Modeling Dynamic Architectures Using Nets-within-Nets

Lawrence Cabac, Michael Duvigneau, Daniel Moldt and Heiko Rölke

University of Hamburg, Department of Computer Science,
Vogt-Kölln-Str. 30, D-22527 Hamburg
{cabac,duvigneau,moldt,roelke}@informatik.uni-hamburg.de

Abstract. Current modeling techniques are not well equipped to design dynamic software architectures. In this work we present the basic concepts for a dynamic architecture modeling using nets-within-nets. Nets-within-nets represent a powerful formalism that allows active elements, i.e. nets, to be nested in arbitrary and dynamically changeable hierarchies. Applying the concepts from nets-within-nets, therefore, allows us to model complex dynamic system architectures in a simple way, which enables us to design the system at different levels of abstractions using refinements of net models.
Additionally to the conceptual modeling of such architecture, we provide a practical example where the concept has been successfully applied in the development of the latest release of Renew (Version 2 of the multi-formalism Petri net IDE[1]). The overall monolithic architecture has been exchanged with a system that is divided into a plug-in management system and plug-ins that provide functionality for the users. By combining plug-ins the system can be adapted to the users' needs. Through the introduction of the Petri net concepts, the new architecture is now – at runtime – dynamically extensible by registering plug-ins with the management system. The introduced architecture is applicable for any kind of architecture but most suitable for applications with dynamic structure.

Keywords: High-level Petri nets, Nets-within-nets, reference nets, Renew, plug-ins, components, dynamic software architecture, modeling

1 Introduction

Today's software systems are getting more and more complex. The amount of functionality and the number of features that are put into a system increases steadily. Moreover, features that are only loosely related are included into a system making the software more attractive for some user and at the same time too complex and bloated with features for other users. The need to switch the design to configurable or adaptable / customizable systems has been recognized for a long time.

Many systems already provide the possibility to extend the functionality with plug-ins. Some of the software systems are component-based, leaving the user in

[1] Integrated development environment.

G. Ciardo, P. Darondeau: Applications and Theory of Petri Nets, LNCS 3536, pp. 148–167, 2005.

charge of the degree of versatility of the utilized system. However, the flexibility of these systems is usually limited. Most systems only allow static configuration, some also allow to extend the functionality at runtime in a very limited way.

Renew serves as an example for a complex system. It is an IDE for several Petri net (and Petri net related) formalisms. Examples of these are reference nets [5], P/T-nets, Workflow nets [4], Feature Structure nets [17], Timed Petri nets and Multi-agent nets [11]. Renew has grown significantly in the last couple of years.

Many users did not need the many specialized formalisms and features in their everyday work with the tool. However, still more new extensions were on the verge to be developed, threatening that the system would grow further. While some users / developers were using the new features, others had to deal with the resulting overhead. The need for a highly customizable architecture became obvious.

The idea was born to redesign Renew in an extensively flexible way that would solve the problems and lead to an architecture that is configurable, customizable and extensible. Moreover, the developers envisioned a system that is even more flexible: A system that is customizable at runtime, i.e. a system whose architecture is dynamically configurable.

The notion of flexibility in the re-design has to be concretized, therefore a model for a dynamic architecture is needed. The modeling process for an architecture design helps significantly to understand the architecture and the dependencies between the involved units. Modeling helps to share and discuss ideas with other software architects and to convey these ideas to developers. Furthermore, it helps to establish the concept, to find and eliminate conceptual problems and to visualize the system. However, to model a system architecture design, an established and expressive modeling technique has to be applied.

There are various architecture modeling techniques like UML [9] or architecture description languages (ADL, see [8,15]) that could be used to design our architecture. Nevertheless, we use the reference net formalism [5] which is based on Valk's nets-within-nets [16] paradigm because it combines many features that are otherwise not available in one single language: With Petri nets we have formal semantics, operational semantics and a plain graphical notation. The reference net formalism adds coverage of dynamic changes in behavior and structure as well as object-based properties like encapsulation, polymorphism and instantiation. So we can use a lot of features that would otherwise not be available within one single technique. However, the reference nets are currently not well equipped to describe static aspects of a system, like interfaces or a type hierarchy. This drawback is of minor relevance since we are interested in a dynamic architecture model in this paper.

We designed our concept model for a dynamic architecture with reference nets. The concept was successfully applied in the re-design of Renew resulting in the current version (2.0). We achieved to transform the old—in some way extensible, but mainly monolithic—design into a highly configurable dynamic architecture.

According to our approach of implementing through model refinement we introduce the abstract concept model and successively refine it until we obtain a functional model. For the reason of efficiency, the functional model is re-implemented in *Java*. Since our models are designed and executed in Renew it is possible to utilize both, the model implementation and the *Java* implementation.

Note that Renew is used as modeling tool for our recursive concept model for a dynamic architecture and also as target system for the application and realization of the concept model. This self-reflective feature is one of the key aspects of our approach.

