Agent Technologies for Plug-in System Architecture Design

Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Heiko Rölke

University of Hamburg, Department of Computer Science,
Vogt-Kölln-Str. 30, D-22527 Hamburg
lastname@informatik.uni-hamburg.de

Abstract.
In this work we present the basic concepts for a dynamic plug-in-based software architecture using concepts from the Petri net-based MAS framework Mulan. By transferring the concepts of agent-orientation to a plug-in-based architecture we are able to design our application and the plug-in-based system on an abstract level. Moreover, general problems that evolve from a highly dynamic and configurable architecture have been solved by basing the conceptual design on multi-agent principles. In this paper we discuss the general properties of extensible systems and the benefits that can be achieved when applying the multi-agent view to their architecture.
In addition to the conceptual modeling of such architectures, we provide a practical example where the concept has been successfully applied in the development of the latest release of Renew. Through the introduction of the multi-agent concepts, the new architecture is now – at runtime – dynamically extensible by registering plug-ins with the management system.
Keywords: Components, dynamic software architectures, high-level Petri nets, modeling, Mulan, multi-agent systems, nets-within-nets, plug-in architectures, reference nets, Renew

1 Introduction

Today's application software has to be adaptable, configurable and customizable to fulfill the needs of the users. Many system developers approach this challenge by introducing extensible systems as plug-in systems to extend or alter the functionality of these applications. Some systems are augmented with simple plug-in mechanisms, others reorganize the architecture of the application towards a system that consists exclusively of plug-ins. While applications of the first category usually resort to simple designs with extremely restrained possibilities of extending, the applications of the second category have to face many challenges to assure consistency and inter-operablility of plug-in components.

Plug-in-frameworks like those of Eclipse [2] or NetBeans [9] provide elaborated plug-in features in their practical environment. However, a conceptual model of a plug-in is not discussed in this context.

By examining plug-in systems from the agent-oriented perspective, restrictions and problems of current systems can be discovered and attacked. Moreover, by basing the architectures of plug-in systems on agent-oriented principles, the designs of the application architectures adopt the advantages of multi-agent systems. These advantages are the handling of problems such as concurrency, conflicting functionality, service dependencies, locality and privacy, compatibility and dynamic extensibility. Challenges of plug-in systems can therefore be addressed in a more general way in Multi-agent systems. One of the foremost benefits is, that the plug-in architecture becomes dynamically extensible i.e. functionality can be altered, added or removed at runtime without the need to restart the application.

Based on agent-oriented Petri nets [8] and the FIPA-compliant MAS framework CAPA (see [1]) we present a conceptual model for plug-in based systems. The idea is to structure and improve such systems using important concepts from the agent-oriented area.

In the following Section 2 we give an introduction for the Petri net-based multi-agent reference architecture MULAN. MULAN, the conceptual basis for CAPA, uses the formalism of reference nets, a high-level Petri net formalism to handle concurrency, distributedness and localities. The focus lies on the concepts in the MAS, which are used in the design of our concept model of a general plug-in architecture. In Section 3 we present our concept model for a dynamic plug-in-based architecture. In Section 4 we present the realization of the concept model in RENEW and discuss some pragmatic design decisions.

Note that the Petri net IDE RENEW is portraited here in two different ways. First, it is used to act as modeling tool and virtual engine for the abstract and the functional MULAN models. Second, it is used as the application, which architecture is examined and presented as a realization of the concept model of a general plug-in architecture. A similar manifold object of discussion is the agent platform CAPA which is realized as a plug-in for RENEW, but also used in the development of the concept model.

2 Agent System Architecture

In this section we will introduce the agent system architecture MULAN together with the CAPA extension. MULAN is implemented using the reference net formalism (and Java) so we start with an overview on reference nets.

2.1 Reference nets

Reference nets [5] are expressive high-level Petri nets that allow nets to be nested within nets in dynamical structures (nets-within-nets [14]). In contrast to ordinary Petri nets, where tokens are passive elements, tokens in nets-within-nets are active elements, i.e. Petri nets. In general we distinguish between two different kinds of token semantics: value semantics and reference semantics. In value semantics tokens can be seen as direct representations of nets. This allows for

nested nets that are structured in a hierarchical order because nets can only be located at one location. In reference semantics arbitrary structures of net-nesting can be achieved because tokens represent references to nets. These structures can be hierarchical, acyclic or even cyclic.

Reference nets may be modelled and executed using the Petri net-IDE Renew [6].

