Java. A first version of the framework has been described in [14]. It differs from the version used to implement PROSYT, because it does not support mobility.

3.1 The JEDI event-based infrastructure

Figure 2 shows the logical architecture of JEDI. The infrastructure is based on the notion of an *active object* (AO). An AO is an autonomous entity that performs an application-specific task. Each active object has its own

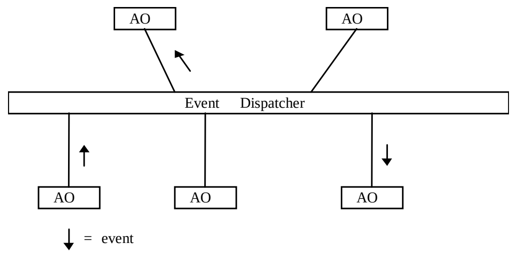

Figure 2. A logical view of the JEDI architecture.

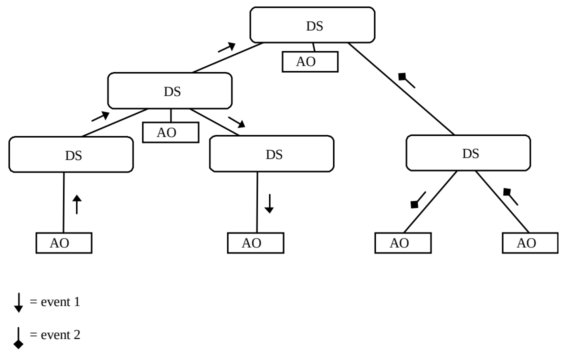

Figure 3. The JEDI distributed architecture.

thread of control and interacts with other AOs by expli itly producing and receiving events[2]. Events can be viewe as a particular type of message, which do not carry info mation about their recipients. An event is generated by an AO by explicitly invoking the `sendEvent` primitive. As a result of the event generation, a specific component of the infrastructure, called the *event dispatcher* (ED), notifies all the AOs that declared an interest in the event. An AO declares the classes of events it is interested in by invoking a `subscribe` operation. A subscription can be dropped by invoking the `unsubscribe` operation. Event subscription and unsubscription can be invoked at any time during the AO lifetime. The notification of events is accomplished asynchronously with respect to their generation.

One of the most important characteristics of an event-based infrastructure is the form of the events delivered by the event dispatcher. Such form has a strong impact on the expressiveness of the communication and the ability to support complex communication patterns. In JEDI, an event is an ordered set of strings $\langle s_1, s_2, ..., s_n\rangle$, where s_1 is the event name and $s_2, ..., s_n$ are the event parameters.

AOs can subscribe either to a specific event or to an *event pattern*. An event pattern is an ordered set of strings each one representing a simplified form of a regular expressions.

For space reasons, it is not possible to provide a detailed account of the features that distinguish JEDI from other event-based infrastructures. In the sequel we concentrate on the architecture of the ED, which was relevant for the implementation of PROSYT, while Section 3.2 focuses on the support provided by JEDI to move active objects from a host to another during execution.

We developed both a centralized and a distributed version of the ED. In the centralized approach, the ED is composed of a single process. This simplifies the implementation of the ED and the deployment of the application built on top of the infrastructure. At the same time, this centralized approach introduces a bottleneck that can be unacceptable as the number of AOs that compose the application and the number of events to be dispatched grows. To optimize the performance of the distribution mechanism, we developed a distributed version of the ED, which is structured as a collection of processes (usually, one for each machine running JEDI), interconnected in a tree structure. Each of these processes is called an *ED component*. Each AO connects to an ED component (not necessarily to a leaf of the tree). The resulting hierarchical architecture is shown in Figure 3. Events are propagated along the tree to the interested components on the basis of the subscriptions posted by each AO. Figure 3 shows the path followed by two different events going from the sender to the recipient. The figure also shows that events are always propagated to the top of the hierarchy. This behavior results from the strategy adopted in distributing subscriptions: an intermediate ED component does not have any knowledge of the subscriptions issued in sibling sub-trees.