In the following section we give an introduction to reference nets. The focus lies on some features of reference nets that we use extensively in the design of our concept model. In Section 3 we present our concept model for a dynamic architecture. In Section 4 we present the realization of the concept model in Renew and discuss some pragmatic design decisions. Section 4.4 relates our concept and modeling technique with other approaches for configurable systems.

2 Nets Within Nets

Note for the experienced reader: If you already know reference nets and are well-acquainted with their concepts we recommend that you skip this section and continue with Section 3.

Nets-within-nets are expressive high-level Petri nets that allow nets to be nested within nets in dynamical structures. In contrast to ordinary 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.

In the following sections we will discuss reference nets because of three reasons. First, they are supported in the tool Renew, second, they show the basic principles of nets-within-nets, and third, they allow for acyclic nesting structures. If the number of references for nets are reduced to one reference for a net, value semantics and reference semantics are equivalent.

2.1 Reference Nets

Reference nets [5] are object-oriented high-level Petri nets, in which tokens can be nets again. For these nets-within-nets [16], referential semantics is assumed. Tokens in one net can be references to other nets. In a simple setting of a

single nesting of nets, the outer net is called system net while a token in the system net refers to an object net. Nevertheless, object nets themselves can again contain tokens that represent nets, and thus a system of nested nets can be obtained. The benefit of this feature is that the modeled system is modular and extensible. Furthermore, transitions in nets can activate and trigger the firing of transitions in other nets, just like method calls of objects, by using synchronous channels [2,5].

Renew (The **Re**ference **Ne**t **W**orkshop [6,7]) combines the nets-within-nets paradigm of reference nets with the implementing power of *Java*. Here tokens can also be *Java*-objects and nets can be regarded as objects.

In comparison to the net elements of P/T-nets, reference nets offer several additional elements that increase the modeling power as well as the convenience of modeling. These additional elements include some arc types, virtual places and a declaration. Several inscription types have been added to the net elements providing functionality for the different net elements. Places can be typed and transitions can be augmented with expressions, actions, guards, synchronous channels and creation inscriptions. In the following paragraph we will focus on aspects of net instances and synchronous channels. Detailed information on nets-within-nets and reference nets can be found in [5] and [16]. It should only be mentioned briefly that we use reserve arcs as a convenient notation. In addition, we also use flexible arcs (see [10]) in our models. These are expressive arcs that can drop all elements of a collection onto a place and withdraw all (pre-known) elements of a collection from a place.

Net Instances and Synchronous Channels 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.

The paradigm of nets-within-nets introduced by Valk [16], allows tokens to be nets again. In reference nets, tokens can be anonymous, basic data types, *Java* objects or net references. Any net instance can create new net instances similar to an object creating new objects. The new net instance is marked with the initial marking according to the specification of the net template.

The notation of the creation inscription with the usage of the keyword `new`, to create a new instance, is displayed in Figure 1. In this example the system net has an initial marking of three integer tokens. Thus the transition can fire three times creating three new net instances.

The three new net instances are bound to the variable `x` and put into the output place. This is displayed in Figure 2, in which the net templates and the

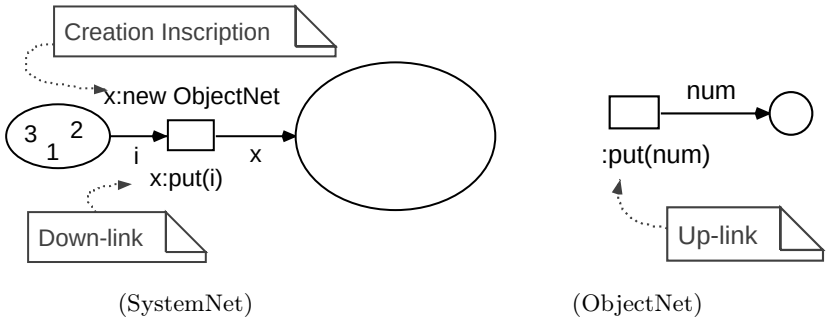

Fig. 1. Example system net and object net.

net instances for both nets are displayed. There is one instance of the system net and three instances of the object net.[2]

The different net instances are each created during one firing of the transition of the system net, which bears the creation inscription. The tokens referring to the net instances are put into the output place. In the net instance `SystemNet[0]` in Figure 2 these three tokens are displayed in the output place. Navigation among the net instances is done in a hypertext fashion by clicking on the reference in a net instance in order to open the referred net instance window.

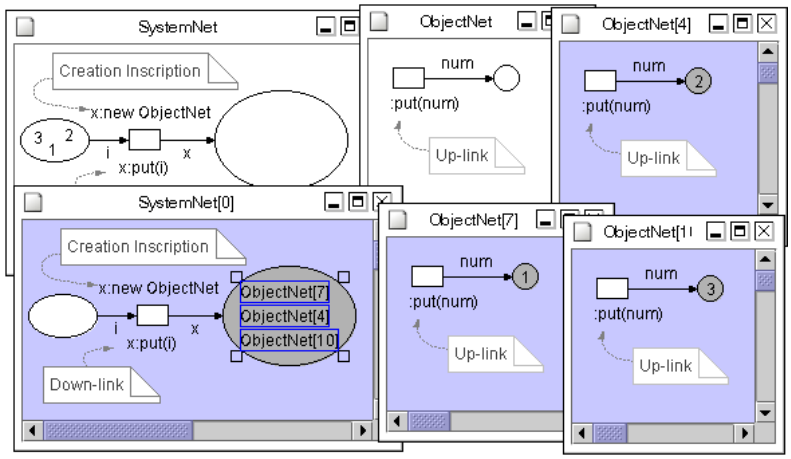

Fig. 2. A screen shot of a system net and an object net, the templates and several net instances.

