Towards Roundtrip Engineering for Java and Agent Interaction Protocols with Petri net-based Semantics*

Niklas Levens, Lukas Seifert, Leon Zander, Philipp Schult, Sonja Kagan, Ole Methler, Marcel Hansson and Daniel Moldt

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences, Department of Informatics
http://www.paose.de

Abstract

Agent Interaction Protocols (AIPs) model interactions in distributed and agent-based systems. While based on UML-inspired notations, their semantics is formally defined using Petri nets, enabling executable models as well as analysis and verification of distributed control flow.

Although various transformations from interaction models to Petri nets or source code exist, a coherent roundtrip engineering approach that integrates AIPs, their Petri net-based semantics, and Java-based implementations is still missing. In particular, it remains an open challenge to consistently relate and co-evolve interaction models, formal semantic representations, and executable code across multiple representations.

This paper outlines a vision towards integrated roundtrip engineering centered on AIPs as the primary interaction model. Petri nets and Java code are treated as complementary executable representations derived from AIPs for execution, simulation, and analysis. To support this vision, we discuss architectural concepts and existing implementations for forward and reverse transformations, traceability, and comparison-based change analysis.

Based on ongoing projects, we summarize current concepts and implementation results, including reverse engineering of AIPs from Java code, forward generation of Java templates from AIPs, and transformations to Petri net-based representations. These building blocks illustrate how essential parts of a roundtrip process can already be supported in practice.

Although a fully automated roundtrip solution is not yet available, the paper demonstrates that combining AIPs with Petri net-based semantics and Java-based implementations provides a promising foundation for roundtrip engineering.

Keywords

Roundtrip Engineering, Agent Interaction Protocols, Java, Petri Nets, Modeling, Software Engineering, Model Transformation

1. Introduction

While Petri nets provide a precise formal basis for describing and analyzing system behavior, their use in software engineering practice is still limited. In many projects, interaction behavior is primarily realized and evolved in executable code, while models and documentation, if present, are often created only initially and are not systematically maintained. As a result, models tend to become incomplete, outdated, or inconsistent with the actual implementation.

Over time, various approaches have tried to address this problem by deriving code from models or by enriching modeling languages with executable semantics. In the context of UML, several tools support roundtrip engineering between models and code in a more or less systematic way. In the area of open source tools ArgoUML provides some means, however, has not reached a professional level[1]. One of the leading commercial tools is the IBM Engineering Systems Design Rhapsody that was originally the Rational Rose tool[2]. The whole approach in these tools is based on programmed solutions, without the options of verification.

PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026

* Supported by participants of our teaching project classes and many student theses.

[1] https://github.com/argouml-tigris-org/argouml

[2] https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/design-rhapsody

Our hypothesis is that all used modeling and coding artifacts must be compatible. To reach this one idea is to use Petri nets semantics as the basis for the modeling techniques and to use a programming language as the inscription that fits the chosen kind of Petri net. In our case these are reference nets [1].

Our long term research question we address is: How far can reference nets support roundtrip engineering?

As this is very ambitious we address in this contribution the first steps for this goal. This results in the questions: Which models, used in software engineering, can be transformed to Java code? How can these models then be transformed to Java code? Which and how can models be derived from Java code? How can models be verified?

To be clear from this point on, we name already here our severe restrictions of our approach: Currently we concentrate on the control structure of the models. We assume communicating entities (like objects, agents or web services). Only the inter-entities processes are addressed with the verification and is addressed with the roundtrip mapping. All entity internal functionality and state is encapsulated by the entity. In the control flow model the access points can be added and used to cover the abstract but precise representation. By this restriction the models build by modelers are grounded on a Petri nets semantics, so that the intention is to reach a solid roundtrip approach based on Petri nets.

In the context of UML sequence diagrams are used to model the interaction of objects. Within our Paose (Petri Net-based Agent- and Organization-oriented Software Engineering) approach, interaction behavior is captured by an extended version of UML sequence diagrams, named Agent interaction protocol diagrams (Aips) [2, 3]. Aips combine the accessible, UML-inspired notation of sequence charts with a Petri net-based semantics, enabling executable models as well as analysis and verification of distributed control flow.

The core idea is to consistently relate Aips, Petri net-based representations, and Java-based artifacts by focusing on conceptual content and traceable structures rather than layout or formatting. Based on ongoing work in Renew, we illustrate how existing building blocks already support essential aspects of this vision and how they can be combined to enable analysis and feedback across representations.

The remainder of this paper is structured as follows. Section 2 introduces the foundations that provide the conceptual and technical background for this work, including the modeling, semantic, and architectural concepts on which the proposed roundtrip approach builds. Section 3 presents our roundtrip vision and the target architecture context in which it is explored. Section 4 summarizes existing building blocks and prototypes that already support parts of the envisioned process. Section 5 integrates these elements into a coherent roundtrip perspective, discussing behavior analysis, change detection with feedback to Aips, and extensibility to further representations. Finally, Section 6 discusses limitations and open challenges, and Section 7 concludes the paper.

2. Foundations

This section introduces the conceptual and technical foundations that form the basis for the proposed roundtrip engineering approach.

2.1. Workflow nets

Workflow nets are a well-established class of Petri nets for modeling processes and control flow [4, 5]. For our approach, they are the central component to verify behavior. A Petri net is a workflow net if it has exactly one start place i with no incoming arcs and one end place o with no outgoing arcs, and all nodes lie on a path from i to o. By adding a short-circuit transition t^* from o back to i, the resulting net becomes strongly connected.

To assess correctness, workflow nets are commonly required to be *sound*. Soundness means that every reachable state from the initial marking i can still reach proper termination, termination leaves exactly one token in o, and no transition is dead [4]. A workflow net is sound iff its short-circuited form is live and bounded.

2.2. Reference Nets

Reference nets extend colored Petri nets (CPN) [6] and form a central modeling and execution formalism in the Renew environment [1]. In CPN, tokens carry values and transition firing may depend on variables and guards [6]. Beyond CPN, reference nets add synchronous channels and the nets-within-nets concept. For our approach, they are used in displaying the protocol net of an Aip.

A synchronous channel (see [7]) resembles a method invocation based on unification algorithm (see [1]). In Renew transitions can be annotated with an uplink and one or more downlinks. For each uplink (and its parameters) there has to be a matching downlink, which are synchronized under a specific binding. All participating transitions of a binding synchronize and fire jointly.

The nets-within-nets concept (see [8]) allows net instances to be created, referenced, and exchanged during execution. This enables the explicit modeling of dynamic system structures in which the entities interact with each other.

2.3. Renew

Renew (Reference Net Workshop) is a graphical modeling and execution environment for Petri nets and serves as the technical foundation of the Paose approach. It is an extensible, plugin-based system that has been continuously developed as an open-source project since the end of the 90s [9].

While Renew primarily focuses on reference nets, its plugin architecture allows the integration of additional modeling languages and tools, including support for agent-based systems. In this context, the Mulan framework extends Renew with a platform for modeling, simulating, and executing multi-agent systems [10]. Agents are represented by reference nets and interact via protocol nets that specify their communication behavior in concrete interaction scenarios.

2.4. Paose

Petri Net-based Agent- and Organization-oriented Software Engineering (Paose) is an approach to software engineering that puts an emphasis on distribution and concurrency [3, 11]. Within the development context of Renew using the Paose approach, we view every contributing person (e.g. developers, product owners, ...) as an agent that fulfills one or multiple roles.

The specific interactions between these roles are modeled using Aips as well as Petri nets. Especially the development processes are modeled using Workflow nets, ensuring that the processes terminate correctly and that developers have clear guidelines on how to act inside of a given development stage.

Renew is mostly expanded and maintained in teaching projects since its release. During the last year we had many students participating. Therefore we adapted the ideas of *Scrum@Scale* for our Paose approach. The students are distributed into various *Scrum*-teams with different focuses and bachelor/master thesis writers usually function as the *Product Owners*. The general idea for the project organization is a multi-agent system of developers as a guiding metaphor, as described in [12].

2.5. Agent Interaction Protocols (Aips)

Agent interaction protocol diagrams are used not only for documentation purposes but also as the basis for multiple model transformations. From Aips, protocol nets can be generated that describe the role-specific behavior within an interaction, which is performed by a set of agents. Aips can be executed directly as part of the agents of a multi-agent system participating in a specific system process.

Aips are used to model the detailed interaction behavior between roles in multi-agent systems. They describe message exchange, control flow, concurrency, and internal agent actions within interactions [3]. Aips provide, beside the constructs for alternatives, parallel execution, and synchronization [3], some extensions for calling internal states and functions of the agents. Aips are based on UML sequence diagrams and extend them with agent-specific concepts similar to the Agent Unified Modeling Language (AUML). Their semantics is formally defined using Petri nets [2, 13]. This formal foundation enables the transformation of the models into executable representations, supporting simulation, verification,