The distributed ED becomes more efficient than the centralized ED whenever a property of *locality* holds, i.e., if AOs tend to communicate with other AOs located on the same or on ”nearby” hosts. In terms of cooperation patterns, this means that the global process can be viewed as a hierarchical federation of cooperative sub-processes. To preserve locality, an AO can be moved closer to its peer AOs. Notice, however, that the decision whether to use the centralized or the hierarchical version of the ED only affects the overall performance of the system, not functionality, since the adopted version of the ED is totally transparent to AOs.

Another critical issue in event-based frameworks is the behavior of the event dispatcher with respect to the ordering of events. The JEDI's ED guarantees that each registered AO receives the events sent by a given sender in the same order in which they were sent.

In summary, the event-based coordination style used in JEDI is characterized by the following properties:

- it is asynchronous;
- it is based on multicast;
- the source of a communication does not specify the destination of the communication;

[2] Although JEDI is implemented in Java, to achieve the goal of openness, AOs are not necessarily written in Java, since they interact with the ED through standard TCP/IP sockets. This allows off-the-shelf tools to be easily integrated in the environment.

- the destination of a communication does not know the identity of the source;
- the events delivered by a sender are guaranteed to be received by the recipients in the same sequence in which they are produced.

3.2 Mobile agents in JEDI

In an event-based environment, most AOs behave according to a standard sequence of operations. Upon activation, the AO subscribes to a set of events and then waits for their occurrence. When an event is notified, the AO performs some operation (possibly generating new events and subscribing or unsubscribing to events) and then goes back to a waiting state. For this reason, we predefined a particular type of active object, called *reactive object*, which exhibits this standard behavior. The programmer simply needs to provide an abstract method (called `processMessage`), which is automatically invoked each time the reactive object receives an event of interest.

In JEDI, reactive objects can hop from host to host keeping their internal state (i.e., they behave as mobile agents [27]). A reactive object can decide to move autonomously to a different host by invoking the `move` operation and specifying the target site. Such an operation causes the following actions to occur:

1. The state of the reactive object (i.e., the value of its attributes) is serialized and stored using standard Java facilities (i.e., object serialization);
2. The reactive object is temporarily disconnected from the ED and the thread of control executing the reactive object loop is stopped.
3. The state of the reactive object is moved to the new location through a network connection. At the destination site the reactive object is restarted and reconnected to the ED.

Migration does not cause any loss of event, since the ED keeps the events that should be received by the migrating reactive object until it reaches its new destination.

The JEDI ability of migrate reactive objects has been used in PROSYT to support nomadic users and to dynamically change the way distributed execution of a process model is carried out. In particular, it has been used to implement the PLAN `move` operation that allows artifacts and folders to be moved from a repository to another. Since artifacts and folders are active elements, by moving them PROSYT moves the site in which process enactment is carried out. As an example, as part of a software development process, it is possible to specify a `deliver` operation, which moves a software component from the development repository to the production repository.

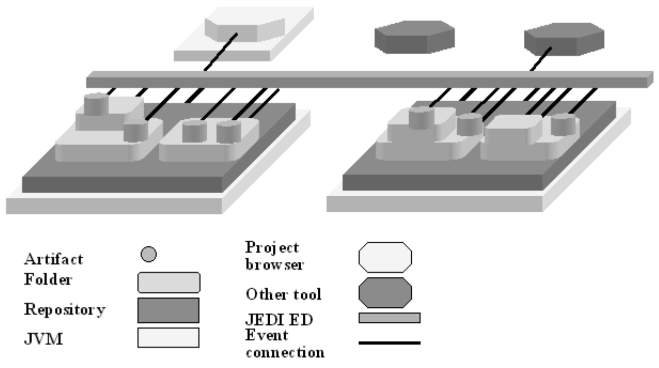

Figure 4. The physical architecture of the first PROSYT prototype.

4 The experience

This section describes how the JEDI framework was used to implement the PROSYT prototype. We argue that the lessons learned have more general validity. They illustrate the benefits of event-based coordination mechanisms and code mobility to support distributed cooperative environments.

