PetriPad – A Collaborative Petri Net Editor

Julian Burkhart, Michael Haustermann

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
{4burkhar, 6hauster} (at) informatik.uni-hamburg.de

Abstract. Collaboration is one of the key aspects of software engineering and commonly includes working in spatially separated teams. Many tools exist to support such a workflow and are used extensively, especially for real-time communication, e.g. instant messaging systems and voice chats. In contrast, programming environments and editors used in general mostly lack synchronous real-time collaboration functionality.
In this work we present an informal specification of such a system in the context of a groupware Petri net editor and the implementation of our model as a proof of concept for the Renew tool. To do this we revisit work on this subject done more than 10 years ago and update the proposed models to the current state of software engineering. As a result we are able to simplify the specification.

Keywords: collaborative editing, Petri nets, Renew, Mulan/Capa, multi-agent systems

1 Introduction

Distributed development of a common code base in a collaborative manner has become one of the key aspects of development in computer science. Many tools exist to support this kind of distributed workflow in different styles. The most common of them are source code management systems (SCM) that enable distributed, concurrent editing of shared documents.

However, SCMs are tailored for sequentially structured textual documents and perform poorly on graphically oriented data files. When using a graphical editor the user usually has only very limited control over the serialized file formats. Even when using a text-based format, slight changes in the editor can result in vast differences in the exported file. This makes it very hard for a SCM system to distinguish the changed parts from the (semantically) identical ones.

Also SCMs only cover scenarios where developers are working independently in the sense that while working at the same project in general the work of others does not immediately impact their own. This is fine for day-to-day development, but for real-time collaborative development it is not feasible that way. Possible applications for real-time collaboration are brainstorming ideas or teaching interactive courses over the web.

For synchronous editing of text documents there are web-based solutions, like Etherpad[1] and desktop applications, like ACE[2] or Gobby[3]. There are plugins for the popular development environment Eclipse[4].

In this work we present a model for a collaborative Petri net editor. It is based on prior research described in Section 2, but varies in that it does not need any locking mechanisms of the graphical components manipulated by the users. The resulting model therefore is much simpler and is presented in Section 3. As a proof of concept we present the implementation of our model for the Petri net editor Renew [17]. It is based on the multi-agent system (MAS) framework Mulan/Capa [10]. The implementation is described in Section 4 and discussed in Section 5. At the end we summarize our work and give an overview of potential extensions of our model and of the possibilities for further research.

2 Related Work

The subject of collaborative Petri net editing is discussed in [1], which serves as a canonical case study for authors to present their approaches of combining Petri nets and object-oriented programming concepts. Some requirements for such an editor are outlined in [2]. The editor proposed is for hierarchically structured Petri nets and allows multiple users to work simultaneously on the same net from different terminals over the network. The users are organized in sessions and each user is able to work on multiple nets in different sessions at the same time.

Each user is capable of having a customized view on the net and a set of access privileges restricts the actions each user can make. For example a user might only be granted reading access to a net or parts of it. The concept of ownership of graphical elements is introduced to restrict the user's actions at a certain point in time. For each operation the appropriate ownership must be acquired beforehand. Some ownerships are exclusive (e.g. delete), others may be acquired from different users at the same time (e.g. modify). The request for some ownership may happen implicitly by selecting an item or explicitly by pressing a button. The case study describes different ownerships, how they could be requested and which restrictions apply if some user holds a certain ownership. We use the term *ownership* in the following section in this sense only unless otherwise specified.

The requirement analysis in the case study is intentionally incomplete. Other authors are encouraged to extend the requirements appropriately to their own concept or to focus on specific points to emphasize certain properties of a chosen formalism.

In the following subsections we describe three different approaches from [1].

[1] `http://etherpad.com`
[2] `http://sourceforge.net/projects/ace`
[3] `http://gobby.0x539.de`
[4] `http://www.saros-project.org`

2.1 Biberstein, Buchs, Guelfi

The authors of [6] describe a centralized approach. They build upon an architecture specified in [5] that is modeled in a formalism they introduced in [7] called *Concurrent Object-Oriented Petri Nets (CO-OOPN/2).*

Their architecture consists of two layers: a graphical interface layer (viewports) and a centralized synchronization layer (server). Documents are stored on the server and cannot be manipulated directly by users but only by the server on request. The viewport's task is to display the current net and send user input to the server. Furthermore updates made by other users received from the server have to be displayed. The server takes requests from users, updates the net and informs all users about that update. Moreover it ensures the consistency of the net and guarantees the compliance with the user rights and ownerships as described in [2].