[2] In Renew net instances can be identified by the names of the windows and the window background colors. Net templates have a white background color and net instances have an integer number attached to their window title bars that identifies the distinct instances. These identifying numbers are also attached to the tokens.

For the communication between net instances, synchronous channels are used. A synchronous channel consists of two (or more) inscribed transitions. There are two types of transition inscriptions for the two ends of the synchronous channel: downlinks and uplinks. The synchronous channel forms a symmetric channel, therefore bidirectional communication is possible. Two transitions that form a synchronous channel can only fire simultaneously and only if both transitions are activated. Downlink and uplink belong to a single net or to different nets. In both cases any object, also another net instance, can be transferred from either transition to the other. If two different net instances are involved, it is thus possible to synchronize these two nets and to transfer objects in either direction through the synchronous channel. For this the system nets must hold the references of the object nets as tokens (as in the output place in Figure 2).

The simple example of Figures 1 and 2 does not only show the creation of net instances, but also the application of synchronous channels. A synchronous channel `put(.)` connects the two transitions of system net and object net. The system net holds the reference to the object net instance `x` that is created during the firing of the transitions. The downlink `x:put(i)` calls the uplink in the object net `:put(num)`. The integers are taken from the input place of the system net and bound to the variable `i` used as an argument in the channel inscription. Both transitions fire simultaneously and the two variables `i` and `num` are unified. Thus `num` is bound to the same integer as `i`, which finally is put into the output place of the object net. So the different numbers in the output places can distinguish the different net instances of the object net, which is the reason why we have chosen numbers as tokens in this example.

By using two (or more) parameters in a channel, information can be transfered in both directions synchronously. For instance a simple database lookup can be implemented by using this feature. One of the parameters serves as key and the other one as value. This will be used in Section 3.

2.2 Renew

With Renew it is possible to draw and simulate Petri nets and reference nets. The simulation engine can execute a net by creating an instance of the net. Any simulated net can instantiate other nets. Hence it is possible to produce many instances of different nets. The relationship between net template, also simply called net, and net instance can be compared to the relationship of class and object (see Section 2.1).

Editor Figure 3 shows the graphical user interface (GUI) of Renew, a simple Petri net in the back and a net instance.

The user interface consists of the menu bar, two palettes and a status line. The menu bar offers menus for general operations, attribute manipulations, layout adjustment and Petri net-specific operations. It also provides the possibility to control the simulation. Of the two palettes the first one consists of usual drawing tools while the second one holds the Petri net drawing tools for the creation

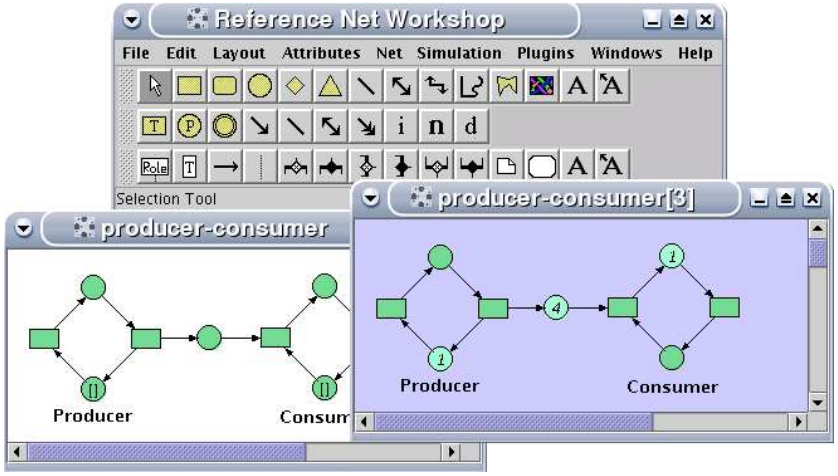

Fig. 3. RENEW GUI, Petri net and net instance (producer-consumer example).

of transitions, places, virtual places, arcs, test arcs, reserve arcs, inscriptions, names and declarations. In addition to these tools, the editor reacts in a context-sensitive manner to facilitate the drawing of nets. One example is the dropping of arcs on the background that creates a new place if the arc starts at a transition and vice versa. Another example is the right click on inscribable elements that produces an inscription for this element with a context sensitive default value.