4.1 The first PROSYT prototype

The components of a PLAN model (i.e., repositories, folders, and artifacts) are implemented as JEDI reactive objects, which communicate among themselves and with external tools via JEDI events. We also chose to run each repository, and the folders and artifacts it includes, within a different Java Virtual Machine (JVM). That is, each repository runs as a different process, while folders and artifacts are threads of the enclosing repository process. The resulting physical architecture of PROSYT is shown in Figure 4. It shows the use of JEDI as a middleware providing the coordination infrastructure; each component of the environment (tools, projects, repositories, folders, and artifacts) can send and receive JEDI events.

To enact a PLAN model, PROSYT adopts a translation based approach. Each element of a PLAN model (i.e., project types, repository types, folder types, an artifact types) is translated into a different Java class. At runtime, instances of such classes are created as needed. To simplify the translation step, the JEDI framework was enriched by introducing several classes which implement specific PROSYT functionalities, as shown in Figure 5. In particular, class `Item` includes the methods to decide if an exported operation invoked by a user has to be executed (as defined by the adopted deviation handling policy) and the methods to react to a violation of some invariant (as defined by the adopted consistency checking policy). Similarly,

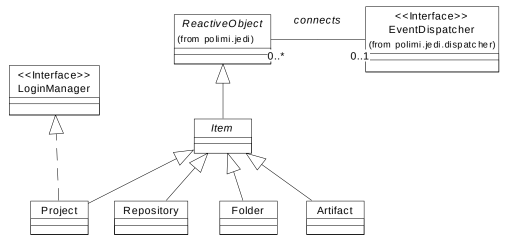

Figure 5. The design of the process engine of the first PROSYT prototype.

classes `Project`, `Repository`, `Folder`, and `Artifact` include methods that implement the operations that are common to all projects, repositories, folders, and artifacts, respectively.

The specific tools that compose the PROSYT environment (i.e., the browser, the administrative tool, the login manager, the monitoring tool, and the reconciling tool) are also developed in Java and run as different processes. They interact with the process engine through JEDI events.

The main advantages of the architecture of this first prototype can be summarized as follows:

- There is a strong decoupling between the different components of the environment. Components interact via events and can ignore the physical locations of other components.
- The event-based coordination makes it easy to modify the system architecture and add new tools. As an example, the monitoring component and the reconciling tool were added with a minimal effort after an initial prototype was finished.
- It is easy to integrate existing off-the-shelf tools into the environment because they do not need to know any specific feature of PROSYT. A wrapping layer must be developed to encapsulate such tools, to let them generate JEDI events when their state changes and to react to JEDI events sent by the running process engine to control their execution.

As we started using the first PROSYT prototype, we realized that the event dispatcher could become a bottleneck for the performance of the system. Although this problem was mitigated by adopting the distributed version of JEDI, it continued to be a problem in presence of large processes in which a large number of artifacts has to be managed. The second PROSYT prototype was developed to overcome this problem.

4.2 The second PROSYT prototype

By analyzing the traces of the events delivered by the components of the first PROSYT prototype, we discovered

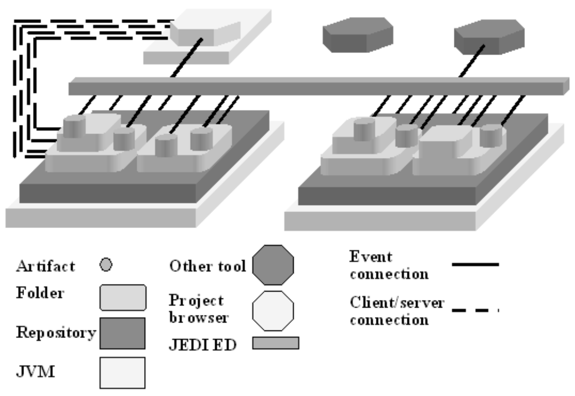

Figure 6. The physical architecture of the second PROSYT prototype.

that most of them were sent by instances of the browsers to the process engines and vice-versa. Each time a user browses through a folder `F`, an event is sent by each item (i.e., artifact and folder) that belongs to `F`. The event includes information about the state of the item and the exported operations that can be invoked upon it. If the user invokes the exported operation `Op` upon one of these items, another event is generated. This event is dispatched to the target item, to all the items that have an automatic operation associated with the invocation of `Op`, and to the monitoring tool. Moreover, each time the state of an item changes, a new event is generated and dispatched to the browsers that were displaying the item.

