Universität Hamburg
Department Informatik
Vogt-Kölln-Str. 30
D-22527 Hamburg

Bericht 290

Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents MOCA´09

FBI-HH-B-290/09

Editors:
Michael Duvigneau
Daniel Moldt
Universität Hamburg
Department Informatik

In die Reihe der Berichte des Fachbereichs Informatik aufgenommen durch
Prof. Dr. R. Valk
Prof. Dr. W. Lamersdorf

September 2009

Abstract
This report contains the proceedings of the fifth International Workshop on *Modelling of Objects, Components and Agents* (MOCA'09) that took place in Hamburg, Germany on September 11, 2009. Modelling is *the* central task in informatics. Objects, components, and agents are fundamental units to organise models. The workshop comprises a wide variety of contributions that address all relations between theoretical foundations of models on the one hand and objects, components, and agents on the other hand with respect to modelling in general.

Zusammenfassung
Dieser Bericht enthält die Proceedings des fünften internationalen Workshops über das *Modellieren von Objekten, Komponenten und Agenten* (MOCA'09), der am 11. September 2009 in Hamburg stattfand. Objekte, Komponenten und Agenten sind fundamentale Einheiten von Modellen. Der Workshop umfasst ein breites Spektrum an Beiträgen, welche alle Beziehungen zwischen theoretischen Grundlagen von Modellen auf der einen Seite und Objekten, Komponenten und Agenten auf der anderen Seite im generellen Zusammenhang des Modellierens angehen.

Editors: Michael Duvigneau and
Daniel Moldt

Proceedings of the Fifth International Workshop on

M odelling of
O bjects
C omponents and
A gents

MOCA'09

University of Hamburg
Department of Informatics

Preface

This booklet contains the proceedings of the International Workshop on *Modelling of Objects, Components and Agents* (MOCA'09) in Hamburg, Germany, September 11, 2009. It is a co-located events of *MATES 2009*, the seventh German conference on Multi-Agent System Technologies and *CLIMA-X 2009* the tenth international workshop on Computational Logic in Multi-Agent Systems.

More information about the workshop, like online-proceedings, can be found at

`http://www.informatik.uni-hamburg.de/TGI/events/moca09/`

Modelling is *the* central task in informatics. Models are used to capture, analyse, understand, discuss, evaluate, specify, design, simulate, validate, test, verify and implement systems. Modelling needs an adequate repertoire of concepts, formalisms, languages, techniques and tools. This enables addressing distributed, concurrent and complex systems.

Objects, components, and agents are fundamental units to organise models. They are also fundamental concepts of the modelling process. Even though software engineers intensively use models based on these fundamental units, and models are the subjects of theoretical research, the relations and potential mutual enhancements between theoretical and practical models have not been sufficiently investigated. There is still the need for better modelling languages, standards and tools. Important research areas are for example UML, BPEL, Petri nets, process algebras, or different kinds of logics. Application areas like business processes, (Web) services, production processes, organisation of systems, communication, cooperation, cooperation, ubiquity, mobility etc. will support the domain dependent modelling perspectives.

Therefore, the workshop addresses all relations between theoretical foundations of models on the one hand and objects, components, and agents on the other hand with respect to modelling in general. The intention is to gather research and application directions to have a lively mutual exchange of ideas, knowledge, viewpoints, and experiences.

The multiple perspectives on modelling and models in informatics are most welcome, since the presentation of them will lead to intensive discussions. Also the way objects, components, and agents are use to build architectures / general system structures and executing units / general system behaviours will provide new ideas for other areas. Therefore, we invited a wide variety of contributions, which were reviewed by an international programme commitee, which was supported by several other international experts, resulting in at least *four* reviews per submitted paper. The programme commitee mem-

bers reflect important areas and perspectives for the Modelling of Objects, Components, and Agents (MOCA).

The program committee consists of:

Bernhard Bauer (Germany)
Rafael Bordini (Brazil)
Jose-Manuel Colom (Spain)
Jörg Desel (Germany)
Torsten Eymann (Germany)
Jorge C. A. de Figueiredo (Brasil)
Paolo Giogini (Italy)
Xudong He (USA)
Koen Hindriks (The Netherlands)
Astrid Kiehn (India)
Radek Koci (Czech Republic)
Johan Lilius (Finland)
Andrea Omicini (Italy)
Birna van Riemsdijk (The Netherlands)
Yann Secq (France)
Mark-Oliver Stehr (USA)
Catherine Tessier (France)
Michael Wahler (Switzerland)
Manuel Wimmer (Austria)
Olivier Boissier (France)
Piotr Chrzastowski-Wachtel (Poland)
Mehdi Dastani (The Netherlands)
Michael Duvigneau (Germany) (Chair)
Berndt Farwer (UK)
Guy Gallasch (Australia)
Esther Guerra (Spain)
Vincent Hilaire (France)
Benjamin Hirsch (Germany)
Franziska Klügl (Sweden)
Michael Köhler-Bußmeier (Germany)
Daniel Moldt (Germany) (Chair)
Pascal Poizat (France)
Heiko Rölke (Germany)
Alexei Sharpanskykh (The Netherlands)
Harald Störrle (Germany)
Rainer Unland (Germany)
Danny Weyns (The Netherlands)
Christian Zirpins (Germany)

We received 13 high-quality contributions. The program committee has accepted six of them for full presentation. Furthermore the committee accepted three papers as short presentations.

The international program committee was supported by the valued work of Tina Balke as an additional reviewer. Her work is highly appreciated.

Furthermore, we would like to thank the organizational team of VSIS in Hamburg for their general organizational support.

Without the enormous efforts of authors, reviewers, PC members and the organizational teams this workshop wouldn't provide such an interesting booklet.

Thanks!

Michael Duvigneau and Daniel Moldt
Hamburg, September 2009

PS: This is the second edition of the workshop proceedings. It includes changes to the ACTAS paper that reached us too late for inclusion in the original printed edition of the proceedings. Furthermore, the document now includes navigation facilities for electronic publication.

Contents

Part I

Invited Talk

Modelling and Verification of Resource-Bounded Multi-Agent Systems

Berndt Farwer

Durham University
School of Engineering and Computing Sciences
South Road, Durham DH1 3LE, UK
berndt.farwer@durham.ac.uk

Abstract. We survey approaches to multi-agent systems' specification and verification. In real-world systems, the notions of resource and location represent important constraints that need to be considered at both the specification level and the verification level. When applying model-checking techniques to multi-agent systems or programs, properties are usually formalised in a modal logic language. We argue that it is imperative to introduce a native concept of resource into such property specification languages.

Part II

Full Presentations

Visual Representation of Mobile Agents

Modeling Mobility within the Prototype MAPA

Lawrence Cabac, Daniel Moldt, Eva Müller, Matthias Wester-Ebbinghaus

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
http://www.informatik.uni-hamburg.de/TGI/

Abstract. Mobile agents in dynamic and open systems are still a challenge, not only within the process of developing mobile agents but also in integrating these in multi-agent systems. Graphical modeling facilitates the development of mobile and concurrent agents within a multi-agent system.
This paper picks up the approach of modeling mobility by the use of reference nets as a high-level Petri net. Hence, these are chosen as the formal basic principle, which incorporates the "nets within nets" paradigm as a suitable framework for mobility. Furthermore, this paper introduces the prototype of the framework MAPA – an architecture for multi-agent systems supporting mobility and its visualization.
The underlying formal principles are depicted by an extended household robot system, introduced in [14]. The enhancement of this case study is based upon and accompanied by MAPA. MAPA not only provides a visual representation of (mobile) agents but also a routing agent, which computes routes between connected platforms. In addition, MAPA allows the use of two tools, namely MulanViewer and Sniffer. The MulanViewer allows the user to keep track of the primary reference nets and the Sniffer tool logs the message flow during run-time.

Key words: mobility, agent, multi-agent system, reference net, Petri net, routing agent

1 Introduction

Talking about multi-agent systems means talking about topics such as agent, concurrency, mobility, distributed systems, communication or autonomy, to name just a few examples. Any of these keywords correlate in a certain manner. Due to the fact that this paper focuses on mobility and its visualization within the context of open multi-agent systems, the particular emphasis is placed on mobility, communication and agent.
We conceptualize the use of the term *agent* by taking Wooldridge's definition:
"An agent is a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives. "[25, p. 29]

As a start, we conceive a multi-agent system as a representation of social and/or geographical structures. In general, the environment, e.g. an area or room, represents the agent platform where agents may reside. For the first time, the Foundation for Intelligent Physical Agents (FIPA) standardized the term *agent platform* in 2002. The aim was to ensure the interoperability between applications which were developed independently. The current standard of 2004 defines: "An Agent Platform (AP) provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system, agent support software, FIPA agent management components (DF[1], AMS[2] and MTS[3]) and agents." [10, cf. document SC00023K]
To understand the meanings as well as the constraints of (the visual representation of) mobility within multi-agent systems, an examination of the underlying concepts and formalisms is necessary.
The prototype of a multi-agent system framework that we present in this paper uses reference nets [16] as the underlying modeling and implementation technique. Reference nets are particularly suitable since they carry forward the "nets within nets" paradigm of Valk [12] and for that reason are able to implement multi-agent systems as defined by the Mulan-reference model [13]. As a matter of course, the migration of an agent between platforms in a multi-agent system characterizes mobility. Even though, we consider not only the migration of an agent but also the transmission of any entity[4] between agents in the context of the Mulan-reference model as mobility.
The here presented prototype supports the visual representation of (migrating) agents and bears the name Mapa (**M**obile **A**gent **P**latform **A**rchitecture). For this, we restructure the already developed framework of a multi-agent system, namely Capa [8], concerning the support of agent visualization. The idea of developing such a prototype arose from the examination of the conceptual and prototypical addressing of mobility in [14]. Herein, the visual representation of mobility was dealt with. However, Mapa perceives itself as a consolidation of previous ideas. We aim to visualize (mobile) agents from top to bottom of the Mulan-reference model, i.e. the multi-agent system, the platforms as well as the agents residing on the platform and to provide flexibility with respect to future applications using Mapa as much as possible.
Structure of the paper. In Section 2 we present the main concepts. First of all, we give a short introduction to the formalism of reference nets. Subsequently, we describe the reference architecture Mulan and its implementation Capa. Finally, we state the use of the term *mobility* relating to this paper. The case-study of a mobile household robot, which was introduced in [14], will be outlined and discussed in Section 3. The results are carried forward to some requirements for the prototype Mapa, we specify in Section 4. Further on, we outline the main features of our prototype. Afterwards, we join and illustrate the results of the

[1] Abbreviation of *Directory Facilitator*.
[2] Abbreviation of *Agent Management System*.
[3] Abbreviation of *Message Transport Service*.
[4] Hence, entities are (for example) messages, artifacts or resources.

previous sections by a modified version of the case-study of a mobile household robot using Mapa. Section 5 discusses related work. We close this paper with Section 6. Herein, we summarize the achievements and give an outlook on further work.

2 Basic Principles

The modeling as well as the visual representation of (mobile) agents requires a formalism in order to provide some kind of graphical representation of the used entities. Also, it has to support complex, concurrent processes. In addition, the incorporation of the "nets within nets" paradigm is necessary in order to model hierarchical structures as specified in the Mulan architecture.

Reference nets. The formalism of reference nets offers some interesting and useful features of which the framework Mapa takes advantage. The extensions regarding (colored) Petri nets[5] are the creation of net instances during run-time, the incorporation of the "nets within nets" paradigm and the possibility to communicate through synchronized channels. For that reason, reference nets are a special kind of colored Petri nets and provide serveral advantageous graphical notation in order to visualize mobility.
The "nets within nets" paradigm formalizes the modeling of tokens as any kind of data type or net [23]. So, the token of a Petri net arises from a usual *black token* to an (active) entity, which may dynamically act itself or even represent a Petri net. This fact enables the modeling of the hierarchical structures of a multi-agent system via reference nets. Thus, entities may be Petri nets, which reside in another Petri net as tokens and therefore also describe the marking of this Petri net.

Due to the above mentioned features, reference nets are object-based and executable by the simulation engine Renew [17]. Within reference nets, tokens may represent references to other entities[6]. If the referenced entity is a net itself, it is possible that more than one token references this entity, regardless of which net the token belongs to. For this reason, a dynamical composition is possible during run-time. In this regard, one may also refer to nets as *system-net* and *object-net*. The *system-net* contains the token, which references the *object-net*. One can also say that the *object-net* resides within the *system-net*. If the *object-net* holds a reference to another net by a token in turn, the denotation is up to the users point of view. Obviously, this relationship between *system-net* and *object-net* refers back to the "nets within nets" paradigm in turn.

Mulan – **Multi Agent Nets.** The framework Mulan was presented in [13] and is an implementation of the Mulan-reference model by means of reference nets

[5] We assume that (colored) Petri nets are well-known. See also [19,11,15].
[6] These entities are control flow token, data types or nets, for instance.

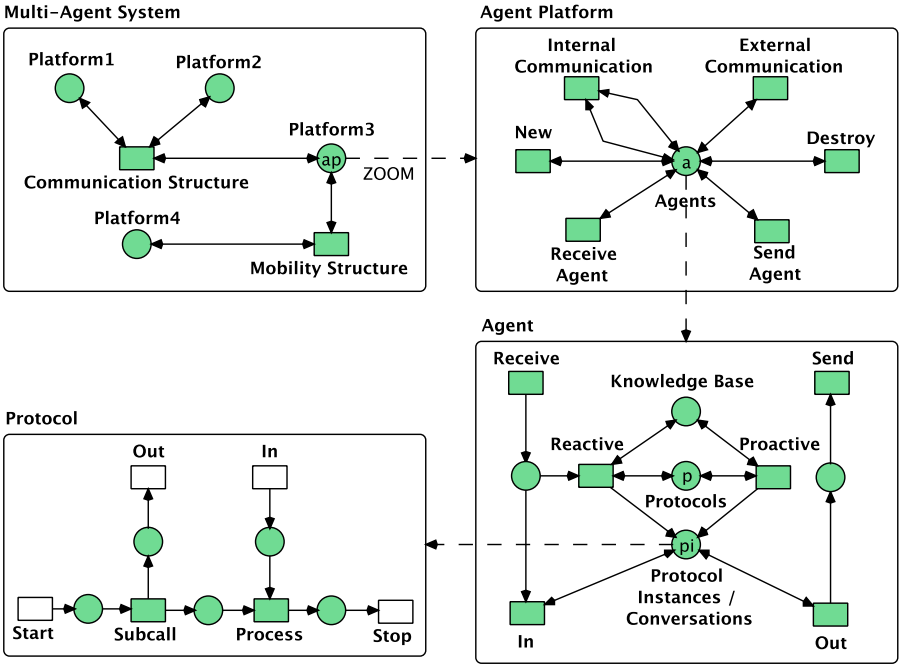


Fig. 1. Agent system as nets within nets

. MULAN establishes the basic principle of the presented agent platform MAPA due to the fact that MULAN is a multi-agent system and incorporates the "nets within nets" paradigm. Therefore, it provides the possibility to model the hierarchical structures of the MULAN-reference model, which is depicted in Figure 1. As mentioned above, each box in Figure 1 "describes one level of abstraction in terms of a *system net*" [14, p. 128]. As well, each *system-net* accommodates an *object-net*. This relationship is represented by the dashed ZOOM arrows. Even though Figure 1 constitutes the MULAN-reference model, it is also a simplified version of the framework MULAN, which is executable if all inscriptions and synchronous channels are replenished. The *Multi-Agent System* net is capable of storing *Agent Platform* nets as tokens in given places. The transition *Communication Structure* as well as the transition *Mobility Structure* establish the infrastructure of the multi-agent system in terms of the direct communication between agents residing on a platform. However, the given *Multi-Agent System* net exemplifies communication structures and is not implemented in MULAN.
The token on place *Platform3* in the *Multi-Agent System* net is described in more detail in the *Agent Platform* net. By zooming into this token, the general structure is made observable. The place *Agents* stores the agents, which are actually residing on this platform. New agents are created by the transition

New and destroyed by the transition *Destroy*. As well, agents are migrate onto this platform or leave the platform by the transitions *Receive Agent* and *Send Agent*. The communication between two agents on the same platform is realized by the transition *Internal Communication*. The overall platform communication between two agents on different platforms is provided by the transition *External communication*. The structure of an agent or rather its agent net becomes visible by zooming into the token on the place *Agents*. The agent is able to communicate with other agents by the transitions *Receive* and *Send*. The intelligent and autonomous behavior of an agent is modeled by the place *Knowledge Base* and the transitions *Reactive* and *Proactive*. These two transitions have access to protocol templates on the place *Protocols*, which are as well nets and model the behavior of the agents. These protocols are instantiated by the two transitions *Reactive* and *Proactive* as a result of an incoming message. The instantiated protocol nets are stored in the place *Protocol Instances / Conversations* and are part of an ongoing conversation. The protocol instance can be inspected by zooming into its token on the place *Protocol Instances / Conversations*. The exemplified structure of a protocol instance is given in the net *Protocol*.

A more detailed description of the four layers, which cover representations of the multi-agent system, platforms, agents and protocols (agent internals), can be found in [13,20]. Mulan uses reference nets to implement these hierarchical structures. According to this, the internal communication structure of Mulan is modeled as a net, too. Unfortunately, this implementation results in an overall platform communication, which requires the very same simulation engine for any participating platform [13]. This lack of interoperability between participating remote platforms led to the development of Capa.

Capa – **Concurrent Agent Platform Architecture.** Capa [8] is a Java and Petri net based implementation of the framework Mulan. Capa enables an overall platform communication beyond used simulation engines, i.e. participating agents neither have to use the same simulation engine nor the same framework. For that purpose, Mulan's conceptional platform net was enlarged as a FIPA-compatible implementation. During the design and implementation process, the most important aspect was to facilitate a high grade of concurrency. Capa is composed of the platform Capa and the agents AMS (Agent Management System) and DF (Directory Facilitator). In addition, Capa provides an internal Message Transport Services (MTS) and an interface to enable an overall communication, the ACC (Agent Communication Channel). For a detailed specification of Capa see [9] and the respective Fipa-specifications [10].

Mobility. The migration or movement of an entity from one location to another is intuitively associated with mobility. Though, we have to distinguish between two kinds of mobility. The first kind refers to physical entities, such as mobile phones and is called *physical mobility*. The other kind characterizes so-called "logical computing entities", such as network applications. In this paper, we term this kind as *logical mobility* and stress it regarding agent-oriented

software engineering.
In order to talk about mobility, we limit its scope to *location*[7], *agent*[8] and *object*[9] as defined in [14]. According to the term *object*, the meaning of communication as a particular kind of mobility is more comprehensive if you consider messages as *objects*, which are exchanged – in terms of sending a message – between agents. In addition, we differentiate between four types of movement, namely subjective, objective, spontaneous and consensual movement[10]. In the context of the "nets within nets" paradigm and the relationship between *system-net* and *object-net*, the subjective movement is triggered by the *object-net* so that the *object-net* is moved within the *system-net*. For instance, an agent starts its migration from one location to another. On the contrary, the objective movement is triggered by the *system-net* such as message passing. If there is no control with regard to the movement of the *object-net* within the *system-net*, this is called spontaneous movement. On the other hand, the consensual movement requires an agreement between *system-net* and *object-net* on the *object-nets'* movement.

3 Case Study of a Household Robot Using MULAN

In the following section, we discuss the implementation of the case-study of a mobile household robot, presented in [14,20]. The case-study describes a mobile household robot that receives and executes assignments. For that purpose, the mobile software entity – the agent *robot* – migrates between rooms within the house. The house represents the overall system and constitutes the multi-agent system according to the previously introduced MULAN architecture. The system is executed within the framework MULAN and is illustrated in Figure 2. Every location in the system – **hall**, **kitchen**, **living room**, **next room** and **frontyard** – is embodied by an agent platform of the multi-agent system[11]. The agent *robot* migrates between these platforms[12]. The migration process is based on reference net semantics, where the robot (agent) token flows through the multi-agent system net, from platform to platform (where each platform rests on a *location*-place, cf. Figure 2). During run-time, the initialized agent platforms are referenced by so-called text-token at the respective *location*-place. For example, the text-token "platform2[745]"[13] at the *location*-place *frontyard*

[7] Locations are bordered, host agents and differ in respect to their control capabilities.
[8] Agents are (mobile) software entities which are able to perform actions.
[9] Objects represent different kinds of physical or data resources, which can be moved and exchanged between agents.
[10] To be within the scope of this paper, we refer you to [14] for a more detailed examination.
[11] Each platform offers dedicated services, e.g. fetch coffee, serve coffee, fetch mail, open or close the door.
[12] The route between start platform and destination platform was given by internal protocol nets of the agent *robot*.
[13] The number in the brackets states the reference to the respective net instance during a specific simulation being set up for the screen shot.

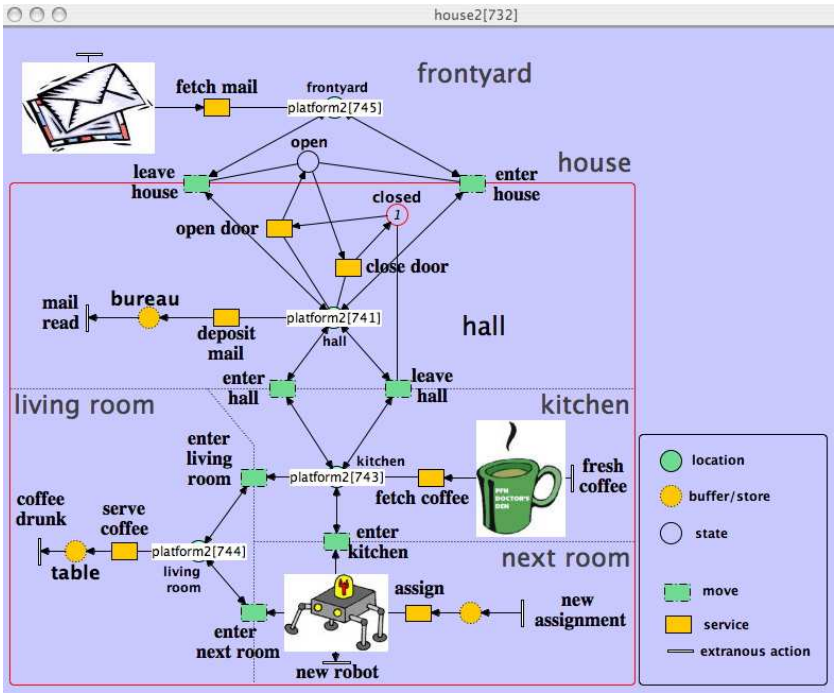


Fig. 2. Snapshot of system at start of the simulation

references the platform net **frontyard**.

Visual representation of mobility. As a fundamental approach concerning the visualization of mobility, [14] introduces the representation of (mobile) agents by a so-called image-token, in addition to the textual representation. For instance, the arrival of a new *letter* at the platform **frontyard** is illustrated by an image-token as shown in Figure 2. The agent *robot* resides at the platform **next room**. This fact is represented by an image-token as well in Figure 2. If the agent *robot* migrates, e.g. from platform **next room** to platform **kitchen**, the image-token of the *location*-place *next room* is replaced by an ordinary text-token, while the text-token "platform2[743]" of the *location*-place *kitchen* is replaced by an image-token. Thus, the *robot*'s movement through the house can be visualized during run-time and an overview of the system's current state is possible.

Discussion. All in all, the presented case-study provides a convincing demonstration of modeling mobility by the "nets within nets" paradigm. The intuitive representation of mobile entities by the usage of image-tokens is a nice feature of visualizing mobility at run-time.
However, the visualization of robot (agent) tokens at the level of the multi-agent system is at odds with the architectural design of MULAN's multi-agent system.

The problem is that robot (agent) image-tokens are used as platform references. This is somewhat counter-intuitive as such a token does rather suggest a reference to the robot/agent itself. In addition, a multi-agent system net hosts objects (mail, coffee, etc.) that are not platforms. Thus, the multi-agent system level is not in accordance with the MULAN architecture. Furthermore, the modeling and implementation of the case-study in such a manner is not suitable for more than one agent per platform. As a related issue, the transport of entities such as a *letter* by the agent *robot* is not displayable within the current implementation.

4 MAPA – Mobile Agent Platform Architecture

The examination of the case-study in Section 3 has led to the idea of developing a dynamic framework that stresses the visual representation of (mobile) agents and is based upon the MULAN-reference model. The visualization of mobility, as presented in Section 3, is not directly implementable in the framework CAPA. This is owed to the fact that the top layer of the MULAN architecture, the multi-agent system, is represented indirectly by the Agent Communication Channel (ACC [9]). Consequently, this layer is not an explicit part of CAPA. Instead of being embedded in a particular infrastructure – as proposed by the MULAN-reference model – all platforms in CAPA are just exhaustively connected with each other. Introducing constraints on communication and mobility channels at the multi-agent system level are not as easily expressed in CAPA as in MULAN. In CAPA, they have to be addressed at the agent level instead. To conclude, the original MULAN architecture offers a more immediate support of inter-platform infrastructures and thus offers a more expressive basis for mobility.
Due to this deficiency of CAPA, the prototype of the framework MAPA (Mobile Agent Platform Architecture) has been implemented. MAPA joins the advantages of both architectures MULAN (explicit multi-agent system infrastructure, usage of images-tokens) and CAPA (implementation of the FIPA-guidelines for interoperability purposes). Furthermore, this consolidation allows the usage of certain tools such as the MulanViewer[14] and the Sniffer[15], which were first developed as CAPA-plug-ins. Before we start with a more detailed specification of the prototype MAPA, we define some necessary requirements.

Requirements relating to MAPA. An intuitive association of mobility within agent-oriented software engineering is the migration of an agent between two platforms, which are hosted on different clients. If the multi-agent system supports multiple mobile agents, some interfaces for communication, coordination and cooperation are necessary. This is already implemented due to the fact that MAPA enhances CAPA, which provides these interfaces.
As well, MAPA has to represent the top layer of the MULAN-reference model

[14] The MulanViewer is a tool-kit to display every set up platform, its agents and (active) protocols.

[15] The Sniffer is a monitoring tool in order to log the messages, which were exchanged between agents.

in an appropriate way. Also, the system net of the multi-agent system has to visualize the migration of an agent. If the migrating agent transports further entities then the visualization of these would be a nice feature, too.
Thus, the implementation of the visual representation of agents within a multi-agent system is an indispensable requirement due to the fact that this is the fundamental basis for any future framework. This comes along with the visualization of multiple agents at one platform. To facilitate re-usability and sustainability of the prototype, it has to provide flexibility concerning its usage by different applications. So, the prototype has to be adaptable and reusable by different applications. This also implies that possible routes between connected platforms are being computed automatically.

Mapa utilizes the simulation engine Renew as IDE (Integrated Development Environment). The plug-in based architecture [22,3,21,4] of Renew allows the development of Mapa as a plug-in itself. The resulting compatibility facilitates the development of further applications, which are able to use the framework Mapa. During the development process of the prototype, an explicit separation was forced between a user view and developer view. The developer view is accessible by the MulanViewer, which allows the survey of (active) protocols and existing (agent-) nets. At this view, a visualization of (mobile) agents has not been implemented.
At the user view, the visual representation of (mobile) agents has been implemented by the use of image-tokens. This means that image-tokens are used as platform references at the multi-agent system. In addition, agents residing on a certain platform are represented by image-tokens as well. This is done by a vertical-hierarchical structuring of the involved platform image-token and its agents image-tokens[16]. In contrast to the implementation of the visual representation in Section 3, the visualization already starts on the level of the multi-agent system. It follows that not only (mobile) agents will be represented by image-tokens but also platforms. Furthermore, the transport of an entity by an agent is illustrated by a labeled image-token. Thus, a four-level, vertical-hierarchical representation arises: multi-agent system $\rightarrow$ platform $\rightarrow$ agent $\rightarrow$ transported entity.
Another particular trait of the Mapa prototype is the implemented routing agent. On the level of the multi-agent system, all platforms are connected by transitions and synchronous channels. The registration of a platform at the multi-agent system causes the routing agent to add every directly connected platform of this newly added platform to its knowledge base as a route. Thus, an agent requests possible routes from the routing agent and migrates, having received a suitable route. The routing agent computes every possible, acyclical route whenever it has no suitable route with regard to the request. The computed route between start platform and destination platform is subsequently stored in its knowledge base.