Simulator Net templates hold the initial marking while net instances hold the current marking. In Figure 3 the producer-consumer example has been started. In the net template (background) one of two black tokens ('`[]`') of the initial marking can be seen in the place labeled `Producer`. While the net instance by default only shows the number of tokens in a place it is also possible to show the contents of the places by clicking on the numbers (compare with Figure 2).

In the following section we model our concept of an dynamic architecture with reference nets. All modeling is done with RENEW, which is also the target for the realization of the concept model.

3 Concept Model

A dynamic architecture is characterized by extensibility and adaptability. In this work we conceive extensibility as a recursive feature. A system is extended by components, which again are extended by plug-ins, which are (specialized) components. Reference nets as nets-within-nets allow nets to be nested within other nets. They are, therefore, capable of modeling an extensible architecture in which a management component can act as a container for components – in

our model other nets. These components again are used as containers for other components. We develop the model successively from a simple one-level view to a full-fledged plug-in-based system.

This chapter introduces the concept model for extensible systems that allows components to extend a system dynamically during runtime. 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. This is modeled here with reference nets starting on an abstract level that is then further concretized throughout this section.

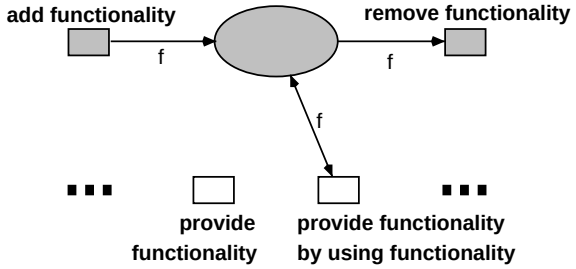

Fig. 4. Model for extensible systems.

Figure 4 illustrates an extensible system in a most general way. The upper grey colored elements of the net define the extension management part of the system. The net shows the system as reference net in which the central place acts as the container for extensions. Functionality is added to the system by a synchronous channel at the transition labeled `add functionality`[3] and then put on the central container place. Functionality is removed by the transition labeled `remove functionality`.

The white transitions in the lower part are representatives for the available domain-specific functionality of the system. Some of the functionality may incorporate the functionality provided by extensions that lie in the central container place. All elements `f` that are extending the system are nets again, according to the nets-within-nets paradigm. An exemplary marking of the net is shown in Figure 5.

In this figure, exemplary channel inscriptions are visible. The small net tokens provide functionality by the channel uplinks `:func1()` or `:func2()`. The system provides functionality to the user through the channel uplinks `:funcA()` and

[3] The channel inscriptions are omitted in this figure because the focus lies on the concepts.

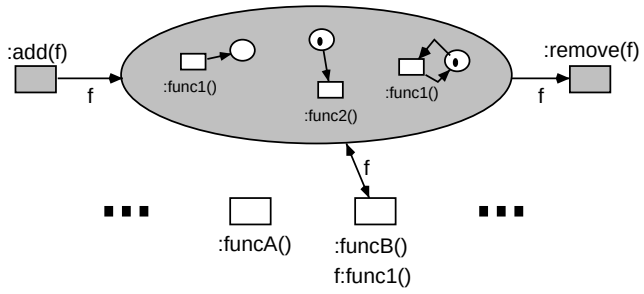

Fig. 5. Extensible system with net tokens.

`:funcB()`. However, to provide `funcB`, the system uses extended functionality. It specifies the extension interface, here represented by the channel downlink `f:func1()`. There are two available extensions providing the specified channel by an uplink, so the system may choose any of these functionality providers. Adding and removing functionality to and from the system is also accessible to the user through channel uplinks `:add(f)` and `:remove(f)`.

This model leads to the concept of components. Components are units of extensibility. The net tokens that represent extensions of functionality in Figure 5 perfectly fit that notion. So we can define components by this net model. A textual definition of components is given by Schumacher in [13].[4]

Definition 1 *Component*

A component is a unit of distribution that comprises executable code accompanied by appropriate documentation and provides domain-specific functionality.

3.2 Recursive Extensibility

In the current model we are able to say that the system is extensible on one level. This notion of *one-level extensibility* [14] expresses the fact that new components can be introduced to the system but these components can not be extended themselves. The concept of extension management of the system can also be applied to the components. Since the extensibility model of the previous section is built with components, we can call the extension management component management. The possibility to extend the components leads us to a notion for a recursively extensible system.

Figure 6 shows the modified system where components may be contained within components. It can be observed that the components implement the same management interface as the system net model. A component only differs from the model of the system or any other component in the domain-specific part, which is not shown explicitly in Figure 6.

[4] This definition is closely related to Sametinger [12].

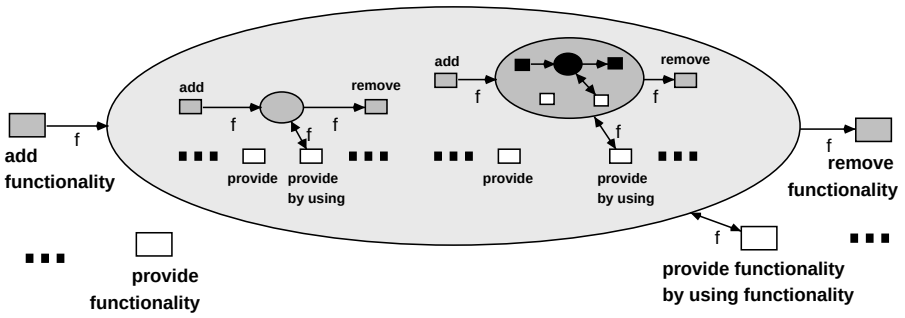

Fig. 6. Model of a recursively extensible system.