Reference nets are object-oriented nets. Similar to objects in object-oriented programming languages, where objects are instantiations of classes, net instances are instantiations of net templates. Net templates define the structure and behavior of nets just like classes define the structure and methods of objects. While the net instance has a marking that determines its status, the net template determines only the behavior and initial marking that is common to all net instances of one type.

Communication between different nets (net instances) is possible via synchronous channels. Synchronous channels resemble method calls in object-oriented programming languages, but they are more powerful. They temporarily fuse two (or more) transitions and allow the passing of arguments in either directions. Furthermore, the enabledness of the channel is determined by all participating transitions, not only by one caller.

2.2 The Multi-agent System Mulan

Today, agents and multi-agent systems (MAS) are one of the most important structuring concepts for complex software systems. By including attributes like autonomy, cooperation, adaptability and mobility agents go well beyond the concept of objects and object-oriented software development.

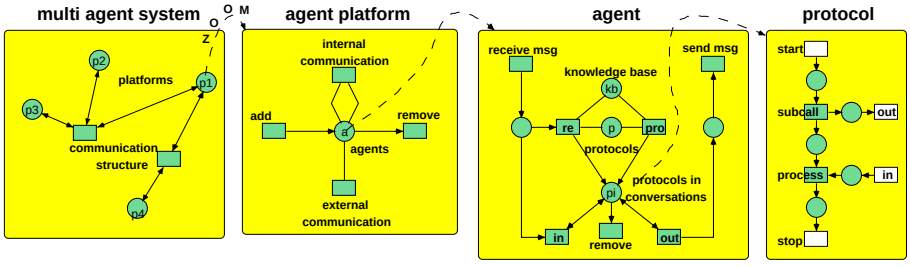

Fig. 1. Agent system as nets-within-nets

The multi-agent system architecture Mulan [8] is based on the nets-within-nets paradigm, which is used to describe the natural hierarchies in an agent system. Mulan is implemented in Reference nets using Renew [6]. Mulan has the general structure as depicted in Figure 1: Each box describes one level of abstraction in terms of the net hierarchy. Each upper level net contains net

tokens, whose structures are made visible by the ZOOM lines.[1] The figure shows a simplified version of MULAN, since for example several inscriptions and all synchronous channels are omitted. Nevertheless, MULAN is an executable model.

The Multi-Agent System View. The net in the upper left of Figure 1 describes an agent system, whose places contain agent platforms as tokens. The transitions describe communication or mobility channels that build up the infrastructure. The multi-agent system net shown in the figure is just an illustrating example, the number of places and transitions or the interconnections have no further meaning.

The Platform View. By zooming into the platform token on place p1, the structure of a platform becomes visible, shown in the upper right box. The central place agents hosts all agents that currently reside on this platform. Each platform offers services to the agents, some of which are indicated in the figure.[2] Agents can be created (transition add) or destroyed (transition remove). Agents can communicate by message exchange. Two agents of the same platform can communicate by the transition internal communication, which binds two agents, the sender and the receiver, to pass one message over a synchronous channel. Transition (external communication) only binds one agent, since the other agent is bound on a second platform somewhere else in the agent system. Also mobility facilities are provided on a platform: agents can leave the platform (via the transition new) or enter the platform (via the transition destroy).

In the diagram some details of the platform are hidden for the reason of simplicity. An important feature that cannot be seen is that a platform may itself act as an agent. By this means, arbitrary hierarchies of agents and platforms are possible, in particular a platform is able to encapsulate its agents from the outside world.

The Agent View. An agent is a message processing entity. It must be able to receive messages, possibly process them and generate messages of its own. Each agent consists of exactly one *agent net* that is its interface to the outside world (in the lower right corner of the figure) and an arbitrary number of *protocols* (lower left corner) defining its behavior. The agent may exchange messages with other agents via the platform. This is done using the transitions receive message and send message. These two transitions are the only interconnection of the agent to the rest of the (multi-) agent system, so the agent is a strongly encapsulated entity.

The central point of activity of a such an agent is the selection of protocols and therewith the commencement of conversations. The protocol selection can

[1] This zooming into net tokens should not to be confused with place refinement.

[2] Note that only mandatory services are mentioned here. A typical platform will offer more and specialized services, for example implemented by special service agents.

basically be performed pro-actively (the agent itself starts a conversation) or re-actively (protocol selection based on a conversation activated by another agent). In the case of the pro-active protocol selection, the place knowledge base is the main enabling condition, the protocols are a side condition.