[16] This form of visualization will be presented on the basis of the modified case-study (see Section 3) at the end of this section.

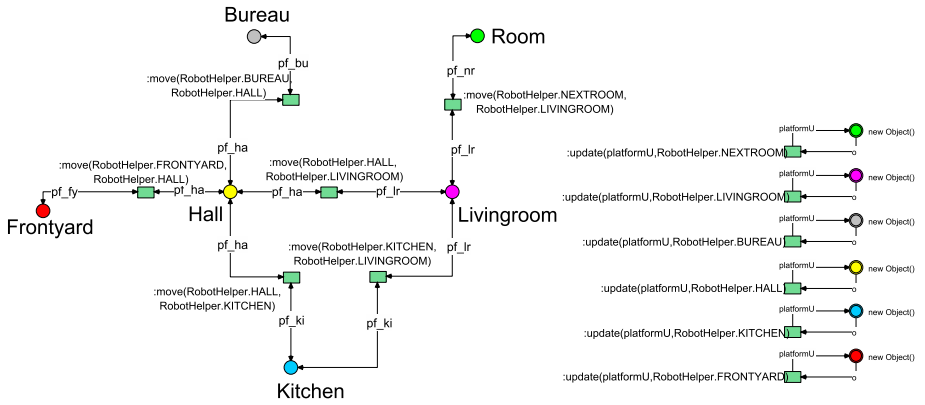


Fig. 3. Multi-agent system's net template.

Execution of the Modified Case-Study by Using Mapa**.** The conversion of the case-study of a mobile household robot to the prototype Mapa requires some modifications. On the one hand, the multi-agent system at the user view has to be re-structured in such a way that platforms are representable by image-tokens. On the other hand, platforms have to be connected explicitly and the update of image-tokens has to be ensured whenever the system state changes. Figure 3 illustrates the adapted multi-agent system. It shows the explicit connection of the platforms so that the routing agent is able to compute possible routes between them. The migration of an agent between two platforms will be done by firing the respective *move*-transition. The platforms itself at the multi-agent system are initialized as objects when the multi-agent system is initialized. If a new platform is started, the *update*-transition will be fired in order to update its image-tokens at the multi-agent system level. Additionally, every arriving or leaving agent at one platform triggers the firing of the respective *update*-transition. Furthermore, agents obtain their own image-token while residing on a platform. The agent image-tokens are placed on top of the image-token of the platform, as you can see in Figure 6 for instance. Therefore, the separation between user and developer view is necessary. Figure 4 depicts this separation. The net instance *Bureau[1787]* illustrates the user view of the platform **Bureau** and its residing agents. For this case-study, the solely information regarding the platform **Bureau** are the residing agents. Another application may show further or other information. On the contrary, the net instance *mapa[1785]* displays the developer view of the platform **Bureau**. This net instance is based on Mapa's default net template *mapa*, which provides and displays the overall structure of a platform. This net template should not be changed.

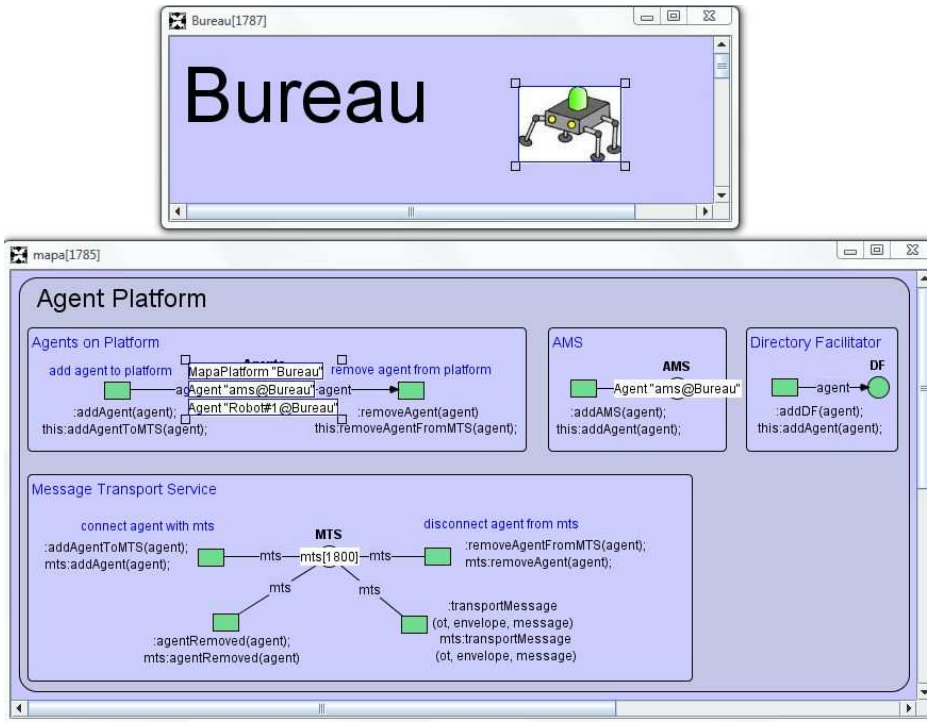


Fig. 4. Illustration of user view (net instance *Bureau[1787]*) and developer view (net instance *mapa[1785]*) for platform **Bureau** after system initialization.

Setting. The following execution setting was chosen: The multi-agent system is initialized with six platforms, namely **Room**, **Bureau**, **Livingroom**, **Kitchen**, **Hall** and **Frontyard** as illustrated in Figure 5[17]. Afterwards the two agents *Robot* and *Postman* are started. Figure 6 shows that the agent *Robot* resides on the platform **Bureau** and the agent *Postman* resides on the platform **Frontyard**. The important point regarding Figure 3 is based on the fact that, for instance, the image-token for platform **Bureau** has an image-token for the agent *Robot* on its top. Nevertheless, the place **Bureau** as shown in Figure 3 contains exactly one token, namely the image-token for the platform **Bureau**. However, Mapa facilitates the representation of hosted agents with additional image-tokens. In other word, the image-token of the *system-net* has the image-tokens of its hosted *object-nets* on its top (cf. Section 2, page 4f).
In this case-study, the agent *Robot*, residing on the platform **Bureau**, is waiting for an assignment. The agent *Postman* resides on the platform **Frontyard** and is transporting a letter (cf. Figure 6). In order to deliver the attached letter, the

[17] Figure 5 is the net instance of the net template in Figure 3. The platforms are represented by image-tokens.

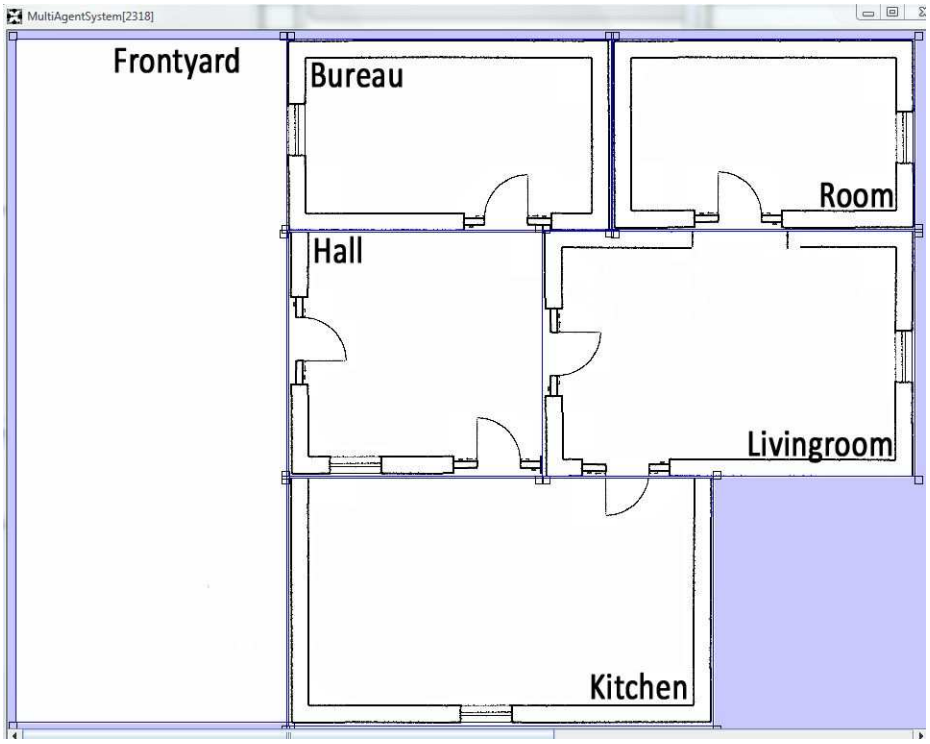


Fig. 5. User view on multi-agent system after system initialization and the start of all platforms.

agent *Postman* initializes a protocol. This protocol assigns the agent *Robot* to fetch the letter. Once the agent *Robot* receives the assignment, it queries the routing agent of the multi-agent system about a suitable route from start platform (**Bureau**) to destination platform (**Frontyard**). Having received the route, namely **Bureau** → **Hall** → **Frontyard**, the agent *Robot* migrates to platform **Frontyard** via platform **Hall** as this is the shortest way (cf. Figure 3). There, the agent *Postman* and the agent *Robot* exchange the letter as illustrated in Figure 7. Also, Figure 8 gives an overview of the related net instances. The inspection of the image-token of the platform **Frontyard** shows the net instance *Frontyard[187]*, where all hosted agents are displayed. Furthermore, the inspection of the image-token of the agent *Robot* returns the net instance *Robot[4216]*. This net instance is accessible as well by the examination of the *Robot* agents' image-token in the net instance *Frontyard[187]*.

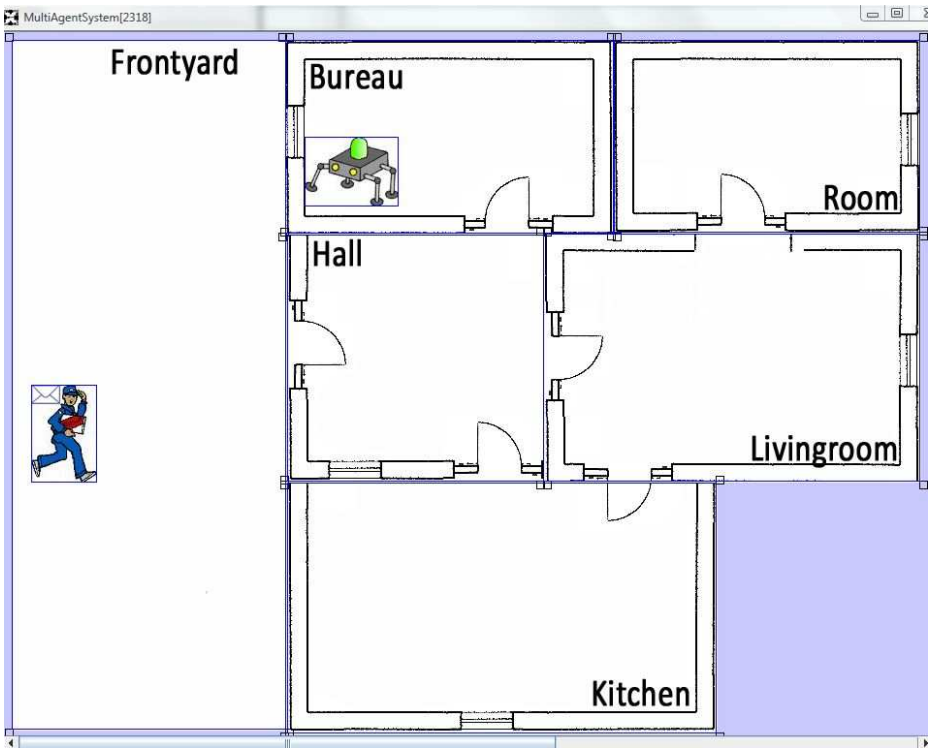


Fig. 6. User view on multi-agent system with agents *Robot* and *Postman*.

Analysis Using the Tool MulanViewer. The CAPA plug-in MulanViewer enables another visualization of the multi-agent system. Due to the implementation of MAPA, as an enhancement of CAPA, the MulanViewer is applicable for MAPA as well. This is demonstrated in the Figures 9, 10 and 11[18].
On the one hand, the MulanViewer displays all set up platforms, agents, decision components and (active) protocols. On the other hand, it may be used as a debugging tool [2,6] during run-time.
The overall system state at startup is illustrated in Figure 9 for the previously specified setting. At platform **Frontyard**, the following agents reside: *Postman#1@Frontyard*, *Frontyard*, *ams@Frontyard* and *Letter#2@Postman#1*[19].
The signifier of the agent *Letter* denotes its residing on the platform **Frontyard**[20]. The implementation of objects as agents themselves, such as the agent

[18] Figure 9, 10 and 11 only display a clipping of the MulanViewer for purposes of clarity.
[19] Within the MulanViewer, the syntax of all listed agents (except the platform agent) complies with the following scheme: <name_agent>#<no>@<name_platform>.
[20] So here we have an implementation of units such as letter as agents. Actually, we have started to soften the four-layer restriction of the Mulan reference model in

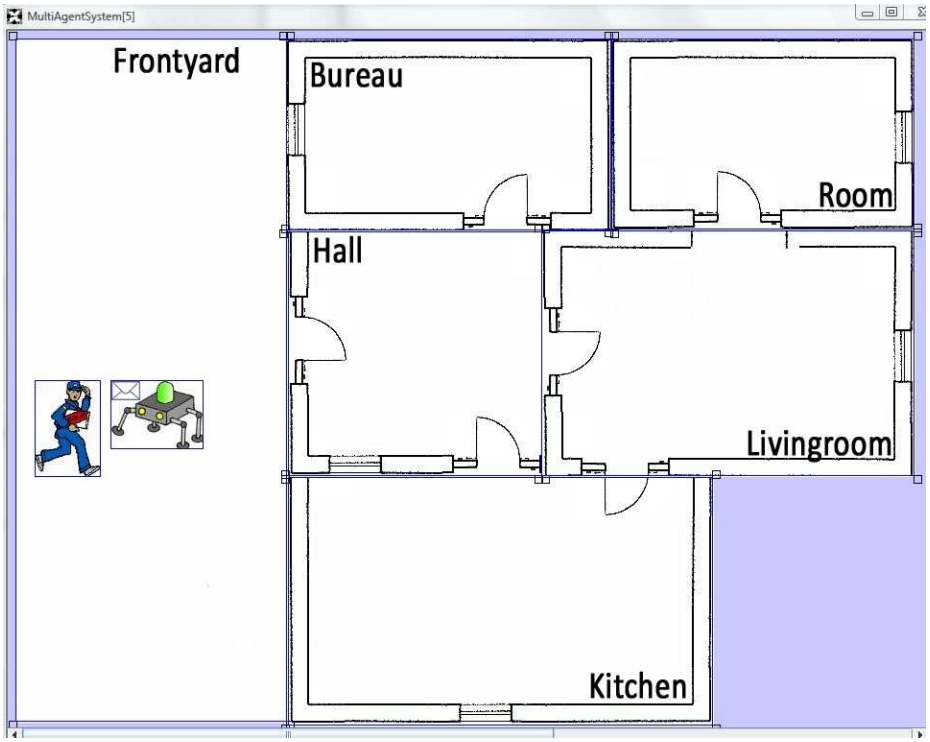


Fig. 7. User view of multi-agent system after exchange.

Letter, results from the consideration to soften the MULAN-reference model concerning its hierarchy in order to transport operational agents. Also, this consideration follows up from the discussion about the topic "Everything is an agent" (see [20, p. 183f])[21]. Nevertheless it is a tentative design choice for the first prototype.

Having arrived at platform **Frontyard**, the agent *Robot* requests the letter from the agent *Postman*. This is illustrated in Figure 10 by the active protocol *moveObject* at agent *Postman*.

Concluding, Figure 11 points out the overall system state at the end of the scenario. Platform **Frontyard** is still hosting the agents *Postman#1@Frontyard*, *Frontyard* and *ams@Frontyard*. In addition, the platform **Frontyard** is now hosting the agent *Robot#1@Frontyard* as well. After processing the protocol

order to be able to regard a specific unit as both an agent and a platform, so that the agent is able to host other agents. This opens up the possibility to build deeper hierarchies of Mulan reference units. However, an in-depth discussion of this topic is beyond the scope of this paper, see [20, p. 183f] and [21] for more details.

[21] The discussion of this topic goes beyond the scope of this paper and is therefore omitted.

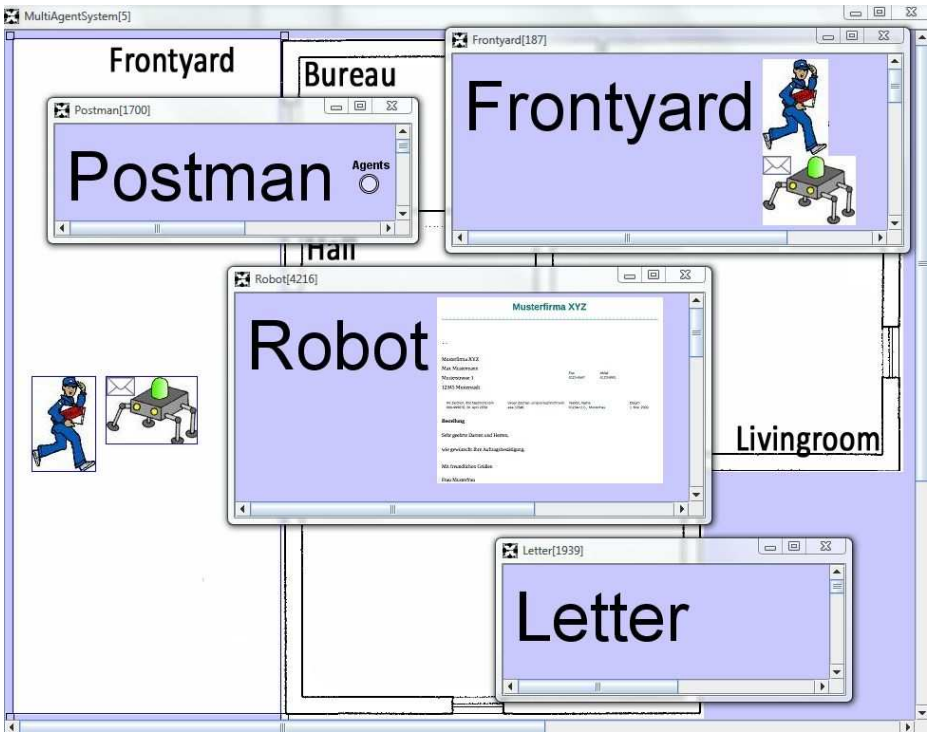


Fig. 8. User view of multi-agent system after exchange with the related net instances.

moveObject, the agent *Letter* has been assigned to agent *Robot*. Due to assignment, the signifier of the agent *Letter* has been changed from *Letter#2@ Postman#1* to *Letter#2@Robot#1*.

Analysis Using the Tool Sniffer. The Capa plug-in Sniffer [2,6] is a monitoring tool in order to log the message exchange between agents. For the given setting, the migration route of the agent *Robot* was traced and is illustrated in Figure 12. Figure 12 shows the filtered[22] trace-log as an Agent Interaction Protocol (AIP) [5,1]. Herein, the migration of the agent *Robot* is illustrated relating to the necessary protocols. Initially, the agent *Robot* (Robot#1@Bureau) sends a "move"-request to its relating AMS (ams@Bureau). The AMS processes the request, sends a "transmit-agent"-request to the remote AMS (ams@Hall) and deregisters the agent *Robot* (Robot#1@Bureau) afterwards. Meanwhile, the AMS at

[22] The filter was set to the keywords "transmit-agent" and "move" in order to refine the generated AIP.

Fig. 9. View on MulanViewer after system initialization.

the platform **Hall** sets up the agent *Robot* (Robot#1@Hall). Due to a particular knowledge base entry, the agent *Robot* (Robot#1@Hall) sends a "move"-request to the AMS (ams@Hall) in turn. The AMS processes the request, sends a "transmit-agent"-request to the remote AMS (ams@Frontyard) and deregisters the agent *Robot* (Robot#1@Hall) afterwards, as well. Meanwhile, the AMS at the platform **Frontyard** sets up the agent *Robot* (Robot#1@Frontyard).

Discussion. As demonstrated by the execution of the case-study within the prototype Mapa, the visual representation of mobility is realizable with a precise separation between a user view and developer view in terms of the Mulan-reference model. As previously mentioned, on the one hand Capa is an enhancement of Mulan concerning the implementation of the Fipa guidelines. On the other hand, Capa loses the representation of the top layer of the Mulan-reference model, the multi-agent system and therefore an appropriate system overview. Due to the consolidation of Mulan and Capa, Mapa is able to represent the vertical nesting of the Mulan-reference model by the usage of image-tokens. Therefore, it illustrates the intuitive association with an agent residing on a platform.
In addition, the representation of multiple agents at one platform can be done without major difficulties, at least if the number of agents is not too large.
As well, the implementation of the routing algorithm of the routing agent turned

Fig. 10. View on MulanViewer before assignment.

Fig. 11. View on MulanViewer after assignment.

out to be effective within different executed settings. If the routing agent computes more than one route between start and destination platform, the first route is chosen in this prototype. Hence, the previously claimed requirements (see Section 3) are successfully fulfilled for this first prototype.
However, the implementation of the prototype and the testing of different settings have pointed out some further issues. First of all, we have to consider an advisable recursion depth for the relationship between platform and agent in the MULAN-reference model, as suggested in [20, p. 156ff]. For instance, if the agent *Postman*, which is residing at the platform **Frontyard**, is transporting a parcel that contains multiple envelopes, each with a letter and another entity, then a visualization of the vertical nesting of all participating entities is most challenging.

5 Related Work

Ambient calculus. The Ambient calculus [7] is a process calculus for describing mobility within a system. An *ambient* is defined as a bounded place where computation happens and which may contain one or more processes and/or sub-ambients that run in parallel within the ambient. A single framework is

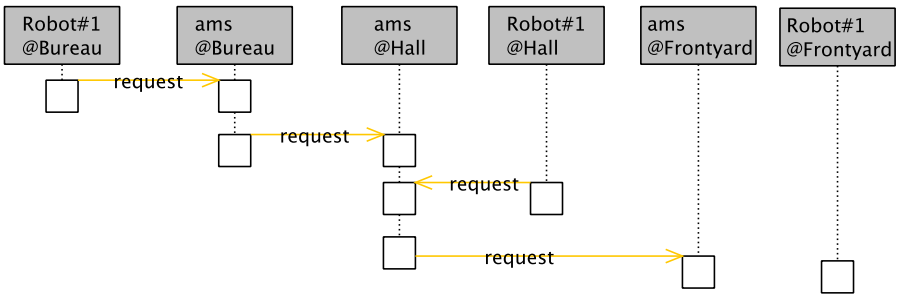


Fig. 12. Filtered message trace using Sniffer.

considered that encompasses movement of ambients and interaction of ambients and processes within ambients. An ambient can move as a unit inside or outside other ambients. Main concepts considered in the Ambient calculus are location, mobility and authorization to move. The calculus models access control and mobility in a way that an ambient may move into or out of a particular ambient only if it possesses an appropriate capability. A process is considered to run inside an ambient. Processes within an ambient may do many activities: cause its enclosing ambient to move; open sub-ambients, which means that a process dissolves an ambient boundary and causes the contents of that ambient to spill into the enclosing ambient; communicate by anonymous asynchronous messages, dropped into the local area. An ability to move and open an ambient is regulated by capabilities that processes must possess by prior knowledge or acquire by communication. A process inside an ambient can be the parallel composition of several processes.
Although the grounded formalism of this paper, namely reference nets, is similar to Mobile Ambients, there are differences. Reference nets, which incorporate the "nets within nets" paradigm, define a model for distributed and mobile processes. The hierarchy of the MULAN-reference model corresponds to the hierarchy of administrative domains of Mobile Ambients. Even though subjective and objective movements are considered in the Ambient calculus as well, the concept of objective movement differs. The difference is when an entity is moved by its environment, it can continue its execution as long as it does not require any resources. The focus of the Ambient calculus is on security and covers the area of process algebra. In our approach concerning MAPA, security is not covered by the formalism of reference nets since it is considered as an application topic that have to be build on top of the introduced case-study[23].

Seal calculus. The Seal calculus [24] is a calculus for mobile computations.

[23] In order to be within the scope of the paper and since there is still an ongoing discussion about security issues, this topic is not covered within this paper.

It represents an extension of the π-calculus with mobility, distribution and protection mechanisms. A seal is defined as a named, hierarchically structured location where computation happens. The seal may contain one or more processes and/or sub-seals. The concept of control access to resources is considered. It is based on portals. Portals are defined as linear revocable capabilities, where linear means that an agent is allowed to use a given resource only once. A hierarchical protection model is proposed: a seal controls all external resources that are used by processes and/or sub-seals residing within the seal. Seals may move (enter/leave) only with the explicit permission of its parent seal. This kind of movement is equivalent to objective movement touched in the paper. Communication between seals is restricted within a single level of hierarchy and is under control of parent seal. A seal may protect itself from parent seal by controlling visibility and access to its own resources (via portals). The model of Petri net for mobility is similar to JavaSeal, a mobile agent platform [24], in the following. In the JavaSeal each agent executes within a protection domain and cannot directly view or modify the data of another object. Protection domains are structured in hierarchy. Communication between objects is based on channels. An agent protection domain can only send a message to its parent domain or to its children. A message sent between two domains is routed through the ancestor of the domains. The main difference between the Seal calculus and reference nets is in the subjective movement that is not allowed within the Seal calculus.

IMAGO Prolog. Inter-agent communication in IMAGO Prolog [18] is exclusively based upon so-called messengers. These messengers are thin, mobile and anonymous agents which aim a reliable message delivery. Thus, they are dedicated to track receiving agents within a dynamic environment. The used agent communication language is based on first order logic regarding Prolog. In order to exchange data or synchronize, the messengers use logic terms and unification. IMAGO Prolog supports (a)synchronous messaging as well as broadcasting and multicasting. Hence, participating agent servers have to provide information about the location of every agent.
In contrast to IMAGO Prolog, the prototype Mapa uses the Fipa-ACL. The inter-agent communication within one platform is realized by the internal Message Transport Service. Due to the fact that Mapa is an enhancement of Capa the overall platform communication is possible as well[24]. Therefore, Mapa does not require specialized agents for communication. But the first prototype of the framework Mapa lacks the ability to track receiving agents as IMAGO Protolog's messengers provide. Nevertheless, this is a feature which has to be discussed and for what IMAGO Prolog provides helpful suggestions.

6 Conclusion

As stated in this paper, the prototype of the framework Mapa is a substantial progress towards the visual representation of mobility within the user view.

[24] For a detailed description of the overall platform communication see [8].