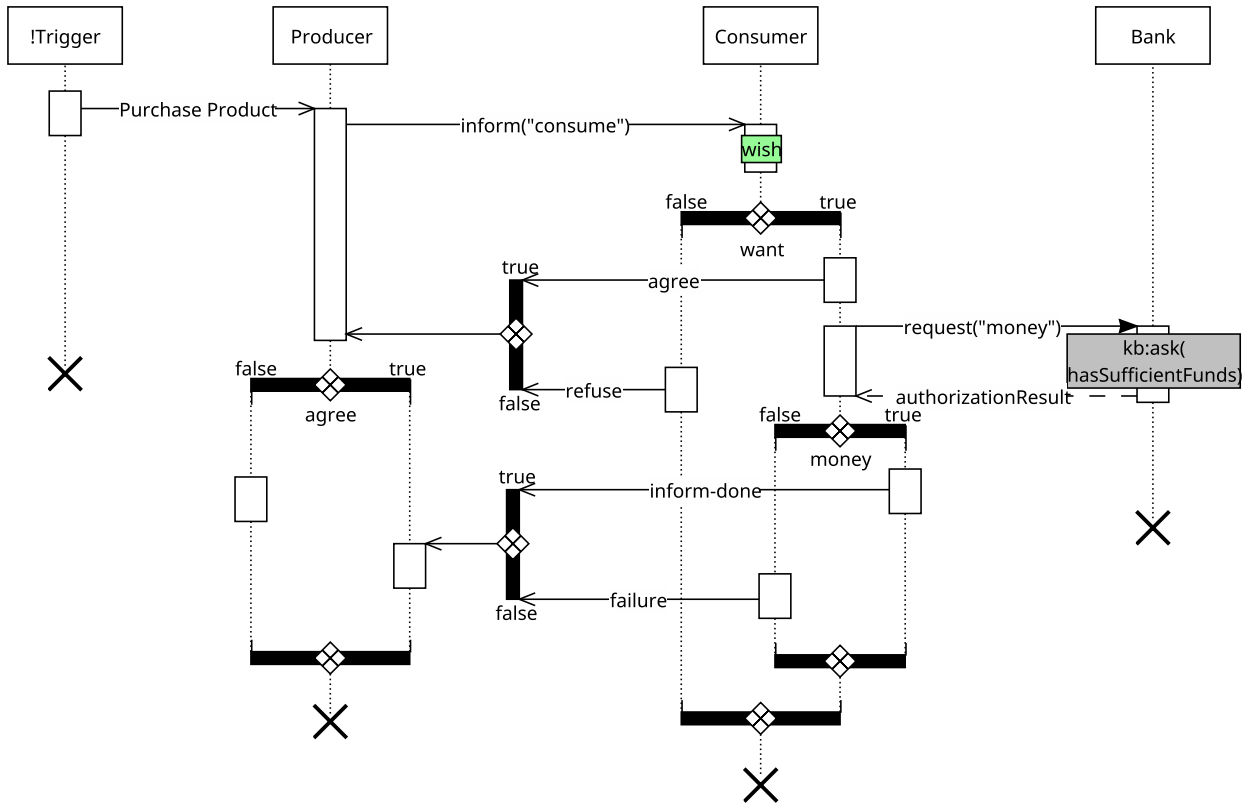


Figure 1: Example AIP (modeled in RENEW).

and analysis of interaction behavior. These representations can be workflownets (see 2.1) or reference nets (see 2.2).

Figure 1 shows an example AIP illustrating AIP-specific constructs such as exclusive splits and joins, knowledge base access, and exchanges with a decision component.

2.6. CDD and Roles Interaction Matrix

In PAOSE, the Coarse Design Diagram (CDD) is one of the first artifacts created in order to roughly model a system and obtain an overview of the overall structure [3].

CDDS utilize the syntax of UML Use Case Diagrams, whereby the meaning of the individual elements has been adapted for the use in a multi-agent context. In contrast to classic UML semantics, use cases in CDDS do not describe the interaction of an external actor with a system, but rather the interactions between the roles of the system itself. Actors represent agent roles, use cases represent interactions, and edges between them represent the participation of a role in the respective interaction [14].

The results of the coarse design can also be represented as a Roles Interaction Matrix. In this matrix, roles are represented as columns, interactions as rows, and the participation of a role in an interaction as a filled cell, usually symbolized by a cross. Both representations describe the same underlying information and can be used simultaneously depending on the desired level of visual or structural abstraction. For each interaction that covers the handling of a trigger event, we provide an AIP.

2.7. Model transformations

Model transformations are a central concept in model-driven software development, where models are treated as primary artifacts for analysis, derivation, and further processing of software systems [15]. In general, model transformations describe the systematic mapping of one model to another while

preserving semantic consistency. Depending on the direction, transformations can be unidirectional or bidirectional and may relate models at different abstraction levels.

In the context of Renew, model transformations have been studied as a means for defining the semantics of modeling languages and for supporting generative tool development. The Renew Meta-Modeling Framework presented in [16] uses transformation-based mappings to Petri net formalisms in order to provide executable and analyzable semantics for high-level models.

Roundtrip engineering extends the notion of model transformation by addressing the co-evolution of multiple related representations. Instead of considering transformations in isolation, roundtrip engineering focuses on maintaining consistency between models and derived artifacts when changes occur in either direction [17]. This is challenging in practice, as transformations are usually not bijective and information may be lost or added during derivation.

A common approach to supporting roundtrip engineering is the use of intermediate representations and explicit traceability relations between corresponding model elements [17, 18]. Changes can be identified by graph-based comparison of intermediate representations, for example on abstract syntax trees that allow structural differences to be mapped back to higher-level models [19].

An early contribution to roundtrip engineering in the context of the Paose approach is presented in [20, 21]. There, roundtrip engineering is understood as the maintenance of consistency between different, partially loosely coupled views of a multi-agent system. Rather than reconstructing models completely, a model-preserving approach is proposed in which changes between overlapping views are either propagated or explicitly treated as inconsistencies. A concrete instantiation of this idea for dependency diagrams and knowledge bases is presented in [22, 23]. However, a concrete, working implementation of change detection between the original Aip diagram tool and generated protocol nets has not been available until the present work.

2.8. Modularity

Most modern software projects rely on the modularization principles described in [24]. Modularization describes the process of splitting a program into smaller distinct parts where each part fulfills a different role. The keywords used in modularization are mainly coupling, cohesion and information hiding.

A modular system offers several advantages when the system is decomposed according to the principle of information hiding. Each module encapsulates a design decision that is likely to change and restricts the assumptions that other modules may make about it. Consequently, modifications to such design decisions can be confined to a single module and do not require changes in other parts of the system. This limits the impact of change, improves maintainability, and makes the system easier to understand, since each module represents a clearly delimited responsibility.

Modular systems can be measured with properties such as *coupling* and *cohesion*. The following definitions are from [25].

Coupling describes the strength of the connection between modules. A modularized system with loose coupling means, that each module has little to no dependencies on other modules, whereas strong coupling means that modules strongly depend on each other. The more a module needs to know about another module, the more they are connected and therefore coupled. For a modular system, loose coupling is preferred, since strongly coupled systems have a high technical debt. Decoupled modules have no connections with each other.

Cohesion is the measure for how strong a module does one task and only one specific task. Modular systems should separate tasks into different modules to avoid a monolithic architecture. With strong cohesion, the purpose of each module is clear and related features are contained in the same module.

Coupling and cohesion are related. A system with higher cohesion will have lower coupling.

Through information hiding, we want to hide our internal implementations of functionality from direct access. Interfaces are defined without giving away the implementation details.

2.9. JPMS

With the introduction of Java 9, a new *module concept* was introduced that enables improved structuring and encapsulation of software. This concept forms the basis for a clear separation between interfaces and implementations and was applied to Renew (see 2.3). The following explanation is based on [26] and [27] as well as the official Java documentation by Oracle ([28]). The target architecture of our approach utilises JPMS.

2.9.1. Basic Idea

Java modules bundle multiple packages into self-contained software components. A module consists of a public interface and a hidden implementation. The goal of modularization is to decompose applications into autonomous, independent units, thereby increasing maintainability and code reuse.

2.9.2. Technical Realization

Each module is described by a `module-info.java` file, which defines dependencies and interfaces. The most important keywords are:

- `requires <module>` – declares dependencies on other modules,
- `exports <package>` – defines which packages are visible to other modules,
- `opens <package>` – enables runtime access via reflection,
- `uses` / `provides` – describes service consumer and service provider relationships.

In addition, modules can be organized into so-called *module layers*, which allow modules to be added or removed dynamically at runtime.

2.9.3. Application of the JPMS to Renew

Since Renew version 4.0 the JPMS is used to improve the Java software basis of Renew. As Renew has a plugin concept for the domain specific separation of its functionality, these plugins were directly supported by the JPMS. We introduced a 1 plugin:1 layer:1 module architecture in version 4.0. The advantages of the language support could therefore be used directly for our architectural design.

As the underlying agent concept of the Paose approach allows a flexible architecture at runtime, several variants of flexibility has be tested for the JPMS. The ideas of control flow between agents, modeled by Aips, and how this is used for better flexibility, is explained in the next sections.

3. Vision: Roundtrip Engineering for Aips

The goal of this work is to establish a shared vision for roundtrip engineering on the level of Aips and Java-based implementations, grounded in a Petri net-based semantics. This vision is motivated by three observations:
(i) verification techniques for Petri nets are well established,
(ii) the need for verified software is steadily increasing, and
(iii) UML sequence diagrams and Java are widely used in practice.

At the same time, using Petri nets directly as the only modeling technique is often perceived as impractical for software engineering. We address this by embedding Petri net semantics into UML-inspired notations and agent-oriented modeling styles (see the Paose approach). This allows us to work with accessible views on system behavior while retaining an executable and analyzable formal foundation in Renew [11].

3.1. From Isolated Transformations to Roundtrip Support

Although many transformations from one technique to another exist, they are typically realized as isolated forward or reverse steps. As a result, consistent roundtrip support across representations is still missing. This challenge has been discussed in our work for many years, but only recently practical progress became possible due to ongoing refactorings and modularization efforts in the Renew Java code base. In particular, the clearer module concept of modern Java supports more systematic mappings between Aips, Petri nets, and Java artifacts, and enables first successful experiments towards integrated roundtrip support.

In our vision, Aips serve as the central interaction model that captures the essential structure and control flow of distributed behavior. Petri nets and Java code represent executable realizations derived from this model. More generally, we assume a central, semantically grounded control-flow model and several concrete representations produced by different techniques. Each representation provides its own perspective and additional information, while sharing control-flow concepts as overlapping content (similar to the separation of concerns known from UML).

Roundtrip engineering is understood as the consistent co-evolution of interaction models, their formal semantics, and executable software artifacts. Instead of treating modeling, analysis, and implementation as separate activities, the envisioned approach aims at systematically relating these representations and supporting changes across them.

3.2. Aips to Model the Control Flow of Systems

In the proposed vision, Aips organize the central control flow model around role interactions. Conceptual content is separated from concrete representation details: semantically relevant information is kept stable, while layout and textual formatting are treated as secondary and excluded from synchronization.

Aips describe message exchange, control flow, and synchronization between roles, while abstracting from implementation-specific realization choices. They model instances that communicate via messages (or method calls in the case of programming languages) and thereby represent domain-specific roles that encapsulate data, functionality, and internal behavior. In Paose, the involved roles and interactions can be identified at the coarse design level using a matrix or a Cdd, and are then refined by Aips [3, 11].