The procedure is as follows: a user changes an element in a net in his viewport. This change is transported from the viewport to the server. The server examines if the changes are compatible with the users rights and ownerships and either includes the change into the net or rejects it. A message is sent to all viewports, if the net changed. The net itself is represented as a tree. Components that may have subcomponents are called hierarchical components and stored as nodes of the tree. Components that may not have subcomponents, called atomic components, are stored as leaves of the tree. A net in this model is itself a hierarchical component.

2.2 Bastide, Palanque

The authors of [3] focus on locking mechanisms for objects to ensure consistency. The majority of the requirements in [2] are not taken into account. The implementation of the locking mechanism is described in great detail and down to a very technical level. The general concept is the locking of graphical elements that are selected in an editor for all other users. [3] describes how the synchronization between graphical elements and graphical editor works. They use the *cooperative objects* formalism to present their approach.

We do not go into further details, since we do not use any locking in our model.

2.3 Guerrero, Figueiredo, Perkusich

Guerrero, Figueiredo, Perkusich describe a decentralized multi-agent architecture [14] for the collaborative editor. They distinguish between two kinds of agents: user agents and manager agents. User agents may work on nets according to their access privileges and join or leave editing sessions. Manager agents are capable of performing administrative tasks for user and session management. They can also grant and revoke ownerships of graphical elements.

Each agent consists of three layers. The lowest layer is the communication layer. It provides message transportation services. The middle layer differs between user agents and manager agents. It is called control layer and management layer respectively. At the top lies the application layer and represents the interface presented to the users of the system.

For a user agent the application layer is a graphical editor that allows the user to view and manipulate Petri nets. User inputs are first passed on to the control layer. The control layer verifies that the net manipulations are compliant with the ownerships the user holds. It may try to request additional ownerships, if needed for the desired action. If the control layer fails to acquire all necessary ownerships, the change is rejected.

The communication layer is used to exchange messages with other agents. Especially to request ownership and to inform other agents about acceptable changes. Incoming changes are passed up through the layers and displayed in the user interface (UI).

The manager agent's application layer is a system console, which enables the execution of administrative commands. These are performed by the manager agent's management layer.

3 Informal Model Specification

In this section we present our own approach. Since it differs from the aforementioned work, we first discuss the main principles behind it. Then we describe the resulting architecture in detail. Therefore we describe the requirements on the user interface, discuss different communication languages and explain how consistency can be guaranteed in our model.

3.1 General Architecture

One of the goals of this work is to update the previous models to the current practice of UI design, especially the design of the collaborative text-editor Etherpad. Comparing different websites offering etherpad-based services[5], the editor offers basic formatting tools only. There are no restrictions in terms of what parts of a document a user may edit. Session management is confined to merely distinguishing sessions, while access privileges are left out completely. The name that uniquely identifies an Etherpad session is incorporated into the URL leading to the edited document and subsequently it can be accessed by that URL without restrictions.

Despite of the absence of administrative capabilities, the result is well fitting to the task of collaborative writing. The UI is highly intuitive and access privileges are non-essential to enable collaborative work (though possibly helpful in some use cases). All conflicts arising from editing the same passage can be discussed in the integrated chat window.

From this brief analysis of Etherpad we draw three main principles for our proposed model.

[5] An overview of some of the available websites can be found on `http://etherpad.org/public-sites/`