Beside the dynamical computation of routes as a major feature of the framework, the visual representation of mobility succeeded by the implementation of a vertical-hierarchical structuring of image-tokens. As well, the embedding of the prototype into the simulation engine Renew provides flexibility in the development of new applications as much as possible.
Despite the convincing illustration of mobility in the context of the prototype Mapa, some issues are still unsolved. The co-existence of multiple robot agents remains unexplained at the multi-agent system presented in Section 4. In addition, an applicable implementation of an effective order maintenance for any operative, mobile agent within the application is necessary. This is required in order to avoid redundant commissioning. In order to gain performance, efficiency and reduction of network traffic in terms of a minimized movement of the operating agents, the movement of an idle agent has to be discussed as well. Therefore, the implementation of a diagnostic tool-kit would be advantageous. This tool may be used in order to locate possible bottle-necks within the usage of mobile agents.
Since the prototype is simply defined and implemented within a local and private system, the problem of connecting remote platforms remains unsolved. I.e. the enhancement to an open network has to be developed. Also, the representation of the movement of a mobile agent between distributed platforms or multi-agent systems poses still a challenge. Not least, this requires an examination of relevant security aspects. If an agent is transporting an entity and wants to migrate to another platform, the agent has to remain in quarantine, for instance, until the content has been successfully checked.

References

1. Lawrence Cabac. Generating code structures for Petri net-based Agent Interaction Protocols using Net Components. In *Workshop: Algorithms and Tools for Petri Nets*, September 2003.
2. Lawrence Cabac and Till Dörges. Tools for testing, debugging and monitoring multi-agent applications. In Daniel Moldt, Fabrice Kordon, Kees van Hee, José-Manuel Colom, and Rémi Bastide, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, pages 209–213, Siedlce, Poland, June 2007. Akademia Podlaska.
3. Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Agent technologies for plug-in system architecture design. In *Proceedings of the Workshop on Agent-oriented Software Engineering (AOSE)*, Utrecht, Netherlands, 2005.
4. Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Benjamin Schleinzer. Plugin-agents as conceptual basis for flexible software structures. In *Multi-Agent Systems and Applications V. Fifth International Central and East European Conference, CEEMAS'07, Leipzig. Proceedings*, volume 4696 of *Lecture Notes in Computer Science*, pages 340–342, Berlin, Heidelberg, New York, 2007. Springer-Verlag.
5. Lawrence Cabac, Daniel Moldt, and Heiko Rölke. A proposal for structuring Petri net-based agent interaction protocols. In Wil van der Aalst and E. Best, editors, *24th International Conference on Application and Theory of Petri Nets, Eindhoven, Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 102–120. Springer-Verlag, June 2003.

6. Lawrence Cabac, Daniel Moldt, and Jan Schlüter. Adding runtime net manipulation features to mulanviewer. In *15. Workshop Algorithmen und Werkzeuge für Petrinetze, AWPN'08*, volume 380 of *CEUR Workshop Proceedings*, pages 87–92. Universität Rostock, September 2008.
7. Luca Cardelli and Andrew Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, pages 140–155. Springer, 1998.
8. Michael Duvigneau. Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2002.
9. Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering III. Third International Workshop, Agent-oriented Software Engineering (AOSE) 2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions*, volume 2585 of *Lecture Notes in Computer Science*, pages 59–72, Berlin, Heidelberg, New York, 2003. Springer-Verlag.
10. FIPA. Foundation for Intelligent Physical Agents. Specifications. http://www.fipa.org, 2001.
11. Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods, and practical use*. Springer, 1992.
12. Eike Jessen and Rüdiger Valk. *Rechensysteme: Grundlagen der Modellbildung*. Studienreihe Informatik. Springer-Verlag, Berlin, Heidelberg, New York, 1987.
13. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the structure and behaviour of Petri net agents. In J.M. Colom and M. Koutny, editors, *Proceedings of the 22nd Conference on Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 224–241. Springer-Verlag, 2001.
14. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling mobility and mobile agents using nets within nets. In Wil van der Aalst and Eike Best, editors, *Proceedings of the 24th International Conference on Application and Theory of Petri Nets 2003 (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 121–139. Springer-Verlag, 2003.
15. Olaf Kummer. Introduction to Petri nets and reference nets. *Sozionik Aktuell*, 1:1–9, 2001. ISSN 1617-2477.
16. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
17. Olaf Kummer, Frank Wienberg, and Michael Duvigneau. *Renew – User Guide*. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, release 2.1.1 edition, 2008. Available at: http://www.renew.de/.
18. Xining Li and Guillaume Autran. Inter-agent communication in IMAGO prolog. In *Programming Multi-Agent Systems*, pages 163–180. 2005.
19. Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc. New York, NY, USA, 1985.
20. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
21. Benjamin Schleinzer. Flexible und hierarchische Multiagentensysteme – Modellierung und prototypische Erweiterung von Mulan und Capa. Diplomarbeit, Universität Hamburg, Department Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2007.
22. Jörn Schumacher. Eine Plugin-Architektur für Renew – Konzepte, Methoden, Umsetzung. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg, October 2003.

23. Rüdiger Valk. Object Petri Nets – Using the Nets-within-Nets Paradigm. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets: Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer-Verlag, Berlin, Heidelberg, New York, 2004.
24. Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. *Lecture Notes in Computer Science*, 1686:47–77, 1999.
25. M. Wooldridge and N. R Jennings. Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152, 1995.

A Centralized Petri Net- and Agent-based Workflow Management System

Thomas Wagner

University of Hamburg
Faculty of Mathematics, Informatics, and Natural Sciences
Department of Informatics
http://www.informatik.uni-hamburg.de/TGI/

Abstract. In this contribution a prototypical agent-based workflow management system is presented. It was modeled using the Mulan and Capa agent architectures and uses workflow nets to realize workflow logic. The system will be described in detail, including the different agent types, the user interface and an example of how the system is used. Concluding the contribution will be a short look at future developments of the system and related work.

1 Introduction

Business process management (BPM) is an important part of any modern company. Generally a business process (BP) is a process inside a company, which, directly or indirectly, adds to the production of goods and services. Companies are very interested in executing their BPs in an efficient way in order to minimize the time needed to complete a process, so that costs can be reduced and profits can be raised. An obvious way to efficiently support the execution of BP is to use computer systems to assist in the execution and, if possible, automate parts of the process, like the transmission of data between steps or the coordination of work units. If a BP, or just a part of a BP, is facilitated or automated by a computer system, this is defined by the Workflow Management Coalition (WfMC), as a workflow ([5]). Advantages gained through the computerization and automation of BP through workflows include higher efficiency and productivity in the execution of BPs and the ability to analyze and optimize BPs through the use of formal techniques and methods ([15]).

In order to fully realize the potential of BPs and workflows, computerized environments must exist that support the creation, administration, maintenance and execution of workflows. These environments are called workflow management systems (WFMS). They take care of the distribution and assignment of work units, as well as providing necessary information to the user.

One way to model a WFMS is described in this contribution. The core functionality of the system is provided on one computer, but it is possible for users to start a client on their computer and log into the system via network access. The WFMS uses a Petri net formalism similar to the workflow nets introduced

in [14] to model the processes and a multi-agent system built on the Mulan and Capa architectures to implement the WFMS functionality.

The reason these technologies were chosen is that the system is aimed at providing a technological foundation for future research and development concerning agents and workflows. These developments will be based upon the theoretical architecture introduced in [11] and [10] and will be shortly described in a later section of this paper. Since this theoretical architecture is modeled with Mulan, Capa and the variation of workflow nets in mind, the selection of these particular technologies for the WFMS is obvious.

The WFMS was developed in a project course at the University of Hamburg and is currently still in a prototypical stage, so it does not yet offer the full functionality that is needed to be used in real-life practical applications. It does however already offer basic workflow functions and will be extended in the future. A more detailed description of the WFMS can be found in [16].

The paper is divided up into five sections. Section 2 gives a short general overview about the technologies used in the creation of this paper's prototype. Section 3 gives some examples of and comparisons to other agent-based WFMS. Section 4 presents the main contribution of this paper and contains the description of the prototype. In Sect. 5 a short practical example for the WFMS is given to illustrate the use of the system and Sect. 6 describes the plans and possibilities for future enhancements of the system. The connections to related work will also be described in that section.

2 Technological Foundations

The WFMS presented in this paper is built using the Mulan and Capa agent architectures. Mulan is an acronym for **Mul**ti-**a**gent **n**ets, which precisely sums up the main idea behind it. It was presented in [12]. Every element of Mulan is modeled using the reference Petri net formalism introduced in [8], which follows the nets-within-nets paradigm like the object nets introduced in [13]. To realize communication between nets synchronous channels are used. These channels are used to synchronize actions between different nets, as well as transmit information. They are similar to the channels described in [2]. All agents share the same net structure, but are distinguished by their knowledge and behavior. Agent behavior is modeled through so called protocol nets, that can be started as a reaction to incoming messages or proactively by the agent itself. Protocol nets make use of the net components introduced in [1]. Capa (**C**oncurrent **A**gent **P**latform **A**rchitecture) is an extension of Mulan, which was introduced in [3]. It is mostly focused on platform external communication and making the Mulan principles fully compliant with the standards of the Foundation for Intelligent Physical Agents (FIPA). The reference net tool Renew serves as both development and runtime environment for the WFMS. A description of Renew can be found in [8] and [9].

Within the WFMS workflows are modeled using a variation of the workflow nets described in [14]. This variation of workflow nets uses the task transition

from [6]. This task transition is represented like a regular transition with two thick sidebars. It actually represents three regular transitions and a place. The three transitions model the request of a work item and the cancellation or confirmation of an activity, while the place indicates that the activity is currently being worked on.

3 Other Agent-based WFMS

This section will give a short description of two other agent-based WFMS developed by other researchers. The goal of this section is to put this paper's WFMS, described in the next section, into a wider context and to show other approaches to the provision of WFMS functionality through multi-agent technology.

The ADEPT WFMS described in [7] uses so-called agencies to execute workflows. These agencies consist of one "responsible agent", a number of "subsidiary agencies" and a set of tasks and resources that the responsible agent controls. The agencies have full control over their own tasks and resources, so the only way interactions between different agencies work is through negotiation for services. A service in this context is (part of) a workflow, that is being managed within the system and can be provided by multiple agencies and their subsidiary agencies. All agents in ADEPT possess the same structure and consist of a number of modules that are responsible for communication, message routing, negotiations, monitoring, task execution, et cetera. The ADEPT system differs greatly from the system described in this paper. The biggest difference is that it only uses one kind of agent to provide WFMS functionality. The agencies are responsible for every aspect of the execution of a workflow, including the negotiation and execution of tasks through services. In contrast to that this paper's WFMS uses several different agent types, all with different responsibilities, and offers task functionality through the user interface.

The JBees WFMS described in [4] uses a number of different agent types to provide WFMS functionality. The execution of workflows is handled in the following way: Management agents, which also offer the administration interface, initiate new workflow instances by creating new process agents. These process agents gather the necessary data from other agents within the system. A resource broker agent then allocates resource agents to that process. Resource agents represent the resources in the system and provide the interface for human users. These allocated resources then execute the process. Other agents in the system are responsible for storing information and monitoring and controlling the system. The JBees WFMS structure is more similar to the WFMS described in this paper than the ADEPT system. It uses different agent types to provide WFMS functionality, but uses a different division of responsibilities. The use of agents to encapsulate processes is not yet implemented into this paper's WFMS, but is planned for the future (see Sect. 6).

4 Description of the Prototype

The entire functionality of the WFMS is provided by a number of agents active within the system. These agents are all responsible for one aspect of the WFMS and will be presented in detail later in this section. Different actions between the agents are modeled as a number of interactions that involve multiple protocols in multiple agents. The interactions will be described at the end of this section. Information needed for workflow execution is stored in a database that can be accessed and edited through an administration interface. The tables of this database contain information about users, roles, workflow definitions, tasks, forms, et cetera. When the system is started the necessary agents are set up and the system is ready for users to log into. This can happen locally (on the same computer the system's core is executed on), but also remotely through a network. When an user is successfully logged into the system he can, depending on his role, instantiate workflows, request and work on tasks or access administration functions.

Workflows are represented as workflow nets. For each new workflow instance a new workflow net is created and maintained by the workflow engine. Available work items are displayed and users can choose which work items to request. There are currently two types of tasks within the system. Simple tasks can merely be requested and then confirmed or canceled by the user. They represent tasks, which are not supported by functions of the WFMS and have to be completed manually by the user. Form tasks on the other hand open a new window when accepted, in which the user can enter data. Forms are attached to tasks through inscriptions in the workflow net and consist of an arbitrary number of labels, text boxes, check boxes and radio buttons which are aligned vertically. A typical form window can be seen in Fig. 5. When the user finishes a form task, the information from the form window is transmitted to the system and can be used in later tasks. The information is stored as a form object which can be transformed for another task with a different form by using form transformation rules. These rules are also stored in the database and define which data from an old form will appear in which position of a new form. Tasks themselves are derived from task definitions within the database, which contain a description of the task, as well as information about which users are allowed to execute a task. When a workflow instance reaches the end of its execution the workflow net is removed from the workflow engine.

The user interface of the WFMS, shown in Fig. 1, provides the basic functions for all user roles. It is the same for all roles, but certain functions are not available for particular roles (i.e. workflow executors are not allowed to edit database tables or initiate workflows). The main interface contains two areas. The upper one lists the currently available work items and the lower area lists the current activities of an user. The user can request work items, instantiate workflows, confirm or cancel activities and enter the administration window by pressing the corresponding buttons in the main interface.

The WFMS consists of eight types of agents. Fig. 2 gives an overview about the different agents of the system. Except for the user agents, all of the agents are

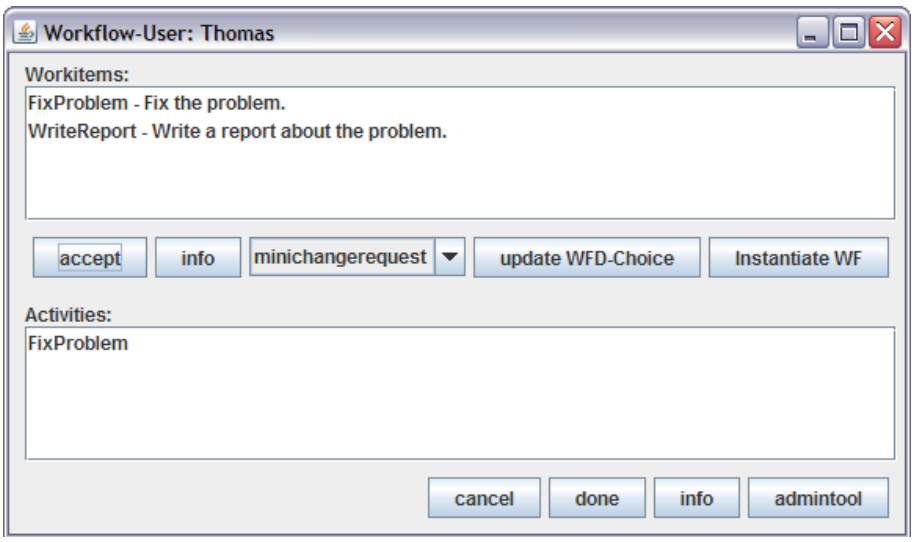


Fig. 1. User interface of the WFMS

unique in one instance of the WFMS. The division of the WFMS functionality was chosen for two reasons. On the one hand this division is very natural, as the agents responsibilities do not overlap and every agent is responsible for a distinct area of the functionality. On the other hand parts of the division were chosen to correspond to elements of the WfMC's workflow reference model described in [5]. It should be noted that the WFMS does follow the reference model but does not yet provide all the interfaces or integrate them fully into one unit. Interface 1 (Process Definition Tools) is given by Renew, which serves as the environment to model workflow nets. Interfaces 2 (Workflow Client Applications) and 5 (Administration & Monitoring Tools) are modeled through the user interface of the WFMS and tools provided by Renew, which allow monitoring the system's internal state. Interfaces 3 (Invoked Applications) and 4 (Other Workflow Enactment Services) are not yet implemented. The eight individual agent types will now be described in detail.

WFMS Agent. This agent serves as a kind of container for all the other agents of the WFMS. It is responsible for starting and setting up all other agents within the system. During its initialization the *setupAgent* interaction is called, which creates the other agents, except for user agents, which are only created when an user wishes to log in. At the end of this interaction the WFMS is completely setup and ready for users to initiate and execute workflows. Besides these setup duties this agent also serves as the WFMS's interface, which user agents use to log into and out of the system. It can also be queried for the individual agent's addresses in the system, of which it holds a registry.

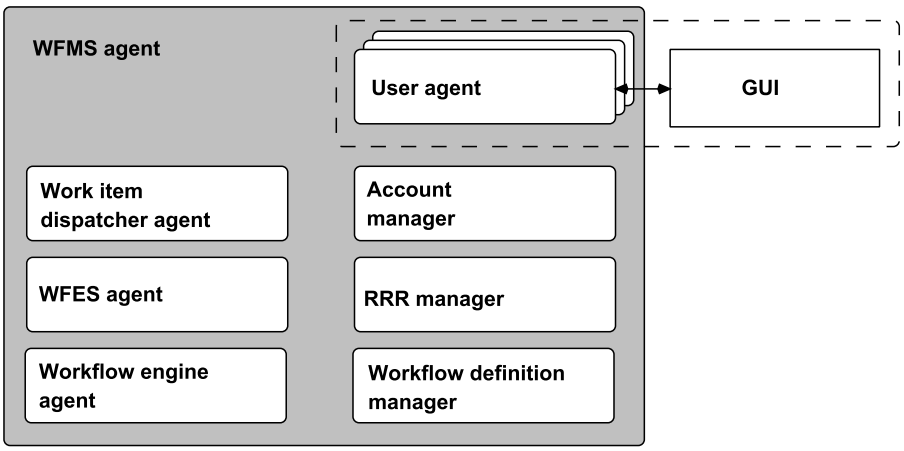


Fig. 2. Overview over the WFMS agents

User Agent. The user agent serves as an interface between the WFMS and one particular user. As mentioned before, this agent is not started along with the rest of the WFMS's agents, but is only created when an user wishes to log into the system. For each user a new user agent is instantiated. The agent then requests to be logged into the system through the WFMS agent and, if the user's credentials (username and password) are valid, connects to the work item dispatcher within the system, to receive work items and activities. Every action the user performs in the graphical user interface (GUI) initiates an interaction, which is started by the user agent. This includes the initiation of workflow instances, requests for work items, the confirmation or cancellation of activities and administrative access to the database (if the user has the required privileges). The user agent is also involved in interactions that are aimed at providing information about the current state of the system to the user. These include updates to the currently available work items, the user's current activities and information for the user about the end of a workflow instance he initiated. This means that this agent is involved in many of the WFMS's interactions, though its role is mostly limited to initiating an interaction or receiving some kind of data to display in the GUI. The connection between the agent and the Java GUI is modeled through a decision component (DC). In general a decision component is a constantly active protocol Petri net within an agent, which can communicate with other active protocols via synchronous channels as it is the usual communication within an agent for all its components. This particular DC transforms and handles the data which is exchanged between the GUI and the agent.

Workflow Enactment Service (WFES) Agent. The WFES agent is responsible for managing the execution of workflows within the WFMS. It can be viewed as an interface between the internal execution of workflows by the workflow engine and the external (towards the user) management of work items and activities by

the work item dispatcher. Its main duty is to accept incoming requests from the user (via the work item dispatcher) like the instantiation of a workflow or the confirmation or cancellation of an activity and forward them to the workflow engine, while maintaining its own information about the current state of the WFMS. Theoretically it is also responsible for distributing the different workflow instances on the different workflow engines, but since there is only one workflow engine active in the current version of the WFMS, this aspect, though already partially modeled through the *chooseWFE* interaction, does not yet apply. The WFES agent mostly corresponds to the WFES element of the WfMC's reference model. It does not contain the actual workflow engine(s) but does manage them.

Workflow Engine Agent. Workflow engine agents realize the functionality of workflow engines according to the reference model of the WfMC ([5]). As such, they are responsible for the actual execution of workflow instances. Currently only one workflow engine agent is active in one instance of the WFMS at any time, but the system can easily be extended to contain multiple workflow engines. Workflow engine agents have some very important functions within the WFMS. As the workflow execution units, they are responsible for instantiating the workflow nets representing a workflow instance and for removing them when execution has finished. During the initiation of a workflow instance, the workflow engine also registers itself as a listener with this instance. The agent's decision component then uses synchronous channels to detect changes in the available work items and current activities, in order to initiate update interactions with other agents, if they are necessary. The firing of the internal transitions of a task transition is also handled by the workflow engine agent. As stated before, the task transition used in the system's workflow nets consists of three individual transitions that represent the assignment/request of work items and the confirmation or cancellation of an activity. When the interactions modeling this behavior are started by an user agent, the work item dispatcher informs the WFES agent, which in turn informs the workflow engine. In this last step the internal transitions are fired by the workflow engine, after which confirmations or disconfirmations are sent to the other agents depending on whether or not the firing was successful.

Work Item Dispatcher Agent. The work item dispatcher agent is responsible for the distribution of work items and activities to the users. During login every user agent registers with this agent and begins, if the user has appropriate roles, receiving updates about available work items and requested activities. Whenever the state of work items or activities within the system changes, the workflow engine informs the work item dispatcher (via the WFES agent). The work item dispatcher then updates his internal lists, as well as the lists of the appropriate users.

Account Manager Agent. The account manager agent is designed to manage access to the WFMS and maintain a directory of all logged-in users. It is mainly responsible for validating if an user has a queried role or privilege. This is done

through the *authenticate* interaction. This interaction can be initiated by any agent that wants to know if an user is allowed to do what he tries to do. For example the work item dispatcher agent has to check the credentials of an user before a work item request can be processed further.

Rules Roles Rights (RRR) Manager Agent. This agent is responsible for the database tables about users, roles and task execution rules. Within the system there are two different types of roles an user has. The first type is the user role. This describes the general function the user has within the system. These user roles are the same for every workflow or application created with the WFMS. Currently there are three different user roles and users are allowed to be assigned more than one. Workflow administrators are responsible for managing and maintaining the system, as well as the database. Workflow initiators have privileges to initiate workflow instances. Workflow executors are responsible for actually working on and completing tasks within a workflow instance. The second type of role is the application role. This is unique to every workflow or application created with the WFMS (though it can of course be reused). An application role defines which tasks in particular a workflow executor is allowed to request. For example application roles can be based on different departments of a company that are all involved in one workflow. The connection between application roles and tasks is made through task execution rules. Every task definition contains the identifier of such a rule. The rules consist of the set of application roles that are allowed to request the tasks that possess this rule. The only interactions this agent is involved in are database access interactions concerning the tables it is responsible for (*dbEdit* interaction) and the interaction to retrieve the set of rules a particular user satisfies (*getSatisfiedRules* interaction).

Workflow Definition Database Agent. This agent is responsible for accessing the database's table for workflow definitions. It is called only to supply the workflow's definition when a new workflow is instantiated, and when the tables are edited. In future enhancements to the WFMS, this agent could for example also be used to handle workflow adaptivity functionality.

The work cycle within the WFMS can be illustrated by following the order in which the user roles become active. At first a workflow administrator has to input the workflow definition into the system's database. This includes the workflow net, appplication roles, the task definitions, task execution rules, form definitons and form transformation rules. When all the necessary data is available in the system, a workflow initiator can start an instance of the workflow. After the instantiation has completed work items, which are activated in the workflow net, are offered to the eligible workflow executors logged into the system. These executors can then request available work items and execute the corresponding activities. When the workflow instance reaches the end of its execution the original workflow initiator is informed about this, which completes the execution of this workflow instance. Currently there is no way for workflow instances to access

results from previous/other workflow instances. This functionality is planned for future versions of the system.

To complete and conclude the description of the WFMS the different interactions of the system will now be described. Figure 3 gives an overview of how the interactions are connected to the agents. The rows represent the interactions and the columns represent the agents. A "X" in the figure means that the column's agent is involved in the row's interaction. The detailed description of the interactions will now follow.

	WFMS Agent	WFES Agent	WFEngine Agent	Work Item Dispatcher	User Agent	Account Manager	RRR Manager	WF Def. Database
login	X				X	X		
logout	X				X	X		
connectToDispatcher				X	X			
disconnectFromDispatcher				X	X			
authenticate						X		
getSatisfiedRules				X			X	
setupAgent	X	(X)	(X)	(X)		(X)	(X)	(X)
dbEdit	X				X	X	X	X
instantiateWorkflow		X	X		X			X
chooseWFE		X						
updateWorkitemList		X	X	X				
offerWorkitemList				X	X			
updateActivityList		X	X	X				
offerActivityList				X	X			
requestWorkitem		X	X	X	X			
confirmActivity		X	X	X	X			
cancelActivity		X	X	X	X			
workflowEndReached		X	X		X			

Fig. 3. The relationship between agents and interactions (Modified from [16])

Login Interaction. The *login* interaction models the behavior of the system when an user wishes to log in. A newly instantiated user agent requests to be logged in at a WFMS agent. The WFMS agent then queries the account manager agent, which checks if the user's username and password are correct and if he is already logged into the system. If the account manager confirms the user's login request, the WFMS agent completes this interactions.

Logout Interaction. The *logout* interaction models the process of logging an user out of the system. When the user closes the main GUI window, this interaction

is automatically started. Basically it removes the user agent from the internal lists of the WFMS agent and the account manager agent.

ConnectToDispatcher Interaction. The *connectToDispatcher* interaction is automatically called, when an user with the workflow executor user role is logged into the system. It registers the user agent with the work item dispatcher agent, which can then begin to offer work items and activities to the user. During the course of this interaction the *authenticate* and *getSatisfiedRules* interactions are called to verify the user and determine the task execution rules he satisfies respectively.

DisconnectFromDispatcher Interaction. When a workflow executor user logs out of the system the *disconnectFromDispatcher* interaction is automatically called to remove him from the work item dispatcher's internal list of connected users.

Authenticate Interaction. The *authenticate* interaction models the authentication of an user within the system. The initiator of this interaction, which can be any agent that needs to authenticate an user, sends a message to the account manager agent containing the unique agent identifier of the user agent, the user's credentials and the privileges, identified by a set of roles, this user is supposed to have. The account manager then tests if the user's credentials are valid, if the user is currently logged into the system and if the roles are correct. It then sends a confirm or disconfirm message to the initiator.

GetSatisfiedRules Interaction. In this interaction the RRR manager agent accesses the database to retrieve and determine the set of task execution rules a given user satisfies. This set of rules is then returned to the initiator of the interaction, which is currently only the work item dispatcher during the *connectToDispatcher* interaction.

SetupAgent Interaction. The *setupAgent* interaction is started automatically when the system is started and the WFMS agent is initiated. Basically the WFMS agent starts all other agents of the system (except user agents) and stores their addresses in internal lists.

DbEdit Interaction. The *dbEdit* interaction implements the database access (read or write) of the system. It is started from the administration window. The user agent of the administrator sends the access request to the WFMS agent, which initiates an authentication of the user and then redirects the request to the agent in charge of the particular table of the database. That agent then accesses the database and returns the result to the user agent.

InstantiateWorkflow Interaction. The *instantiateWorkflow* interaction models the instantiation of a new workflow instance. A workflow initiator's user agent requests the instantiation from the WFES agent. The WFES agent then gathers the workflow net definition from the workflow definition database agent and

forwards this definition to the workflow engine agent. The workflow engine agent uses this to create a new workflow net instance and registers as a listener with this net in order to be able to react to changes (e.g. newly activated work items) in that net.

ChooseWFE Interaction. The *chooseWFE* interaction will model the selection of one of multiple workflow engines active in the system. Currently it is used in the *instantiateWorkflow* interaction by the WFES agent, but only returns the one active workflow engine of the system. This will be extended in the future.

UpdateWorkitemList Interaction. The *updateWorkitemList* interaction is started whenever the available work items have changed in any way (i.e. an activity has been requested, confirmed or canceled or a new workflow has been instantiated). The changes are detected by the workflow engine which determines the exact set of changed work items and initiates this interaction. The workflow engine informs the WFES agent and the work item dispatcher, which distributes the work items to eligible users by starting the *offerWorkitemList* interaction.