Other representations enrich the central control flow model with additional information required for specific purposes. Graphical notations emphasize usability and comprehension, Petri net-based models enable formal analysis and verification, and Java-based artifacts provide executable implementations. Although these representations differ in structure and level of detail, they refer to the same interaction concepts captured by Aips.

In our setting, this is supported by stable intermediate representations and explicit identifiers for traceability, while diagram layout and code formatting are excluded from the synchronization scope. This allows roundtrip engineering to focus on semantic evolution rather than presentation details.

3.3. Petri net-based Semantics

The semantics of Aips is formally defined using Petri nets [3], providing a precise and executable representation of interaction behavior. This enables analysis and verification of distributed control flow, including properties such as soundness, deadlock freedom, and correct synchronization.

Within the envisioned roundtrip process, Petri nets and Java-based implementations are treated as complementary executable artifacts. Petri nets support simulation and formal reasoning, while Java code supports integration into concrete runtime architectures. Keeping both consistent with the underlying interaction models is essential for sustainable evolution of complex systems.

Petri nets do not replace Aips as the central model but serve as their semantic realization. They provide a structured basis for comparing different realizations of interaction behavior and for deriving feedback that can be reflected back into the interaction models. This layer therefore plays a key role in detecting and interpreting changes across representations.

3.4. Roundtrip Process Across Representations

Roundtrip engineering is realized as an iterative process. AIPs may be derived from existing implementations or created during design and are used to generate Petri net-based models and Java-based artifacts. Both analysis results and implementation changes may trigger adaptations of the interaction models.

Rather than aiming at fully automated reconstruction of models, the roundtrip process relies on explicit relationships between representations and comparison-based techniques to identify relevant differences. These differences are interpreted with respect to the AIPs and can often be resolved automatically, but in complex modifications manual action is required.

Changes to AIPs can be directly propagated forward by regenerating the corresponding Petri nets. These nets have successfully been used in the context of the settler game of Catan, which we build based on our PAOSE approach. For a detailed description see e.g. [3, p. 373–392]). Generated nets have Java as inscriptions (see also Figure 2 in this paper) and are just skeletons. They need to be extended by reference nets and Java inscriptions. Propagating changes made in generated artifacts back to the AIP is the harder problem. Main cause of this is that transformations are not bijective in general. Not all changes in derived representations may have a direct counterpart in the AIP.

What is important here is that the respective changes are made in one model. The other models, also related to the central model, need to be adapted in the case that the change is relevant for this perspective.

The target architecture described next provides a concrete context in which this vision can be explored. It reflects the current plugin architecture of RENEW and serves as a reference structure for integrating roundtrip support.

3.5. Target Architecture

There have been many different ideas on how to structure a modular system to use agents. One agent oriented software development approach has been proposed by Rana Mostafa in [29]. She references the Java-Agent-Architecture (JAA) which was first described by Laif-Oke Clasen in [30].

The proposed Java-Agent-Architecture is based on multi-agent systems with the PAOSE approach (see 2.4) and its MULAN agents. Each plugin can then be interpreted as a unique role. A multi-agent system needs an agent platform for agents to interact on. The agent platform is the bootlayer of the system.

The bootlayer has a few core components: [29]

- A factory to create agents on demand
- Communication channels for agents to interact with each other
- A module to define global types and data structures, similar to the ontology of an agent

By utilizing this bootlayer, we don't have to define concrete module dependencies, since these are solved dynamically during runtime. Because of this we avoid problems such as cycles.

Each plugin then consists of 4 core modules.

- An API to handle communication with the communication channels of the bootlayer.
- A control module which defines all possible actions an agent can perform.
- An internal interface module which acts as an interface between the control module and the implementation. With this module it is possible to load and unload different implementations during runtime.
- The implementation module which defines how features are implemented. The features are then provided as services with the *provides* clause in the module info file.

The implementation module provides services to the internal interface module using JPMS service features. The internal interface module then needs a component to connect service providers with

service consumers. In Renew we have the ServiceLookupInfrastructure to find and connect services. This ServiceLookupInfrastructure is described in [31].

The proposed architecture works well in a completely new system. For existing systems however it is not trivial on how to implement this architecture without disrupting current functionality. Because of this we propose a slightly altered approach.

The module structure for plugins stays the same, but no agent specific bootlayer is introduced. Instead, plugins are called by its interface module which then calls a process of its control module. With this approach we still have the benefits of agent oriented architecture, but we can still rely on older systems which haven't been updated yet. However, we lose the ability to create and destroy agents on demand and agent autonomy.

4. Existing Building Blocks

Roundtrip engineering for Aips and Java-based implementations remains challenging, especially for existing code bases without explicit interaction specifications. We address this with complementary building blocks as described in the following sections.

4.1. Reverse Engineering: From Java Source Code to Aips

To enable a roundtrip approach including Java, there must be some way to map Java code onto our Aips. Within the Paose approach, the Cdd offers an abstract overview of larger parts of the system based refering to the Aips.

Our current approach is to use static code analysis of the Renew source code to extract interactions. The main advantage with this is that we can extract the control flow more easily, as compared to dynamic analysis.

The static analysis starts by parsing the source files into an abstract syntax tree (AST). This is a data structure commonly used when working with source code, which allows working with the syntactic elements while ignoring irrelevant textual layout properties.

The goal is to obtain Aips with a high precision, meaning they accurately describe the control flow of existing source code. The reason is that we intend to use the resulting Aips for code generation of the control flow between modeled entities. The more detail we keep, the fewer we need to re-add later.

Based on the semantic elements in the abstract syntax tree, we can translate elements into Aip counterparts. A simple example is translating an `if` statement into a corresponding lifeline split element. Method calls are generally translated into synchronous messages and replies. Roles are based on class-level at first.

The generated Aip should have exactly one lifeline termination for each role. This is meant to ease the verification process later on. Non-block structured code is a challenge in this regard. Early return or throw statements are an example which can obstruct the regular control flow.

These translations can be guided by the Petri net semantic of the Aip. Information that might not have a corresponding translation defined yet is displayed as an action inscription. With this approach, we can incrementally extract and represent more information into the Aip representation.

The Aip elements reverse engineered from source code initially do not have any layout information that is critical for understanding. Based on the source code however, one can use the call stack as a guide on where to place the visual elements.

We use the notion of a coarsening operation on the AST to elevate the interactions from class to module-level. If we are only interested in gathering information on the interactions with respect to the Cdd, we can coarsen the AST significantly. By default, the AST provides us with class-level information and also each and every statement in all methods. Thus, the roles generated are related to classes. To coarsen the interaction to module-level the core idea is to only keep method calls that cross the plugin boundary. Every other method is logically inlined. Then we can also suppress any statement that is not related to inter module communication. Overall, this way we can technically reverse engineer how the modules communicate to accomplish the interactions.

4.2. Forward Engineering: From AIPs to Java Code

Within the development context of RENEW, we aim to document every software interaction with AIPs. These models serve both as an entry point for understanding the technical workings of RENEW, as well as the foundation of roundtrip-based refactorings.

Ideally, AIPs are created prior to implementing new interactions. In practice, however, this is not always the case, and even if such models are created, it is not guaranteed that the AIP correctly reflects the implementation. Moreover, maintaining consistency between AIPs and evolving code is challenging, and updates are frequently neglected, leading to outdated or incorrect models.

To address these issues, we introduce Code Generation from AIPs. This approach parses the contents of an AIP and produces Java code templates that reflect the structure specified in the model. By employing dedicated annotation conventions within the AIP, the generated templates integrate seamlessly into the RENEW codebase.

By modeling interactions in AIPs prior to implementation and deriving code templates from these models, we ensure that the resulting software conforms to the intended structural design.

When combined with the AIP generation from code described in 4.1, this approach enables structural roundtrip refactorings. Specifically, an existing implementation can be transformed into an AIP, modified at the model level, and subsequently regenerated as refactored code.

Although the current Code Generation operates directly on AIPs, it can be readily adapted to use the intermediate representation introduced in 2.7, further supporting roundtrip engineering workflows.

Even if the general interaction between different components should remain the same, the Code Generation is engineered in a way that should allow for internal refactorings of individual components. It can adapt to different architectures that are used within RENEW. This could allow us to use our roundtrip engineering approach to adapt an existing software component to a new architecture.

4.3. From AIPs to Petri net-based Representations

AIPs can be transformed into Petri net-based representations within RENEW, thereby providing executable and analyzable models of interaction behavior. Originally, this transformation focused on the generation of protocol nets for MULAN agents [3], which describe the role-specific behavior of agents during an interaction. More recent work extends this approach to additional Petri net-based representations, in particular workflow nets, enabling the use of AIPs for different analysis and execution purposes [32].

All transformations follow a pattern-based approach. Elements of an AIP are mapped to predefined Petri net components that capture their semantics and are composed into larger nets according to the structure of the interaction. Roles are processed along their lifelines, and messages, control-flow constructs, and synchronization elements are translated into corresponding Petri net fragments.

For protocol nets, a separate Petri net is generated for each role involved in the interaction. These nets are realized as reference nets and encode the detailed communication and control behavior of agents. They form the executable interaction layer within MULAN-based multi-agent systems and can be manually refined where interaction aspects cannot be fully captured at the modeling level.

In contrast, workflow nets provide an abstracted view on interaction behavior. From an AIP, a single workflow net with a dedicated start and end node is generated, focusing exclusively on the control flow of the interaction. Internal agent behavior and execution details are deliberately omitted, allowing workflow nets to be used for verification purposes such as soundness checks or deadlock analysis.

Besides protocol and workflow nets, further Petri net-based representations are possible. For example, interface nets provide modular views on interaction behavior and have been used in the context of compositional agent architectures [33]. They illustrate the extensibility of the transformation approach and the role of AIPs as a common source model for multiple Petri net-based views.

4.4. Consistency between Interaction Views

The CDD and the Roles Interaction Matrix provide two alternative views of the same underlying information about roles and interactions. Since the coarse design is continuously modified during the development process in an incremental approach, for example through refinements, deletions, or restructuring, it is sensible to keep the two views consistent and to aim for a roundtrip between them. The basis for this is a central model that contains only the overlapping information from the two views. This includes the roles, interactions, their respective names, and their associations. View-specific information is deliberately excluded from the central model, as it is not relevant to the other view. This refers to the graphical information in the CDD and the order of roles and interactions in the matrix.