The Interaction View. The activities of an agent are modeled as protocol Petri nets (or short: protocols) – an example is given in the lower left corner of the figure. The variety of protocols ranges from simple linear step-by-step plans to complex dynamic workflows. Petri nets are well suited for the modeling of procedures or process flows, which can be seen by their wide-spread use in the area of (business) process modeling [15].

2.3 Agents as Platforms

As stated before, platforms in a full featured MULAN system may act like agents and encapsulate the hosted agents. It is therefore no problem to implement e.g. a holonic MAS using MULAN agents. The logical consequence of this approach is to exclusively use these "platform agents" as agents in the MAS. Following this idea leads to a dynamically reconfigurable MAS structure, i.e. a new hierarchy level may be introduced at run-time simply by creating a new (platform) agent and migrating other agents into it.

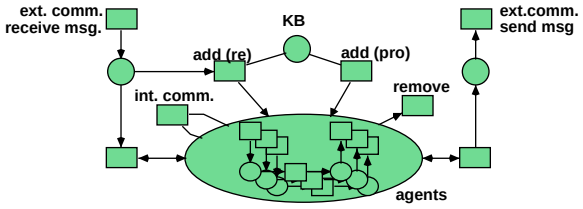

Fig. 2. Agent as a platform

Figure 2 shows an agent that may serve as a platform for other agents. Instead of protocol nets agents serve as description of the platform agent's behavior. The internal agents are depicted in an abstract way. Each of them may be a full-featured (platform) agent.

2.4 CAPA

CAPA (Concurrent Agent Platform Architecture) [1] is a partial re-implementation of the MULAN framework. CAPA ensures the compatibility of the MULAN framework to the FIPA specifications [3]. The internal structure of the agents and the possibilities sketched above are not touched by CAPA.

A part of the compliance to the FIPA specifications concerns the management of an agent platform. In particular, an AMS (agent management system) and a DF (directory facilitator) have to be provided. This is done by placing special agents on each platform that offer the mandatory services. Additional services may be offered by agents residing on the platform. Agents migrating to a platform may offer new services previously lacking on this platform.

This migration idea may serve as a basis for the conceptualization of plug-in architectures. This will be demonstrated in the next chapter.

3 Concept Model for a Dynamic Plug-in Architecture

A dynamic architecture is characterized by extensibility and adaptability. We sketch some general concepts of plug-in systems and map these concepts to agent-oriented concepts. In this work we conceive extensibility as a recursive feature. We apply the idea of nested platform agents to our concept, which leads to recursive extensibility. A system is extended by components, which again are extended by plug-ins, which are (specialized) components. Finally, we show that the recursive agent-oriented plug-in model is a full-fledged plug-in system that allows for dynamic configuration. The realization of this concept in Renew 2.0 is described in Section 4.

3.1 Extensibility

To construct extensible systems it is useful to get a notion of what is meant by extensibility. In software engineering, components have been introduced as units of extensibility. Sametinger gives a definition of a component:

Definition 1 *Component [10, p. 68]*
Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status.

Likewise, extensibility in the agent-oriented view is a first-order concept. An agent system is extended by creating or migrating agents onto a platform. These agents provide additional functionality to all other agents in the system as long as they exist within the system. Removing the agent subtracts its provided functionality from the system.

Obviously we can map the concepts of components on the concepts of agents. Agents are encapsulated (self-contained, clearly identifiable artifacts). They have the capability of action and reaction (specific functions). In FIPA-compliant platforms, the service directory, communication languages and ontologies can provide clear interfaces and documentation. MULAN agents are composed of reusable protocols.

The platform net in Figure 1 visualizes the idea of extensibility: Net tokens that provide functionality – agents – can be put onto and removed from the

central place of this net. These primitive platform services provide the component management of the software system. In the platform net of Figure 1 we are able to say that the system is extensible on one level. This notion of *one-level extensibility* [12] expresses the fact that new components can be introduced to the system but these components can not be extended themselves. The possibility to extend the components leads us to a notion for a recursively extensible system.

The extended agent model shown in Figure 2 already provides recursive extensibility. Every agent can serve as a platform for an arbitrary number of agents, which can be platforms again. Components that recursively extend other components are plug-ins. We take the following definition from Schumacher:

Definition 2 *Plug-in [11, p. 34]*

Plug-ins are components that change the behavior of one or more other components in the system. This is done by using the provided interface of the components.