OfferWorkitemList Interaction. The *offerWorkitemList* interaction models the distribution of work items to the user. It merely accepts the message send by the work item dispatcher to a particular user agent at the end of the *update-WorkitemList* interaction and forwards it to the DC responsible for the GUI connection. The DC then initiates the display of these work items in the GUI.

UpdateActivityList Interaction. The *updateActivityList* interaction is the equivalent of the *updateWorkitemList* interaction for activities. Whenever the activities within the system change in any way (i.e. an activity is successfully requested, completed or canceled) the workflow engine detects these changes and initiates this interaction. As in the *updateWorkitemList* interaction the WFES and work item dispatcher agents are informed and the latter informs the users of their activities by initiating the *offerActivityList* interaction.

OfferActivityList Interaction. As with the *offerWorkitemList* interaction the *offerActivityList* interaction merely accepts the message containing a particular user's current activities and forwards it to the DC, to be displayed in the user's GUI.

RequestWorkitem Interaction. The *requestWorkitem* interaction models the behavior of the system when an user requests a work item. Basically the user agent requests the work item from the work item dispatcher, which initiates the *authenticate* interaction and, if the user was successfully authenticated, forwards the request to the WFES agent. The WFES agent then tasks the workflow engine to fire the task's internal transition that represents the request of the work item. If every step of this interaction is successful the activity is assigned to the user and update interactions are automatically started.

ConfirmActivity Interaction. The *confirmActivity* interaction implements the successful completion of an activity. Internally it works the same way as the *requestWorkitem* interaction, only that it fires the internal transition that represents the completion of the activity.

CancelActivity Interaction. The *cancelActivity* interaction models the cancellation of an activity. It follows the same internal structure as the *requestWorkitem* and *confirmActivity* interactions, but fires the internal transition representing the cancellation of the activity. This causes the workflow net to revert (locally) to the state before the activity was requested.

WorkflowEndReached Interaction. The *workflowEndReached* interaction is automatically started when the end of a workflow instance is reached. It removes the references to the workflow instance from the WFES agent and informs the initiator of the workflow that it has been completed.

5 Example Workflow

To exemplify the use of the WFMS, a short example of how to use it will now be given. The workflow net for this example is shown in Fig. 4. It represents a very simple process that models the behavior of a kind of technical hotline. Costumers of a product call this hotline to report bugs or other problems with the product. Call center employees then gather all the necessary data that is needed by the developers to fix the problem. After the problem is fixed (also if it couldn't be fixed) the developer will write a report to finish the process.

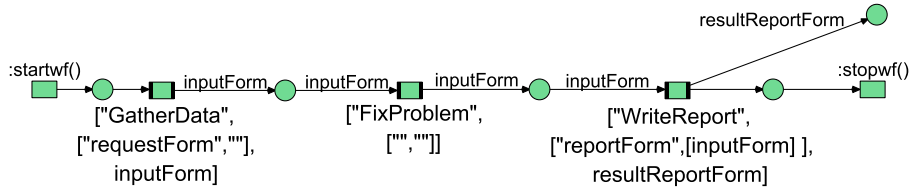


Fig. 4. Example workflow net

The workflow net of Fig. 4 consists of three different tasks. The first task, called "GatherData", represents the call center employee gathering all the necessary data. It is a form task, that opens the form "requestForm", in which the employee enters the data. This form is shown in Fig. 5. It contains one label "New Request" and three text boxes. The call center employee enters a request number, the name of the user and a description of the problem. He can cancel the task by pressing the "Cancel" button or finish the task by pressing the "OK"

button. When the task is completed the data from the form window will be used to create the result object *inputForm*. The second task, called "FixProblem", is a simple task and models the developer's work on the problem. This task does not have a result parameter. The *inputForm* object is merely passed through the task transition. While it is not explicitly supported by the system, the information in this object is available to the developer during the execution of the task. The final task is called "WriteReport". It models the last part of the process, in which the developer writes a report about the problem, what he did to fix it and if he was successful. It uses the data from *inputForm* as input data for its own form called "reportForm". When the task is finished a new object *resultReportForm* is created with data from the form window. The process is then finished.

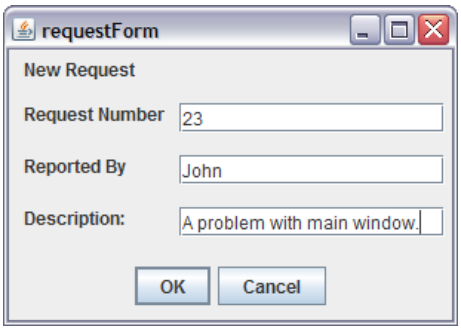


Fig. 5. A typical form window

In this example the system is designed to work in the following way. There are two different application roles: Call center employee and developer. Call center employees are workflow initiators and executors, while developers are only workflow executors. When a call center employee takes a call and ascertains that the problem has to be looked into by a developer, he initiates a new workflow instance. Since in this case he is also the executor of the first task the work item is displayed for him and he requests the work item and begins to fill out the form that is automatically opened. When he is done he closes the form window and confirms the task. The second task then appears on the developer's work item list. He can request it and begin to fix the problem with the (externally) provided information. When he is done he confirms this task and the final task appears in his work item list. He then requests this work item, writes the report into the form and confirms the task. This finishes the process.

A snapshot of execution of this process is shown in Fig. 1. This figure is taken from an user with the developer application role. It shows that the developer is currently fixing a problem, which is shown by the "FixProblem" entry in his activity list, and can request another problem to fix and one report to write, which is shown by the two entries in his work item list.

6 Future Developments and Related Work

Further enhancements to the system will mostly aim at providing a higher degree of distribution. Currently, the agent technology used in the system is not yet used to its full potential. The agents provide an explicit structure to the system, by clearly encapsulating the different parts of the WFMS, but agent principles do not greatly contribute to the system. This will change with the introduction of the agent-workflow. This agent will encapsulate a single workflow instance, which is being executed in the system. In contrast to the current system, it will be possible for workflows using agent-workflows to be executed on different instances of the WFMS, thus turning the centralized version described in this contribution into a much more distributed system.

One way to model this is to use an agent, which is basically responsible for the execution of the structure of the workflow. The agent executes his own version of the workflow net, but for every task he activates, he starts a new workflow instance locally (i.e. the way it is done in the current version) on the WFMS instance the task should be executed on. After completion, the agent receives the result to complete his own version of the task. These local workflows can be generic mini-workflows, which always have the same structure and only change the one particular task, or complex sub-workflows. It is also possible to extend the system even further and have the sub-workflows be executed by agents as well to create a complex hierarchy.

There are more ways to use the agent technology already in place to improve on the system. For example mobile agents can be used to make the system more flexible or autonomous agents can be used to automate certain parts of a workflow or even the entire system. It is also possible to exploit Petri net features in order to enhance the system. For example the workflow nets could be analyzed and optimized in order to raise the efficiency of execution or the workflow nets could be used to visualize information for the workflow executors about their current process.

Of course there are many more areas, like usability or efficiency, which can be improved in future versions. For example the display of work items and activities could be extended to show information specific to a particular work item or activity. At the moment, only the general information of work items or activities is displayed (as seen in Fig. 1).

Strongly related to this contribution is the work presented in [11] and [10]. In it, an architecture is introduced that aims to merge both workflow and agent technology. This architecture consists of five stages, with each stage being based on the previous one. Starting from pure WFMS and agent management systems on stage one, each consecutive stage merges the technologies more, until in the final stage both technologies are completely integrated into each other and both are fully available to the user. The WFMS described in this contribution represents a practical implementation of a system, that can be classified as lying somewhere between the second and third stage of this theoretical architecture. The implementation of the third and fourth stages are the topic of the author's

diploma thesis. This paper's WFMS serves as the technological foundation for that thesis.

7 Summary

In this contribution a centralized workflow management system based on Petri nets and multi-agent technology was presented. After giving a short overview of the particular technologies that were used in the creation of the WFMS and describing some examples of other agent-based WFMS, the prototype was described in detail. The structure of the overall system, as well as the individual agents making up the system, was depicted in the main section. The graphical user interface was also briefly described and the section was concluded with an overview of the interactions within the system. A short example was then given to illustrate how the system can be used in practice. The contribution concluded with an outlook at planned and possible future developments of the system and related work.

The centralized WFMS described in this contribution does not yet fully benefit from the advantages of the agent technology, which was used to model it. As such the system described in this text is more of a kind of groundwork for future development and enhancement. Such further development will turn the WFMS into a more versatile and distributed system and can also make use of other aspects of the agent-oriented paradigm, such as autonomy and mobility, to improve the system. Such improvement will be eased by the fact that the agent-technology is already in place and the resulting system will make full use of the benefits of this programming paradigm.

References

1. Lawrence Cabac. Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen. Bachelor thesis (equiv.), University of Hamburg, Department of Computer Science, 2002.
2. Soren Christensen and Niels Damgaard Hansen. Coloured Petri nets extended with channels for synchronous communication. *Lecture Notes in Computer Science*, 815/1994:159–178, 1994. Application and Theory of Petri Nets 1994.
3. Michael Duvigneau. Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, Germany, December 2002.
4. Lars Ehrler, Martin Fleurke, Maryam Purvis, and Bastin Tony Roy Savarimuthu. Agent-based workflow management systems (WfMSs) - JBees: a distributed and adaptive WfMS with monitoring and controlling capabilities. *Information Systems and E-Business Management*, 4, Number 1 / January, 2006:5–23, 2005.
5. David Hollingsworth. *The Workflow Reference Model.* Workflow Management Coalition. Available at http://www.wfmc.org/.
6. Thomas Jacob. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, 2002.

7. N. R. Jennings, T.J. Norman, and P. Faratin. Adept: An agent-based approach to business process management. *ACM SIGMOD Record*, 27:32–39, 1998.
8. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
9. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. *Lecture Notes in Computer Science*, 3099/2004:484–493, 2004.
10. Christine Reese. *Prozess-Infrastruktur für Agentenanwendungen*. PhD thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, Germany, 2009. submitted; not yet published.
11. Christine Reese, Matthias Wester-Ebbinghaus, Till Dörges, Lawrence Cabac, and Daniel Moldt. Introducing a process infrastructure for agent systems. *Lecture Notes in Computer Science*, 5118/2008:225–242, 2008.
12. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
13. Rüdiger Valk. On processes of object Petri nets. Report of the Department of Informatics FBI-HH-B-185/96, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, Germany, June 1996.
14. Wil M.P. van der Aalst. Verification of workflow nets. *Lecture Notes in Computer Science*, 1248/1997:407–426, 1997. Application and Theory of Petri Nets 1997.
15. Wil M.P. van der Aalst and Kees van Hee. *Workflow Management - Models, Methods, and Systems*. The MIT Press, 2002.
16. Thomas Wagner. Modeling of a centralized Petri net- and agent-based workflow management system. Bachelor thesis (equiv.), University of Hamburg, Department of Informatics, 2009.

Identifying the structure of a narrative via an agent-based logic of preferences and beliefs: Formalizations of episodes from *CSI: Crime Scene Investigation*™

Benedikt Löwe[1,2,3], Eric Pacuit[4,5], Sanchit Saraf[6]*

[1]Institute for Logic, Language and Computation, Universiteit van Amsterdam, Postbus 94242, 1090 GE Amsterdam, The Netherlands
[2]Department Mathematik, Universität Hamburg, Bundesstrasse 55, 20146 Hamburg, Germany
[3]Mathematisches Institut, Rheinische Friedrich-Wilhelms-Universität Bonn, Endenicher Allee 60, 53115 Bonn, Germany;
[4] Department of Philosophy, Stanford University, Building 90, Stanford, CA 94305-2155, United States of America
[5]Center for Logic and Philosophy of Science, Tilburg University, PO Box 90153, 5000 LE Tilburg, The Netherlands
[6]Department of Mathematics and Statistics, Indian Institute of Technology, Kanpur 208016, India
bloewe@science.uva.nl, epacuit@stanford.edu, sanchit@iitk.ac.in

Abstract. Finding out what makes two stories equivalent is a daunting task for a formalization of narratives. Using a high-level language of beliefs and preferences for describing stories and a simple algorithm for analyzing them, we determine the doxastic game fragment of actual narratives from the TV crime series *CSI: Crime Scene Investigation*™, and identify a small number of basic building blocks sufficient to construct the doxastic game structure of these narratives.

1 Introduction

1.1 General Motivation

As theorists working on narrative-based computer games, we are interested in understanding the relevant structural properties that makes narratives more or less interesting, or more or less interesting for a particular target group, or, in general, to understand our notion of two stories being "essentially the same"

* The research project reported on in this paper was partially supported by the European Commission (Early Stage Research Training Mono-Host Fellowship GLoRiClass MEST-CT-2005-020841). The third author would like to thank the Institute for Logic, Language and Computation of the *Universiteit van Amsterdam* and the *Department Mathematik* of the *Universität Hamburg* for their hospitality during the time the formalizations for this paper were done.

that human agents seem to be able to grasp easily but which escapes a proper formalization so far.[1]

Any formalization of narratives provides an obvious answer to this most general question: given a formal language to describe narratives, two narratives are "essentially the same" if they are structurally isomorphic in that formal language. Whether the answer given by a fixed formalization is good depends very much on the formal language chosen. If you choose too rich a language, then minute differences between narratives become expressible, and thus the derived notion of isomorphism will fail to identify some narratives as identical even though human readers would think that they are "essentially the same". On the other hand, if your language is not very expressive, then all too many narratives will be considered equivalent by the system.

So, what is the right level of detail that allows us to identify the right notion of isomorphism? Only an empirical investigation of narratives and our willingness to identify them as equivalent will help.

Beyond the obvious general interest in understanding our perception of narratives as structurally equivalent, there are various applications for such an understanding. If we had empirical data on which structural elements tend to make a narrative more interesting, or which structural elements would be more appropriate for certain genres or audiences, we could use this in combination with existing story synthesis engines (e.g., MEXICA [19] or *Façade* [15]; both of which still use human intervention for story creation) for automated story production in computer games.

1.2 This paper

We do not claim that we have a definitive or good answer to our above questions: the formalization given in this paper gives a first approximation based on an agent-language with beliefs and preferences that might be a step towards a more complete description.

In [13], the authors proposed a simple algorithm for analyzing narratives in terms of belief states based on notions of doxastic logic. The algorithm requires focusing on the purely doxastic part of the narratives, i.e., the game structure in which all actions are determined by iterated beliefs about preferences of the agents. Then, the narrative can be analyzed as a perfect information game in which all agents may be mistaken about their iterated beliefs.

Whereas in [13, § 4], the algorithm was used to analyze a fictitious narrative about love and deceit, in this paper, we focus on narratives commercially produced for television broadcasting. In a *descriptive-empirical* approach we investigate their common structural properties based on a formalization in our system, reducing the rich narrative structure of the stories to their doxastic game trees. The empirical results of this paper point towards the possible conclusion

[1] Cf. the discussions of the notion of "analogy" in the cognitive science literature [22,10]; cf. [11, p. 791–792] for an overview of existing formal models.

that from a large number of possible formal structures, commercial crime narratives only use a very small number of doxastically simple basic building blocks (§ 2.4).

1.3 Related Work and Background.

We are interested in a fragment of the formal structure of narratives, so we aim at ignoring their presentation (i.e., choice of actors, details of dialogue, facial expressions of actors, lighting, cuts, etc.) unless it is relevant for determining the formal structure. In narratology, these components are normally called "story" and "discourse" (alternatively, "фабула"/"сюжет" or "histoire"/"récit") [4]. From now on, we shall use the term "*discourse*" to refer to the presentation of the narrative. The abstraction of a narrative to a part of its formal structure relates our research to the vast literature on "*Story Understanding*"[2] which has made tremendous progress towards analysing and synthetizing narratives:

> "there is now a considerable body of work in artificial intelligence and multi-agent systems addressing the many research challenges raised by such applications, including modeling engaging virtual characters ... that have personality ..., that act emotionally ..., and that can interact with users using spoken natural language." [26, p. 21]

Most of the work on *Story Understanding* goes into far more detail than our formalization, including the *discourse* of the narrative. Especially applications of logic for *Story Understanding* deal with the understanding of the grammatical structure of the *discourse* (cf. [25]). Even models just focusing on the *story*/фабула in general take more into account than our doxastic fragment.[3] In terms of Mueller's "shallow"/"deep" distinction [16, § 1.3], the depth of our formalization is below that of the shallow understanding. Relatively close to our approach are *Story Grammars* [23], invented by Rumelhart inspired by the structuralist investigation of fairy tales by the Russian narratologist Propp [21], the *Story Beats* in *Façade* [15], and Lehnert's *Plot units* [12].

Almost none of these approaches model beliefs and knowledge of agents in an explicit way[4]. A rare exception is the AIIDE 2008 paper by Chang and Soo [3] which is very programmatic and preliminary. The restrictions to doxastically simple building blocks and explicit modelling of theories of mind clearly relates our formalization to work in cognitive science. For these relations, cf. § 5.1.

---

[2] There is "a great variety of applications, which differ widely in the way they use, create or tell stories [24]". Cf. [1,17] for surveys, and [6,7,27] for work on interactive story telling ("Interactive story creation ... takes place in role-playing games that can be seen as emergent narratives of multiple authorship. ... Interactive story telling instead relies on a predefined story, a specific plot concerning facts and occurrences. [27, p. 32]").

[3] Cf. Young's characterization of the *story*/*discourse* divide: "A *story* consists of a complete conceptualization of the world in which the narrative is set [32]".

[4] Cf. [31] for a discussion of the lack of modelling of higher order knowledge in artificial intelligence.

1.4 Structure of the Paper.

In §2 of this paper, we shall introduce our system, modified from [13, § 3] to incorporate event nodes (at which no agent is playing) and partial states. We also discuss the basic building blocks of belief structures that we shall later encounter in the analyzed narratives. In §3, we discuss the process of taking an actual narrative and transforming it into a game of mistaken and changing beliefs, focusing in particular about the restrictions that we imposed upon ourselves by the choice of our formal framework. Finally, in §4, we then present the formalization of six narratives from the first four episodes of the TV series *CSI: Crime Scene Investigation*™ in which we can see that the eight doxastic building blocks from § 2.4 are enough to formalize all narratives. In § 5, we summarize the findings of the paper, connect them to phenomena in cognitive science about iterated beliefs (§ 5.1), and discuss future directions (§ 5.2).

2 Definitions and fundamental structures

2.1 Definitions

We give a short version of the definitions from [13, § 3]. As opposed to the discussion there, we shall explicitly use *event nodes*, i.e., nodes in which none of the agents makes a decision, but instead an event happens. Structurally, these nodes do not differ from the standard *action nodes*, but beliefs about events are theoretically on a lower level (of theory of mind[5]) than beliefs about beliefs.

Let I be the finite set of agents whom we denote with boldface capital letters. We reserve the symbol $\mathbf{E} \in I$ for the event nodes. If $\vec{\mathbf{P}} = \langle \mathbf{P}_0, ..., \mathbf{P}_n \rangle$ is a finite sequence of agent symbols, we write $\vec{\mathbf{P}}\mathbf{P}$ for the extension of the sequence by another player symbol $\mathbf{P}$, i.e.,

$$\vec{\mathbf{P}}\mathbf{P} := \langle \mathbf{P}_0, ..., \mathbf{P}_n, \mathbf{P} \rangle.$$

A tree T is a finite set of nodes together with an edge relation (in which any two nodes are connected by exactly one path). Let $\mathsf{tn}(T)$ denote the set of terminal nodes of T, and for $t \in T$, let $\mathbf{succ}_T(t)$ denote the set of immediate T-successors of t. The **depth** of the tree T is the number of elements of a longest path in T, and we denote it by $\mathrm{dp}(T)$.

We fix I and T and a **moving function** $\mu : T \backslash \mathbf{tn}(T) \to I$, where $\mu(t) = \mathbf{P}$ indicates that it is $\mathbf{P}$'s move at node t. If $\mu(t) = \mathbf{E}$ we call t an **event node**, otherwise we call it an **action node**. We call total orders $\succeq$ on $\mathbf{tn}(T)$ **preferences** and denote its set by $\mathcal{P}$. A map $\succeq : I \to \mathcal{P}$ is called a **description**. We call functions

$$S : T \times I^{\leq \mathrm{dp}(T)} \to \mathcal{P}^I$$

states, interpreting the description $S(t, \varnothing)$ as the **true state of affairs** at position t. If $S(t, \vec{\mathbf{P}})$ is one of the descriptions defined by the state S, we interpret $S(t, \mathbf{P}\vec{\mathbf{P}})$ as player $\mathbf{P}$'s belief about $S(t, \vec{\mathbf{P}})$.

[5] Cf. §§ 2.4 and 5.1.

2.2 The analysis

Given a tuple $\langle I, T, \mu, S\rangle$, we can now fully analyze the game and predict its outcome (assuming that the agents follow the backward induction solution). In order to do this analysis, we shall construct labellings $\ell_{S_{\vec{\mathbf{P}}}} : T \to \mathsf{tn}(T)$ where $\ell_{S_{\vec{\mathbf{P}}}}$ is interpreted as the subjective belief relative to $\vec{\mathbf{P}}$ of the outcome of the game if it has reached the node t. For instance, $\ell_{S_{\mathbf{A}}}(t) = t^* \in \mathsf{tn}(T)$, then player $\mathbf{A}$ believes that if the game reaches t, the eventual outcome is t^*.

The labelling algorithm If t is a terminal node, we just let $\ell_U := t$ for all states U. In order to calculate the label of a node t controlled by player $\mathbf{P}$, we need the $\mathbf{P}$-subjective labels of all of its successors. More precisely: if $t \in T$, $\mu(t) = \mathbf{P}$ and we fix a state U, then we can define ℓ_U as follows: find the U-true preference of player $\mathbf{P}$, i.e., $\succeq = U(t, \varnothing)(\mathbf{P})$. Then consider the labels $\ell_{U_{\mathbf{P}}}(t')$ for all $t' \in \mathsf{succ}(t)$ and pick the $\succeq$-maximal of these, say, t^*. Then $\ell_U(t) := t^*$. Concisely, $\ell_U(t)$ is the $U(t, \varnothing)(\mu(t))$-maximal element of the set $\{\ell_{U_{\mu(t)}}(t')\,;\, t' \in \mathsf{succ}(t)\}$.

Computing the true run of the game After we have defined all subjective labellings, the true run can be read off recursively. Since our labels are the terminal nodes, for each t with $\mu(t) = \mathbf{P}$ and S, there is a unique $t' \in \mathsf{succ}(t)$ such that $\ell_{S_{\mathbf{P}}}(t') = \ell_S(t)$. Starting from the root, take at each step the unique successor determined by $\ell_S(t)$ until you reach a terminal node.

2.3 Partial states, notation, and isomorphism

Note that in actual narratives (as opposed to narratives invented for the purpose of formalization, such as the narrative in [13, § 2]), we cannot expect to have full states. Instead, we shall have some information about agents' preferences and beliefs that is enough to run the algorithm described in § 2.2. If $\mathcal{P}^{\mathrm{p}}$ is the set of partial preferences (i.e., linear orders of subsets of $\mathsf{tn}(T)$) and $\mathrm{PF}(X, Y)$ is the set of partial functions from X to Y, then we call partial functions from $T \times I^{\mathrm{dp}(T)}$ to $\mathrm{PF}(I, \mathcal{P}^{\mathrm{p}})$ **partial states**.

In the following, we shall use the letters v_i for non-terminal nodes of T and t_i for terminal nodes. If we write

$$S(v_i, \vec{\mathbf{P}})(\mathbf{P}) = (t_{i_0}, t_{i_1}, ..., t_{i_n}),$$

we mean that in the ordering $\succeq := S(v_i, \vec{\mathbf{P}})(\mathbf{P})$, we have $t_{i_0} \succeq t_{i_1} \succeq ... \succeq t_{i_n}$. If in such a sequence, we include a non-terminal node v_i, e.g.,

$$S(v_i, \vec{\mathbf{P}})(\mathbf{P}) = (t_j, v_k),$$

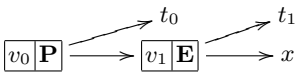
we mean that t_j is preferred over *all* nodes following v_k. Similarly,

$$S(v_i, \vec{\mathbf{P}})(\mathbf{P}) = (v_j, v_k)$$

means that every outcome following v_j is preferred over every outcome following v_k. We normally phrase preferences in these terms. When we are drawing our

game trees, we represent non-terminal nodes by $\boxed{v_i|\mathbf{P}}$ indicating $\mu(v_i) = \mathbf{P}$. In our discussions, we shall assume introspection of all agents, i.e., agents are aware of their own preferences and iterations thereof, even though there is evidence that introspection is not necessarily a feature of human mental processes and awareness [18]. This simplifies notation considerably, and there are no indications that failure of introspection is relevant in any of the narratives we analyzed.

To illustrate this, let us look at the two building blocks *Expected Event* and *Unexpected Event* in Figure 1. In both cases, agent $\mathbf{P}$ prefers outcome t_1 over t_0. Also in both cases, he thinks that the event will produce outcome t_1 (expressed in our language, somewhat awkwardly, as "the event agent prefers t_1 over x"). In $\mathsf{ExEv}(\mathbf{P})$, the latter belief is correct; in $\mathsf{UnEv}(\mathbf{P})$, it is incorrect.



$\mathsf{ExEv}(\mathbf{P})$: $S(v_0, \varnothing)(\mathbf{P}) = (t_1, t_0)$; $S(v_0, \mathbf{P})(\mathbf{E}) = (t_1, x)$; $S(v_1, \varnothing)(\mathbf{E}) = (t_1, x)$
$\mathsf{UnEv}(\mathbf{P})$: $S(v_0, \varnothing)(\mathbf{P}) = (t_1, t_0)$; $S(v_0, \mathbf{P})(\mathbf{E}) = (t_1, x)$; $S(v_1, \varnothing)(\mathbf{E}) = (x, t_1)$

Fig. 1. The basic building blocks $\mathsf{ExEv}(\mathbf{P})$ and $\mathsf{UnEv}(\mathbf{P})$ of *Expected Event* and *Unexpected Event*.

The notion of partial states give an obvious definition of **isomorphism** of two formalized versions of narratives: if $\langle I, T, \mu, S\rangle$ and $\langle I^*, T^*, \mu^*, S^*\rangle$ describes two narratives (where S and S^* are partial states), then they are isomorphic if there are bijections $\pi_0 : I \to I^*$ and $\pi_1 : T \to T^*$ such that

1. π_1 is an isomorphism of trees,
2. $\pi_0(\mathbf{E}) = \mathbf{E}$,
3. $\mu^*(\pi_1(x)) = \pi_0(\mu(x))$, and
4. $S^*(\pi_1(x), \pi_0(\vec{\mathbf{P}}))(\pi_0(\mathbf{P})) = (\pi_1(t), \pi_1(t'))$ if and only if $S(x, \vec{\mathbf{P}})(\mathbf{P}) = (t, t')$ (where $\pi_0(\vec{\mathbf{P}})$ is the obvious extension of π_0 to finite sequences of elements of I).

2.4 Building blocks of narratives

While working with the actual narratives, we identified a number of fundamental building blocks that recur in the investigated narratives and that can describe all of the narratives under discussion. For our reconstruction of the narratives, we need eight building blocks.

These building blocks can be stacked. We use the symbol x in our building blocks to indicate that this could either be a terminal node (at the end of the narrative) or a non-terminal node which would now become the top node of the next stack. If the last node of a building block is controlled by an agent, then the doxastic structure of the building blocks overlaps, as the first node of the

second block becomes the last node of the first block. In the case of blocks of length 3, there could also be larger overlap, but we did not find instances of this in the narratives investigated.