When changes are made in one view, the central model is updated and serves as an intermediary to inform the other view about relevant changes through an observer technique. Although the structural information about the change is available in the central model, view-specific decisions cannot be derived from it. For instance, if a new role is added in the CDD, the Roles Interaction Matrix has no information about the intended position of this role, since no mapping between the views exists for such layout decisions. As a result, a complete reconstruction of the target view is not possible and instead only the specific changes are transferred, while information existing only in the target model remains untouched [21]. To address this standard strategies can be used, such as automatically inserting new roles at a predefined position. This shows that there is no bijective mapping between a CDD and the Roles Interaction matrix, but only a partial, information preserving synchronization of the common elements is possible.

However, the relevance of the coarse design in the context of roundtrip is not limited to the interaction between the two views, but also extends to the subsequent refinement of interactions. The interactions identified in the CDD can be further specified using AIPS, in which the roles involved in the interaction also represent the agent roles within the AIP [3].

4.5. Substitution between Petri Nets and Java

RENEW's ability to run Petri nets is based on Java and there is little difference between any regular Java object and the NetInstance object which represents a Petri net. In the context of roundtrip, this could allow a developer to substitute a Petri net for a Java class with the same functionality or vice versa. This could be used to build an application from the AIPS described in earlier sections. A developer could model an interaction in the application using an AIP, generate protocol nets which can already run the interaction and then write Java classes based on the generated nets that implement the interaction one by one.

In practice however, the use of more than one synchronous channel attached to a transition prevents this substitution to be a simple exchange of class names. Communication between Petri nets in applications takes place over synchronous channels instead of method calls, switching between Petri nets and Java classes requires the use of so-called stubs (see the handbook of RENEW, Section 3.9). Stubs are Java classes which contain a NetInstance object and take method calls and refer them to this NetInstance by addressing the synchronous channels within the net.

A minimal approach to substitute a Petri net with a Java-class involves creating a Java interface, which contains the uplinks present in the Petri net which is to be substituted. Then, a stub which implements the interface is created for the Petri net. Finally, the downlinks to the net are replaced with references to methods of the stub-class. The creation of the stub is not technically necessary, but it makes the substitution of the Petri net easily reversible and generates a class which can later be copied and edited. A new Java class which implements the interface can then be created and the functionality of the net reproduced. As methods are implemented, the references to the stub-class can be replaced by those to the new class to test immediately. For the reverse, the interface implements the public methods of a Java class. A Petri net is then created which should minimally contain one uplink per method in the interface. A stub is again created for the new Petri net implementing the interface, and the references to the initial class are replaced with references to the stub.

5. Towards Integrated Roundtrip Support

The previously introduced building blocks address individual aspects of roundtrip engineering, such as model transformations and code generation. This section integrates these elements into a coherent roundtrip perspective and outlines how the artifacts are related throughout the development process. Rather than proposing a fully automated solution, the focus lies on establishing traceable connections and systematic feedback mechanisms that support controlled evolution across representations.

In particular, we discuss Petri net-based behavior analysis, protocol net change detection with feedback to Aips, and how the same principles can be transferred to workflow nets and code-level AST representations.

5.1. Overview of the Integrated Roundtrip Process

Our integrated roundtrip process is centered on Aips as the conceptual source model for interaction behavior. From Aips, Petri nets and Java artifacts can be derived via forward transformations: Petri nets enable execution and analysis based on the formal semantics, while Java code integrates the interaction into the target software architecture. Reverse transformations complement this by recovering Aips from existing implementations, which supports roundtrip scenarios for legacy systems.

Consistency across representations is maintained via explicit traceability relations that enable comparison-based change detection and feedback.

To illustrate the integrated roundtrip process, we use a small running example based on the Aip from section 2.5. Table 1 shows a Role-Interaction matrix with three roles: a *Producer*, a *Consumer*, and a *Bank*. The matrix captures several interactions between these roles, reflecting that overall system behavior typically consists of multiple interactions.

One of these interactions is *Purchase Product*, which is refined by an Aip (see Figure 1). In this interaction, the *Producer* informs the *Consumer* about an available product. The *Consumer* then decides whether to buy the product and, if so, requests payment authorization from the *Bank*. Depending on the response of the *Bank*, the purchase is either completed or aborted.

The Aip can be transformed into protocol or workflow nets as well as into Java code. Changes in generated nets or code can be related back to the Aip of this interaction, while other interactions in the Cdd remain unaffected. To illustrate this refinement chain, Figure 2 shows the protocol net generated for the *Producer* role, representing the role-specific executable view of its participation in the interaction.

5.2. Behavior Analysis Based on Petri Net Semantics

Because we implement the processes generated from code into the codebase again, it is imperative, that the generated structures have a sound workflow. To do this, we can convert Aips into workflow nets. These nets can then be analyzed manually or by using automated tools to check for boundedness and liveness (of the short circuited net, see 2.1) to check soundness properties. Only if the workflow net is sound we can be sure that the program will still work correctly without causing deadlocks.

The tools we use to check workflow nets, which are integrated into Renew, are Lola and Averto. Lola displays a GUI which tells the user the properties of the net, including boundedness and liveness.

If a workflow net is not sound, the underlying Aip has to be modified to generate a sound workflow net. This process guarantees, that the resulting program doesn't deadlock.

Table 1
Role-Interaction matrix

Interaktion	Producer	Consumer	Bank
Offer Product	X	X	
Purchase Product	X	X	X
Cancel Purchase	X	X	
Authorize Payment		X	X

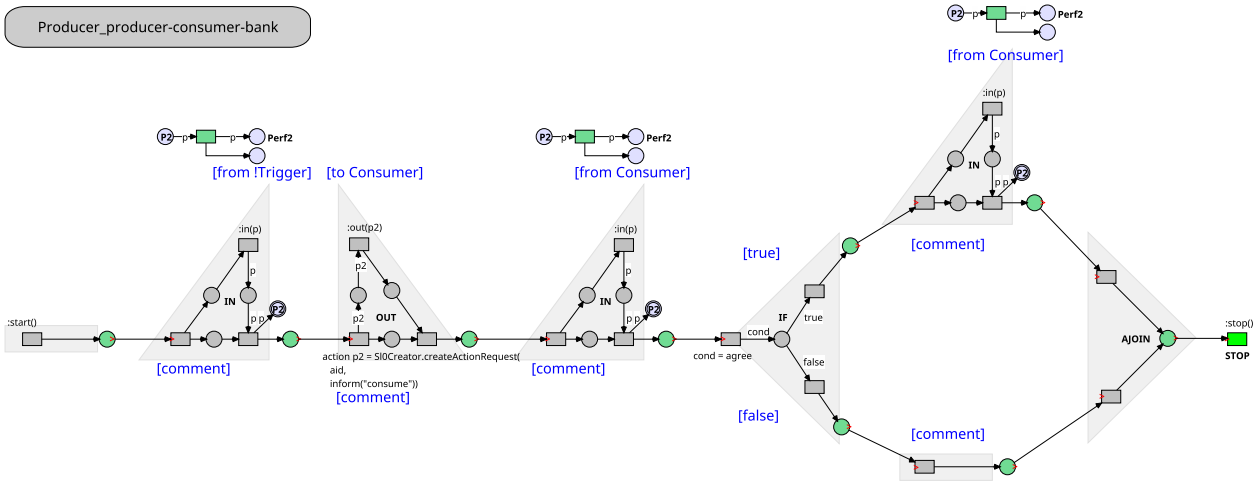


Figure 2: Role-specific protocol net for the *Producer* role (generated in Renew).

There are still a few minor problems with this approach:

- **Aips → Petri nets:** Not all Aip structures can be converted into a workflow net yet. In Renew we have two defined loops, *iterator* and *for all*, which can't yet be translated into their respective Petri net semantics. Currently, we work on full support for automated workflow net translation.
- **Java → Aips:** There are code structures which are inherently incompatible with Aips. Try-Catch-blocks have no defined way to display them. Exceptions cause a break in the lifeline of an Aip and therefore break the defined conventions of one defined end per role. Recursion needs the ability to create new roles, or to save decisions for splits, which would need to be defined.

5.3. Change Detection and Feedback

A central prerequisite for roundtrip engineering is the ability to relate changes in derived artifacts back to the originating interaction model. In our current implementation, change detection is realized between Aips and generated protocol nets. After generation, protocol nets are often adapted manually in Renew, which makes a simple re-generation unsuitable for synchronizing the models.

To enable comparison independent of layout and drawing details, protocol nets are transformed into a stable graph representation in which each net component corresponds to a node and control-flow connections are represented as directed edges. The baseline graph is reconstructed from the current Aip via generation, while the modified net is decomposed from the open drawing into a corresponding graph structure. Both graphs are persisted and compared using a structure-based matching strategy [34].

Detected differences are classified (e.g., inserted, deleted, or rewired components) and mapped back to the Aip using traceability information recorded during generation. This allows the tool to highlight affected Aip elements and to provide user-oriented feedback. At this stage, the feedback supports controlled manual updates of the Aip model; automatic propagation of changes is left open for now.

The change detection is designed to work with nets generated from the Aip, which carry named component annotations, and with manual edits that are performed using the provided component palette. This ensures that the graph representation captures all relevant structural changes. Manual additions required for the execution in Mulan cannot be unambiguously related back to Aip elements and therefore fall outside the synchronization scope, as do content-level modifications such as changes to message inscriptions.

5.4. Extensibility to Further Representations

The change detection mechanism described above is currently instantiated for the roundtrip between Aips and protocol nets, but the underlying principle is transferable. The key idea is to compare two

versions of a representation on a structural level that abstracts from irrelevant layout or textual formatting, while maintaining traceable identifiers that support mapping differences back to AIPs elements. A direct extension target are workflow nets generated from AIPs. Since workflow nets are generated using predefined Petri net components as well, the same notion of a graph-based intermediate representation and comparison-based change detection can be applied, enabling analysis guided adjustments of interaction behavior and corresponding feedback at the AIP level.

A similar approach is conceivable for Java-based artifacts. The reverse engineering (see section 4.1) already relies on abstract syntax trees (ASTs), which naturally provide a graph-like representation of code structure. To support feedback in the roundtrip process, traceability would need to be established across generation and extraction, for example by relating element identifiers in the AIP intermediate structure to AST nodes. This would allow structural differences at AST level to be interpreted and reported with respect to the originating AIP concepts, even if full automation is not feasible.