While the events generated by a browser instance may be relevant for different AOs, some of the events generated by process items (i.e., artifacts, folders, repositories, and projects) are relevant only for the browsers that are currently showing those items. In addition, artifacts and folders can move during their lifetime, while browsers run on the same host until they are closed. This suggests a strategy to reduce the number of events managed by the event dispatcher, still retaining the advantages of an event-based architecture, by adopting direct RMI connections to implement the communication from process engines to browsers. In particular, the browsers play the role of the servers exporting the functionality of showing process items like artifacts and folders. The resulting physical architecture is shown in Figure 6.

To implement this new architecture we added several Java classes. They implement the general communication protocol between a reactive object which is interested in showing its state on a *viewer tool* and the viewer tool itself.

5 Related Work

Process support systems, event-based coordination frameworks, and mobile agents are three very active research areas. Several prototypes and even products exists for each of them. Due to space reasons we cannot describe related work in these specific areas. For a survey of mobile code languages and systems, the interested readers may refer to [16], while [22] compares several mobile agent systems based on Java. Similarly, [14] compares JEDI with other event-based systems, while [12] and [13] include an extensive comparison between PROSYT and other PSSs.

To the best of our knowledge, there have been no published experiences of the combined use of mobile agents and event-based coordination frameworks and of their use to implement cooperative and distributed business processes. In this section, we briefly compare PROSYT with other PSEE with respect to system architecture.

Most of the PSEEs developed during the last years adopts a standard client-server architecture. Process enactment is centralized and tools communicate with the engine through point-to-point connections. Examples of such PSEEs are Adele [8], Arcadia [26], EPOS [11], JIL [25], Marvel [21], Merlin [23], Oikos [1], Process Weaver [19], Provence [7], SENTINEL [15], and SPADE [6, 5].

Endeavors [9] adopts a complex architecture to distribute process enactment. Several process engines may coexist. They communicate with standard point-to-point connections. As mentioned, this approach reduces the possibility of changing the system architecture at run-time to cope with changes in the underlying environment.

The APEL [17] architecture has been explicitly conceived to support large, distributed processes. It takes benefit of a composite communication paradigm among components, which includes both point-to-point and event-based connections. It differs from the PROSYT architecture because it does not takes benefit of code mobility to change the way process enactment is distributed at run-time.

6 Conclusions and future work

The design of a distributed process support system like PROSYT has been a challenging task. Since the traditional client-server design paradigm did not satisfy PROSYT requirements, we had to look for less established approaches, like event-based integration frameworks and mobile agents. The resulting hybrid design paradigm, which integrates the two approaches and a limited use of client-server interconnections, proved to be quite useful. We argue that this experience can be useful for others who are implementing similar kinds of large, distributed, reactive, and cooperative applications.

Our experience has also shown the drawbacks of event-base and mobile code infrastructures. As for event-based infrastructures, the main drawback we had to face was the difficulty in supporting synchronous cooperation in cases where this form of cooperation is necessary. The second PROSYT prototype shows how this problem can be partially solved by adopting a mixed architecture, which involves a limited form of client-server communication. In the future, we plan to investigate the possibility of integrating synchronous communication primitives into JEDI to eliminate the need of using client-server technologies.

Another difficulty related to event-based infrastructures has to do with scalability. Although JEDI provides a solution to this problem in terms of distribution of the event manager, more is needed to achieve full scalability at the Internet level [24].

Regarding agent mobility, in our system we took a rather simple and ad-hoc approach, by providing a primitive that allows a reactive object to move to specific locations. It would be nice in the future to be able to use languages that provide a richer set of integrated abstractions to describe mobile computations in a structured, natural, and elegant way. This is an active research field in which we may expect progress in the future [16].

Acknowledgements

We are grateful to Alfonso Fuggetta and Elisabetta Di Nitto for their contribution to the design of JEDI. We wish also to thank S. Beretta, F. Ferrari, and A. Nava who provided an essential support in the development and implementation of JEDI and PROSYT.