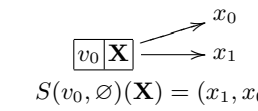
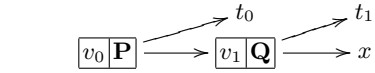


Fig. 2. The basic building block $\mathsf{Act}(\mathbf{X})$ of *Action*.

The trivial building blocks are just actions that happen with no relevant reasoning about them (described in Figure 2); these could be called doxastic blocks of level -1. We denote it by $\mathsf{Act}(\mathbf{P})$ for an action by player $\mathbf{P}$. Typical examples are actions where agents just follow their whim without deliberation. Note that being represented by a building block of level -1 does not mean that the *discourse* of the narrative shows no deliberation; in fact, in our investigated narratives we find examples of CSI agents discussing whether they should follow their beliefs (i.e., perform a higher level action) or not, and finally decide to perform the action without taking their beliefs into account. These would still be formalized as blocks of level -1.

The next level of basic building blocks are those that have reasoning based on beliefs, but not require any theory of mind at all, i.e., building blocks of level 0. The two fundamental building blocks here are *expected event* ($\mathsf{ExEv}(\mathbf{P})$) and *unexpected event* ($\mathsf{UnEv}(\mathbf{P})$), explained before and described in Figure 1.



$\mathsf{ExAc}(\mathbf{P}, \mathbf{Q})$: $S(v_0, \varnothing)(\mathbf{P}) = (t_1, t_0)$; $S(v_0, \mathbf{P})(\mathbf{Q}) = (t_1, x)$; $S(v_1, \varnothing)(\mathbf{Q}) = (t_1, x)$
$\mathsf{UnAc}(\mathbf{P}, \mathbf{Q})$: $S(v_0, \varnothing)(\mathbf{P}) = (t_1, t_0)$; $S(v_0, \mathbf{P})(\mathbf{Q}) = (t_1, x)$; $S(v_1, \varnothing)(\mathbf{Q}) = (x, t_1)$

Fig. 3. The basic building blocks $\mathsf{ExAc}(\mathbf{P}, \mathbf{Q})$ and $\mathsf{UnAc}(\mathbf{P}, \mathbf{Q})$ of *Expected Action* and *Unexpected Action*.

Moving beyond zeroth order theory of mind, we now proceed to building blocks that require beliefs about beliefs. There are two such building blocks used in our narratives, *Expected Action* ($\mathsf{ExAc}(\mathbf{P}, \mathbf{Q})$), *Unexpected Action* ($\mathsf{UnAc}(\mathbf{P}, \mathbf{Q})$), and *Collaboration gone wrong* ($\mathsf{CoGW}(\mathbf{P}, \mathbf{Q})$) whose structure we give in Figures 3 and 4. Let us give examples from the investigated narratives from § 4. In the narrative *The severed leg* (cf. Figure 14), agent Willows informs the victim's husband of the state of the investigation. Based on this information, the husband concludes that the current suspect Phil Swelco has murdered his

wife and kills Swelco. In the tree in Figure 3, the node t_0 corresponds to "Willows does not give information to the husband" and t_1 corresponds to "Willows is nice to the husband, and the husband does not do anything with the information given to him", whereas x is the actual outcome. Willows believes that the husband prefers t_1 over x and prefers t_1 over t_0 herself.

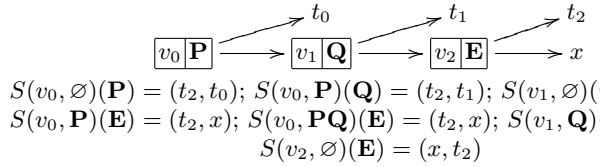


Fig. 4. The basic building block $\mathsf{CoGW}(\mathbf{P}, \mathbf{Q})$ of *Collaboration gone wrong*.

The building block *Collaboration gone wrong* is discussed in more detail in § 3.3. Kyle kills James and expects Matt to cooperate in covering up the murder as a suicide. Matt actually helps Kyle in that respect, but it doesn't work, as the autopsy reveals that James did not hang himself (cf. Figure 15).

Finally, we move to the building blocks that use second order beliefs. In our narratives, there are only two such building blocks: *Betrayal* ($\mathsf{Betr}(\mathbf{P}, \mathbf{Q})$) and *Unsuccessful Collaboration with a Third* ($\mathsf{UnCT}(\mathbf{P}, \mathbf{Q}, \mathbf{R})$) (given in Figures 5 and 6).

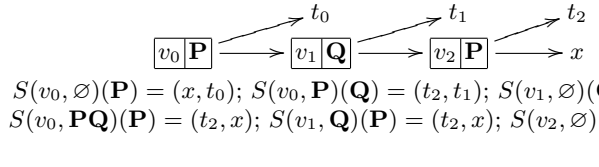
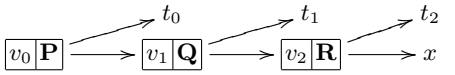


Fig. 5. The basic building block $\mathsf{Betr}(\mathbf{P}, \mathbf{Q})$ of *Betrayal*.

To give an example for *Betrayal* from the narrative *Faked Kidnapping* (cf. Figure 12): Chip and Laura plan to fake a kidnapping of Laura in order to get money from Laura's husband. Laura agrees to this, but Chip betrays her and buries her in a crate in the Nevada desert. Notice that we model the joint plan to fake the kidnapping as a sequence of actions by Chip ("proposing the faked kidnapping") and Laura ("agreeing to the faked kidnapping") with outcomes x ("Laura is buried in the desert"), t_2 ("Laura and Chip get the money from her husband"), t_1 ("Laura does not want to be part of the faked kidnapping"), and t_0 ("Chip does not propose a faked kidnapping").



$$S(v_0,\varnothing)(\mathbf{P}) = (t_2,t_0);\ S(v_0,\mathbf{P})(\mathbf{Q}) = (t_2,t_1);\ S(v_1,\varnothing)(\mathbf{Q}) = (t_2,t_1)$$
$$S(v_0,\mathbf{P})(\mathbf{R}) = (t_2,x);\ S(v_0,\mathbf{PQ})(\mathbf{R}) = (t_2,x);\ S(v_1,\mathbf{Q})(\mathbf{R}) = (t_2,x)$$
$$S(v_2,\varnothing)(\mathbf{R}) = (x,t_2)$$

Fig. 6. The basic building block $\mathsf{UnCT}(\mathbf{P},\mathbf{Q},\mathbf{R})$ of *Unsuccessful Collaboration with a Third*.

3 Methodological issues

In the introduction (§ 1.1), we pointed out that finding the right notion of formal representation for narratives is subtle and difficult. If you allow your formal language to be too expressive, then narratives that are considered "equivalent" by human audiences would be separated, whereas if your language is too coarse, then non-identical narratives will be identified.

It is not at all obvious what elements a formalization with the right balance should contain, and we consider this study as part of the endeavour of finding out how much detail we need. Certainly, the system we propose here errs on the side of being too coarse: Already separating *story* from *discourse* is a difficult task, and reducing the narrative to our parsimonious doxastic fragment from §2 requires a number of hand-crafted modelling decisions in order to fit the narratives into our framework. In this section, we discuss a number of issues related to the formalization of narratives in our formal language.[6]

3.1 The sequence of events

The narrative of a TV crime episode rarely proceeds chronologically. Often, it starts when the corpse is found, and then proceeds to tell the story of the detectives unearthing the sequence of events that led to the murder. Sometimes, we see scenes of the past in flashbacks, sometimes, they are being reported by agents. We consider all this part of the *discourse* of the narrative and shall build our structures of actions and events in chronological order. Note that one consequence of this is that our models do not take into account the beliefs of the audience.

3.2 Imperfect or incomplete information

Our model is based on perfect information games with mistaken beliefs. However, in many cases, imperfect or incomplete information can be mimicked in our system by event nodes. Let us give a simple examples:

[6] The corresponding caveat for Lehnert's set-up of *Plot units* is the problem of "Recognizing plot units" [12, § 10].

Example. *Detective Miller thinks that Jeff is Anne's murderer while, in fact, it is Peter. Miller believes that Jeff will show up during the night in Anne's apartment to destroy evidence and thus hides behind a shower curtain to surprise Jeff. However, Peter shows up to destroy the evidence, and is arrested.*

The natural formalization would be an imperfect or incomplete information game, but the structure given in Figure 7 can be used to formalize the narrative with **M** representing Miller, **J** Jeff, and **P** Peter. The event node v_1 should be read as "Peter turns out to be Anne's murderer". Nodes t_1 and t_3 are "Peter (Jeff) is the murderer, returns to the apartment and is caught", respectively; nodes t_2 and t_4 are "Peter (Jeff) is the murderer and does not return to the apartment".

We let $S(v_0, \mathbf{M})(\mathbf{E}) = (v_3, v_2)$ (i.e., Miller believes that Jeff will turn out to be the murderer), $S(v_3, \mathbf{M})(\mathbf{J}) = t_3$, $S(v_1, \varnothing)(\mathbf{E}) = (v_2, v_3)$ (i.e., Peter is the actual murderer), and $S(v_2, \varnothing)(\mathbf{P}) = (t_1, t_2)$ (i.e., Peter in fact plans to return to the apartment).

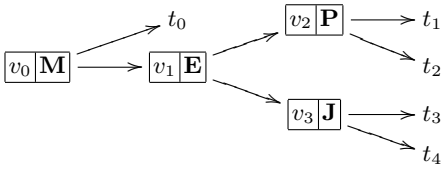


Fig. 7. Mimicking imperfect information by an event node v_1 representing "Peter turns out to be the murderer".

Note that this is not a natural way of modelling imperfect information and future proposals for a formalization would have to deal with this by having a more liberal underlying structure. However, we found that for the chosen narratives from the series *CSI: Crime Scene Investigation*™, the impact on the adequacy of our formalizations was relatively minor.[7]

3.3 Not enough information

As mentioned in § 2.3, we often do not have enough information to give the full state, but only enough of the state that allows us to formally reconstruct the sequence of events and actions. In general, this is not a problem, but sometimes, the narrative is ambiguous on what happened or why it happened, and we are not even able to reconstruct the formal structure without any doubts.

[7] We suspect that one of the reasons is that "strictly go by the evidence" is one of the often repeated explicit creeds of the CSI members, prohibiting the actors from letting beliefs about facts influence their actions. This has its formal reflection in the fact that the investigators play only a minor rôle in our formalizations, often occurring in event nodes, and rarely making any decisions.

We can give an example from the narratives investigated in § 4: In the narrative *Pledging gone wrong*, we see in a brief flashback scene that (the student) Kyle murders (his fellow student) James. There is a cut, and after that we see that (the student) Matt enters, and Kyle and Matt discuss what to do. The whole scene lasts but a few seconds, and the narrative does not give any clue whether Kyle was expecting Matt to enter or not. There are various different ways to formalize this brief sequence of events as described in Figure 8. In option (a), we consider Kyle's action almost as a joint action: he is murdering James under the (correct and never discussed) assumption that Matt will help him to cover this up. In option (b), we allow Matt to consider not helping Kyle, and then have to model Kyle as correctly assuming that Matt will help him, i.e., $S(v_1, \mathbf{K})(\mathbf{M}) = (x, t_1)$ and $S(v_1, \varnothing)(\mathbf{M}) = (x, t_1)$. In option (c), we now model the entering of Matt after the murder as an event and have to decide whether Kyle expected that this happens or not. One could take the casual tone of Kyle when Matt enters as an indication of lack of surprise, and therefore choose $S(v_1, \mathbf{K})(\mathbf{E}) = (v_2, t_1)$.

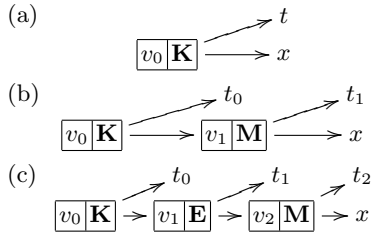


Fig. 8. Three different formalizations of the interaction between Kyle and Matt in the narrative *Pledging gone wrong*.

Which of the three options is correct? We believe that there is no good answer that does not take into account the narrative as a whole. In this particular case (see § 4), we decided to go with option (b), as Matt's decision is explicitly relevant in the last scenes of the narrative when Matt decides to tell the truth. We therefore decided that having a decision node for Matt represents the character of the narrative most appropriately. It is unlikely that modelling decisions like this can always be uncontroversial. The problem of judging what is the natural formalization from the narrative is exemplified once more in § 3.4.

3.4 Relevant information

In § 3.3 we have seen that the narrative sometimes does not allow us to uncontroversially choose the formalization. The dual problem to this is that the *discourse* is often much richer than the structure necessitates. Let us explain this in the following three examples:

Example 1. *John and Sue are a happily married couple when John's old friend, Peter, suddenly shows up after no contact for seven years, inviting himself for dinner. Peter asks John for a large amount of money without giving any reasons. Sue had always disliked Peter, and after Peter had left, Sue urged her husband not to give him any money. After a long discussion, John sighs and agrees to Sue's request. The couple goes to bed, but after Sue is sound asleep, John sneaks into the living room, gives Peter a call and promises to pay. After two weeks, Sue finds out that a large amount of money is missing from their joint bank account.*

Example 2. *... The couple goes to bed, but after Sue is sound asleep, John sneaks into the living room, gives Peter a call and promises to pay. Peter is honestly surprised, as he had not expected this after the rather icy atmosphere at the dinner table. After two weeks, ...*

Example 3. *... John sneaks into the living room, and gives Peter a call, intending to give him the money. However, John did not know how deep in trouble Peter was. After Peter noticed the icy atmosphere at the dinner table, he had taken the elevator to the rooftop of John's apartment building. There, he takes John's call, says "Good bye, John, you were always a good friend", and jumps, before John can tell him that he'll give him the money. John shouts "I'll give you the money" into the phone, but it is too late. When he turns around, Sue is standing behind him.*

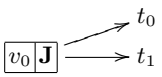


Fig. 9. The tree diagram for all three example narratives about John, Sue and Peter.

The tree structure of all of these narratives is the same, viz. the one depicted in Figure 9. Only the partial states differ slightly. In Example 1, we have $S(v_0, \mathbf{S})(\mathbf{J}) = (t_0, t_1)$ and $S(v_0, \varnothing)(\mathbf{J}) = (t_1, t_0)$ which explains Sue's surprise. In Examples 2 and 3, we have in addition $S(v_0, \mathbf{P})(\mathbf{J}) = (t_0, t_1)$ representing Peter's belief in both narratives that John will not give him the money.

Structurally, Examples 2 and 3 are isomorphic in the sense of § 2.3 and slightly different from Example 1. However, we are sure that most readers will agree that Examples 1 and 2 are closer to each other than to Example 3. This difference does not lie in the event and action structure of the narratives, but in the *discourse*. In Example 3, Peter's disbelief in John giving him the money intensifies the emotional difference between the terminal nodes t_0 and t_1, and thus creates a different feeling. As the modeller, we should have to make the decision of whether we include $S(v_0, \mathbf{P})(\mathbf{J}) = (t_0, t_1)$ in the formalization of Example 2.

4 The six narratives formalized

In this section, we shall give the formal structure of six narratives from the first four episodes of season one of the drama series *CSI: Crime Scene Investigation*™. These four episodes contain ten narratives some of which involved material from other episodes than the first four and others had interlinking events between narratives; we left these unconsidered for the sake of simplicity.[8]

[8] Cf. [2]. Episode 1, entitled "Pilot", was written by Anthony E. Zuiker and directed by Danny Cannon; Episode 2, entitled "Cool Change" was written by Anthony E. Zuiker and directed by Michael W. Watkins; Episode 3, entitled "Crate 'n Burial",

Trick roll (episode 1; agents victim, **V**, Kristy Hopkins, **K**)

> A prostitute, Kristy Hopkins, puts the drug scopolamine on her breasts to knock out her customers and steal their possessions. A victim is found robbed at a crime scene by agent Nick Stokes with a discolouration around his mouth. Shortly afterwards, Hopkins loses consciousness while driving. Agent Stokes connects the two cases and finds a similar discoloration on Hopkins's breast.

Winning a fortune (episode 2; agents Jamie Smith, **J**, Ted Sallanger, **T**)

> Jamie Smith and her boyfriend Ted Sallanger are gambling in Las Vegas. Smith urges Sallanger to continue and Sallanger wins the $40 million jackpot. Shortly afterwards, Sallanger breaks up with Smith, and they have a fight during which she hurts him with a bottle and leaves the apartment. Later on, she returns to kill him with a candlestick. Her return is not properly filed by the key card system of the hotel, so at first the key card records of the hotel seem to confirm her story that she did not return to the room after the fight.

Faked kidnapping (episode 3; agents Chip Rundle, **C**, Laura Garris, **L**, the CSI unit, **U**)

> Chip Rundle and Laura Garris plan to fake a kidnapping and get a ransom from Garris's husband. However, after the staged kidnapping, Rundle turns on Garris and buries her in a crate in the Nevada desert. Based on some dirt on the bedroom carpet, the CSI unit manages to find Garris before she dies. In the meantime, Garris's husband has paid the ransom. When he collects the ransom, Rundle is arrested. Confronted with the facts, Garris does not tell the police that Rundle was the kidnapper, but his voice is matched to the voice of the ransom phone call. The CSI unit decides to investigate further and finds that the evidence is not consistent with a real kidnapping. A blood test confirms that Garris was never drugged and leads to Garris's arrest.

Hit and run (episode 3; agents Charles Moore, **C**, James Moore, **J**)

> The young James Moore kills a young girl in a car accident and flees the scene. The CSI unit finds an imprint of the license plate on a bruise on the body of the victim and traces Moore. Moore's grandfather Charles wants to protect his grandson and claims that he was the driver. The CSI unit finds that the position of the car seat is not consistent with this claim. The grandfather modifies the story and claims that he was the driver at the time of the accident, but after that, the grandson took the wheel as Charles had banged his head during the accident. Further investigation brings forward a piece of tooth that the driver lost during the accident and the CSI unit matches this to James Moore.

The severed leg (episode 4; agents Catherine Willows, **C**, Winston Barger, **W**)

> A female body with a severed leg is found in Lake Mead. Her stomach contents lead the CSI to a restaurant near the lake where it is established that she had dinner with a Phil Swelco. Swelco admits that he was having an affair with the victim. The discussion between the CSI and Swelco is observed by the victim's husband, Winston Barger, who asks how Swelco is related to Wendys death. CSI Willows informs Barger about the state of the investigation. The CSI find the boat and establish that the victim tried to restart the engine, dislocated her shoulder, lost her balance, hit her head, and fell into the water. When the CSI come to Swelco, they find him dead in his house, murdered by Barger who thought he was avenging his wife.

---

was written by Ann Donahue and directed by Danny Cannon; Episode 4, entitled "Pledging Mr. Johnson", was written by Josh Berman and Anthony E. Zuiker and directed by Richard J. Lewis.

Pledging gone wrong (episode 4; agents James Johnson, **J**, Jill Wentworth, **W**, Kyle Travis, **K**, Matt Daniels, **M**)

During a pledging ceremony in a fraternity, James Johnson is being bullied by Kyle Travis. The new students have to go to a sorority and get some body part signed by the female students. Johnson asks Travis's girlfriend, Jill Wentworth, to sign his private parts and she agrees. Travis is very angry and asks Johnson privately to allow him to insert a piece of raw liver on a noose. When Travis tries to pull it out, the noose breaks and Johnson chokes to death while Travis watches. Matt Daniels enters and is convinced by Travis to cover up the murder. They stage Johnson's death as a hanging suicide. However, the autopsy reveals that the death was not death by hanging, and the piece of raw liver is found. Travis and Daniels change their story and tell that they tried to save Johnson by performing the Heimlich maneuver, but no evidence of this is found. The CSI unit finds out that the signature on Johnson's private parts belongs to Travis's girlfriend, and finally Daniels tells the truth.

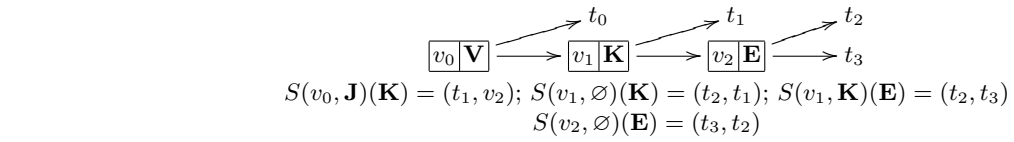


Fig. 10. The formalization of *Trick roll*, consisting of $\mathsf{UnAc}(\mathbf{V}, \mathbf{K})$ and $\mathsf{UnEv}(\mathbf{K})$.

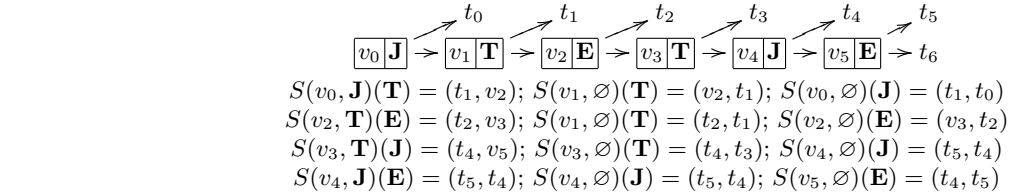
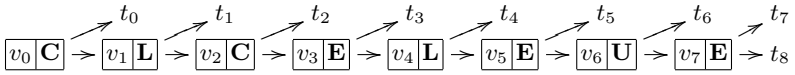


Fig. 11. The formalization of *Winning a fortune*, consisting of $\mathsf{ExAc}(\mathbf{J}, \mathbf{T})$, $\mathsf{UnEv}(\mathbf{T})$, $\mathsf{UnAc}(\mathbf{T}, \mathbf{J}$, and $\mathsf{UnEv}(\mathbf{J})$.

Here, we shall reconstruct all six narratives in terms of the basic building blocks given in § 2.4.

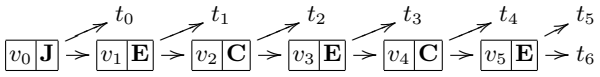
One of our narratives does not even contain first-order beliefs: *Hit and run*, formalized as Figure 13.

Half of our narratives involves basic building blocks of at most level 1, formalized in Figures 10, 11, and 14. The remaining two narratives have blocks of level 2. These are *Faked kidnapping*, formalized in Figure 12 and *Pledging gone wrong*, formalized in Figure 15.



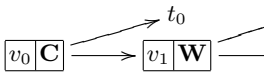
$S(v_0, \varnothing)(\mathbf{C}) = (t_3, t_2, t_0, t_1)$; $S(v_2, \varnothing)(\mathbf{C}) = (t_3, t_2)$; $S(v_2, \mathbf{C})(\mathbf{E}) = (t_3, v_4)$
$S(v_0, \mathbf{C})(\mathbf{L}) = (t_2, t_1, v_3)$; $S(v_0, \mathbf{CL})(\mathbf{C}) = (t_2, v_3)$; $S(v_1, \mathbf{L})(\mathbf{C}) = (t_2, v_3)$
$S(v_1, \varnothing)(\mathbf{L}) = (t_2, t_1, v_3)$; $S(v_3, \varnothing)(\mathbf{E}) = (v_4, t_3)$
$S(v_4, \mathbf{L})(\mathbf{E}) = (t_5, v_6)$; $S(v_4, \varnothing)(\mathbf{L}) = (t_5, t_4)$; $S(v_5, \varnothing)(\mathbf{E}) = (v_6, t_5)$
$S(v_6, \varnothing)(\mathbf{U}) = (v_7, t_6)$; $S(v_7, \varnothing)(\mathbf{E}) = (t_7, t_8)$

Fig. 12. The formalization of *Faked kidnappping*, consisting of $\mathsf{Betr}(\mathbf{C}, \mathbf{F})$, $\mathsf{UnEv}(\mathbf{C})$, $\mathsf{UnEv}(\mathbf{J})$, and $\mathsf{ExEv}(\mathbf{U})$.



$S(v_0, \mathbf{J})(\mathbf{E}) = (t_1, v_2)$; $S(v_0, \varnothing)(\mathbf{J}) = (t_1, t_0)$; $S(v_1, \varnothing)(\mathbf{E}) = (v_2, t_1)$
$S(v_2, \mathbf{C})(\mathbf{E}) = (t_3, v_4)$; $S(v_2, \varnothing)(\mathbf{C}) = (t_3, t_2)$; $S(v_3, \varnothing)(\mathbf{E}) = (v_4, t_3)$
$S(v_4, \mathbf{C})(\mathbf{E}) = (t_5, t_6)$; $S(v_4, \varnothing)(\mathbf{C}) = (t_5, t_4)$; $S(v_5, \varnothing)(\mathbf{E}) = (t_6, t_5)$

Fig. 13. The formalization of *Hit and run*, consisting of $\mathsf{UnEv}(\mathbf{J})$, $\mathsf{UnEv}(\mathbf{C})$, and $\mathsf{UnEv}(\mathbf{C})$.



$S(v_0, \mathbf{C})(\mathbf{W}) = (t_1, t_2)$; $S(v_0, \varnothing)(\mathbf{C}) = (t_1, t_0)$; $S(v_1, \varnothing)(\mathbf{W}) = (t_2, t_1)$

Fig. 14. The formalization of *The severed leg*, consisting of $\mathsf{UnAc}(\mathbf{C}, \mathbf{W})$.

5 General conclusion

In § 4, we have seen that ten narratives from a crime series commercially produced for TV entertainment show a lot of recurring structures. A total number of eight basic building blocks is able to describe the event and action structure of all of the six narratives; most of the building blocks involve only zeroth- and first-order beliefs, and there are only two instances of genuine second-order beliefs. Not surprisingly, we see that second-order beliefs typically show up in those parts of the crime narratives that do not directly related to solving the crime, but to interpersonal interaction between the agents. While mistaken belief is a relatively common phenomenon, changing preferences and beliefs did not occur in any of the formalized narratives.

5.1 Restrictions on orders of theory of mind

The fact that in concretely given narratives, we only encounter building blocks of level 2 and lower corresponds very well to experimental research in orders of

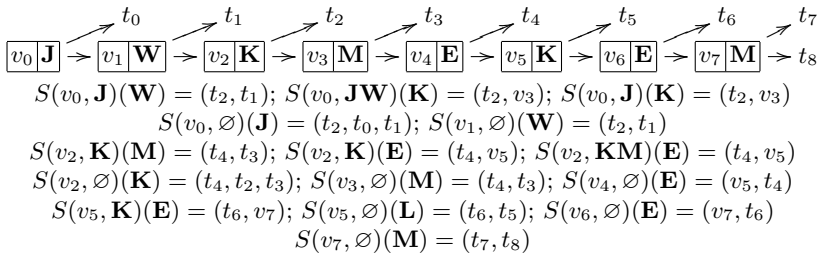


Fig. 15. The formalization of *Pledging gone wrong*, consisting of $\mathsf{UnCT}(\mathbf{J}, \mathbf{W}, \mathbf{K})$, $\mathsf{CoGW}(\mathbf{K}, \mathbf{M})$, $\mathsf{UnEv}(\mathbf{K})$, and $\mathsf{Ac}(\mathbf{M})$.

theory of mind. Both in experimental game theory (as a reaction to the fact that human beings do not seem to follow the mathematical predictions of game theory) and in psychology and cognitive science, researchers have investigated the limits of the capacity of human cognition to reason about iterated beliefs.

In game theory, this led to Herbert Simon's notion of "Bounded Rationality". Stahl and Wilson have investigated levels of belief in games [29] and identified "most participants' behavior ... as being observationally equivalent with one specific type" from their list of five types: 'level–0', 'level–1', 'level–2', 'naïve Nash', and 'worldly'. There is evidence from evolutionary game theory [28] that even in a population with players of arbitrary depth of theories of mind, the simple types will never be driven out of the population (this argument is the foundation of the decision of Stahl and Wilson to restrict their attention to the above mentioned five types as there is little advantage to move beyond level–2 [29, p. 220]).