6. Discussion

This paper presents a vision for roundtrip engineering centered on AIPs and substantiates it through a set of integrated building blocks. Rather than proposing a fully automated solution, the contribution lies in aligning existing concepts, tools, and prototypes within a shared conceptual framework.

A key design decision is the role of AIPs as the central interaction model. Executable artifacts such as Petri nets and Java code are treated as complementary realizations, each serving specific purposes such as analysis or execution.

The presented building blocks illustrate that roundtrip engineering is an incremental process. Transformations are typically not bijective, and full reconstruction of models is not realistic. Instead, comparison-based techniques, intermediate representations, and user-guided feedback provide a practical way to maintain consistency. This is reflected in the Petri net-based behavior analysis and the change detection between AIPs and protocol nets.

Existing industrial tools already provide limited forms of roundtrip support between interaction models and code. For example, tools such as IBM Rhapsody, Enterprise Architect, MagicDraw/Cameo, and Visual Paradigm support partial roundtrip scenarios between UML sequence diagrams and Java. Typically, this includes forward transformations from sequence diagrams to Java code skeletons and reverse reconstruction of sequence diagrams from execution traces.

However, these approaches are constrained by the nature of sequence diagrams. They lack a full semantics, which limits their suitability as a stable basis for consistency management. In particular, the expressive power of Java allows complex concurrency and control flows, which cannot be captured by sequence diagram fragments. As a result, reverse engineering from code can only reconstruct incomplete or approximated interaction models.

The approach presented in this paper differs in that AIPs allow additional control flow elements and internal action. However, like in SysML (see https://www.omg.org/spec/SysML) we restrict the allowed elements in all modeling techniques and have strong Java coding conventions.

At the same time, limitations remain in the current realization. Not all programming constructs can yet be faithfully reflected at the interaction level, and change detection is currently restricted to selected representations. Extending these mechanisms, for example to workflow nets or to Java code via AST-based comparisons, requires further conceptual and implementation work.

The automatic extraction of AIPs is limited by the dynamic nature of Java due to its polymorphism and runtime types. With only static analysis, we can only make use of static types available in the source code. During runtime, the dynamic type might differ thus potentially leading to a different control flow. For example, we might encounter a method call of `foo()` on a variable of static type `A` in the source code but during runtime, this object has a dynamic type of `B` leading to a call of `B#foo()` instead of `A#foo()`. This problem generally requires class hierarchy analysis to identify all possible methods that could be called, which overapproximates the actual possible control flow based on the assumption that all possible types are known for analysis.

A dynamic analysis could resolve this ambiguity and lead to a hybrid approach. Instrumentation of the source code could be used to determine which methods are actually called when program is executed. However, this comes with a huge performance penalty.

Another limitation to keep in mind is that the automatic extraction is not able to differentiate between implementation details and relevant steps on the domain level. So far, our approach is to extract as many details as possible in our ongoing approach. With coarsening, we attempt to abstract from class to higher levels.

Overall, the presented results indicate that the combination of AIPs, Petri net-based semantics, and Java-based implementations provides a promising foundation for roundtrip engineering. The work highlights the feasibility of core concepts, while also outlining aspects that require refinement and further investigation in future work.

7. Conclusion

7.1. Summary

This paper presented a vision towards roundtrip engineering centered on AIPs as a central interaction model with a formally defined Petri net-based semantics. Rather than proposing a single, fully automated solution, we structured the problem along concrete representations and transformations and discussed how existing concepts and tools can be combined to support roundtrip scenarios in practice.

We showed how AIPs can serve as a stable conceptual core from which executable artifacts such as Petri nets and Java-based implementations are derived. Existing building blocks already support essential parts of this vision, including reverse engineering of interactions from code, forward generation of plugin interfaces, Petri net-based behavior analysis, and comparison-based feedback from modified protocol nets back to AIPs. In addition, we demonstrated that similar roundtrip principles can be applied at earlier design stages by maintaining consistency between CDDs and Roles Interaction Matrices, using a shared central model and partial, information-preserving synchronization. Together, these elements illustrate how consistency between interaction models, their semantic realizations, and executable artifacts can be supported in a controlled and traceable manner.

Additional aspects of the different models (carrying semantical parts of the system) are kept separately in their own model. Syntactic parts of all models are kept aside in further models as they are relevant for the usability of the models of the different perspectives. This is similar to those approaches from industry (see e.g. IBM Rhapsody), however, we have a Petri nets semantics behind all relevant control flow parts between the modeled entities of the system. Due to the later we assume that we can integrate the presented transformations much stronger than it is the case today. A central precondition is that we restrict the allowed syntactic elements in all models in such a way that we can provide a proper Petri net-based semantics for all resulting models.

7.2. Outlook

While the presented results demonstrate the viability of key roundtrip concepts, several aspects remain open. In particular, further work is required to generalize change detection and feedback mechanisms beyond protocol nets, for example to workflow nets and Java-based artifacts. Although similar principles can be applied, differences in abstraction level and representation structure require careful adaptation and additional tooling support.

Beyond technical extensions, future work should further investigate how the proposed roundtrip approach integrates into larger development processes. This includes questions such as the role of manual intervention, the handling of inconsistencies, and the evolution of architectures over time. Overall, the presented vision and building blocks provide a solid foundation for continued research on roundtrip engineering based on AIPs, Petri net semantics, and executable software artifacts.

Declaration on Generative AI

During the preparation of this work, the authors used Claude, Quillbot in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] O. Kummer, Referenznetze, Logos Verlag, Berlin, 2002. URL: http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=.

[2] L. Cabac, D. Moldt, H. Rölke, A proposal for structuring Petri net-based agent interaction protocols, in: W. van der Aalst, E. Best (Eds.), 24th International Conference on Application and Theory of Petri Nets, Eindhoven, Netherlands, June 2003, volume 2679 of *Lecture Notes in Computer Science*, Springer-Verlag, 2003, pp. 102–120. URL: http://dx.doi.org/10.1007/3-540-44919-1_10.

[3] L. Cabac, Modeling Petri Net-Based Multi-Agent Applications, Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2010. URL: https://ediss.sub.uni-hamburg.de/handle/ediss/3691.

[4] W. v. d. Aalst, Verification of workflow nets, in: P. Azéma, G. Balbo (Eds.), Application and Theory of Petri Nets 1997, number 1248 in Lecture Notes in Computer Science, Springer Verlag, Berlin Heidelberg New York, 1997, pp. 407–426.

[5] H. M. W. Verbeek, T. Basten, W. van der Aalst, Diagnosing workflow processes using woflan, Comput. J. 44 (2001) 246–279.

[6] K. Jensen, L. M. Kristensen, Coloured Petri Nets - Modelling and Validation of Concurrent Systems, Springer, 2009. URL: https://doi.org/10.1007/b95112. doi:10.1007/B95112.

[7] S. Christensen, N. D. Hansen, Coloured Petri Nets Extended with Channels for Synchronous Communication, Technical Report DAIMI PB-390, Aarhus University, 1992. URL: https://tidsskrift.dk/daimipb/article/view/6625.

[8] R. Valk, Petri nets as token objects - an introduction to elementary object nets, in: J. Desel, M. Silva (Eds.), 19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal, number 1420 in Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York, 1998, pp. 1–25. URL: https://doi.org/10.1007/3-540-69108-1_1.

[9] O. Kummer, F. Wienberg, Renew – the Reference Net Workshop, Available at: http://www.renew.de/, 1999. URL: http://www.renew.de/, release 1.0.

[10] L. Cabac, T. Dörges, M. Duvigneau, D. Moldt, C. Reese, M. Wester-Ebbinghaus, Agent models for concurrent software systems, in: R. Bergmann, G. Lindemann (Eds.), Proceedings of the Sixth German Conference on Multiagent System Technologies, MATES'08, volume 5244 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin Heidelberg New York, 2008, pp. 37–48. URL: http://dx.doi.org/10.1007/978-3-540-87805-6_5.

[11] L. Cabac, M. Haustermann, D. Mosteller, Software development with Petri nets and agents: Approach, frameworks and tool set, Sci. Comput. Program. 157 (2018) 56–70. URL: https://doi.org/10.1016/j.scico.2017.12.003.

[12] L. Cabac, Multi-agent system: A guiding metaphor for the organization of software development projects, in: P. Petta (Ed.), Proceedings of the Fifth German Conference on Multiagent System Technologies, volume 4687 of *Lecture Notes in Computer Science*, Springer-Verlag, Leipzig, Germany, 2007, pp. 1–12. URL: https://doi.org/10.1007/978-3-540-74949-3_1.

[13] L. Cabac, D. Moldt, Formal semantics for AUML agent interaction protocol diagrams, in: Agent-Oriented Software Engineering V: 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004. Revised Selected Papers, volume 3382 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005, pp. 47–61. URL: http://dx.doi.org/10.1007/978-3-540-30578-1_4.