Up to now, we have a hierarchical structure of the system. The extension relation is strongly tree-structured. The use of reference semantics enables us to relax this condition. With reference nets we would be able to create arbitrary structured extension relations, e.g. acyclic graphs. This would also be desirable for a plug-in system, however, as long as we regard the containment relation of agents within agents as a physical relationship, one agent cannot be located at two platforms at the same time. Nevertheless, the logical platform concept allows an agent to be residing at multiple platforms. Von Lüde et. al. [13] stress the necessity of the use of multiple memberships of agents in platforms from a sociological viewpoint in analogy to the membership of humans in communities.

3.2 Communication Between Components

In a multi-agent system, communication between agents is always carried out through messages. These are transported horizontally by the platforms – this is a basic platform service. In addition, the nesting of agent platforms introduces vertical message passing as depicted in Figure 2 (ext. comm.).

In fact, the communication services provided by each platform allow us to see the functionality of all agents inside that platform as functionality of that platform, including the management services of that platform and its children. Thus, the distinction between management and functionality that we made earlier can be dropped now.

One of the advantages of the component-orientation is the re-usability (cf. [10, p. 68]). This means for instance that the functionality that is offered by the plug-in can be utilized by all components that need this functionality. Therefore, a component has to be able to address another component / plug-in. For this we need the notion of services that are offered by components. Services have to be published and made accessible for other components. The service directory provided by the directory facilitator (DF) of FIPA-compliant platforms serves that purpose. If the DF is modeled as an agent, we can provide its service at any

platform in the hierarchy. However, for a software system it is more practical to have one global service directory. Therefore, we require a DF only at the top-level platform. Likewise, we demand for our system that every component (agent) is a direct member of the top level platform. Although this is an enormous restriction of our general model it simplifies the management of the plug-in system. If each extensible component declares its extension management interface as a public accessible service, potential plug-ins can query the platform for that service and register themselves directly.

A drawback of the current model is that the autonomy of a plug-in / agent is restrained when it comes to its own addition to or removal from selected platforms / components. The agent may initiate its migration, but it has no possibility to prevent migration requests issued by third parties. It is passed as a passive object through the communication channels during the process. The model also does not exactly determine the moment of extension registration and configuration. Therefore, we enforce a life cycle for all plug-ins within the platform.

The participation of a plug-in can be realized by simply adding a synchronous channel request to the add and remove transitions of a platform. These channel requests must be confirmed by the added / removed plug-ins. This ensures that the services of a plug-in cannot be used by components before the plug-in has been properly initialized.

It should be noted that each containment relationship is accompanied by its own life cycle. In this view we can map the agent life cycle, as standardized by the FIPA [4], onto the life cycle of the plug-in on the top-level platform. The life cycles in each containing sub-level should be handled by interaction protocols, defined by the extended component.

A multi-agent system already defines per definition an extensible system. By mapping some of the multi-agents concepts back to the model of a plug-in architecture we are able to design a system that offers extensibility as first-order concept. In addition, an agent-based view also takes into account that extensible systems have to deal with conflicts, concurrency, redundant functionality and also missing functionality. We have shown in this section that a concept model for a dynamic plug-in system architecture can be done on the basis of multi-agent principles. We have achieved a formalization of plug-in systems that enables plug-ins to be loaded dynamically at runtime. In the following section we demonstrate the feasibility of our concepts in a real-world example.

4 Renew Plug-in Architecture

We use Renew itself for a case study where the plug-in concept is applied. The Renew tool has grown enormously since its first release in 1999, and many application-specific extensions have been created in the meantime. These extensions, like a workflow engine, an agent platform or an editor for UML interaction diagrams, are themselves already grown to applications with their own extensions. Up to Renew 1.6, all extensions were compiled into one large application.

Some sets of functionality could be selected by specifying a mode at startup, but mode switching at runtime was not possible. However, a user would normally not need all extensions at the same time, but possibly in arbitrary combinations. Altogether, RENEW is very well suited as a case study for a dynamic, recursive plug-in system.

The plug-in system along with the decomposed application has been released as RENEW 2.0 and presented from the user's point of view last year in [7]. In this section we want to show how the concepts developed in the previous section are applied to the RENEW plug-in system.

4.1 Functional Decomposition

From the user's point of view, RENEW comprises two main components: the simulation engine and the graphical net editor. Already in the first release it has been stated that RENEW supports multiple formalisms, since new formalisms could easily be added by implementing the appropriate compiler. Clearly it is desirable to separate each formalism into its own plug-in. A formalism management component can then provide a registry for all loaded formalisms as well as the basic functionality needed by many formalisms.