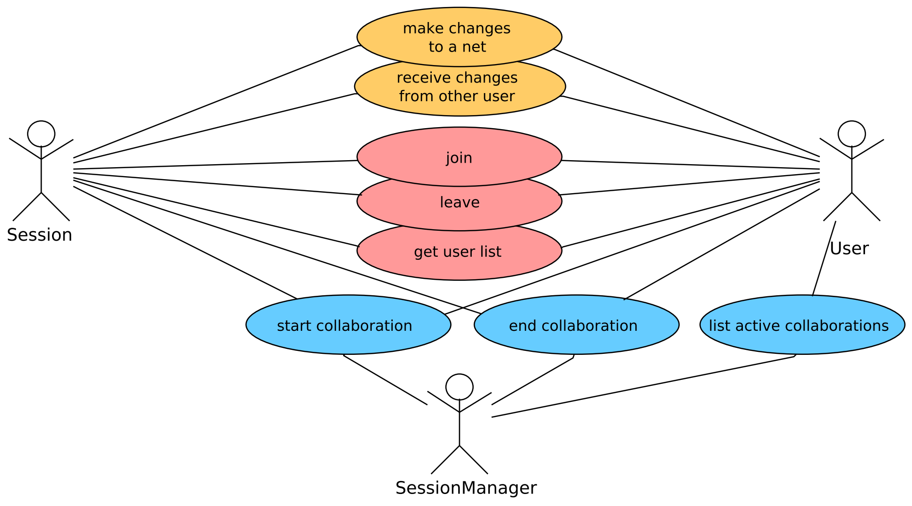

Fig. 1: The typical use cases of a collaborative editing system, i.e. manipulating shared Petri nets (the upper two use cases) and session handling (the rest).

1. All users have the same access privileges, i.e. no access privileges are present at all.[6]
2. The user is enabled to make any change he wants, i.e. no change is rejected afterwards or prohibited in the first place. Furthermore every change is implemented immediately in the users view.
3. Any arising conflicts are dealt with automatically and in a sensible way.

From the second point it is obvious that we had to omit locking mechanisms as described in [3] and [14] (cf. Section 2). This is also the primary source for simplification for our model.

We adopt a number of requirements from the original case study. The session management enables users to participate in multiple collaboration sessions and collaborate on multiple nets at the same time. A user's view on the shared nets is independent of the other users.

Additional requirement on the system is the separation of communication infrastructure and editor, so that the graphical user interface (GUI) could be exchanged. For example it should be possible for one of two collaborating users to work in a full-blown desktop client, while the other works in a web-based editor in an internet browser.

[6] It should be mentioned that we are not opposed to access privileges in general. A number of use cases can be thought of that require such a mechanism, e.g. tutoring or presentation purposes, but these are not the use cases we considered for this work.

The system is modeled as a multi-agent application similar to [14], but with some important differences. We identify three kinds of agents in the system (cf. Figure 1).

User agents represent the users of the system. Since we want the UI to be interchangeable, a user agent needs to be modular by design. It consists of a communication module with a well-defined interface to which an arbitrary editor can be connected.

Session management agents offer an entry point to user agents. They are autonomous and can start or terminate sessions on request of user agents or be queried for a list of existing collaboration sessions.

Session agents represent individual collaboration sessions and keep track of the edited nets and the participating users. It also functions as a central message relay between the participating user agents and resolve conflicts arising from concurrency. It also stores the current state of each net.

3.2 Requirements on User Interface

The UI connected to a user agent's communication module has two essential tasks. First of all, it has to observe the actions the user takes. This follows from the manner changes are not requested at a central authority as in [6], but simply made and then passed on to others.

The UI should also recognize what actions result in meaningful changes. In this case meaningful refers to the completion of an action. While the user drags an element from one point to another, it might not be wise to flood the network with updates for every single pixel that the element is moved. Especially, because in the context of nets, dragging one element usually impacts the position of other elements that are connected to it as well.

To further reduce the number of messages, actions may be grouped together to batches and sent in one message to the session agent. The most straight forward way for any receiving agent to deal with batches is to first execute all operations that add new elements, then perform all operations that change existing elements and lastly all remove operations.

The second job of the UI is to integrate incoming changes from the session agent. A crucial requirement is for the integration to be done atomically and between user manipulations, so that it does not interfere with changes the user makes.

3.3 Communication Language

We consider compliance with the standards of the Foundation for Intelligent Physical Agents[7] (FIPA) [20] a baseline for our model. These include defining the message format (Agent Communication Language – ACL [11]) for all inter-agent communication. Choice exists however on the part of the content languages

[7] `http://www.fipa.org/`

for the actual message payload. Two possibilities will be discussed, namely the Petri Net Markup Language [24] (PNML)and using ontologies.

PNML is a ISO/IEC standard for higher order Petri nets that is still in development. Many Petri net tools have adopted it [8] and type definitions for various different flavors of Petri nets have been created. The latter is what makes PNML especially attractive for our work. The set goal of building a universal platform for collaborative editing agnostic to the actual editor in use would greatly benefit from a widely adopted standard. Renew as our main aim for application of our work already has support for importing and exporting PNML.