References

[1] V. Ambriola, P. Ciancarini, and C. Montangero. Enacting software processes in Oikos. In *Proceedings of the 4th ACM SIGSOFT Symphosium on Software Development Environments*, volume 15 of *Software Engineering Notes*, Irvine (California USA), December 1990.

[2] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing process-centered environments. *ACM Transactions on Software Engineering and Methodology*, 6(1), July 1997.

[3] S. Baker. *CORBA Distributed Objects*. ACM Press Books. Addison Wesley Longman Ltd., 1997.

[4] M. Baldi and G. Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In R. Kemmerer and e. K. Futatsugi, editors, *Proceedings of the 20th International Conference on Software Engineering (ICSE'99)*, Kyoto (Japan), April 1998.

[5] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering*, 19(12), December 1993.

[6] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: an environment for Software Process Analysis, Design, and Enactment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.

[7] N. Barghouti and B. Krishnamurthy. Using event contexts and matching constraints to monitor software processes. In *Proceedings of 17th International Conference on Software Engineering*, Seattle (Washington - USA), April 1995.

[8] N. Belkhatir, J. Estublier, and W. L. Melo. Adele2: A support to large software development process. In *Proceedings of the 1st International Conference on the Software Process,* Redondo Beach CA (USA), October 1991.

[9] G. A. Bolcer and R. N. Taylor. Endeavors: A process system integration infrastructure. In *Proceedings of the Fourth International Conference on Software Process (ICSP4)*, Brighton, UK, December 2-6 1996.

[10] A. Carzaniga, G. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering,* Boston, MA, May 1997.

[11] R. Conradi et al. Design of the Kernel EPOS Software Engineering Environment. In *Proceedings of the First International Conference on System Development Environments and Factories*. Pitmann Publishing, 1990.

[12] G. Cugola. *Inconsistencies and Deviations in Process Support Systems*. PhD thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, Feb 1998.

[13] G. Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Transactions on Software Engineering*, 24(11), Nov 1998.

[14] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE98)*, Kyoto (Japan), April 1998.

[15] G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione. How to deal with deviations during process model enactment. In *Proceedings of the 17th International Conference on Software Engineering,* Seattle (Washington - USA), April 1995.

[16] G. Cugola, C. Ghezzi, G. Picco, and G. Vigna. Analyzing Mobile Code Languages. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, pages 93–111. Springer, April 1997.

[17] J. Estublier, P. Y. Cunin, and N. Belkhatir. Architectures for process support system interoperability. In *Prooceedings of the Fifth International Conference on the Software Process,* Lisle, IL, Jun 1998.

[18] J. Farley. *Java Distributed Computing*. The Java Series. O'Reilly & Associates, Inc., 1997.

[19] C. Fernström. PROCESS WEAVER: adding process support to UNIX. In *Proceedings of the 2nd International Conference on the Software Process,* Berlin (Germany), February 1993.

[20] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.

[21] G. Kaiser, P. Feiler, and S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, May 1988.

[22] J. Kiniry and D. Zimmerman. A hands-on look at java mobile agents. *IEEE Internet Computing*, 1(4), July/August 1997.

[23] B. Peuschel and W. Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proceedings of the 14th International Conference on Software Engineering,* Melbourne (Australia), May 1992. ACM-IEEE.

[24] D. S. Rosemblum and A. L. Wolf. A design framework for internet-scale event observation and notification. In M. Jazayeri and H. Schauer, editors, *Proceeding of the 6th European Software Engineering Conference (ESEC'97),* LNCS 1301, Zurich, Switzerland, September 1997. Springer-Verlag.

[25] M. S. Sutton and J. L. Osterweil. The design of a next-generation process language. In *Proceedings of the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering,* number 1301 in Lecture Notes in Computer Science, pages 142–158, Zurich, Switzerland, Sep 1997. Springer-Verlag.

[26] R. Taylor, R. Selby, M. Young, F. Belz, L. Clark, J. Wileden, and L. O. A. Wolf. Foundations of the Arcadia environment architecture. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Symphosium on Software Development Environments*. ACM, 1988.

[27] J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet*. LNCS 1222. Springer, April 1997.