In fact, the management of formalisms is divided into two plug-ins, and an analogous partitioning is suggested for each individual formalism as well as each other extension to the simulation engine: One plug-in extends the simulation engine and/or formalism management components. It provides the pure functionality extension without graphical adornments, e.g. the formalism management, or a compiler. The second component is a plug-in of the editor component and provides additional menu entries, graphical representations of net structure and tokens, formalism-specific tools, etc. to the user.

The user benefits from the introduction of the dynamic plug-in system for example in a server scenario: An application that has been implemented using reference nets and *Java* should normally run without the graphical net editor or animated token game. In this case, a user can start a reduced RENEW system where only the non-graphical components are installed. The simulation engine and formalism components are sufficient to run the application. The user may also want to install a so-called Prompt plug-in so he can control the plug-in system and the simulation engine from the command line interface.

Suppose that, after the application has run undisturbed for some time, something gets stuck. The user then has the possibility to load the graphical editor and related plug-ins into the running system to debug the situation. The animated token game, although started long after the simulation setup, will show the current state of the system. So the user can search for the bug and hopefully fix it. Afterward, he can restart the simulation engine. When the application runs again, the editor and related plug-ins can be unloaded from the system and free their resources.

4.2 Applied Concepts

The RENEW plug-in system is implemented along the concepts developed in Section 3. In the system there exists a so-called `PluginManager` that acts like the platform net shown in Figure 1. There is no distinction between components and plug-ins because any component may also act as a plug-in to any other component.

The two transitions that add and remove plug-ins become slightly refined in the RENEW `PluginManager`: Plug-ins can enter the system in two ways. At startup, a plug-in finder looks in a specific location for pre-installed plug-ins, and during runtime plug-ins can be loaded dynamically by supplying an URL to the plug-in loader. Analogously, all plug-ins are unloaded when the system shuts down. The removal of components can also be initiated at runtime through a `unload` command.

All plug-ins are accompanied by a description of their provided services. All plug-ins follow a simple life cycle with mainly two transitions: initialization and shutdown. Plug-ins are asked to confirm these transitions, this gives them back some of their autonomy as demanded in Section 3.2).

The maintenance of the service directory is automated: Each plug-in is accompanied by a description of its provided services, the `PluginManager` maintains the directory during the initialization and shutdown transitions.

Optionally, the `PluginManager` may enforce dependencies between plug-ins. If a plug-in is also accompanied by a description of required services, the `PluginManager` will not include it in the system unless the required services are available, that is provided by some other plug-ins. This mechanism delegates some commonly needed autonomous decisions of the plug-in to the `PluginManager` to simplify the plug-in development process. Likewise, the unloading of a plug-in is prohibited as long as another plug-in requires a service provided by the plug-in to remove. A command to recursively unload all dependent plug-ins is provided. Of course, this dependency enforcement only works for static service requirements—but this is exactly what a *Java* programmer needs to ensure the availability of required class definitions.

Recursive extensibility (as introduced in Section 3.1) is included in the system as a chain of component extensions. An example is the MULAN viewer plug-in, which extends the CAPA component to monitor agent activity on the platform. The CAPA plug-in in turn extends the simulation engine component to include its platform services to all simulated net instances. Since the CAPA plug-in additionally extends the editor plug-in to provide graphical representations for agent tokens, we also have an example for the extension of multiple components by one plug-in.

4.3 Pragmatism

The side condition that the plug-in system should not reduce the application's execution speed necessitated some pragmatic solutions. The concessions we made

are restricted to the *Java* implementation of the plug-in system, the precise and concurrent Petri net semantics of the model in Section 3 have not been weakened.

The most important pragmatic decision is to not follow our usual paradigm of implementation by specification because the two uses of RENEW – as runtime engine *and* as case study – do not mix well. It would be possible to use the nets presented in Section 3 as code base for the plug-in system. By augmenting the nets with *Java* inscriptions that call the existing functionality of the application components, we would get an executable plug-in system rather easily. However, then we would have to set up the Petri net execution environment before the plug-in system in order to execute the net implementation. This would introduce a circular precedence because the simulation engine is a part of the application on top of the plug-in system. Therefore, we decided to use the insights gained from the concept model to re-implement the plug-in system in pure *Java*.

A consequence is the different behaviour of the RENEW and Java runtime environment with respect to dynamic linking. While in the reference net formalism the communicating plug-ins are linked at runtime for each communication individually, the Java virtual machine links the classes once when they are loaded. For full dynamic behaviour we would have to implement a dynamic linking layer of our own, but we decided to skip that part. The needed indirection would slow down inter-plug-in communication of repetitive jobs.