The main downside of applying PNML to this work is that it only represents the net itself and not manipulation operations on it. Describing the actions performed directly, e.g. a moveElement or addMarking operation, is out of the question. PNML can only be used to describe the set of elements that changed and their relevant properties. That is however a viable solution and receiving agents can simply overwrite the received elements in their copies of the net.

Another possibility is to use an ontology to model Petri nets and the operations. Ontologies becoming increasingly popular in software engineering. They can be used as a glossary throughout all development phases and as a meta-language for specification. The de facto standard ontology modeling tool Protégé[8] has a built-in code generator, which can generate Java classes directly from an ontology. Protege is based on the Web Ontology Language (OWL) developed for the Semantic Web[9]. It has a variety of different syntaxes to choose from [19,15]. They can be machine-readable like the RDF-based XML syntax or better suited to be edited by humans like the Manchester Syntax, thus lowering the bar for adoption of OWL 2 considerably over its predecessor OWL 1.

In multi-agent systems ontologies form the basis for communication [13]. Without a common ontology to give meaning to objects and statements, there can be no exchange of knowledge, both between agent or humans. And specifically in the context of our implementation detailed in Section 4, the Mulan/Capa framework relies heavily on ontologies. Not only for communication, but for modeling purposes as well.

The two possibilities, using PNML or ontologies, are not necessarily mutually exclusive either. Attempts have been made to define ontologies for higher order Petri nets, that are compliant with PNML [12,23]. Since we are aiming at interoperability to some extent in our model, we suggest [23] to be used to embed PNML into use in our multi-agent system.

3.4 Consistency Guarantee and Conflict Treatment

Guaranteeing consistency needs special care in our approach since users have their own synchronized copies of the shared nets. To ensure that identifiers for net elements are globally unique, we let the central session agent determine them. So when a user adds an element locally, a temporary identifier is assigned

[8] `http://protege.stanford.edu/`
[9] `http://www.w3.org/standards/semanticweb/`

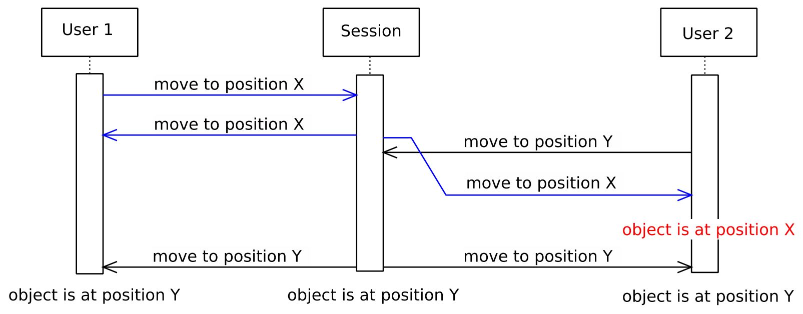

Fig. 2: Possible inconsistencies from varying message transit times.

to that element. The final identifier is received as a result message, when he informs the session agent about the new element. All changes to objects with temporary identifiers are buffered in the user clients until the final identifier has been determined.

To address the problem of messages overtaking one another, we add sequence numbers to each message. A global order of all messages is determined by the session agent, who orders them per user and integrates them into a global order by time of arrival. The session agent accordingly sets new sequence numbers to all messages before distributing them.

Our approach enables users to modify the same object concurrently. Thus we allow conflicting modifications that have to be dealt with. For conflicting changes to the same property of an element, the change processed last by the session agent wins. To enable a user to determine which change won in such a case, the session agent distributes change operations (but not additions or deletions) to all users including the sender (cf. Figure 2).

A minor inconvenience of the scenario in Figure 2 is that User 2 would implement the change of User 1 for a short period of time after he made his own change which would be undone shortly after that, when his own update is sent back to him. A possible remedy for this is that if a user made a change to property p of element a, all incoming updates to p of a can be ignored until his own change appears in the stream of updates.

Lastly we have to deal with incoming changes to elements that were already deleted. This situation occurs at the session agent, when a user makes a change to an element before receiving the delete message. It can also occur at the user agent when an element was deleted locally, but the delete message was not yet distributed. In either case the change can be dropped safely. The user from whom the change originated will eventually receive the delete message and subsequently delete the element himself. That way a consistent state is reached at quiescence, i.e. when all messages have been distributed and processed at each site.