[14] J. Sieranski, Diskussion und Weiterentwicklung der Rollen-Interaktionsmatrix zur Unterstützung

des Projektmanagements in der PAOSE, Master thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2016.
[15] T. Stahl, M. Völter, K. Czarnecki, Model-driven software development: technology, engineering, management, John Wiley & Sons, Inc., 2006.
[16] D. Mosteller, L. Cabac, M. Haustermann, Providing Petri net-based semantics in model driven-development for the Renew meta-modeling framework, in: D. Moldt, H. Rölke, H. Störrle (Eds.), Petri Nets and Software Engineering. International Workshop, PNSE'15, Brussels, Belgium, June 22-23, 2015. Proceedings, volume 1372 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015, pp. 99–114. URL: http://CEUR-WS.org/Vol-1372.
[17] T. Hettel, M. Lawley, K. Raymond, Model synchronisation: Definitions for round-trip engineering, in: International Conference on Theory and Practice of Model Transformations, Springer, 2008, pp. 31–45.
[18] R. F. Paige, Engineering bidirectional transformations, in: Bidirectional Transformations: International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures, Springer, 2018, pp. 151–187.
[19] L. Angyal, L. Lengyel, H. Charaf, A synchronizing technique for syntactic model-code round-trip engineering, in: 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008), IEEE, 2008, pp. 463–472.
[20] R. Dirkner, A. Lehning, Grundlagen des Round-trip Engineerings in der agentenorientierten Softwareentwicklung am Beispiel von Interaktionsdiagrammen und Mulanprotokollen, Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2005.
[21] R. Dirkner, Roundtrip-Engineering im PAOSE-Ansatz, Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2006.
[22] L. Cabac, R. Dirkner, H. Rölke, Modelling service dependencies for the analysis and design of multi-agent applications, in: D. Moldt (Ed.), Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents. MOCA'06, number FBI-HH-B-272/06 in Report of the Department of Informatics, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, Germany, 2006, pp. 291–298.
[23] L. Cabac, R. Dirkner, D. Moldt, Modeling with service dependency diagrams, in: D. Moldt, U. Ultes-Nitsche, J. C. Augusto (Eds.), Proceedings of the 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS-2008, In conjunction with ICEIS 2008, Barcelona, Spain, June 2008, INSTICC PRESS, Portugal, 2008, pp. 109–118.
[24] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (1972) 1053–1058.
[25] E. Yourdon, L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Englewood Cliffs, NJ, 1979.
[26] A. Daschkewitsch, Modularisierung des Renew-Plugin Systems, Master thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2019.
[27] M. Hansson, Konsolidierung workflowbezogener Renew-Plugins, Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2020.
[28] Oracle, Java Platform, Standard Edition 17 API Specification, Oracle, 2021. URL: https://docs.oracle.com/en/java/javase/17/docs/api/, accessed: 2026-01-24.
[29] R. Mostafa, Analyse und entwurf einer mustersprache eines agentenorientierten architekturvorschlags im kontext der open-source-software renew durch prototypische umsetzung eines producer-storage-consumer-beispiels auf basis des jpms, 2026.
[30] L. Clasen, Untersuchung von Softwarearchitektur und Projektorganisation unter Berücksichtigung der Wechselwirkungen im Kontext von Lehrprojekten am Beispiel verteilter Entwicklung der Open-Source Software Renew, Master thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2023.
[31] K. Ehlers, Untersuchung des JPMS für Java Programmsysteme zur Dekomposition mittels

Modulschnittstellen für einzelne Plugins der Open-Source Software Renew, Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2023.

[32] L. Seifert, K. Ihlenfeldt, D. Moldt, L. Clasen, M. Hansson, Modellierung von komplexen abläufen im paose kontext mit dem diagram tool, in: H. Giese, K. Rosenthal (Eds.), Modellierung 2024 - Workshop Proceedings, Potsdam, Germany, March 12-15, 2024, Gesellschaft für Informatik e.V., 2024, p. 19. URL: https://doi.org/10.18420/modellierung2024-ws-019. doi:10.18420/MODELLIERUNG2024-WS-019.

[33] D. Moldt, M. Hansson, L. Seifert, K. Ihlenfeldt, L. Clasen, K. Ehlers, M. Feldmann, Enriching heraklit modules by agent interaction diagrams, in: L. Gomes, R. Lorenz (Eds.), Application and Theory of Petri Nets and Concurrency - 44th International Conference, PETRI NETS 2023, Lisbon, Portugal, June 25-30, 2023, Proceedings, volume 13929 of *Lecture Notes in Computer Science*, Springer Nature Switzerland AG, Cham, Switzerland, 2023, pp. 440–463. URL: https://doi.org/10.1007/978-3-031-33620-1_23.

[34] L. Seifert, Graphbasierte Analyse und architektonische Grundlagen für Roundtrip-Engineering zwischen Agent Interaction Protocols und Petrinetzen in Renew, Masterarbeit, Universität Hamburg, Department Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, 2026.

For reviewers

Contents

1. Introduction **1**

2. Foundations **2**
2.1. Workflow nets . . . 2
2.2. Reference Nets . . . 3
2.3. Renew . . . 3
2.4. Paose . . . 3
2.5. Agent Interaction Protocols (Aips) . . . 3
2.6. Cdd and Roles Interaction Matrix . . . 4
2.7. Model transformations . . . 4
2.8. Modularity . . . 5
2.9. JPMS . . . 6
2.9.1. Basic Idea . . . 6
2.9.2. Technical Realization . . . 6
2.9.3. Application of the JPMS to Renew . . . 6

3. Vision: Roundtrip Engineering for Aips **6**
3.1. From Isolated Transformations to Roundtrip Support . . . 7
3.2. Aips to Model the Control Flow of Systems . . . 7
3.3. Petri net-based Semantics . . . 7
3.4. Roundtrip Process Across Representations . . . 8
3.5. Target Architecture . . . 8

4. Existing Building Blocks **9**
4.1. Reverse Engineering: From Java Source Code to Aips . . . 9
4.2. Forward Engineering: From Aips to Java Code . . . 10
4.3. From Aips to Petri net-based Representations . . . 10
4.4. Consistency between Interaction Views . . . 11
4.5. Substitution between Petri Nets and Java . . . 11

5. Towards Integrated Roundtrip Support **12**
5.1. Overview of the Integrated Roundtrip Process . . . 12
5.2. Behavior Analysis Based on Petri Net Semantics . . . 12
5.3. Change Detection and Feedback . . . 13
5.4. Extensibility to Further Representations . . . 13

6. Discussion **14**

7. Conclusion **15**
7.1. Summary . . . 15
7.2. Outlook . . . 15

A. Examples of Generated and Extracted Artifacts **20**
A.1. From Java Code to Aip . . . 20
A.2. From Aip to Java Code . . . 22
A.3. From Aip to Workflow Net . . . 24
A.4. From Aip to Protocol Nets . . . 25

A. Examples of Generated and Extracted Artifacts

This appendix is just for the review process. In case of publication the content will be provided via a website with more details. The following examples illustrate the transformations described in Sections 4.1, 4.2, and 4.3. Each subsection corresponds to one aspect of the roundtrip process.

A.1. From Java Code to AIP

As an example for this transformation, we consider the code shown in Listing 1 which originates from the RENEW source code.

Listing 1: Example Java code from RENEW for AIP extraction

```java
package CH.ifa.draw.util;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JMenu;

import de.renew.draw.ui.ontology.AbstractCommand;


public class CommandMenu extends JMenu implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source instanceof CommandMenuItem cmi) {
            AbstractCommand cmd = cmi.getCommand();
            cmd.execute();
        }
    }
}

/** Next source file */

package CH.ifa.draw.util;

import javax.swing.JMenuItem;

import de.renew.draw.ui.ontology.AbstractCommand;


public class CommandMenuItem extends JMenuItem {

    public AbstractCommand getCommand() {
        return this._cmd;
    }
}

/** Next source file */

package de.renew.draw.ui.ontology;

public abstract class AbstractCommand {
    public abstract void execute();
}
```

The extracted Aip can be seen in Figure 3. The role names are the fully qualified class names. Note that the additional role *!Trigger* was added manually to account for technical reasons in the workflow generation. Method calls are mapped onto different elements depending on context.

- Method calls to other classes in the system to analyze are mapped onto synchronous messages.
- Method calls to core Java libraries are mapped onto actions.
- Method calls to a static type of an interface are mapped onto a decision component.

Please note: All following figures are vector graphics: You can zoom in for more details if desired.

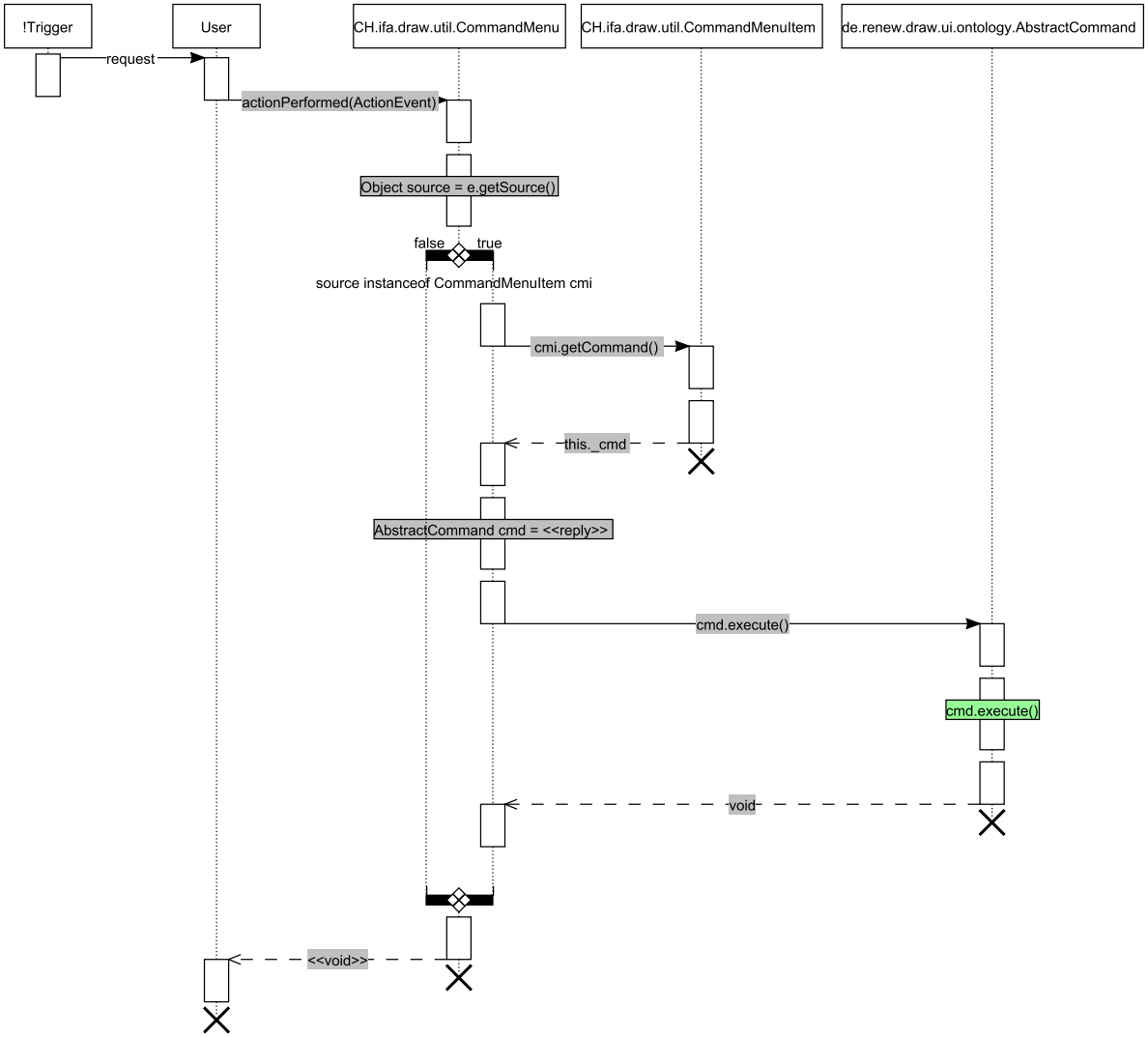


Figure 3: Example Aip generated from Renew code shown in Listing 1

A.2. From Aip to Java Code

To demonstrate the transformation of an Aip to Java code, we will inspect the generated code for the role `DiagramPluginGenerator` within the Diagram Plugin of Renew. Note that code for the role `Generator` of the PluginDevelopmentPlugin of Renew is also generated. However, since the role is not specified further besides having two interfaces for incoming messages, the generated code for this role is not too interesting. The exclamation mark at the beginning of the role name for the `GeneratePluginCommand` indicates that it is the start of the interaction. As such, it is assumed that this component already exists and serves as the entry point of the interaction, and is therefore not generated.