We can now regard the components that recursively extend other components as plug-ins. From Schumacher's viewpoint a system is composed of components and plug-ins. Plug-ins are special components that change the behavior of the system by changing the behavior of components. The full textual definition given in [13, p. 34] is:

Definition 2 *Plug-in*

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.

With the introduction of the plug-in concept we can regard the component management as plug-in management.

However, in our net model, there is no difference between the component nets and the system net. So we cannot distinguish components from plug-ins. The system as well as all components can be extended by plug-ins. In the further refinement of the net model it is possible to observe a difference between system and components, and thus the distinction between plug-ins and components becomes significant.

Up to now, we have a hierarchical structure of the system. The extension relation is strongly tree-structured. The use of reference semantics as described in Section 2.1 enables us to relax this condition. It is possible to add one plug-in to multiple different components, so the extension relation forms an acyclic graph. Even a cyclic relation is possible, although that may lead to endless recursion.

In order to concretize the model further, we will now describe how the adding of plug-ins is introduced into the model. Figure 7 shows the simple mechanism of adding plug-ins to plug-ins. A plug-in `p2` is added to another plug-in `p` by using the derived functionality of the component represented by the main net. Here we have a chain of channel synchronizations: When the uplink `:subadd(p2)` is called by some other net instance's downlink, then the downlink `p:add(p2)` at the same transition synchronizes with the uplink `:add(p)` of a third net instance.

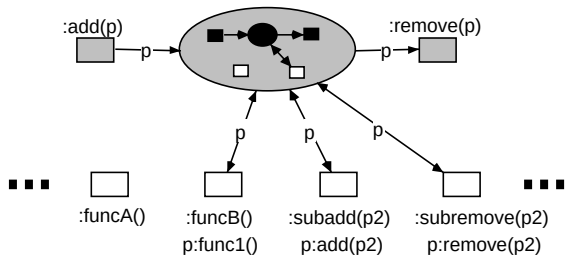

Fig. 7. A component allows the extension of its plug-ins.

The management part of that third net instance looks the same as in Figure 7, it may also be another instance of the same net. The unloading of the plug-in is done in a similar way.

This mechanism enables the system to be recursively pluggable through plugging plug-ins into components at an arbitrary number of levels, as long as every level provides such a call-through functionality. The mechanism also shows that management of one plug-in can be seen as functionality of another plug-in. The distinction between management and functionality that we made in the first models can be dropped now.

A drawback of the current model is that the plug-in `p2` cannot control its adding to or removal from selected components. It is passed as a passive object through the channels during the process. This problem will be discussed in the following section.

3.3 Communication Between Components

One of the advantages of the component-orientation is the re-usability. This means that the functionality that is offered by the plug-in is utilized by all components that need this functionality. Therefore, a component has to be able to address another component / plug-in. For this we introduce the notion of services that are offered by components to other components.

Services have to be published and made accessible for other components. Each component provides an interface by which the descriptions of the offered services are accessible. A global service directory is needed so that components can look up service provider components. We refine the abstract model of Figure 7 and introduce a net that has the management of plug-ins and their services as its only functionality. This net is called the plug-in management system (PMS).

Figure 8 shows the model of a PMS with the service lookup infrastructure (SLI). When a new plug-in `p` is added to the system, the PMS asks for the description of the services that are offered (`p:getServices(sd)`) by this plug-in. This is in this net indicated by the *flexible arcs* (two arrow tips) which are able to drop all elements of a collection `sd` onto a place or to withdraw all elements of a collection simultaneously from a place, respectively.

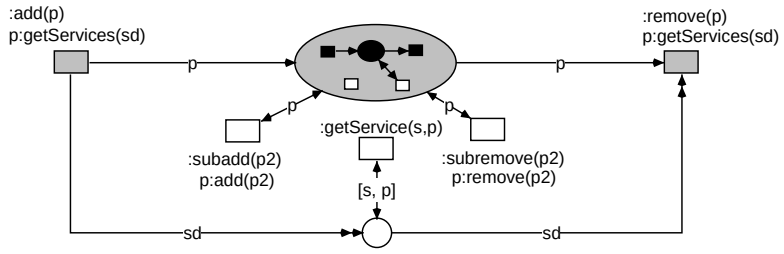

Fig. 8. A PMS that supports plug-in services.

Each service description is stored as a key-value tuple (`[s,p]`). Other components can get information about available services and their suppliers by using the channel uplink `:getService(s,p)`.[5]

To be able to use the lookup functionality of the PMS, all components need a reference to the PMS. Therefore we define that the PMS net should have exactly one instance. This instance can be regarded as the root of the graph of the extensibility relation. So the PMS is our explicit top-level net instance of the whole system.