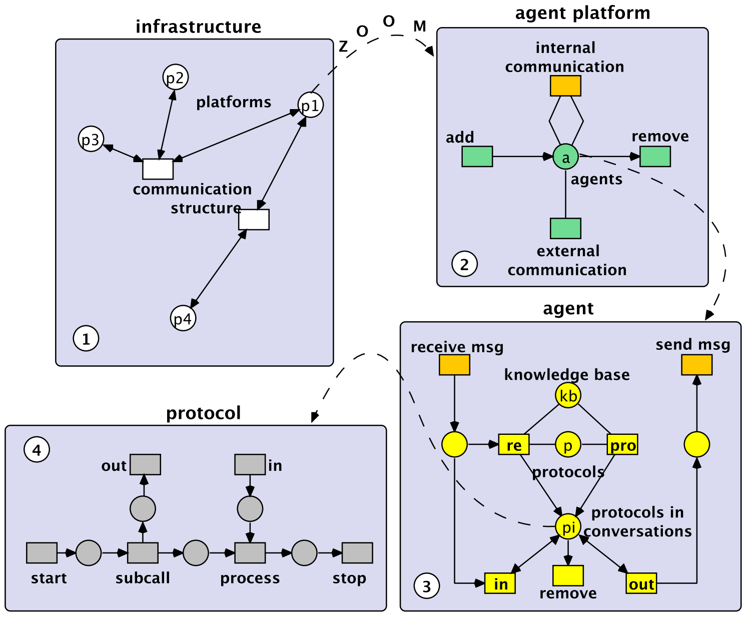

Fig. 3: The four levels of the MULAN model.

4 Implementation for Renew

In this section we give an overview of our proof-of-concept implementation. We will shortly introduce the RENEW Petri net editor and the MULAN/CAPA agent framework and then describe the implementation and the Ontology we use for communication.

4.1 Context of Implementation

RENEW. The Reference Net Workshop (RENEW) [17] is an editor for Petri net formalisms developed by the theoretical foundations of computer science group at the department of computer science at the University of Hamburg. It is highly modular and includes plugins for different Petri net formalisms. The integrated simulator can execute basic P/T-nets as well as higher order nets, e.g. colored Petri nets and reference nets, which are object-oriented Petri nets with synchronous channels that allow for tokens to be nets themselves [16,22]. Transitions of reference nets can also be inscribed with Java code, that is executed during simulation. This mechanism provides a seamless integration of object-oriented programming with specification and simulation of Petri nets.

MULAN/CAPA. In order to facilitate the implementation of our model, we built upon a framework for multi-agent applications called MULAN/CAPA.

Mulan (**Mul**ti-**a**gent **n**ets) [21] is a reference architecture for multi-agent systems. It is modeled completely in reference nets and consists of four different levels as seen in Figure 3.

The highest level (level 1) is the infrastructure. The infrastructure connects multiple agent platforms to a network. A platform (level 2) provides the environment for agents. Apart from starting and terminating agents, it provides means for communication between agents. If sending and receiving agent are identical the communication is considered to be an internal communication, e.g. between different active protocols of the same agent. The next level are the actual agents (level 3). The agents can send and receive messages to and from other agents and the platform. This is the only externally observable behavior of an agent. Each communication between agents and agents or agents and a platform is described by protocols (level 4). It models the behavior of an agent or how agents communicate with each other. A protocol is developed in complementary parts for each participating agent and orders the flow of information and the messages sent.

Capa (**C**oncurrent **A**gent **P**latform **A**rchitecture) [10] is a FIPA-compliant implementation of the Mulan model on top of Renew and Java. Capa facilitates building multi-agent applications based on the Mulan model. Heterogeneous systems with platforms implemented in other frameworks are possible due to the FIPA-compliance. The infrastructure level of the Mulan model is not implemented in Capa. It emerges when combining an arbitrary number of instances of Capa platforms that can be interconnected over a network.

The combination of model and implementation, plus the development environment, monitoring and debugging tools comprise the Mulan/Capa framework. It provides a high level of concurrency since it is developed with reference nets that are by design concurrent.

4.2 Coarse Design of the PetriPad Plugin

PetriPad consists of two parts. The multi-agent model is implemented in the Mulan/Capa framework and we extend Renew with a plugin, which connects it to the MAS. In terms of our informal model communication module is the agent connected to the editor.