In psychology, the study of the development and use of second-order beliefs started with Perner and Wimmer [20] and was continued in experiments by Hedden and Zhang [8], Keysar, Lin, and Barr [9], Verbrugge and Mol [30], and Flobbe, Verbrugge, Hendriks, and Krämer [5], to name but a few. The experimental evidence suggests that many adults only apply first-order theory of mind (even this is not always done without errors, cf. [9, Experiment 1]) and few progress to second-order theory of mind and beyond. Our results are perfectly in line with this.

5.2 Future work

A lot of the suspense and enjoyment in crime narratives comes from the fact that the audience (and the detectives) do not know who committed the crime. As a consequence, the most natural way to model crime narratives would be by imperfect information games or incomplete information games or games involving awareness. Our formal model described in § 2 is purely based on a perfect information game model. In § 3.2, we saw that this was not a serious restriction for the investigated narratives, but in general, we feel that a formal language

should be able to express these phenomena. We see it as a major task for the future to develop a version of our formal model that incorporates some aspects of imperfect or incomplete information or awareness. Such a model would be able to deal much more easily and naturally with the issues discussed in § 3.2. Another component that could turn out to be important is the representation of plans of agents (cf. [33] for the inclusion of planning into a story engine, and [14] for the inclusion of a planning engine for artificial agents) in the formal language. This leads to the natural proposal to enhance our formal system by including these aspects; however, this will have to be done with caution in order to retain the simplicity of the system: there are many formal models that can powerfully deal with various aspects of communication and reasoning, but we do not want to jeopardize perspicuity and ease of use of our formal system.

Once a system has been developed that can capture many relevant aspects of narratives, larger numbers of narratives, also from different genres could be translated into this formal system in order to form a corpus for investigating various important and wide-ranging empirical questions.

References

1. R. Alterman. Understanding and summarization. *Artificial Intelligence Review*, 5:239–254, 1991.
2. D. Cannon, M. W. Watkins, R. J. Lewis, L. Antonio, K. Fink, M. Shapiro, T. J. Wright, and O. Scott, directors. *CSI : Crime Scene Investigation™. Seizoen Één. Aflevering 1.1–1.12.* CBS Broadcasting Inc./RTL Nederland B.V., 2008. DVD.
3. H.-M. Chang and V.-W. Soo. Simulation-based story generation with a theory of mind. In M. Mateas and C. Darken, editors, *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE 2008)*, pages 16–21. AAAI Press, 2008.
4. S. B. Chatman. *Story and Discourse: Narrative Structure in Fiction and Film.* Cornell University Press, 1980.
5. L. Flobbe, R. Verbrugge, P. Hendriks, and I. Krämer. Children's application of theory of mind in reasoning and language. *Journal of Logic, Language and Information*, 2008. to appear.
6. D. Grasbon. Konzeption und prototypische Implementation einer Storyengine: Dynamisch-reaktives System zum Erzählen nichtlinear-interaktiver Geschichten bei größtmöglicher Spannung, gedanklicher Immersion, Identifikation und Motivation des Spielers, 2001. Diplomarbeit, Technische Universität Darmstadt.
7. D. Grasbon and N. Braun. A morphological approach to interactive storytelling. *netzspannung.org/journal*, special issue:337–340, 2001. Proceedings: cast01 // living in mixed realities, September 21 22, 2001, Schloss Birlinghoven, Conference on artistic, cultural and scientific aspects of experimental media spaces.
8. T. Hedden and J. Zhang. What do you think I think you think? Strategic reasoning in matrix games. *Cognition*, 85:1–36, 2002.
9. B. Keysar, S. Lin, and D. J. Barr. Limits on theory of mind use in adults. *Cognition*, 89:25–41, 2003.
10. S. Lam. Affective analogical learning and reasoning. Master's thesis, School of Informatics, University of Edinburgh, 2008.

11. L. B. Larkey and B. C. Love. CAB: Connectionist analogy builder. *Cognitive Science*, 27:781–794, 2003.
12. W. G. Lehnert. Plot units and narrative summarization. *Cognitive Science*, 4:293–331, 1981.
13. B. Löwe and E. Pacuit. An abstract approach to reasoning about games with mistaken and changing beliefs. *Australasian Journal of Logic*, 6:162–181, 2008.
14. M. Magnusson. Deductive planning and composite actions in Temporal Action Logic, 2007. Licentiate Thesis, Linköping University.
15. M. Mateas and A. Stern. Integrating plot, character and natural language processing in the interactive drama *Façade*. In S. Göbel, N. Braun, U. Spierling, J. Dechau, and H. Diener, editors, *Technologies for Interactive Digital Storytelling and Entertainment. TIDSE 03 Proceedings*, volume 9 of *Computer Graphik Edition*. Fraunhofer IRB Verlag, 2003.
16. E. T. Mueller. Story understanding through multi-representation model construction. In S. Nirenburg, editor, *Human Language Technology Conference, Proceedings of the HLT-NAACL 2003 workshop on Text meaning – Volume 9*, pages 46–53. Association for Computing Machinery, 2003.
17. E. T. Mueller. Story understanding. In L. Nadel, editor, *Encyclopedia of Cognitive Science*. John Wiley & Sons, Inc., 2008.
18. R. E. Nisbett and T. D. Wilson. Telling more than we can know: Verbal reports on mental processes. *Psychological Review*, 84:231–259, 1977.
19. R. Pérez y Pérez and M. Sharples. Mexica: A computer model of a cognitive account of creative writing. *Journal of Experimental and Theoretical Artificial Intelligence*, 13(2):119–139, 2001.
20. J. Perner and H. Wimmer. "John *thinks* that Mary *thinks* that ...", Attribution of second-order beliefs by 5- to 10-year-old children. *Journal of Experimental Child Psychology*, 39:437–471, 1985.
21. V. Propp. *Морфология сказки*. Akademija, Leningrad, 1928.
22. M. J. Rattermann and D. Gentner. Analogy and similarity: Determinants of accessibility and inferential soundness. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, pages 23–35, Hillsdale NJ, 1987. Lawrence Erlbaum.
23. D. E. Rumelhart. Notes on a schema for stories. In D. G. Bobrow and C. A. M., editors, *Representation and Understanding: Studies in cognitive science*, pages 211–236. Academic Press, 1975.
24. L. Schäfer. Models for digital storytelling and interactive narratives. In A. Clarke, editor, *COSIGN 2004 Proceedings*, pages 148–155, 2004.
25. L. K. Schubert and C. H. Hwang. Episodic Logic meets Little Red Riding Hood: A comprehensive natural representation for language understanding. In L. Iwanska and S. C. Shapiro, editors, *Natural language processing and knowledge representation: Language for Knowledge and Knowledge for Language*, pages 111–174. MIT/AAAI Press, 2000.
26. M. Si, S. C. Marsella, and D. V. Pynadath. Thespian: Using multi-agent fitting to craft interactive drama. In M. Pechoucek, D. Steiner, and S. Thompson, editors, *International Conference on Autonomous Agents. Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems. Utrecht, The Netherlands, July 25–29, 2005*, pages 21–28, 2005.
27. U. Spierling, D. Grasbon, N. Braun, and I. Iurgel. Setting the scene: playing digital director in interactive storytelling and creation. *Computers & Graphics*, 26(1):31–44, 2002.
28. D. O. Stahl. Evolution of smart$_n$ players. *Games and Economic Behaviour*, 5:604–617, 1993.

29. D. O. Stahl and P. W. Wilson. On players' models of other players: Theory and experimental evidence. *Games and Economic Behavior*, 10:218–254, 1995.
30. R. Verbrugge and L. Mol. Learning to apply theory of mind. *Journal for Logic, Language and Information*, 2008. to appear.
31. A. Witzel and J. Zvesper. Higher-order knowledge in computer games. In F. Guerin, B. Löwe, and W. Vasconcelos, editors, *AISB 2008 Convention. Communication, Interaction and Social Intelligence, 1st–4th April 2008, University of Aberdeen. Volume 9: Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning*, pages 68–72, Aberdeen, 2008.
32. R. M. Young. Story and discourse: A bipartite model of narrative generation in virtual worlds. *Interaction Studies*, 8:177–208, 2007.
33. R. M. Young, M. O. Riedl, M. Branly, A. Jhala, R. Martin, and C. Saretto. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development*, 1, 2004.

A Petri Net based Prototype for MAS Organisation Middleware

Michael Köhler-Bußmeier, Matthias Wester-Ebbinghaus

University of Hamburg, Department for Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
(koehler,wester)@informatik.uni-hamburg.de

Abstract. This contribution presents a prototype middleware for the complete organisational teamwork, like team formation, negotiation, team planning, coordination, and transformation.
Organisations are modelled in SONAR, a Petri net based specification formalism for multi-agent organisations.
SONAR models are specific enough to generate all the configuration data – in the spirit of the model driven architecture idea – for our middleware in an automated way.

1 Organisation-centered Design of Multi-Agent Systems

Organisation-oriented software engineering is a discipline which incorporates research trend from distributed artificial intelligence, agent-oriented software engineering, and business information systems (cf. [1, 2] for an overview). The basic metaphors built around the interplay of the macro level (i.e. the organisation) and the micro level (i.e. the agent). Organisation-oriented software models are very interesting especially for self- and re-organising systems since the systems structure is taken into account by representing it explicitly.

The following work is based on the organisation model SONAR which we have presented in [3], and its theoretical properties in [4]. The paper presents a prototype middleware for the complete organisational teamwork.

First of all we like to have a rapid development of our middleware prototype. Therefore we need a specification language with powerful and high-level features, like pattern matching or synchronisation patterns. As a second requirement we like the specifications to be executable. As a third requirement we are interested in well established analysis techniques to study the prototype's behaviour. As a fourth requirement we like the specification to be as close to the formal SONAR model as possible. And as a fifth requirement we want that the middleware can be generated from the SONAR model automatically.

Since SONAR-models are based on Petri nets we have chosen high-level Petri nets [5] as our specification language, so we can reuse the models basic structures by enriching them with high-level features, like data types, arc inscription function etc. Petri nets are well known for their precise and intuitive semantic

and their well established analysis techniques, including model checking or linear algebraic techniques. We choose the formalism of reference nets, a dialect of high-level nets which are simulated by the RENEW simulator [6].

The contribution is structured as follows: Section 2 gives a formal model of organisations based on Petri nets. The set of possible teams is modelled as a so-called R/D net. Section 3 describes stratified SONAR models which are used for self-organisation where the higher-levels of the system reorganise the lower ones. Section 4 illustrates the structure of our target system: SONAR-models are compiled into a multi-agent system consisting of so called Position-Agents, i.e. agents that are responsible for the organisational constraints. Section 5 describes the middleware prototype generated from SONAR-models. The middleware executes and controls all the organisational activities, like team formation, negotiation, team planning, coordination, and transformation. The paper closes with a conclusion and an outlook.

2 The Underlying Theoretical Model: SONAR

In the following we give a short introduction into our modelling formalism, called SONAR. A detailed discussion of the formalism can be found in [3], its theoretical properties are studied in [4]. A SONAR-model encompasses (i) a data ontology, (ii) a set of interaction models (called *distributed workflow nets*), (iii) a model, that describes the team-based delegation of tasks (called *role/delegation nets*), (iv) a network of organisational positions, and (v) a set of transformation rules (cf. [4, 3] for details).

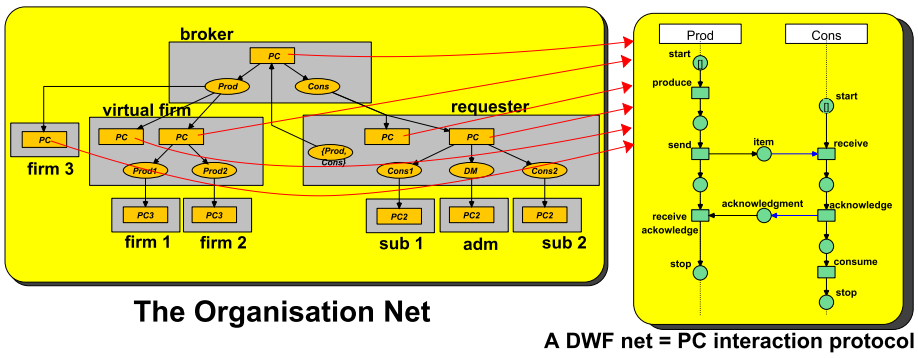


Fig. 1. A SONAR-Model

Figure 1 illustrates the relationship between the SONAR interaction model, the delegation model and the position network (aspects (ii) to (iv), omitting data-related aspects (i) and deferring transformation rules (v) until Section 3). It describes the relationship between some positions (*broker*, *virtual firm*, *requester*, etc.) in terms of their respective roles (*Producer*, *Consumer* etc.) and

associated delegation links. In this scenario, we have a requester and two suppliers of some product. Coupling between them is provided by a broker. (Note that for this simplified model brokerage is an easy job, since there are exactly two producers and one consumer. In general, we have several instances for both groups with a broad variety of quality parameters making brokerage a real problem.) From a more fine-grained perspective, the requester and one of the suppliers consist of delegation networks themselves. For example, in the case of the *virtual firm* supplier, we can identify a management level and two subcontractors. The two subcontractors may be legally independent firms that integrate their core competencies in order to form a virtual enterprise (e.g. separating fabrication of product parts from their assembly). The coupling between the firms constituting the virtual enterprise is apt to be tighter and more persistent than between requester and supplier at the next higher system level, which provides more of a market-based and on-the-spot connection.

Basic Notations SONAR relies on the formalism of Petri nets. A *Petri net* is a tuple $N = (P, T, F)$ where P is a set of places, T is a set of transitions, disjoint from P, i.e. $P \cap T = \emptyset$, and $F \subseteq (P \times T \cup T \times P)$ is the flow relation. In the following we identify the relation F with its characteristic function $F : (P \times T \cup T \times P) \to \{0, 1\}$. A Petri net $K = (B, E, \lessdot)$ is called a *causal net* whenever $\lessdot^*$ is a partial order. A marking $\boldsymbol{m} : P \to \mathbb{N}$ assigns to each place the number of tokens lying there. The *preset* of a node y is ${}^\bullet y := (\_\ F\ y)$ and *postset* is $y^\bullet := (y\ F\ \_)$. The set of places with empty preset is ${}^\circ N = \{x \in P \cup T \mid {}^\bullet x = \emptyset\}$. The set of places with empty postset is $N^\circ = \{x \in P \cup T \mid x^\bullet = \emptyset\}$. For causal nets ${}^\circ N$ are the minimal and N° the maximal elements. A transition $t \in T$ of a net N is enabled in the marking $\boldsymbol{m}$ iff $\forall p \in P : \boldsymbol{m}(p) \geq F(p, t)$ holds. The successor marking when firing t is $\boldsymbol{m}'(p) = \boldsymbol{m}(p) - F(p, t) + F(t, p)$. We denote the enabling of t in marking $\boldsymbol{m}$ by $\boldsymbol{m} \xrightarrow[N]{t}$. Firing of t is denoted by $\boldsymbol{m} \xrightarrow[N]{t} \boldsymbol{m}'$. The net N is omitted if it is clear from the context. The notation is extended to words $w = t_1 \cdots t_n \in T^*$ of transitions in the obvious way. For a general introduction into Petri nets we refer to [5].

Role/Delegation Nets In SONAR a formal organisation is characterised as a network of *organisational positions*. Each position is responsible for the execution or delegation of several tasks. In our model *Role/Delegation (R/D) nets* [4] are used to describe all the information about task delegation. A R/D net is a Petri net (P, T, F) where each task is modelled by a place p and each task implementation (delegation/execution) is modelled by a transition t. A place p with ${}^\bullet p = \emptyset$ models an *initial task*, while ${}^\bullet p \neq \emptyset$ models a *subtask*. Transitions $t \in T$ with $t^\bullet \neq \emptyset$ are called delegative, transitions with $t^\bullet = \emptyset$ are called executive. Each place p is labelled by a role $R(p)$ and each transition t with a DWF net $D(t)$ (see below). An example R/D net is given on the left side of Figure 1. The positions of our example model in Figure 1 are drawn as grey boxes. It has the positions: *broker*, *virtual firm*, *requester*, etc.

Distributed Workflow Nets A *distributed workflow net* (DWF net) is a multi-party version of the well-known workflow nets [7] where the parties are called *roles*. Roles are used in DWF nets to abstract from concrete agents. For example, the two roles *Producer* and *Consumer* have the same form of trading interaction no matter which agent is producing or consuming. The right side of Figure 1 shows the DWF net *PC* that describes the interaction between both roles: First the producer executes the activity *produce*, then *sends* the produced *item* to the consumer, who *receives* it. The consumer sends an *acknowledge* to the producer before he *consumes* the item.[1] Technically speaking roles are some kind of type for an agent describing its behaviour. Note that agents usually implement several roles.

Positions and Organisations Positions define which entity is responsible for the existing delegation tasks.[2] This relationship is modelled as a set of disjoint subsets of the nodes $P \cup T$ of the R/D net.[3] Each initial task (i.e. the places p with ${}^\bullet p = \emptyset$) are the starting points of organisational activity.

Definition 1. *Let $\mathcal{D}$ be a DWF universe and $\mathcal{R}$ the role universe. A* (formal) organisation net *is the tuple $Org = (N, \mathcal{O}, R, D)$ where:*

1. *$N = (P, T, F)$ is a Petri net with $|p^\bullet| > 0$ for $p \in P$ and $|{}^\bullet t| = 1$ for $t \in T$.*
2. *$\mathcal{O}$ is a partition on the set $P \cup T$. An element $O \in \mathcal{O}$ is called* position.
3. *$R : P \to \mathcal{R}$ is the role assignment.*
4. *$D : T \to \mathcal{D}$ is the DWF net assignment.*

In a well-formed organisation the roles of the DWF net $D(t)$ are consistently related to the roles of the places in the preset and the postset of t such that no role behaviour is lost or added during the delegation. In a well-formed organisation, termination of the interaction described by a DWF net is guaranteed. Cf. [4] for details.

In general a delegation t comes along with a behaviour refinement. In our example, the position *requester* implements the role *Cons* by generating subtasks for the roles *Cons 1*, *DM*, and *Cons 2*. These subtasks are handled by the positions *sub 1*, *adm*, and *sub 2* that implement their respective roles according to the DWF PC_2 (given in Figure 2) which decomposes the behaviour of role *Cons* into the composition of *Cons 1*,

[1] To simplify the presentation we have omitted all data-related aspects in our discussion of distributed workflow nets. In Sonar each DWF net uses data object based on the model's ontology.

[2] The main distinction between roles and position ist that – unlike roles – positions are situated in the organisational network, they implement roles and are equipped with resources.

[3] There is a close connection between organisation nets and the commonly used organisation charts. In fact, organisation charts are a special sub-case of our model. Organisation nets encode the information about delegation structures – similar to charts – and also about the delegation/execution choices of tasks, which is not present in charts. If one fuses all nodes of each position $O \in \mathcal{O}$ into one single node, one obtains a graph which represents the organisation's chart. Obviously, this construction removes all information about the organisational processes.

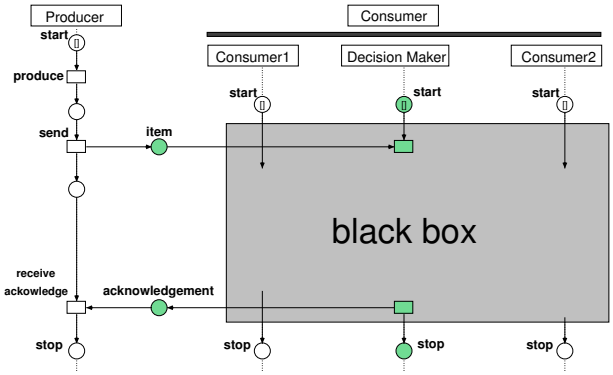


Fig. 2. Protocol with Refined Consumer Role

DM, and *Cons 2*. The role *Prod* is defined equally in both DWFs PC and PC_2. In well formed organisations it is guaranteed that the service generated from the refined DWF PC_2 and the roles *Cons 1*, *DM*, and *Cons 2* has the same communication behaviour as the service generated from the original DWF PC and the role *Cons*:

$$PC[Cons] \simeq PC_2[Cons_1, DM, Cons_2]$$

Similarily we have a DWF PC_3 (not shown here) which decomposes the behaviour of role *Prod* into *Prod1* and *Prod 2*. This refinement is used by the positions *firm 1* and *firm 2* in Figure 1.

Teams Team formation can be expressed in a very elegant way: If one marks one initial place of an organisation net *Org* with a token, each firing process of the Petri net models a possible delegation process. More precisely, the *token game* is identical to the team formation process (cf. Theorem 4.2 in [4]). It generates a *team net* and a *team DWF*: Teams are modelled as an acyclic R/D nets. More precisely: An R/D net G is called a *team net* if it is a connected causal net (i.e. an acyclic net) with exactly one minimal node: $|^\circ G| = 1$.[4] The team DWF is derived from the DWF inscriptions $D(t)$ of the team's maximal nodes $t \in G^\circ$, i.e. the leaves of the team net.

3 Stratified SONAR-Models

As another aspect, SONAR-models are equipped with transformation rules. Transformation rules describe which modifications of the given model are allowed. They are specified as graph rewrite rules [8]. The minimal requirement for rules in SONAR is that they must preserve the correctness of the given organisational model. Cf. [3] for more details.

[4] As a nice theoretical byproduct we can generate all the team nets using unfoldings, a well known construction for Petri nets (cf. [4] for details).

In SONAR transformations are not performed by the modeller – they are part of the model itself. Therefore we assume that a SONAR model is *stratified* by models of different levels. The main idea is that the parts of an organisation model that belong to the level n are allowed to modify those parts that belong to levels $k < n$ but not to higher ones.

To define a stratified model we start to define Petri nets of level n. For each $n \in \mathbb{N}$ we assume a countable infinite set of places, denoted $\mathcal{P}_n$, and a countable infinite set of transitions of level n, denoted by $\mathcal{T}_n$. We assume that for all m and n $\mathcal{P}_m$ and $\mathcal{T}_n$ are disjoint.

The universe of places is $\mathcal{P}_u := \bigcup_{n\in\mathbb{N}} \mathcal{P}_n$ and the universe of transitions is $\mathcal{T}_u := \bigcup_{n\in\mathbb{N}} \mathcal{T}_n$.

A Petri net $N = (P, T, F)$ is of level n whenever its places and transitions are, i.e. if $P \subseteq \mathcal{P}_n$ and $T \subseteq \mathcal{T}_n$ hold.

Roles are typed, too. For each $n \in \mathbb{N}$ we assume the roleset of level n, denoted $\mathcal{R}_n$. The whole role universe is $\mathcal{R} := \bigcup_{n\in\mathbb{N}} \mathcal{R}_n$.

Now we define the set of allowed transformations of level n, which is denoted $\mathcal{TM}_n$. Assume that we have already defined the set of transformations $\mathcal{TM}_k$ for each level $k < n$. A pair (D, λ) is called a *DWF of level* n, whenever D is a DWF and $\lambda : T_D \to \left(\{id\} \cup \bigcup_{k=1}^{n-1} \mathcal{TM}_k\right)$ is a transformation mapping, that assigns to each transition t of the net D a transformation $\lambda(t)$ of a level less than n. The set of all DWF nets of level $n > 0$ is denoted by $\mathcal{D}_n$. The whole DWF net universe is $\mathcal{D} := \bigcup_{n\in\mathbb{N}} \mathcal{D}_n$.

We can assume that the organisation $(N, \mathcal{O}, R, D)$ as defined in Def 1 is a model over the unions, i.e. over $\mathcal{P}$, $\mathcal{T}$, $\mathcal{R}$ and $\mathcal{D}$.

We define an organisation of level $n > 0$ as the following special case.

Definition 2. *An organisation $(N, \mathcal{O}, R, D)$ is of level $n > 0$, whenever:*

1. *$N = (P, T, F)$ is a Petri net of level n, i.e. $P \subseteq \mathcal{P}_n$ and $T \subseteq \mathcal{T}_n$.*
2. *R is of level n, i.e. we have $R : P_N \to \mathcal{R}_n$.*
3. *D is of level n, i.e. we have $D : T_N \to \mathcal{D}_n$.*

The set of all organisations of level n is denoted by $\mathcal{ORG}_n$. The set of all organisations by $\mathcal{ORG}$.

A transformation of level $n > 0$ is a function that maps the set $\mathcal{ORG}_n$ onto itself: $f_n : \mathcal{ORG}_n \to \mathcal{ORG}_n$. With $\mathcal{TM}_n$ we denote the set of all these transformations.

Note that we have a cyclic dependency on the concepts: Transformations modify organisations and organisations are built up from DWF nets, that incorporate transformations. This definitorial cycle is broken by the stratification, since transformations of level n are defined on organisations of level n which incorporate DWF nets that must have transformations of a level less than n.

Definition 3. *For an organisation $Org = (N, \mathcal{O}, R, D)$ with $N = (P, T, F)$ we define for each $n \in \mathbb{N}$ the* restriction *to the level n as:*

$$Org_n := (N_n, \mathcal{O}_n, R|P_n, D|T_n)$$

1. $N_n := (P_n, T_n, F_n)$ *with* $P_n := P \cap \mathcal{P}_n$, $T_n := T \cap \mathcal{T}_n$ *and* $F_n := F \cap (P_n \cup T_n)^2$.
2. $\mathcal{O}_n := \{O \cap (\mathcal{P}_n \cup \mathcal{T}_n) \mid O \in \mathcal{O}\}$

A stratified organisation *is an organisation* $Org^* = (N, \mathcal{O}, R, D)$ *such that each restriction* Org_n *is an organisation of level* n *and there are no "mixed" arcs, i.e.* $F \subseteq \bigcup_{n \in \mathbb{N}} (\mathcal{P}_n \times \mathcal{T}_n) \cup (\mathcal{T}_n \times \mathcal{P}_n)$ *holds.*

The set of all stratified organisations is denoted by $\mathcal{ORG}^*$.

A stratified organisation Org^* can be understood by the set of its restrictions: Figure 3 shows the restriction Org_1, Figure 4 is the restriction Org_2. For both restrictions we have depicted the portions which are global for the whole organisation.