Provided each plug-in has a reference to the PMS, it could also call the `subadd` and `subremove` channels of the PMS to control its own registration with other components. A more elegant approach is to use the SLI: If each extensible component declares its extension management interface as a public accessible service, potential plug-ins can query the PMS for that service and register themselves directly. So we can omit the `subadd` and `subremove` channels from the PMS (and from all other components, too).

Our current model is lacking a mechanism that passes the PMS reference to each component. 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 PMS.

In Figure 9 the functionality of managing the life cycle of components is added to the PMS by two extra transitions with channel downlinks `p:init(pms)` and `p:shutdown()`). At the `init` transition, the added component gets informed about its addition to the system, receives the PMS reference and gets the chance to connect to other components that provide required services. Note that the retrieval of service descriptions has also moved from the transition where the component is added to the transition where the component is initialized. This ensures that the services of a component cannot be used by other components before the component has been properly initialized.

The introduction of the PMS and its service influences our model of a component. The updated model is shown in Figure 10. The gray management part has grown because of three things: First, right below the central extension place,

[5] Modeling database lookups is explained in Section 2.1.

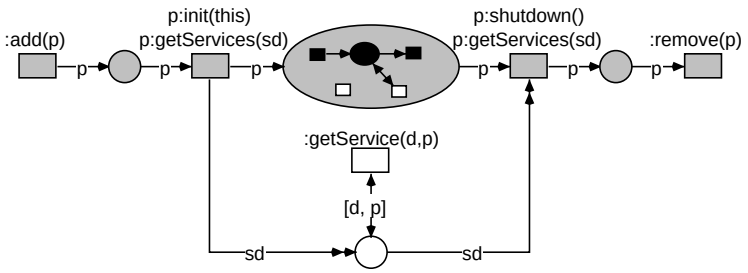

Fig. 9. A PMS with life cycle management.

the life cycle support has been added, the new place holds the PMS reference. Second, at the right, the channel uplink `:getServices(sd)` has been added. Here the PMS can extract information about the component. And last, left and right of the central extension place, the life cycle management of the PMS is repeated for plug-ins.

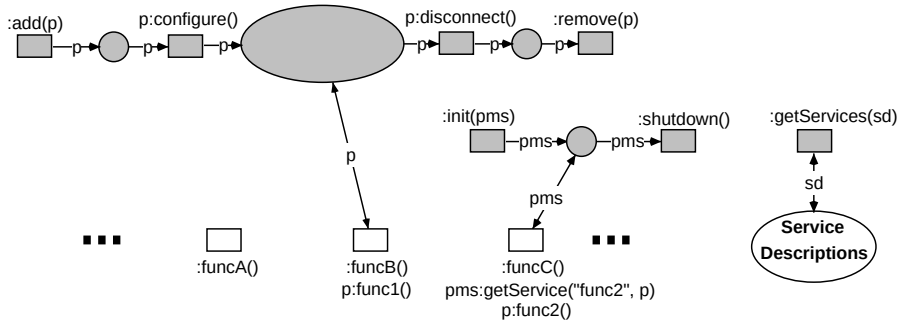

Fig. 10. Refined model of a component.

The life cycle management for plug-ins is introduced for the same reasons as the system-wide life cycle is introduced at the PMS level. However, the channel names are intentionally changed to `configure` and `disconnect` to indicate that there is exactly one global life cycle for each component. If the component also is a plug-in, it must additionally support the life cycle specific to the component where it can be plugged in. If one component can be plugged into multiple other components, it has to support each life cycle, respectively. So we now have refined the plug-in concept, we speak of plug-ins with respect to certain components.

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, this was not flexible enough: 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. First we sketch how the functionalities of the application have been decomposed into several components. We will show where the plug-in concepts can be found in some exemplary components and where the dynamics come in. Last we will mention some concessions we had to make to keep the application usable.

4.1 Functional Decomposition

From the user's point of view, Renew comprises two main components: the simulation engine and the 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.

Figure 11 shows some plug-ins of the current decomposition.[6] At the bottom, there are some unnamed class libraries that are used by many or all plug-ins. Some of these libraries are integrated into the application as a plug-in of their own, but they do not provide any extension interfaces. At the right there is the main plug-in of Renew, the simulation engine. This plug-in also includes the input and output interfaces of the simulation engine, e.g. non-graphical net representation classes, token game feedback, remote control or database backup. With this plug-in and a formalism, it is possible to execute a Petri net system (without graphical feedback).

The graphical editor comprises two plug-ins: JHotDraw and Gui. This is due to historical reasons, we decided to re-surface the JHotDraw framework that had

[6] It has to be noted that the decomposition of an existing application with approximately 900 classes in 30 packages into several components is not unique and therefore some functionalities might be reassigned between components in future releases. The refactoring of Renew is still work-in-progress.