To exchange data between the editor and the agent we use the WebGateway plugin for Mulan/Capa. It implements a gateway architecture [4], which facilitates connecting HTML5-based web services to the MAS. The WebGateway acts as a bridge using the WebSocket protocol and although web applications are its original aim, it can hook up arbitrary software systems.

Our utilization of WebGateway can be seen in Figure 4. The Renew Petri net editor serves as UI for an agent running in a remote Mulan/Capa platform. We call that agent the *modeler agent*. The Renew plugin tracks the changes made by the user and passes them through a WebSocket channel to the WebGateway, which relays them to the modeler agent in the PetriPad MAS.

The primary motivation for this architecture is that users do not need to be running a instance of the Mulan/Capa platform locally. Instead only one

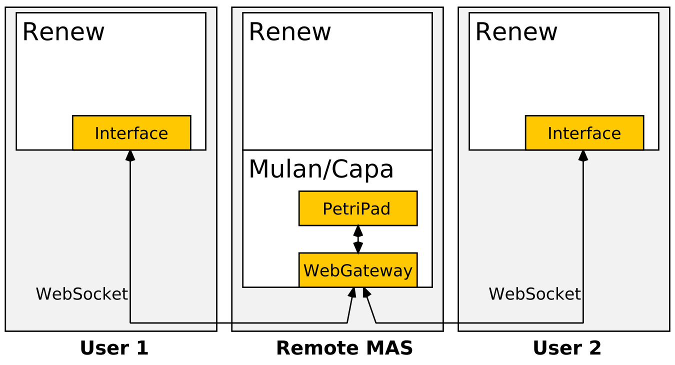

Fig. 4: Our proposed architecture of the communication infrastructure.

platform is needed to which an arbitrary number of editors can be connected and it is possible to use editors other than Renew.

Our use case diagram (Figure 1) showed three agent types and a number of interactions between them. These translate directly to levels 3 and 4 of the Mulan model. Each interaction is implemented as a protocol for the participating agents.

4.3 Ontology

Although we argued in favor of using PNML compliant ontologies, we had to deviate from it to some extent. This is mostly due to the limited expressive power of the Mulan/Capa default modeling formalism for ontologies. It is called *concept diagrams* [9] and can define a taxonomy of concepts in a UML-based graphical notation. Each concept can be described by a set of key-value-tuples and the values' respective domains. A domain may be a Java data type, another concept in the ontology or a list of either of them. Semantically *concept diagrams* coincide with the idea of frames [18] limited to defining concepts only and subsumption as the only relation.

Figure 5 shows a subset of our ontology dealing with reference nets and possible operations on them. The semantics are as follows. The nodes in the graph represent the defined concepts and the arcs define the subsumption hierarchy. Every concept has a name (bold) and attributes of the form *k: t*, where *k* is the name of the attribute and *t* its domain. A star ('*') at the end of the domain definition denotes that an instance of the concept can have multiple values for that attribute, whereas the other attributes must be single-valued.

Our ontology distinguishes between the structural elements (transition, place, arc, virtual-place[10]) and the textual elements. Arcs in our model have type (e.g.

[10] A virtual place is a reference to a place. A place and its virtual copies are semantically identical.

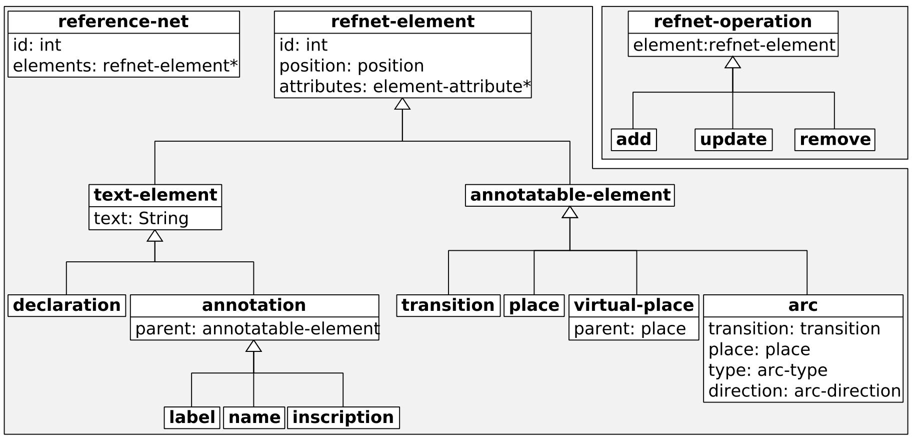

Fig. 5: A part of the ontology defining reference nets. As much as it is practical it is modeled with their formal properties in mind rather than their representation in Renew.

normal, inhibitory) and direction attributes that define the way they interact with the associated place and transition.