5 Conclusion

MULAN is an expressive modeling framework, based on the reference net formalism, that is capable of modeling dynamic system architectures. Models that are built with MULAN agents can profit from the multi-agent architecture, which among other benefits provide the ability to construct arbitrary and dynamic structures. The agent/platform model allows to express extensibility and dependency relationships of system components. Furthermore, the possibility to concretize the model by refinement leading to a functional model is of great advantage when designing, discussing, prototyping and (re-)designing a system.

The presented generic – reference net multi-agent based – concept model for a dynamic architecture proves to be an approach that is both, sufficiently abstract for expressive modeling and sufficiently concrete to be able to transfer it to a real-world application. Moreover, it is the only modeling technique – to our knowledge – that is able to represent a flexible, adaptable and dynamic design of an application architecture. The level of abstractness is a benefit to the general design decisions. The level of concreteness helps the architect and developer to experiment and evaluate the model prior to the implementation.

The concept model comes with an explicit top-level net, the platform or plug-in management system. Furthermore, the similarity of structures on the top level and all other levels allows for the introduction of independent service and extension management units on every level. Our model is capable of describing a pluggable plug-in mechanism. Such a model is useful to merge multiple systems with independent management architectures.

The Petri net IDE Renew has undergone major refactorings and this process is still in progress. However, the preliminary results are promising. It is safe to say that the decision to refactor the system was the right way to go. We achieved a lean and flexible plug-in mechanism that permits arbitrarily nested plug-ins.

Beside just another plug-in mechanism with specific features that are very valuable in the context of our research and development, a visual modeling concept for plug-ins has been presented. In fact, currently well-established modeling techniques are highly elaborated and powerful but also oriented towards static architecture design and very resistant against paradigm shifts. In order to improve modern architecture design many dynamic aspects have to be included as first-order concepts. Extensibility is one of them.

We are looking forward to unleashing the full power of our architecture model by supporting an interleaved multi-formalism simulation support. Thereby, several advantages of different formalisms can be combined to the advantage of the designed model.

References

1. M. Duvigneau, D. Moldt, and H. Rölke. Concurrent architecture for a multi-agent platform. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Proc. of AOSE 2002*, volume 2585 of *LNCS*, Berlin, 2003. Springer Verlag.
2. Eclipse Homepage. http://www.eclipse.org, 2004.
3. FIPA. Foundation for Intelligent Physical Agents. http://www.fipa.org, June 2004.
4. Foundation for Intelligent Physical Agents. *FIPA Agent Management Spec.*, 2004.
5. Olaf Kummer. *Referenznetze*. Logos-Verlag, Berlin, 2002.
6. Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – The Reference Net Workshop. http://www.renew.de, June 2004. Release 2.0.
7. O. Kummer et. al. An extensible editor and simulation engine for Petri nets: Renew. In Jordi Cortadella and Wolfgang Reisig, editors, *Proc. of ICATPN 2004*, number 3099 in LNCS, pages 484–493, Berlin, 2004. Springer Verlag.
8. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the structure and behaviour of Petri net agents. In *Proc. of ICATPN 2001*, volume 2075 of *LNCS*, pages 224–242, Berlin, 2001. Springer Verlag.
9. NetBeans Homepage. http://www.netbeans.com, 2004.
10. J. Sametinger. *Software Engineering with Reusable Components*. Springer Verlag, Berlin, 1997.
11. Jörn Schumacher. Eine Plug-in-Architektur für Renew: Konzepte, Methoden, Umsetzung. Diplomarbeit, University of Hamburg, Department of Computer Science, October 2003.
12. Clemens Szyperski. *Component software: beyond object-oriented programming*. ACM Press books. Addison-Wesley, 2. edition, 2002.
13. R. v. Lüde, D. Moldt, and R. Valk. *Sozionik: Modellierung soziologischer Theorie*, volume 2 of *Reihe: Wirtschaft – Arbeit – Technik*. Lit-Verlag, Münster - Hamburg - London, 2003.
14. Rüdiger Valk. Petri Nets as Token Objects - An Introduction to Elementary Object Nets. In J. Desel and M. Silva, editors, *19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal*, number 1420 in LNCS, pages 1–25, Berlin, 1998. Springer Verlag.
15. Wil van der Aalst, Jörg Desel, and Andreas Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*. Number 1806 in LNCS. Springer-Verlag Berlin, 2000.