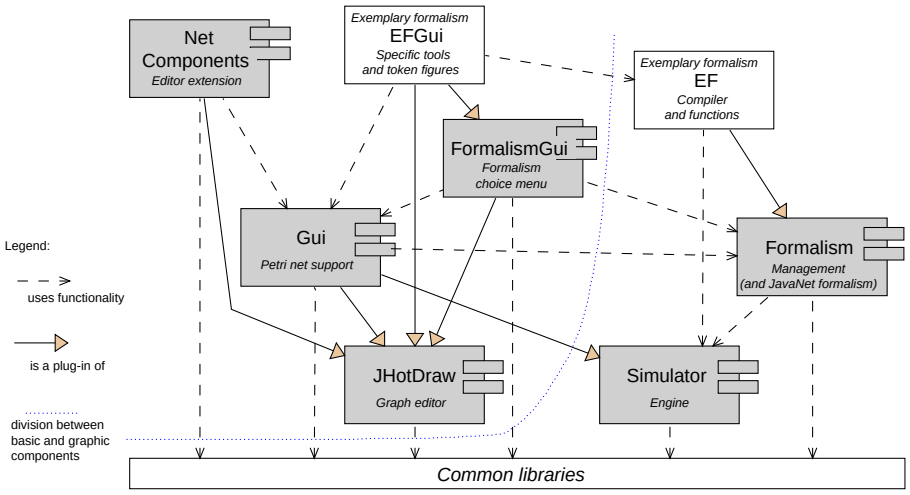

Fig. 11. Plug-ins and their dependencies as of Renew2.0

served as the basis for the net editor. The Gui plug-in enhances the JHotDraw application by Petri net specific figures and control commands for the token game. The NetComponents plug-in serves as an example for a new plug-in that has been added after the application decomposition. It has been presented in [1] and extends the editor by a tool-bar with commonly used patterns of net elements.

The management of formalisms has been divided into two plug-ins. Formalism manages the registry of known (i.e. loaded) formalisms and provides an API to select a formalism. The FormalismGui plug-in establishes the connection to the editor by presenting the available choices to the user. It also tailors the editor's menu and tool bars to the currently selected formalism. The two white components at the top of Figure 11 represent an arbitrary formalism. The standard formalism of Renew—the reference net formalism—is integrated in the Formalism plug-in for the time being.

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 Simulator and Formalism plug-ins (generally speaking, all components and common libraries right or below the dotted line in Figure 11) are sufficient to run the application. The user may also want to install a Prompt plug-in[7] so he can control the plug-in system and the simulation engine from the command line interface.

[7] The Prompt plug-in is not included in Figure 11, but is available for download from the Renew homepage [6].

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 components 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 acts like the platform shown in Figure 8. There is no distinction between components and plug-ins because any component may also act as a plug-in.

The two PMS transitions that add and remove components become slightly refined in the Renew PMS: 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. The removal of components is realized by an `unload` command provided by the PMS.

The Renew PMS is a flat-topped PMS as proposed in Figure 9. All plug-ins are accompanied by a description of their provided services. All components follow the life cycle shown in Figure 10. When a plug-in is loaded, the PMS calls its `init()` method. To unload a plug-in, there is a two-step process: First the PMS queries the plug-in whether it `canClose()`, afterward it may call `cleanup()` and remove it from the system.[8]

Optionally, the PMS may enforce dependencies between plug-ins. If a plug-in is also accompanied by a description of required services, the PMS will not include it in the system unless the required services are available, that is provided by some other plug-ins. 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.

All plug-ins in Figure 11 that are marked with two boxes at their right side, provide extension interfaces that follow the idea from Figure 10. There exist three refinements of the general idea: Either an extending plug-in provides additional implementations of existing interfaces that are seamlessly integrated into a framework, or it extends the basic plug-in by observing and reacting on events, or it just registers its own service in a database where it can be queried by other plug-ins.

[8] The two-step process (ask first, remove afterward) emulates the functionality of the net in Figure 9: There the plug-in also has the possibility to block its removal by not activating its `shutdown` channel uplink.

In the case of the JHotDraw plug-in, other plug-ins can extend the graph editor by registering additional drawing tools, file types and menu commands. The main editor window integrates the registered functionality seamlessly. Due to the elementary functionality of the JHotDraw plug-in, all other plug-ins that provide some editing functionality are extending this plug-in (as can be seen in Figure 11).

The Simulator plug-in can be extended mainly by the notification of observer objects that can react on simulation-related events like transition firings, marking changes, or initialization and termination of the engine.

The extension interfaces of the Formalism and FormalismGui plug-ins have already been sketched above. From the technical viewpoint, these plug-ins provide all three types of extension: There is a registry of known formalisms, a notification about the choice of the default formalism, and an formalism-dependent integration of additional tools and menus into the editor.

Recursive extensibility (as introduced in Section 3.2) is represented in Figure 11 by a chain of extension arrows. An example is the EFGui plug-in which extends the FormalismGui plug-in which in turn extends the JHotDraw plug-in. Since the EFGui plug-in additionally directly extends the JHotDraw plug-in, 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.