The leafs of the text-element subtree model the four types of textual elements that are used in Renew. A structural element can be named by a name annotation and inscriptions cover all semantically relevant annotations, e.g. markings, guards, weights, etc. All other annotations are labels and are ignored during simulation of the net. Declarations are free-standing textual elements that contain all declarations of Java objects used in the net and all imports for Java packages that are needed.

The top concept for elements (refnet-element) has an attribute attributes that may contain the graphical information of the element. The concept hierarchy for its type element-attributes is not depicted here, but contains e.g. color and size.

An operation on an element is modeled as refnet-operation. We distinguish three different kinds of operations: add, update and remove. In case of an update the element attribute contains the entire new state of the element that was updated.

Note that Figure 5 describes only part of our reference net ontology.

5 Discussion

Our decision to abandon locking mechanisms resulted in a huge simplification of the overall design of the collaborative Petri net editor. On the other hand abandonment of locking mechanisms entails certain problems with concurrent editing of the same elements. It can be confusing for users for example to have their changes overwritten immediately by other users. One such case has been discussed in Section 3.4.

In this case the developers have to communicate to clear up the confusion and possibly revert some changes manually. We argue that in the targeted use case of constructive collaborative work this is a corner case that seldom arises and thus can be neglected.

The choice of using WebSocket as communication channel and a well-defined ontology for message contents leads to loose coupling of the infrastructure and the editor. This has two main benefits. Firstly different editors can be connected to the MAS and secondly the MAS can be offered as a service, which compatible editors can connect to from anywhere over the net.

Not mentioned before, the WebGateway can convert ontology communication into JSON or XML. These languages are common in web applications and thus facilitate connecting web-based editors in particular.

Modeling the system as a MAS resulted in a very modular architecture and naturally, it inherits certain properties from the Mulan/Capa framework. On the one hand, implementing large parts of the system in Petri nets is a huge gain in terms of concurrency. On the other hand the message transport through the various layers of the MAS is far slower than direct peer-to-peer communication.

The ontology we designed for inter-agent communication is not yet compliant with the PNML standard, but in general the differences are minor. Using an ontology has the additional benefit of serving as a glossary in the development phase. It also provides a more abstract means of modeling in conjunction with the code generator of the Mulan/Capa framework.

6 Summary

In the preceding chapters we gave a short overview of the prior work on collaborative Petri net editors. We argued that the specifications are incomplete in the sense that they only consider parts of the overall system.

The new model proposed in this work differs from all prior approaches in that it completely foregoes locking of elements in the edited nets. The goal is not to restrict the user in what he can manipulate and give immediate feedback to all inputs. We achieve this by using a multi-agent approach with a central message relay. By serializing all communication at the central relay we were able to implement user input immediately in the user's view and only subsequently send it to the relay.

Session management is incorporated in our model as well. A user can participate in multiple sessions simultaneously and each session can hold an arbitrary number of documents available to all users in that session.

As a proof of concept we implemented our model as a plugin for the Petri net editor Renew and a multi-agent application based on the Mulan/Capa framework.

7 Outlook

Since our approach uses WebSocket for communication it allows for using a web application as Petri net editor. With the canvas element and the WebSocket channel HTML5 offers all the required components to build such an editor.

Another topic of great significance for developing Petri nets is the synchronized simulation. The simulation of the models may reveal modeling errors. In a collaborative development process, on-line debugging a net has the benefit that every participant can observe the exact firing sequence in the net. Simulating the nets individually produces various different firing sequences for each participant due to the innate concurrency of Petri nets and is therefor of little use to collaborative debugging.