The Aip is depicted in Figure 4, and the code that was generated from this Aip can be seen in Listing 2

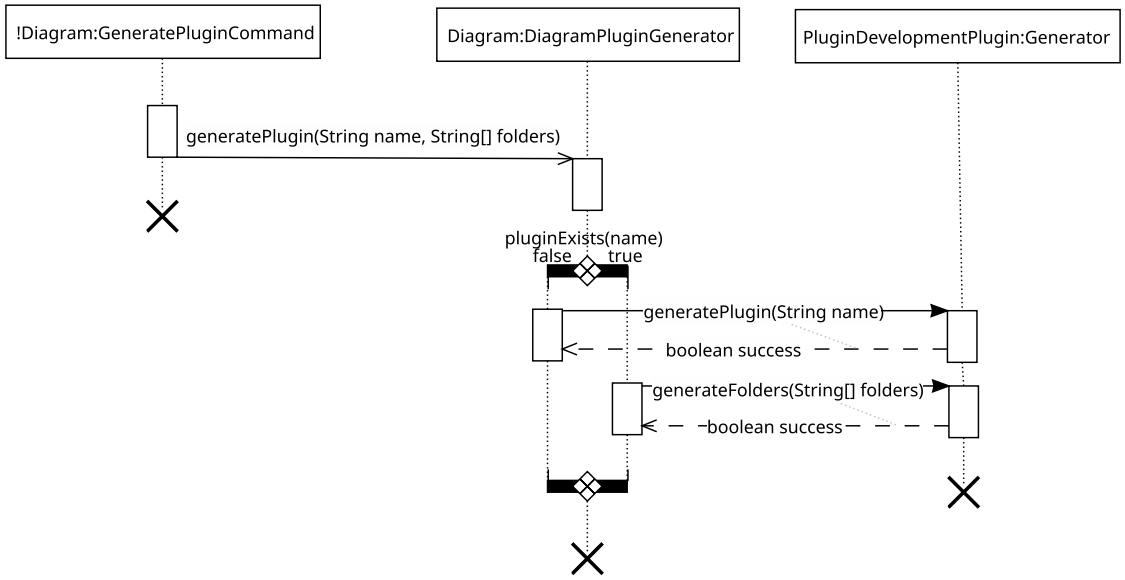


Figure 4: Example Aip to generate code templates for a new interaction in Renew

Listing 2: The code template generated from the AIP depicted in Figure 4

```java
package de.renew.diagram;

import de.renew.plugindevelopmentplugin.Generator;

/**
 * This is a generated file. Generated by Javagora.
 *
 * @author DiagramClassGenerator
 */
public class DiagramPluginGenerator {

    public void generatePlugin(String name, String[] folders) {
        if (pluginExists(name)) {
            generateFolders(folders);
        } else {
            generatePluginAndFolders(name, folders);
        }

    }

    private void generateFolders() {
        Generator generator;
        boolean success = generator.generateFolders(folders);
    }

    private void generatePluginAndFolders() {
        Generator generator;
        boolean success = generator.generatePlugin(name);
    }
}
```

A.3. From AIP to Workflow Net

In this example the AIP from Figure 3 is converted into a workflow net. The workflow net depicts the control flow of the AIP, therefore the inscriptions on AIP elements are ignored. The different roles of the AIP are shown in the visible rows of the net.

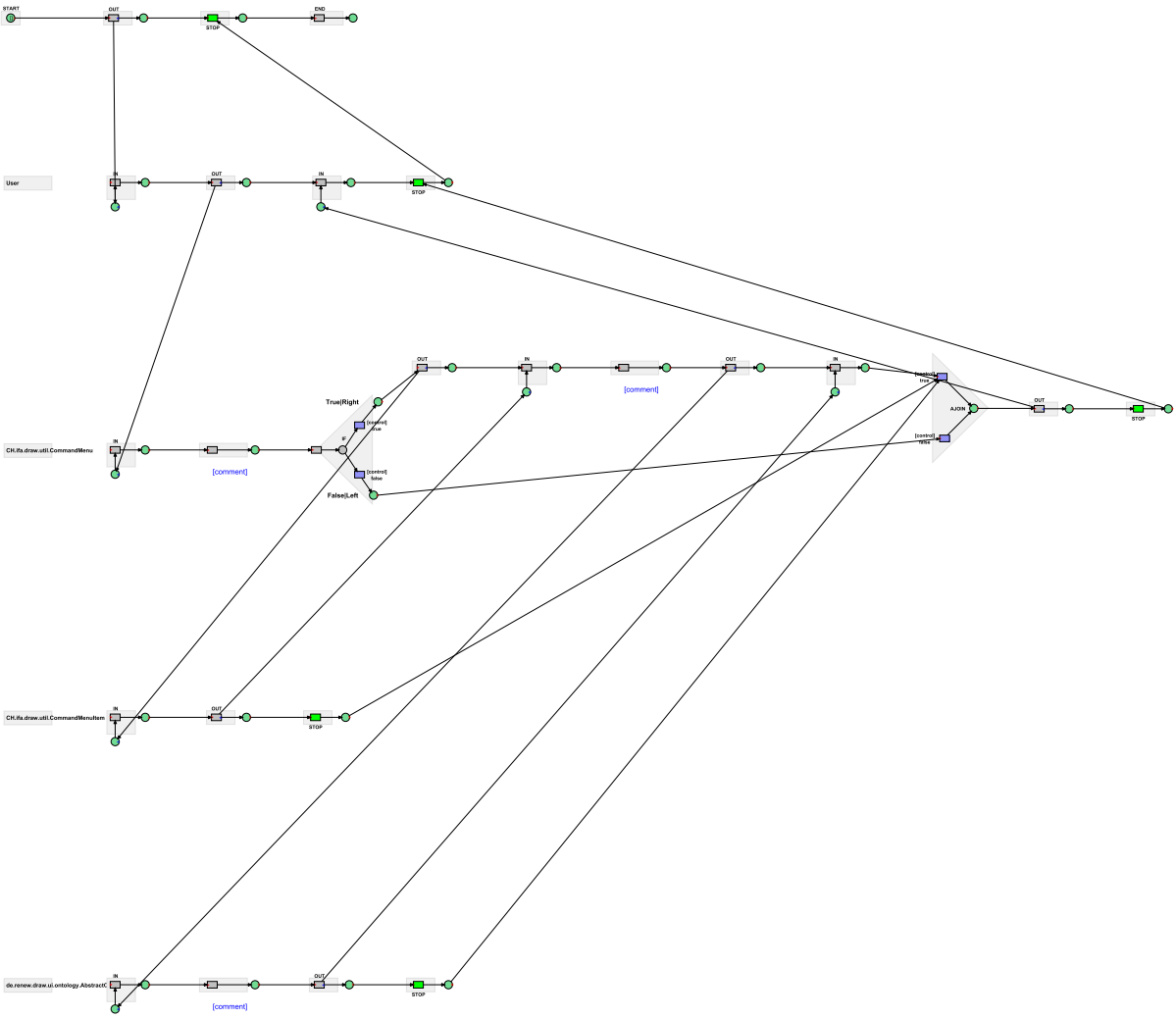


Figure 5: The corresponding workflow net

A.4. From AIP to Protocol Nets

In this example each role of the AIP Figure 3 is converted into separate protocol nets. A protocol net represents the underlying Petri net semantics.