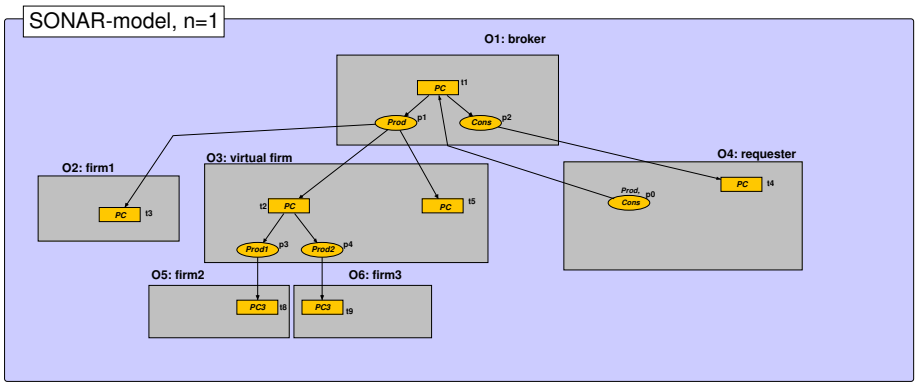


Fig. 3. The Organisation Model Org_1

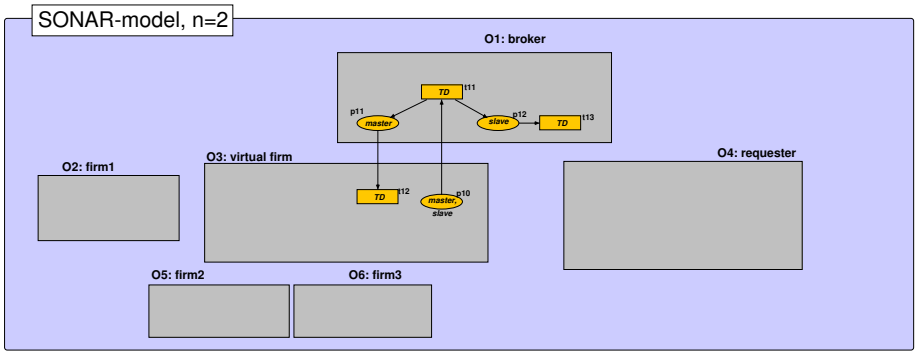


Fig. 4. The Organisation Model Org_2

For the level $n = 2$ the DWF nets are label by transformations $\lambda(t)$. Figure 5 shows a DWF net of level $n = 2$. It has the two roles *master* and *slave*. The *master* has the objective to delete the transition t_5 of the organisation model Org_1, while *slave* has the objective to delete all referencing arcs to t_5 (here: the arc from p_1 to t_5). It is clear that the position that implements the role *master* has to be the position that owns t_5, i.e. $O(t_5) = O_3$. Similarly, the role *slave* has to be implemented from the position that references t_5, i.e. $O(^\bullet t_5) = O_1$. These constraints are directly expressed by the model Org_2 in Figure 4.

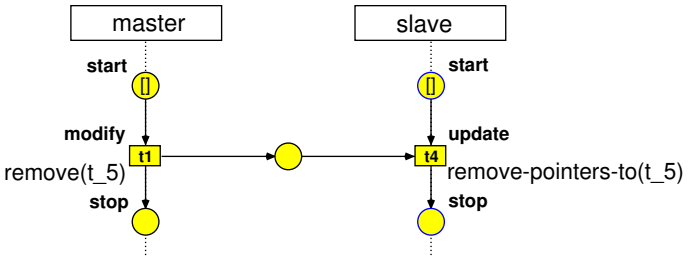


Fig. 5. A Transformation DWF Net

Note that for all practical models most of the restrictions Org_n must be empty since Org is a finite net.

Each transformation $f_n : \mathcal{ORG}_n \to \mathcal{ORG}_n$ is extended to a transformation $\hat{f} : \mathcal{ORG}^* \to \mathcal{ORG}^*$ on the whole class, by choosing the identity for all the other elements:

$$\hat{f}(x) = \begin{cases} f_n(x), & \text{if } x \in \mathcal{ORG}_n \\ id, & \text{otherwise} \end{cases}$$

Note that the organisational teamwork respects the stratification: Assume a given stratified organisation Org^*. Since there are no mixed arcs each team G belongs to exactly one level n. This team generates a team transformation $\lambda(G) : \bigcup_{k=1}^{n-1} \mathcal{ORG}_k \to \bigcup_{k=1}^{n-1} \mathcal{ORG}_k$. By construction, this team transformation effects only those parts of the organisation, that are of lower level, i.e. the models $(Org_k)_{k<n}$. The other ones $(Org_k)_{k\geq n}$ remain unmodified.

If we look at the definition we can see that the basic organisational models can be seen as a stratified organisation which consists of level $n = 1$ only.

4 Organisational Position Network Activities

Now that we have obtained a precise picture of what constitutes a formal organisation according to our approach, we can elaborate on the activities of a multi-agent systems behaving according to a SONAR-model. The basic idea is

quite simple: With each position of the SONAR-model we associate one dedicated agent, called the *organisational position agent* (OPA).

Figure 6 illustrates our specific philosophy concerning MAS organisations utilising the middleware approach. In SONAR, we describe a formal organisation in terms of interrelated *organisational positions*. Compared to other middleware layers, we advocate complete distribution. Instead of introducing one or more middleware managers that watch over the whole organisation (cf. the manager in $\mathcal{S}$-$\mathcal{M}$OISE$^+$ [9]) or at least over considerable parts (cf. institution, scene, transition managers in AMELIE [10]), we associate each position with its own OPA.[5]

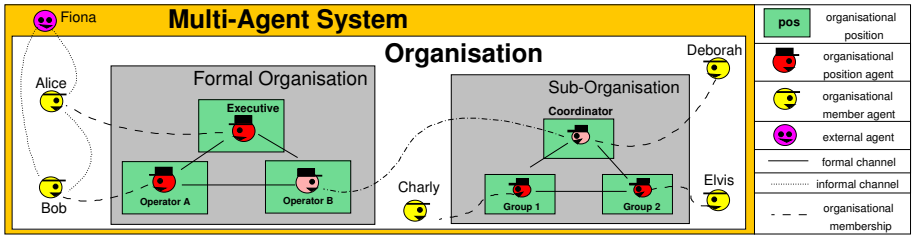


Fig. 6. An Organisation as an OPA/OMA Network

An OPA network embodies a formal organisation. An OPA represents an organisational *artifact* and not a *member/employee* of the organisation. However, each OPA represents a conceptual connection point for an organisational member agent (OMA). An organisation is not complete without OMAs. It depends on domain agents that actually carry out organisational tasks, make decisions where required and thus implement/occupy the formal positions. Note that an OMA can be an artificial as well as a human agent. An OPA both enables and constrains organisational behaviour of its associated OMA. Only via an OPA an OMA can effect the organisation and only in a way that is in conformance with the OPA's specification. In addition, the OPA network as a whole relieves its associated OMAs of a considerable amount of organisational overhead by automating coordination and administration. To put it differently, an OPA offers its OMA a "behaviour corridor" for organisational membership. OMAs might of course only be partially involved in an organisation and have relationships to multiple other agents than their OPA (even to agents completely external to the organisation). From the perspective of the organisation, all other ties than the OPA-OMA link are considered as informal connections.

To conclude, an OPA embodies two conceptual interfaces, the first one between micro and macro level (one OPA versus a network of OPAs) and the

[5] We provide a more detailed comparison of our approach to other MAS middleware approaches in [11] where we also derive conclusions concerning best fits between different approaches and application contexts.

second one between formal and informal aspects of an organisation (OPA versus OMA). We can make additional use of this twofold interface. Whenever we have a system of systems setting with multiple scopes or domains of authority (e.g. virtual organisations strategic alliances, organisational fields), we can let an OPA of a given (sub-)organisation act as a member towards another OPA of another organisation. This basically combines the middleware perspective with a holonic perspective (cf. [12]) and is not as easily to be conceptualised in the context of other middleware approaches that take a less distributed/modular perspective. In this paper the aspect of holonic systems is not discussed any further – cf. [13] for an in depth discussion.

4.1 Organisational Teamwork

As explicated in Section 3, positions are global for all levels of a stratified formal organisation. Consequently, a position agent network as described in the previous subsection is *re-used* for any level of a stratified organisation. For each given level n, we distinguish between organisational activities of first- and of second-order (where second-order activities correspond to first-order activities of levels $k > n$). First-order activities are carried out within the same organisational level n and this level is referred to as a static context.

- *Team Formation:* Teams are formed in the course of an iterated delegation procedure in a top-down manner. Starting with an initial organisational task to be carried out, successive task decompositions (i.e. role refinements) are carried out and subtasks (i.e. sub-roles to be implemented) are delegated further. A team net according to Section 2 results that consists of the positions that were involved in the delegation procedure.
- *Team Plan Formation:* After a team has been formed, a compromise has to be found concerning how the corresponding team DWF (cf. Section 2) is to be carried out as it typically leaves various alternatives of going one way or the other. A compromise is found in a bottom-up manner. The "leaf" positions of the team net (those that actually participate in the DWF) tell their preferences and the intermediary team positions iteratively seek compromises between the preferences/compromise results of subordinates. The final compromise is the team plan.
- *Team Plan Execution:* As the team plan is still a DWF (cleansed of ambiguities) that describes interactions between team positions, team plan execution follows straightforward.[6]
- *Hierarchic propagation:* If a holonic approach as illustrated in Figure 6 is chosen, team activities that span multiple organisations are propagated accordingly.

Second-order activities of level n (first-order activities of level $k > n$) target at the reorganisation of level n. In this case, level n is consequently regarded as a variable.

[6] For the time being, we do not address the topic that team plan execution might fail and what rescue efforts this might entail.

- *Evaluation:* Organisational performance is monitored and evaluated in order to estimate prospects of transformations. To estimate whether an organisational transformation would improve organisational performance, we introduce *metrics* that assign a multi-dimensional assessment to a formal organisation. In addition to the Petri net-based specifications of the previous section, there may exist additional teamwork constraints and parameters that may be referred to. How to measure the quality of an organisational structure is generally a very difficult topic and highly contingent. We will not pursue it further in this paper.
- *Organisational Transformations:* As described in Section 2, transformations can either be applied to a formal organisation externally or be carried out by the positions themselves as transformation teams (cf. exogenous versus endogenous reorganisation [14]). In the latter case, transformations are typically triggered by the above mentioned evaluations. But it might also be the case that a new constraint or directive has been imposed and the organisation has to comply.

So far we have described the activities just with reference to positions. But actually, each position consists of a formal (the OPA as an organisational artifact) and an informal part (the OMA as a domain member). We have further stated that an OPA network relieves its associated OMAs of a great part of the organisational overhead by automation of administrative and coordination activities. It is exactly the *generic* part of the above mentioned activities that is automated by the OPA network: Team formation, team plan formation, team plan execution and applying given metrics always follow the same mechanics and OMAs only have to enter the equation where domain actions have to be carried out or domain-dependent decisions have to be made.

In the following section we present the specification of the OPA teamwork. All OPAs share a common structure which we call the *generic OPA (GOPA)*. An OPA O is an instance of this GOPA that is parametrised by that part of the organisational model that describes O, i.e. its inner structure (subtask and delegation/execution activities) and all the surrounding OPAs. In the following presentation we will focus on the interaction between position agents and will neither discuss the internal architecture of OPAs (i.e. their reasoner etc.) nor the OPA-OMA interaction itself. A more detailed discussion of the GOPA-architecture can be found in [15].

5 The Middleware Prototype based on High-Level Nets

In the following we demonstrate the compilation of an organisation SONAR-model Org^* into a middleware that executes the associated teamwork. A SONAR-model is semantically rich enough to provide all the information to perform this MDA-like operation. The aspects of this compilation and the resulting prototype are discussed using the stratified organisation as introduced above in Figures 3 and 4. The organisational model induces the complete teamwork, i.e. on level $n = 1$ the

"basic" business processes (for the example defined by the DWF nets PC and PC_3) and on level $n = 2$ the reorganisational processes (here: the transformation by deleting t_5 in the position *virtual firm* – specified by the DWF in Figure 5).

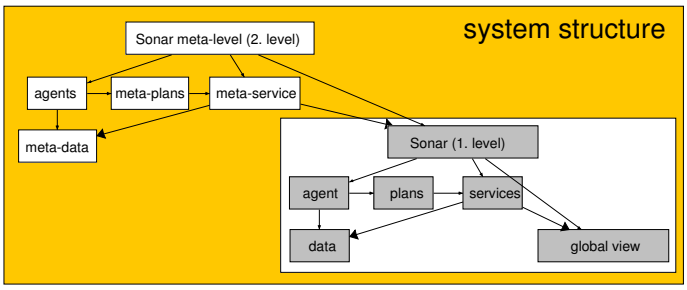


Fig. 7. Architecture of the Middleware generated from the SONAR-model of Fig. 3/4

The teamwork prototype generated from the SONAR-model of Fig. 3 and 4 is specified by a high-level Petri net, namely as a reference net. This is beneficial for two reasons: (i) the translation result is very close to the original specification, since the prototype directly incorporates the main net structure of the SONAR-model; and (2) the prototype is directly executable using the open-source Petri net simulator RENEW [6].

The middleware has a recursive system architecture, as depicted in Figure 7. The subsystem for the level $n = 1$ contains all the position *agents*, the *services* (i.e. the DWF nets), and the *plans* (i.e. objects that control the teamwork execution of the team DWF nets; team DWF nets allow multiple processes and this nondeterminism is solved by plans). The execution of a team DWF has effects on the agents' local data and due to this also on the *global view* on the system data.

On the meta-level, i.e. for the level $n = 2$, we have the same basic structure, consisting of the same *agents*, their *meta-services* (i.e. their DWF nets used for transformations), and the *meta-plans*. The meta-level teamwork has effects on its data which is the complete model of level $n = 1$.

In the following we explain the high-level net specification. It is out of the scope of this paper to explain every detail so we restrict ourself to an explanation of the overall process.

First, we start with the team DWF nets. For their middleware versions we add inscriptions which synchronise each instance of a team DWF net with the agents implementing the net's roles. More precisely: The team DWF net synchronises with the agents' plans. If an agent implements the role r in a team DWF net N it generates a plan c (short for controller). The plan c selects among all possible events assigned to r the next event to be executed. Since a team DWF net usually has several roles it is synchronised with several plans. In general, each agent may participate in several team DWFs at the same time. It lies in the

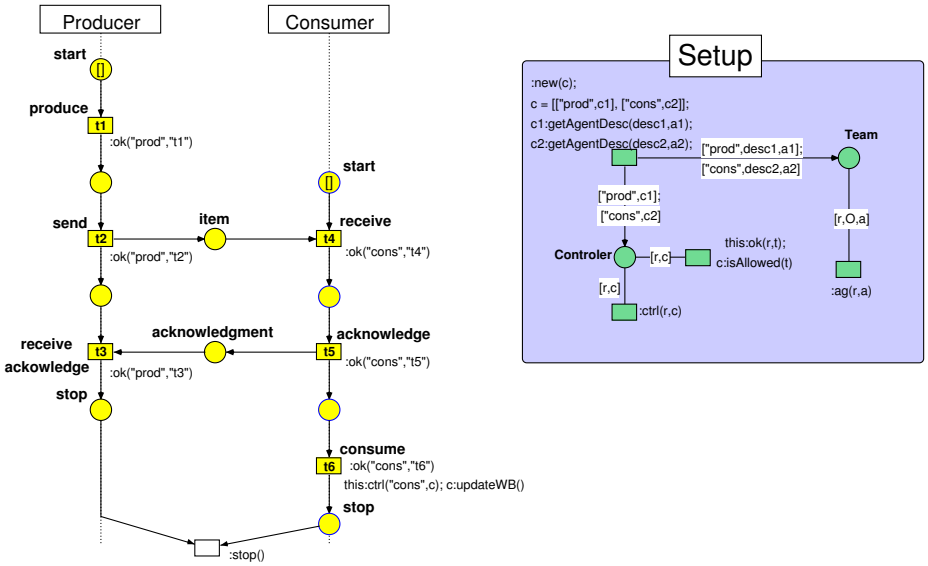


Fig. 8. Prototype: The controlled DWF net PC

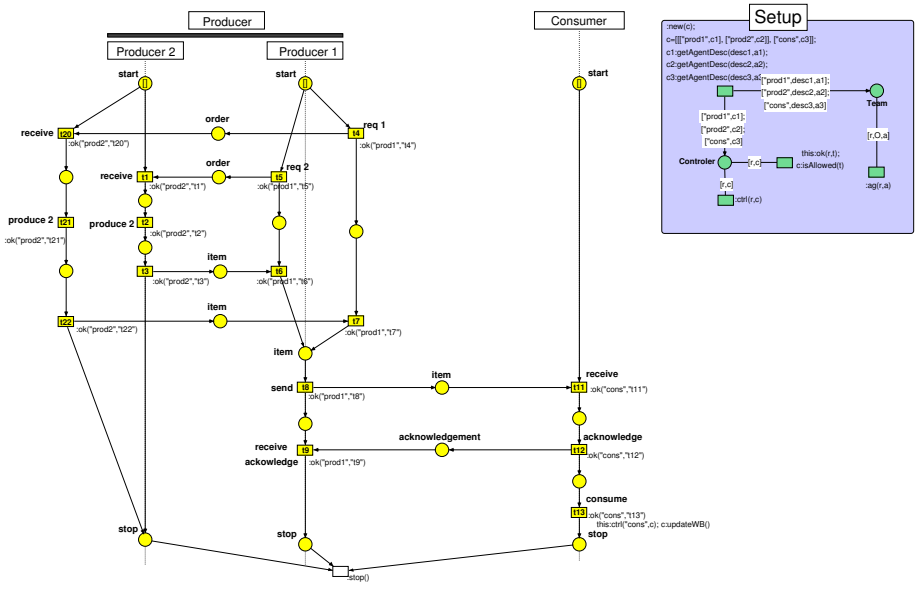


Fig. 9. Prototype: The controlled DWF net PC_3

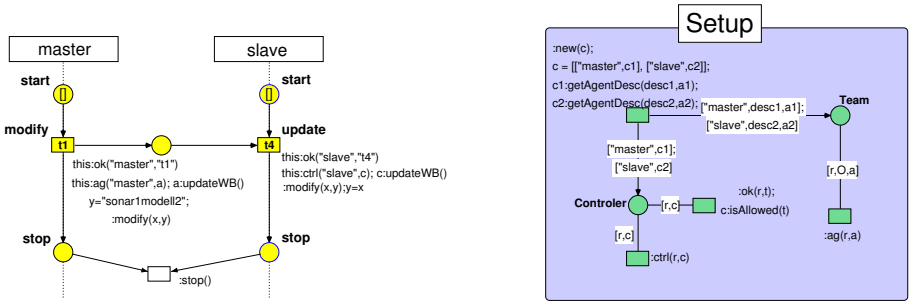


Fig. 10. Prototype: The Transformation DWF net

agent's responsibility to coordinate all its plans, e.g. to avoid race conditions, dirty read/writes etc.

For the team DWF nets PC and PC_3 of our running example the result is shown in Fig. 8 and Fig. 9. The synchronisation with the plan is provided by synchronisations channels of the form $:\mathsf{ok}(\mathsf{prod}, t_1)$ at the DWF net's event t_1. To enable the event t_1 the plan c has to enable the counterpart of this channel at his side: $c{:}\mathsf{isAllowed}(t_1)$. For the team DWF net used for the transformation, given in Fig. 5, we obtain the prototype from Fig. 10. In all cases, we omit details concerning data exchange between agents.

In SONAR, teams, team DWFs and team plans are the result of organisational processes. Organisational processes are identical for all levels. They consists of the following phases:

1. In the first phase the organisation is explored to select the team agents.
2. In the second phase the team as an object of discurs is generated by composing teamtrees recursively .
3. The teamtree is announced among all the team member agents, starting at the root of the teamtree.
4. The executing team agents (i.e. the leaves of the teamtree) construct partial local plans related to the team DWF net. These partial plans are recursively processed via negotiation by the ancestors in the team tree, resulting in a global plan. This global plan is a compromise of the partial local plans.
5. The global teamplan is send back to all the team members and localised by them.
6. The team generates an instance of the team DWF net, assigns all the local plans to it, and starts the execution.

The prototype directly expresses these six phases. The translation of the organisation model Org_1 is shown in Figures 11/12. One clearly recognises six parts of the nets which all share the basic structure of the original specification of Org_1 in Figure 3. Similarly we obtain the prototype of Figure 13/14 from the specification Org_2 in Fig. 4.

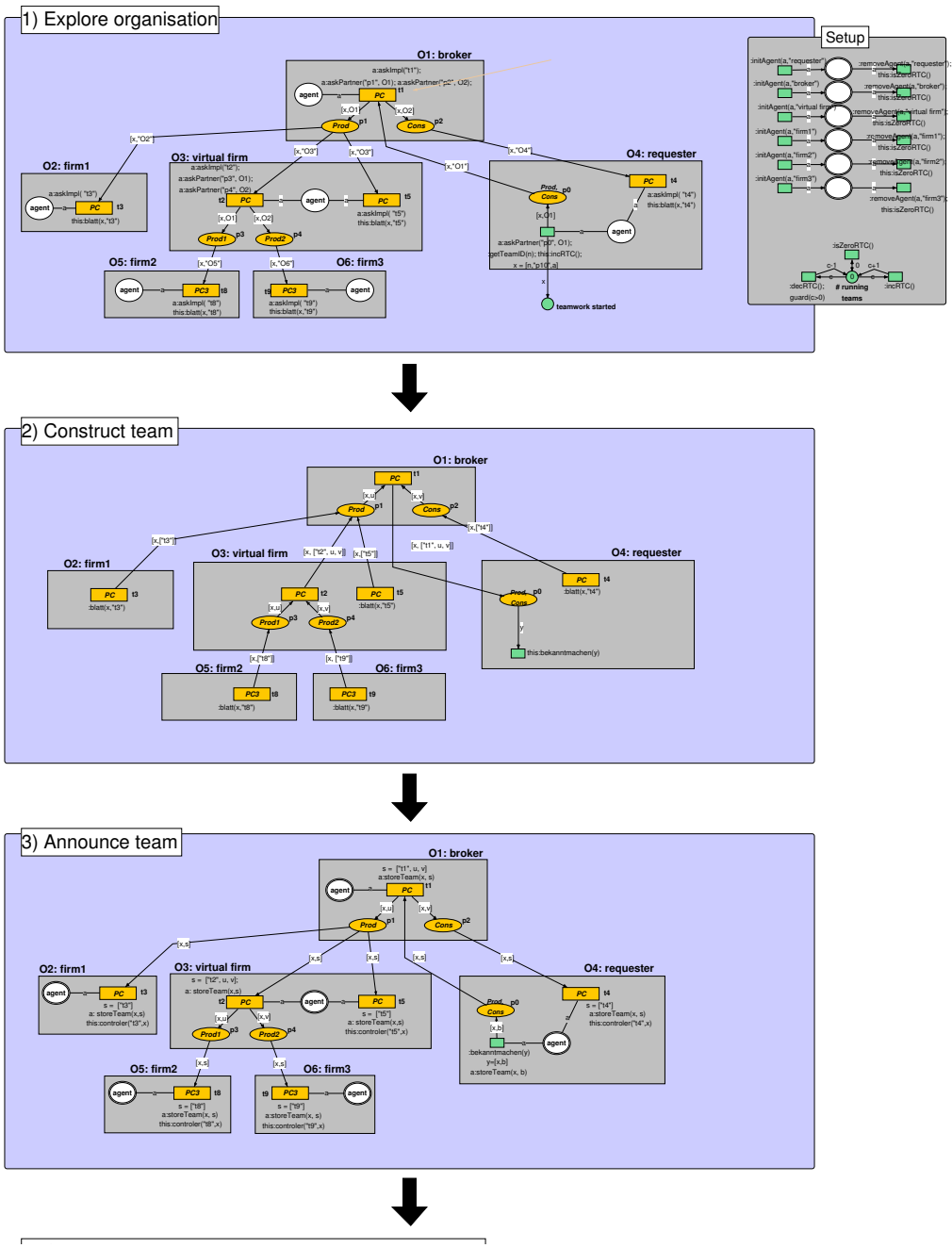


Fig. 11. SONAR-middleware for the organisation Org_1 of Fig. 3

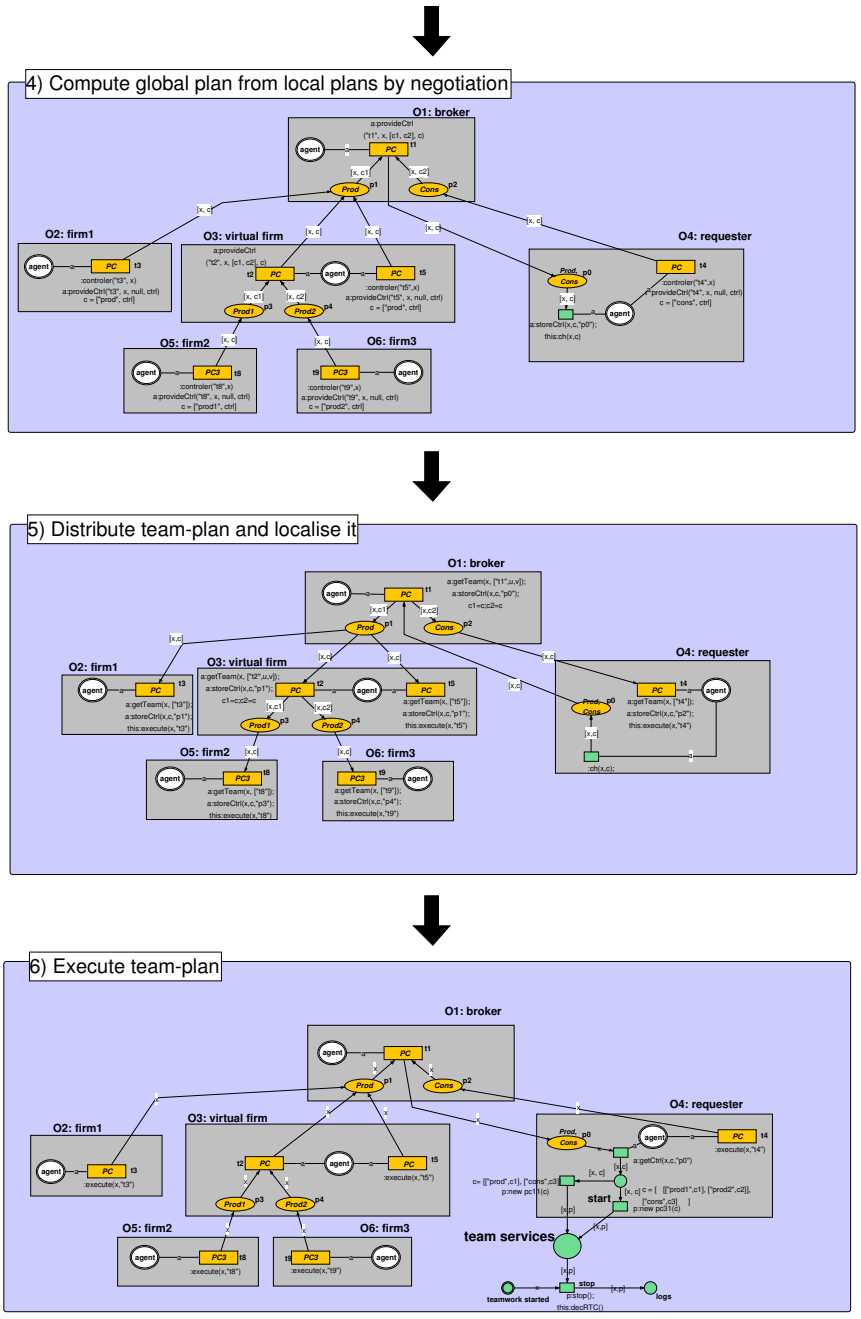


Fig. 12. SONAR-middleware for the organisation Org_1 of Fig. 3 (continued)