Looking a little farther behind the horizon, the MAS specified in our model is by itself agnostic to the subject of the collaborative work. We defined very basic manipulation operations that can be applied to a number of collaborative editing scenarios. The MAS could thus be developed into a service platform providing an infrastructure for collaborative editing. In order to build such an infrastructure a meta ontology for collaborative editing needs to be developed. It will facilitate building subject ontologies that describe particular subjects of collaborative work and the permissible manipulations and their effects. Compliant subject ontologies can then be used in conjunction with the collaboration infrastructure.

References

1. Gul Agha, Fiorella De Cindio, and Grzegorz Rozenberg, editors. *Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
2. Rémi Bastide, Charles Lakos, and Philippe A. Palanque. A Cooperative Petri Net Editor. In Agha et al. [1], pages 534–535.
3. Rémi Bastide and Philippe A. Palanque. Modeling a groupware editing tool with cooperative objects. In Agha et al. [1], pages 305–318.
4. Tobias Betz, Lawrence Cabac, and Matthias Wester-Ebbinghaus. Gateway architecture for Web-based agent services. In Franziska Kügl and Sascha Ossowski, editors, *Multiagent System Technologies*, volume 6973 of *Lecture Notes in Computer Science*, pages 165–172. Springer Berlin / Heidelberg, 2011.
5. O. Biberstein, D. Buchs, and N. Guel. CO-OPN/2 applied to the modeling of cooperative structured editors. In *Tech. Report 96/184, Swiss Federal Institure of Technology (EPFL), Software Engineering Laboratory.* Citeseer, 1996.
6. O. Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In Agha et al. [1], pages 73–130.
7. Olivier Biberstein and Didier Buchs. Structured Algebraic Nets with Object-Orientation. In G. Agha and F. de Cindio, editors, *Workshop on Object-Oriented Programming and Models of Concurrency'95*, pages 131–145, 1995. Turin.
8. Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The petri net markup language: Concepts, technology, and tools. In *Applications and Theory*

of Petri Nets 2003: 24th International Conference, pages 1023–1024, Eindhoven, The Netherlands, June 2003.

9. Lawrence Cabac. *Modeling Petri Net-Based Multi-Agent Applications.* Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, April 2010. `http://www.sub.uni-hamburg.de/opus/volltexte/2010/4666/`.
10. Michael Duvigneau. Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2002.
11. Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. `http://fipa.org/specs/fipa00061/`.
12. Dragan Gašević and Vladan Devedžić. Petri net ontology. *Knowledge-Based Systems*, 19(4):220 – 234, 2006.
13. Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199 – 220, 1993.
14. Dalton Serey Guerrero, Jorge C. A. de Figueiredo, and Angelo Perkusich. An object-based modular cpn approach: Its application to the specification of a cooperative editing environment. In Agha et al. [1], pages 338–354.
15. Matthew Horridge and Peter F. Patel-Schneider. OWL 2 web ontology language manchester syntax. `http://www.w3.org/TR/owl2-manchester-syntax/`.
16. Olaf Kummer. Introduction to Petri nets and reference nets. *Sozionik Aktuell*, 1:1–9, 2001. ISSN 1617-2477.
17. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Michael Köhler, Daniel Moldt, and Heiko Rölke. Renew – the Reference Net Workshop. In Eric Veerbeek, editor, *Tool Demonstrations. 24th International Conference on Application and Theory of Petri Nets (ATPN 2003). International Conference on Business Process Management (BPM 2003).*, pages 99–102. Department of Technology Management, Technische Universiteit Eindhoven, Beta Research School for Operations Management and Logistics, June 2003.
18. Marvin Minsky. A framework for representing knowledge. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
19. Peter F. Patel-Schneider and Boris Motik. OWL 2 web ontology language mapping to RDF graphs. `http://www.w3.org/TR/owl2-mapping-to-rdf/`.
20. S. Poslad and P. Charlton. Standardizing agent interoperability: The FIPA approach. *Multi-Agent Systems and Applications*, pages 98–117, 2006.
21. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications.* Logos Verlag, Berlin, 2004.
22. Rüdiger Valk. Petri nets as token objects - an introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal*, number 1420 in Lecture Notes in Computer Science, pages 1–25, Berlin Heidelberg New York, 1998. Springer-Verlag.
23. J.C. Vidal, M. Lama, and A. Bugarín. A high-level petri net ontology compatible with PNML. *Petri Net Newsletter*, 17:11 – 23, 2006.
24. Michael Weber and Ekkart Kindler. The petri net markup language. In Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-40022-6_7.