However, the `shutdown` of the PMS life cycle needs special care in this system: Due to the tight connections between components and their plug-ins, each

component has to tidy up its references very carefully. Due to the already mentioned restriction of the Java class loader mechanism, we currently cannot truly remove a plug-in from memory when it is unloaded. But in most cases the cleanup process is sufficient to emulate the desired behavior.

4.4 Related Architectures

Plug-ins are more and more used for software architectures. We mention some well known products and their used concepts. The architectures are classified by their authors either as component or as plug-in systems. However, since the distinction between those systems is ambiguous, we list them here without discussion of details.

The model of JavaBeans (see `http://www.sun.com`) defines a mechanism for reusable components of Java application servers. Enterprise JavaBeans extend this again to allow for the design of standardized server components.

The OpenIDE is an integrated development environment which is based on NetBeans (see `http://www.netbeans.org`). The main advantages are the dynamic mechanism for plug-ins and the openness of the architecture. The disadvantages are the specific overhead and the missing visualization of the mechanisms used.

Gimp (Gnu Image Manipulation Program, see `http://www.gimp.org`) is a open source product with a static plug-in concept. The concept is strictly oriented towards its main purpose of image manipulation.

Netscape see (see `http://www.netscape.com`) installs its components on demand. This allows for a lean version. However, the basic mechanism is designed for the presentation of Web pages. It is unclear whether this architecture can be used in general for the design of software architectures.

Poseidon (see `http://www.gentleware.com`) uses the NetBeans mechanism for its plug-in architecture. The migration from a monolithic to a plug-in-based architecture (like the one presented in this paper) was done in [3]. Poseidon demonstrates the potential of NetBeans.

Eclipse is an open source product that mainly provides an architecture for static plug-ins, which are loaded once at startup time. Usually a restart of Eclipse is required when installing new plug-ins. Dynamic plug-ins are not fully realized in the current version of 3.1.

5 Conclusion

Nets-within-nets is an expressive modeling technique that is capable of modeling dynamic system architectures. Models that are built with these nets can profit from their ability to construct arbitrary and dynamic structures. The reference semantics that is applied in reference nets 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 and redesigning a system.

Our generic 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 architecture design. 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 PMS. 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. The IDE has become more flexible and it can now be configured according to the needs of the users—even individually within a multi-user setting. Extending the functionality as a developer has become much easier, due to the fact that extensibility is a first order concept in the system.

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 believe that this approach can be transferred to other application areas and applied as a general concept for various kinds of domains. Moreover, it is possible to generalize the implementation of the concept model as done in Renew to achieve a generalized core application that together with a conceptual approach can form a base for component-based application design of any kind.

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. Such an approach would be very difficult to handle in a monolithic system.

References

1. Lawrence Cabac, Daniel Moldt, and Heiko Rölke. A proposal for structuring Petri net-based agent interaction protocols. In W.M.P. van der Aalst and E. Best, editors, *Lecture Notes in Computer Science: 24th International Conference on Application and Theory of Petri Nets, ICATPN 2003, Netherlands, Eindhoven*, volume 2679, pages 102–120, Berlin Heidelberg: Springer, June 2003.

2. Søren Christensen and Niels Damgaard Hansen. Coloured Petri nets extended with channels for synchronous communication. Technical Report DAIMI PB–390, Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark, April 1992.
3. Clemens Eichler. Entwicklung einer Plug-In-Architektur für dynamische Komponenten. Diplomarbeit, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2002.
4. Thomas Jacob. Implementation einer sicheren und rollenbasierten Workflow-Managementkomponente für ein Petrinetzwerkzeug. Diplomarbeit, University of Hamburg, Department of Computer Science, 2002.
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`, October 2004. Release 2.0.1.
7. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets 2004: 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings*, number 3099 in Lecture Notes in Computer Science, pages 484–493. Springer, 2004.
8. Neno Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1):70–93, January 2000.
9. Object Management Group (OMG). *Unified Modeling Language (UML)*, 2004. `http://www.uml.org`.
10. Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag New York, October 1997.
11. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
12. J. Sametinger. *Software Engineering with Reusable Components*. Springer, Berlin, 1997.
13. Jörn Schumacher. Eine Plug-in-Architektur für Renew: Konzepte, Methoden, Umsetzung. Diplomarbeit, University of Hamburg, Department of Computer Science, October 2003.
14. Clemens Szyperski. *Component software: beyond object-oriented programming*. ACM Press books. Addison-Wesley, 2. edition, 2002.
15. Richard Torkar. Dynamic software architectures. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-based Systems*, chapter 3, pages 21–28. Artech House, 2002.
16. Rüdiger Valk. Petri nets as dynamical objects. In Gul Agha and Fiorella De Cindio, editors, *Workshop Proc. 16th International Conf. on Application and Theory of Petri Nets, Torino, Italy*, June 1995.
17. Frank Wienberg. *Informations- und prozeßorientierte Modellierung verteilter Systeme auf der Basis von Feature-Structure-Netzen*. Dissertation, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2001.