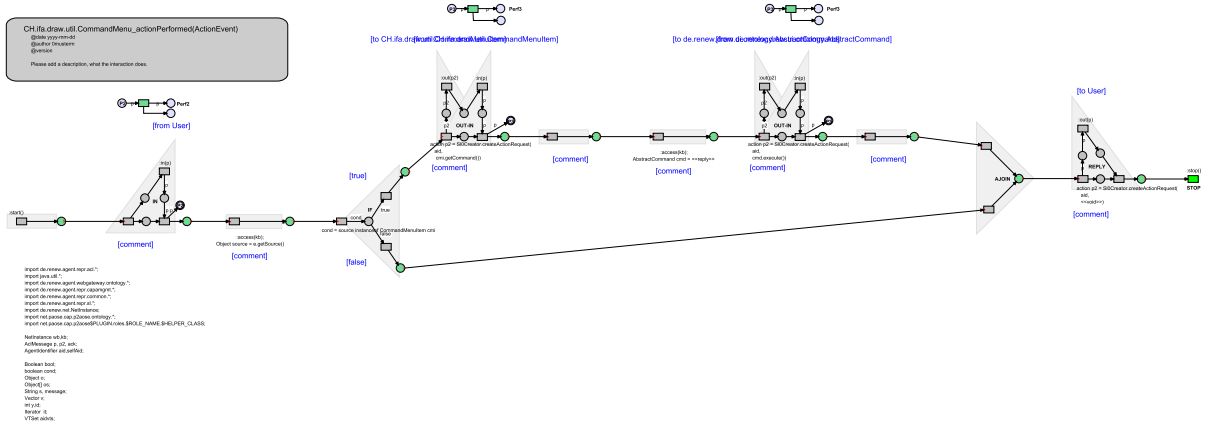
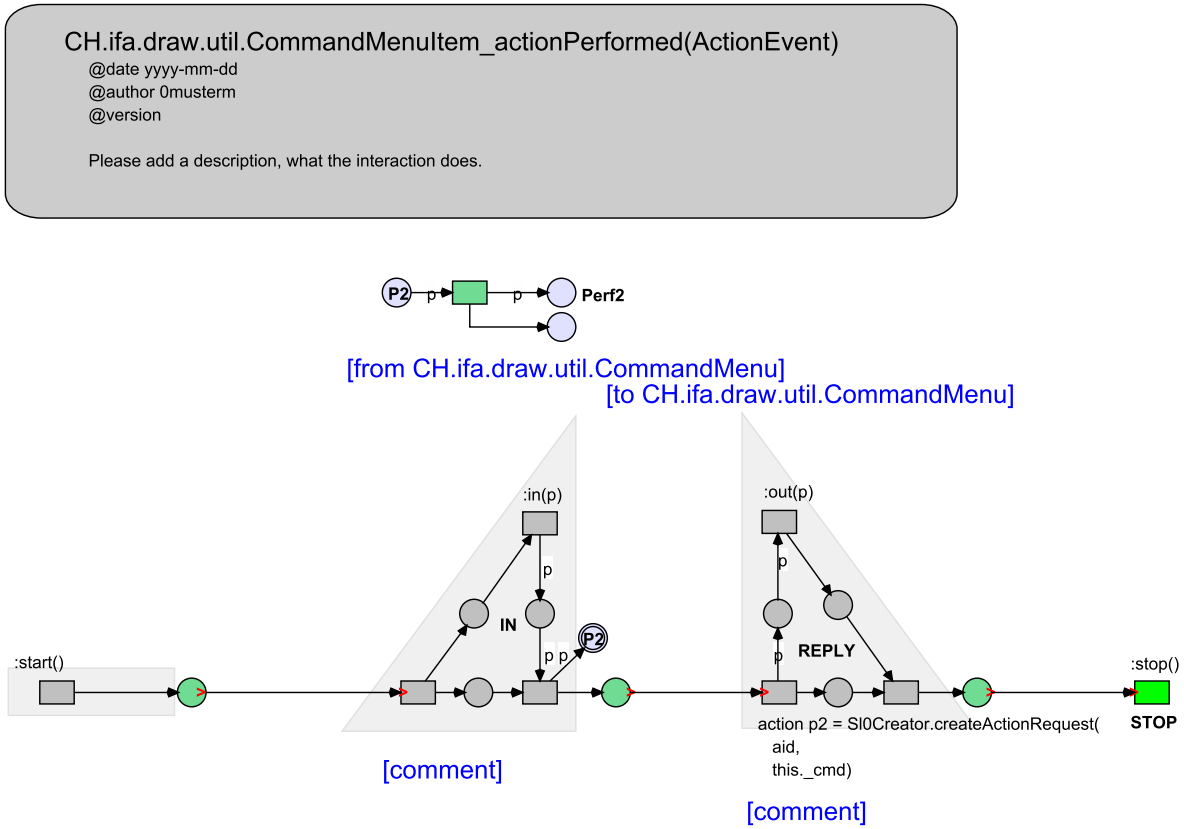


Figure 6: The corresponding protocol net for CH.ifa.draw.util.CommandMenu

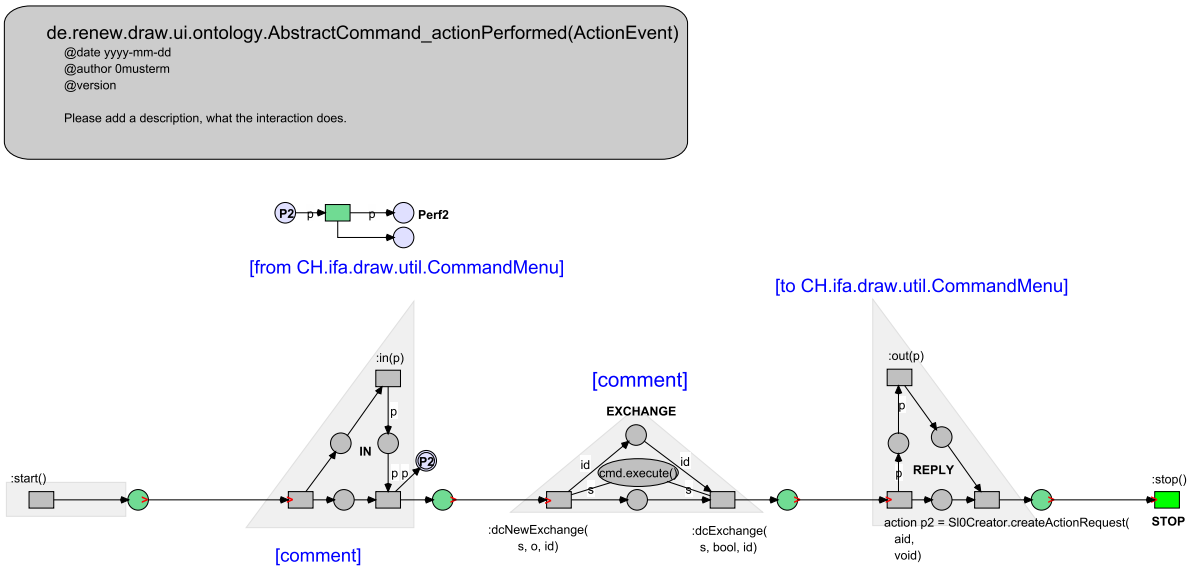



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.renew.agent.webgateway.ontology.*;
import de.renew.agent.repr.capamgmt.*;
import de.renew.agent.repr.common.*;
import de.renew.agent.repr.sl.*;
import de.renew.net.NetInstance;
import net.paose.cap.p2aose.ontology.*;
import net.paose.cap.p2aose$PLUGIN.roles.$ROLE_NAME.$HELPER_CLASS;

NetInstance wb,kb;
AclMessage p, p2, ack;
AgentIdentifier aid,selfAid;

Boolean bool;
boolean cond;
Object o;
Object[] os;
String s, message;
Vector v;
int y,id;
Iterator  it;
VTSet aidvts;
```

Figure 7: The corresponding protocol net for CH.ifa.draw.util.CommandMenuItem

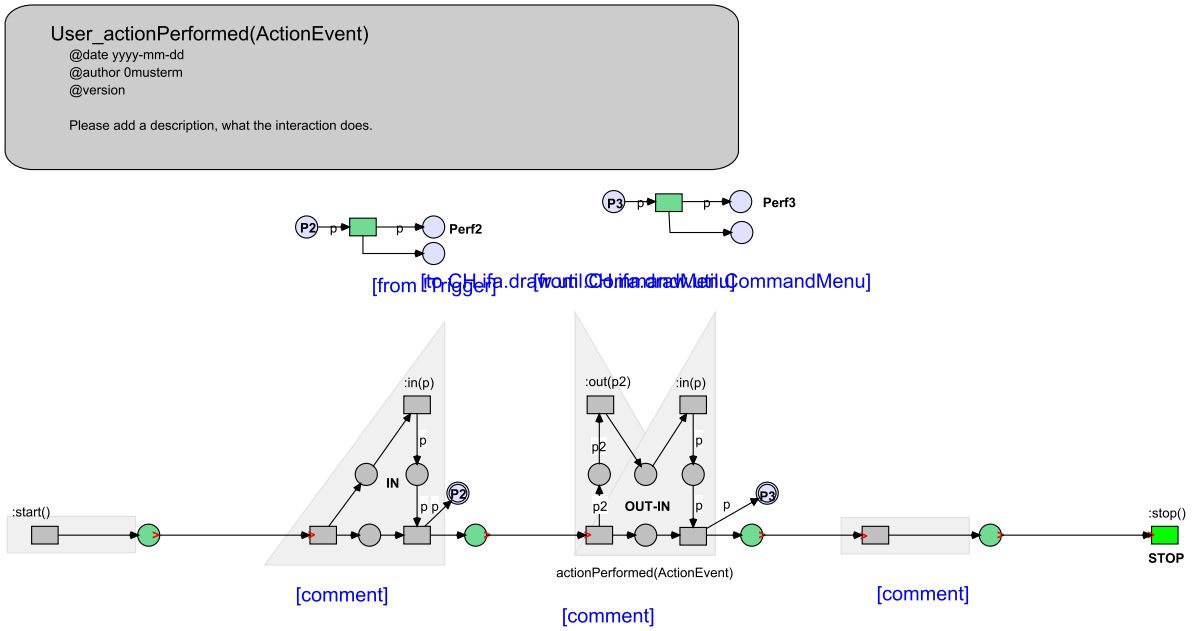



```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.renew.agent.webgateway.ontology.*;
import de.renew.agent.repr.capamgmt.*;
import de.renew.agent.repr.common.*;
import de.renew.agent.repr.sl.*;
import de.renew.net.NetInstance;
import net.paose.cap.p2aose.ontology.*;
import net.paose.cap.p2aose$PLUGIN.roles.$ROLE_NAME.$HELPER_CLASS;

NetInstance wb,kb;
AclMessage p, p2, ack;
AgentIdentifier aid,selfAid;

Boolean bool;
boolean cond;
Object o;
Object[] os;
String s, message;
Vector v;
int y,id;
Iterator  it;
VTSet aidvts;
```

Figure 8: The corresponding protocol net for de.renew.draw.ui.ontology.AbstractCommand




```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.renew.agent.webgateway.ontology.*;
import de.renew.agent.repr.capamgmt.*;
import de.renew.agent.repr.common.*;
import de.renew.agent.repr.sl.*;
import de.renew.net.NetInstance;
import net.paose.cap.p2aose.ontology.*;
import net.paose.cap.p2aose$PLUGIN.roles.$ROLE_NAME.$HELPER_CLASS;

NetInstance wb,kb;
AclMessage p, p2, ack;
AgentIdentifier aid,selfAid;

Boolean bool;
boolean cond;
Object o;
Object[] os;
String s, message;
Vector v;
int y,id;
Iterator  it;
VTSet aidvts;
```

Figure 9: The corresponding protocol net for User