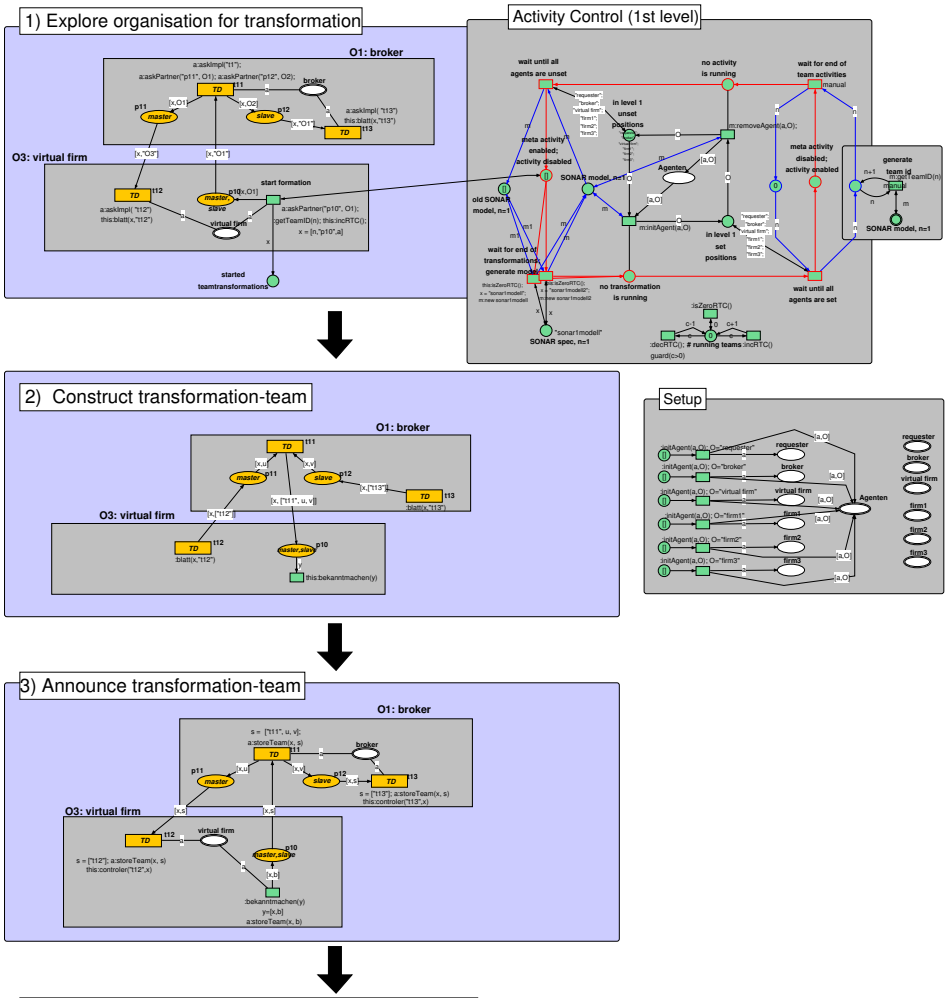


Fig. 13. SONAR-middleware for the organisation Org_2 of Fig. 4

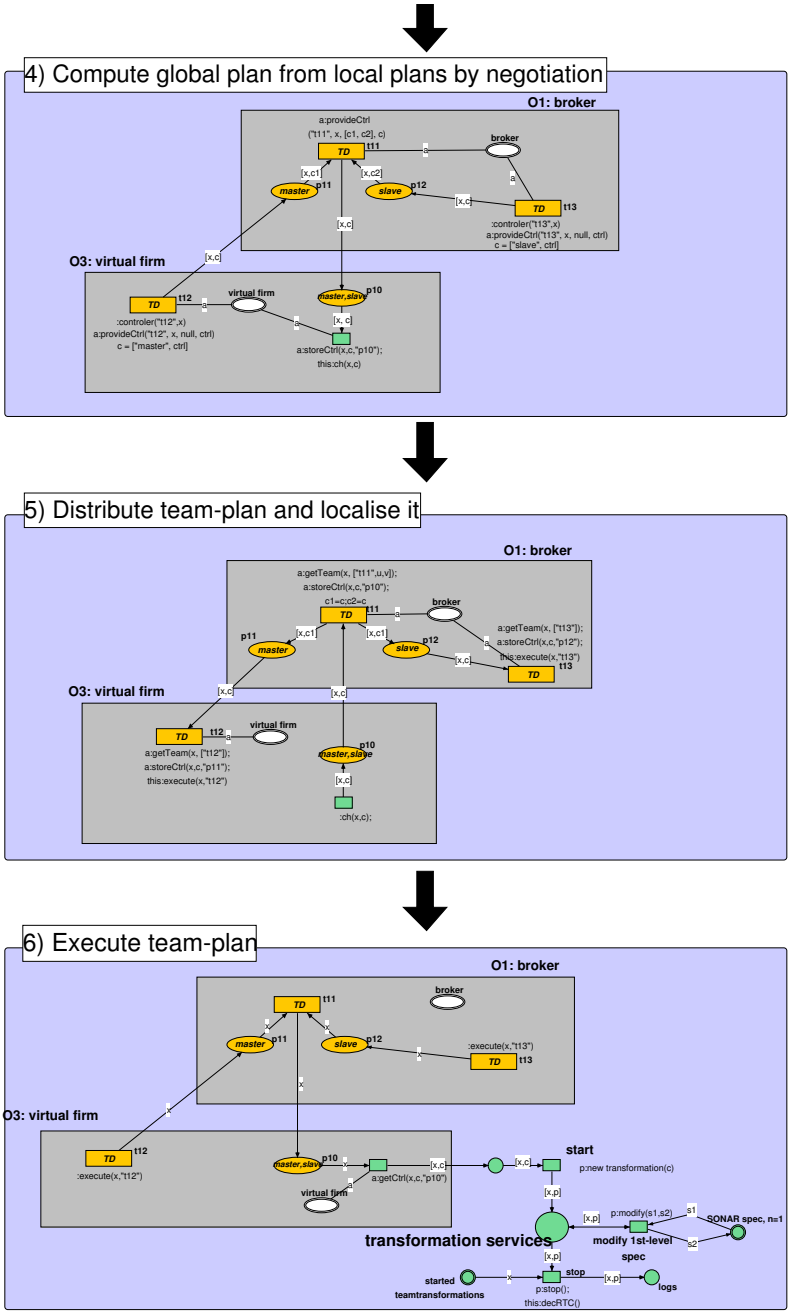


Fig. 14. SONAR-middleware for the organisation Org_2 of Fig. 4 (continued)

The six parts are not independent – they are connected via synchronisation inscriptions, so the end of an phase is synchronised with the start of the succeeding phase.

The six parts share the same net structure but have different inscriptions. This reflects the fact that all teamwork is generated from the organisational structure, but in different phases different information is needed. Also if one carefully investigates the prototype one will recognise that all the arcs for parts of phases 2, 4, and 6 are reverted since in the phases 1, 3, and 5 the information flows from the root to the leaves, but in phases 2, 4, and 6 the flow is in the opposite direction.

We now discuss the six phases in more detail. In the initial phase the RENEW simulator generates an instance of the meta-level model, i.e. the net in Fig. 13 together with six agents – one for each position – and registers them. The position agents are modelled again as nets (not shown here). After this the meta-level instance for Org_2 generates an instance for Org_2 and registers all the agents there as well. After this step the initialisation is finished and the teamwork may start.

For the given SONAR-model at level $n = 1$ we have only one position, that is able to start a team: O_4 since it is the only position having a place with an empty preset (i.e. the place p_0). Whenever the position agent O_4 decides to begin teamwork it starts the first phase: exploration. The only possibility for the task p_0 is to delegate it further to O_1 which delegates further via t_1 and generates the two subtasks p_1 and p_2. The agent O_1 chooses the agents these subtasks are delegated to. For p_1 he has a choice between O_2 and O_3, but for p_2 there is only one partner: O_4. In the prototype the agents are queried for partner choices via a synchronisation $\mathsf{a{:}askPartner}(p, O)$ which means that the agent a provides a binding for the partner O whenever the task p has to be delegated. Assume that the agent O_1 decides in favour of O_3, then the control is handed over to O_3 which has a choice how to implement the task: either by t_2 or by t_5. These decision is transferred into the model via the channel $\mathsf{a{:}askImpl}(t)$ which is activated by the agent a only if t has to be used for delegation/implementation.

After this iterated delegation has come to an end – which is guaranteed for well-formed SONAR-models – all subtask have been assigned to team agents and the first phase ends. At this point the agents know whether they are team members or not, but they do not know each other by now. To establish such mutual knowledge we start the second phase and traverse the team tree in reverse order, from the leaves to the root. At the root we have generated an object that represents the team tree.

In the third phase this team tree object is sent from the root to the leaves. After this all members know each other and they know the team structure, too. The formation of the team is complete.

After team formation we start the fourth phase: the team negotiation. The executing agents, i.e. the leaves in the tree, generate a partial local plan for the team DWF net $D(G)$. The plans are local in the sense that they describes only actions relevant for the particular role each agents plays in the team. They

are partial since they integrate different possibilities, like the best choice, the second best etc. A plan is considered as a *controller* since it is useful only when combined with the team DWF net it has been created for. Each agent a provides such a plan/controller via the channel a:provideCtrl() (we omit technical details concerning exchanged parameters). These plans are objects so they can be sent to the direct ancestors in the teamtree which compute a compromise from these local plans. The result is again a local plan, but now for this subtree. If this process reaches the root we have computed a local plan for the whole team, i.e. a global plan.

The global plan is yet unknown within the team. So we start the fifth phase and publish the global plan from the root to the leaves. Each position agent computes the part from it that is relevant for his respective role, i.e. its local plan, which is no longer partial anymore.

After this negotiation phases, team DWF execution starts in the sixth phase. An instance of the team DWF net is generated on the place team services and this instance is coupled with all the agents' plans/controllers. The team DWF net instance starts and the actions selected by the local plans are carried out.

The meta-level teamwork is more or less the same. The main difference is that the actions executed in the team DWF net instance manipulate not only data but the organisation itself, here: the organisation of level $n = 1$. The model Org_2 defines a transformation that deletes the transition t_5 in the model Org_1.

To perform such a transformation we assume that there are no ongoing team activities in the instance of level $n = 1$. So a meta-level team activity disables the start of new low-level team activities, waits until the end of all currently ongoing low-level team activities, and then deregisters all the position agents from level $n = 1$. This is all done by the block *activity control* in Figure 13. It then starts its own (level 2) six phases: a transformation team is formed, a transformation plan is negotiated, a transformation DWF net is instantiated and executed. The transformation DWF net is shown in Fig. 10. The main difference when executing the transformation DWF net is that it modifies the model Org_1. In the prototype we have the channel modify(s_1, s_2), which couples the DWF with the meta-model Org_2. This channel has the effect that is replaces the old specification s_1 with the result of the transformation $s_2 = \lambda(G)(s_1)$.

6 Conclusion

In this paper, we have built upon our previous work SONAR on formalising organisational models for MAS by means of Petri nets [4, 3]. Here, we have supplemented our original formal model with a recursive approach for reorganisation. We have introduced stratified SONAR models with multiple organisational levels where an organisation of a given level is allowed to transform organisational models of lower levels.

In addition, we have presented a prototypical middleware for our formal model. As SONAR specifications are formalised with Petri nets, they inherently

have an operational semantics and thus already lend themselves towards immediate implementation. We have taken advantage of this possibility and have chosen the reference net formalism as an implementation means. Reference nets implement the nets-in-nets concept [16] and thus allow us to build stratified Sonar organisations as nested Petri net systems. In addition, the reference net tool Renew [6] offer a comprehensive integration of the JAVA programming language, allowing us to refine/extend the Sonar specifications into fully executable prototypes.

This leaves us with a quite close link between a Sonar specification of an organisation and its accompanying middleware implementation. The structure and behaviour of the resulting software system is directly derived and compiled from the underlying formal model. For example, we have explicitly shown how the organisation net of a formal Sonar specification can be utilised for the middleware support of six different phases of teamwork. In each phase, the original net is used differently (with different inscriptions and arrow directions) but nonetheless, it ís always the same underlying organisation net. Basically, this MDA approach of deploying Sonar models does not only relieve the developer of much otherwise tedious programming. It also allows to preserve desirable properties that could be proven for the formal model and that now carry over to the software technical implementation.

Concerning future work, there exist several driving forces. First of all, we need to elaborate and extend our prototype with convenient tools for modelling, monitoring and controlling Sonar organisations. In addition, the mechanism for generating a reference nets implementation from a formal Sonar models is not generic enough at the moment.

Secondly, our purely Petri net-based approach will carry us only so far. Contrary to our philosophy of a (also physically) distributed OPA network from Section 4, our current Renew middleware prototype is of course centralised. Nevertheless, we expect to be able to reuse a great deal of our prototypical basis when transferring it to a distributed MAS framework.

Finally, although we have introduced stratified Sonar models in this paper, this stratification deals with levels of reorganisation authority. We have not addressed the topic of organisations that exhibit multiple system levels where organisational units of one system level embed organisational units of lower system level. Such a form of layering is orthogonal to the stratification introduced in this paper. We study this this subject in [17], but on an even more abstract/generic level than Sonar offers. Nevertheless, we have already begun to transfer the results to Sonar.

References

1. Carley, K.M., Gasser, L.: Computational organisation theory. In Weiß, G., ed.: Multiagent Systems. MIT Press (1999) 229–330
2. Dignum, V., ed.: Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models. IGI Global, Information Science Reference (2009)

3. Köhler-Bußmeier, M., Wester-Ebbinghaus, M., Moldt, D.: A formal model for organisational structures behind process-aware information systems. Transactions on Petri Nets and Other Models of Concurrency. Special Issue on Concurrency in Process-Aware Information Systems **5460** (2009) 98–114
4. Köhler, M.: A formal model of multi-agent organisations. Fundamenta Informaticae **79**(3-4) (2007) 415 – 430
5. Girault, C., Valk, R., eds.: Petri Nets for System Engineering – A Guide to Modeling, Verification, and Applications. Springer (2003)
6. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: Renew. In Cortadella, J., Reisig, W., eds.: Petri Nets 2004. Volume 3099 of LNCS, Springer (2004) 484 – 493
7. Aalst, W.v.d.: Verification of workflow nets. In Azeme, P., Balbo, G., eds.: Petri Nets 1997. Volume 1248 of LNCS, Springer (1997)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer (2006)
9. Hübner, J.F., Sichman, J.S., Boissier, O.: S-moise: A middleware for developing organised multi-agent systems. In: Organizations in Multi-Agent Systems: From Organizations to Organization-Oriented Programming (OOOP 2005). (2005) 107–120
10. Esteva, M., Rodriguez-Aguilar, J., Rosell, B., Arcos, J.: Ameli: An agent-based middleware for electronic institutions. In Sierra, C., Sonenberg, L., Tambe, M., eds.: Autonomous Agents and Multi-Agent Systems (AAMAS 2004). (2004) 236–243
11. Wester-Ebbinghaus, M., Köhler-Bußmeier, M., Moldt, D.: From multi-agent to multi-organization systems: Utilizing middleware approaches. In Artikis, A., Picard, G., Vercouter, L., eds.: Engineering Societies in the Agents World (ESAW 08). (2008)
12. Fischer, K., Schillo, M., Siekmann, J.: Holonic multiagent systems: A foundation for the organization of multiagent systems. Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS). Volume 2744 of LNCS, Springer (2003) 71–80
13. Wester-Ebbinghaus, M., Moldt, D.: Modelling an open and controlled system unit as a modular component of systems of systems. In Köhler-Bußmeier, M., Moldt, D., Boissier, O., eds.: International Workshop on Organizational Modelling (OrgMod'09), University of Paris (2009) 81–100
14. Boissier, O., Hübner, J., Sichman, J.S.: Organization oriented programming: From closed to open organizations. In O'Hare, G., Ricci, A., O'Grady, M., Dikenelli, O., eds.: Engineering Societies in the Agents World VII. Volume 4457 of LNCS, Springer (2007) 86–105
15. Köhler-Bußmeier, M., Wester-Ebbinghaus, M.: SONAR: A multi-agent infrastructure for active application architectures and inter-organisational information systems. In Lamersdorf, W. et al., eds.: Conference on Multi-Agent System Technologies, MATES 2009. LNAI (2009)
16. Valk, R.: Object Petri nets: Using the nets-within-nets paradigm. In Desel, J., Reisig, W., Rozenberg, G., eds.: Advanced Course on Petri Nets 2003. Volume 3098 of LNCS, Springer (2003) 819–848
17. Wester-Ebbinghaus, M., Moldt, D.: Structure in threes: Modelling organization-oriented software architectures built upon multi-agent systems. In: 7th International Conference an Autonomous Agents and Multi-Agent Systems (AAMAS'2008). (2008) 1307–1311

Generalized Hypernets and their Semantics

Marco Mascheroni

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di MilanoBicocca
Viale Sarca, 336, I-20126 Milano (Italy)**

Abstract. Hypernets were introduced for modelling systems of mobile agents using the nets-within-nets paradigm. In this model agents are structured in a containment hierarchy which can dynamically change. In this paper, hypernets are generalized by relaxing some constraints. The main results of the paper are the proof that the containment relation induces a tree structure on the hierarchy of agents in any reachable configuration, and the translation of a generalized hypernet into a 1-safe Petri net with an equivalent behavior.

1 Introduction

Systems of mobile agents have been widely studied by people who work in computer science and software engineering areas in recent years. They are composed of dynamic agents, which interact with each other and the environment, and modify their behavior with respect to interactions happened, or take autonomous decisions. Often, these agents are pieces of software travelling across a network of hosts, where they can be executed in a local environment. Modeling such systems is an important activity, and several models that deal with aspects of systems of mobile agents have been defined in the past. The interest of this paper is on Petri net based models that use the *nets-within-nets* paradigm.

With the nets-within-nets paradigm, tokens of a Petri net have the structure of a Petri net themselves. This idea was introduced by Valk in 1987 [13], who defined the model of Elementary Object Systems (EOS) in [14]. Other works about EOS, related models (like Object Nets), and their properties have been done by the same research group and can be found here for example: [15][16][9]. The nets-within-nets approach has been implemented in the tool RENEW ([11]).

Nested Petri nets was one of the first approaches that used the nets-within-nets paradigm to model systems of mobile agents [12], whereas other approaches were used in [8] and in [7].

Hypernets were introduced in [2] as a new nets-within-nets based framework for modelling systems of mobile agents. Like in nested Petri nets each agent is situated in one location in every marking, but agents cannot be created nor destroyed. Moreover, two peculiar characteristics of hypernets are a non-static hierarchical structure (agents can exchange tokens with their sub- or super-agents, and thereby change the hierarchy arisen from the containment relation

** Partially supported by MIUR and CNR - IPI PAN.

between agents) and the use of modularity (each agent is composed of modules of a certain sort which are state machines and can communicate with other modules of the same sort in other agents). Agents have a sort themselves, determining in which module of other agents they can be located. The use of a dynamical hierarchical structure has also been studied for Object Nets model [10].

Several works about Petri hypernets have been developed in the literature. In [3] it was shown the existence of a morphism from hypernets to 1-safe Petri nets. This translation allows us to reinterpret all the properties of the model one can derive on the 1-safe net by means of the techniques developed in the literature on hypernets. Moreover, a class of transition systems, called Agent Aware Transition Systems, was defined in order to describe the behavior of hypernets [4][1].

Hypernets have some constraints that limit their expressive power for modelling purposes. For example, it is not possible to associate a weight to an arc of a hypernet, and the modularity subdivision is sometimes too strict. Therefore, a generalization of the basic model that relaxes these constraints was defined and called generalized hypernets. In this paper, the 1-safe nets semantics is preserved, as long as the tree-like structure of the hierarchy arising from the containment relation between agents. Module subdivision is maintained too, replacing the state machine module structure constraint with the notion of paths, which are themselves of a certain sort. Generalized hypernets were employed, for example, to model a class of P-Systems [4] (a computational model based upon the architecture of a biological cell).

Section 2 discusses some basic concepts about generalized Hypernets which are formalized in section 3. In Section 4 it is shown that markings of a hypernet have always a tree structure. Finally, in Section 5 a theorem showing a correspondence between a hypernet and a 1-safe Petri net is shown.

In the remainder the term *Basic hypernets* is used when referring to the original version of the formalism, while the term hypernets is used when referring to generalized hypernets.

2 Generalized Hypernet: Concepts

The focus of this paper is modeling systems of mobile agents using the nets-within-nets paradigm. The term agent is used in this paper to denote a component of a concurrent system, and it is not related to the concept of agent in other disciplines such as Artificial Intelligence. Each agent is represented by an *open* net, that is a Petri net enriched with particular places for managing the communication of tokens between agents. The characteristic that each agent is located in another agent is reflected in the model by the fact that each net but one is a token of another open net. The only exception is an agent A which contains, directly or indirectly, all other agents. These agents are indirectly contained in A if they are located in another agent distinct from A which in turn is direcly or indirectly located in A

2.1 Paths

The first obvious difference between hypernets and Petri nets is that tokens can be nets and have an identity. Indeed, there is a distinction between *structured* tokens and *simple* tokens: simple tokens are very similar to black tokens in Petri nets, while structured tokens have an internal state represented by a Petri net. Structured tokens can either change their state by means of internal autonomous transitions, or by means of *interactions* with other structured tokens. These interactions can be enabled or disabled depending on the internal state of the structured tokens and are performed using a token exchange mechanism between structured tokens.

In Petri nets there is no distinction between tokens because a state is represented by a function which assigns the number of token to each place. Therefore, each token is identical to others, but in hypernets this is not necessarily true because of the structure and the internal state of tokens. Thus, the way tokens are manipulated in hypernets must be different. In the basic model it is not necessary to distinguish between each single token. However, since hypernet tokens are structured and have their own internal state, firing a simple transition that takes a token from a place p putting it in another place could produce different results if there are several tokens in p: moving a token instead of another one could change the future behavior of the entire hypernet because certain future interactions could be enabled or not, depending on the internal state of the moved token. A mechanism to select which tokens will be moved when a transition is fired is needed.

Another issue that must be taken into account is in which place a token must be placed after the execution of a transition that has more than one output place. Indeed, the way tokens are moved from input places to output places is also important and could produce different results. Looking at picture 1 it is possible to notice that again the usual Petri nets firing rule is not sufficient because it does not contain informations about token identity. The two tokens in the place $p1$ could be both moved to place $p3$ (figure 1(b)), or could be separated and moved one to $p3$ and one to $p4$ (figure 1(c)).

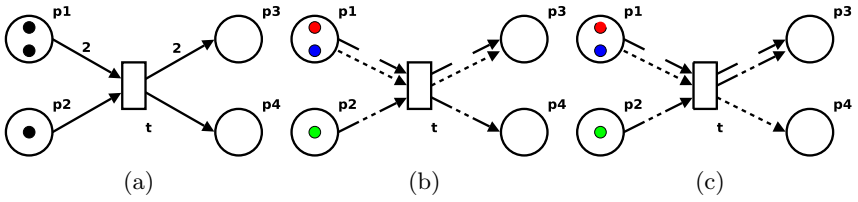


Fig. 1.

To take into account these two problems *paths* are introduced. A path is a triple place-transition-place that is used to uniquely identify which tokens will

perform a transition (together with the γ function in the Definition 6) and where each token will be placed after a transition is fired. In each of the two figures 1(b) and 1(c) there are three paths: two paths that insist on (p_1, t, p_3) and one that insists on (p_2, t, p_4) in figure 1(b), and paths $(p_1, t, p_3), (p_1, t, p_4), (p_2, t, p_3)$ are present in figure 1(c). p_1 and p_2 are called input places of the path, whereas p_3 and p_4 are called output places of the path. As it will be clearer later when the graphical notation is introduced in example 4, the same effect of paths can be obtained in high level nets using annotation on arcs: it is enough to connect the input place of the path to the transition with an arc annotated g, and to connect the transition to the output place of the path with an arc annotated with the same label g.

Basic hypernets solve these problems by means of synchronized state machines. Each agent is made of a set of state machines that cooperate in performing a transition. Since each transition in a state machine has exactly one input place and one output place, then it is possible to identify which structured token is moved and the place where it will be placed after the execution of the transition. However, such state machines decomposition is too strict for some application contexts, for instance, because of the needs of weighted arcs. For example, in [4] transformation rules that take any quantity of molecules of several types and put them out of a membrane were modeled. These kinds of rules are easilly modeled if the formalism allows the use of weighted arcs, whereas with synchronized state machines the model is more constrained.

2.2 Virtual Places

Each agent of a hypernet is located in another agent, with the exception of one special agent that is considered as an environment. Thus, there is a containment relation between nets that can be represented as a graph. This graph as it will be demonstrated in theorem 1, is always a tree. Virtual places are introduced to manage the passage of tokens between *close* structured tokens. Two tokens are close when one is located in a place of the other or vice-versa. A virtual place acts as a communication link between structured tokens and is graphically represented with a triangle inscribed in a circle. Consider an example in which a system with a structured token A that sends a simple token to a structured token B is modeled. The simple token can be sent up if token A is located in one place of token B 2(a), or down if token B is located in a place of token A 2(b). Thus, two kinds of virtual places are used: the one with the triangle's base on the bottom is used for down to up passage of token, while the one with the triangle's base on the top is used for up to down passage.

3 Generalized Hypernets: Formal Definition

3.1 Preliminaries

In the following it will be used the notation: $f\langle q_0, q_1, \ \dots\ , q_n\rangle = \langle r_0, r_1, \ \dots\ , r_n\rangle$ when f is a function $f : Q \to R$ and $\{q_0, q_1, \ \dots\ , q_n\} \subseteq Q$ and $\{r_0, r_1, \ \dots\ , r_n\} \subseteq R$ and $f(q_0) = r_0, f(q_1) = r_1, \ \dots\ , f(q_n) = r_n$.

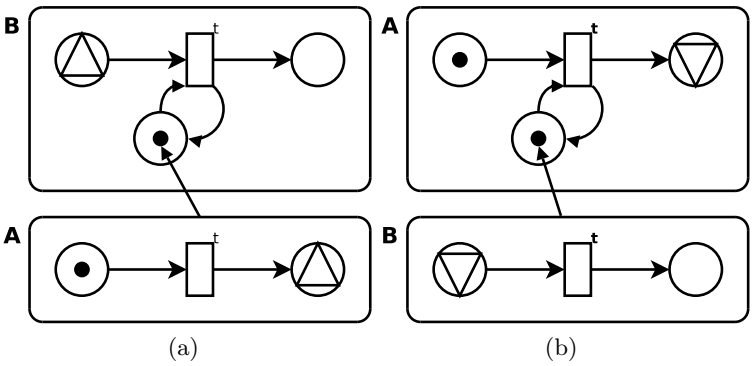


Fig. 2.

A *Petri net* N is a tuple $N = (P, T, F)$ where P is the set containing the *places* of the net, T is the set containing *transitions* with $P \cap T = \emptyset$, and $F : (P \times Q) \cup (Q \times P)$ is the *flow relation*. A *marked Petri net* is a tuple $N = (P, T, F, M_0)$ where $N = (P, T, F)$ is a Petri net and $m_0 : P \to \mathbb{N}$ is the initial marking assigning to each place the number of tokens. The *input places* of a transition t, denoted ${}^\bullet t$, are $p \in P : \exists t, (p, t) \in F$. In a similar way *output places* $t^\bullet$ are defined. A transition t is *enabled* in a marking m, denoted $m[t\rangle$, iff $\forall p \in {}^\bullet t, m(p) > 0$. *Firing* the enabled transition t leads to a new marking $m' : \forall p \in {}^\bullet t \backslash t^\bullet, m'(p) = m(p) - 1 \wedge \forall p \in t^\bullet \backslash {}^\bullet t, m'(p) = m(p) + 1$. A marking m is *reachable* from m_0 if there exists a sequence of transitions that lead to m. A marked Petri net is *1-safe* iff the number of tokens on each place is 0 or 1 in every reachable marking.

3.2 Static Structure

Definition 1. *An* agent *is a tuple* $A_i = (P_i, T_i, G_i, \phi_i)$*, where:*

- $P_i = L_i \cup V_i$ *is the set of* local places L_i *and* virtual places V_i *(or communication places), with* $L_i \cap V_i = \emptyset$*; an agent can send tokens in two* directions*: up or down, so it is possible to distinguish the two kinds of virtual places* V_i^{Up} *and* V_i^{Down} *such that* $V_i = V_i^{Up} \cup V_i^{Down}$ *with* $V_i^{Up} \cap V_i^{Down} = \emptyset$
- T_i *is the set of* transitions *with* $T_i \cap P_i = \emptyset$*;*
- G_i *is the set of* paths*;*
- $\phi_i \colon G_i \to P_i \times T_i \times P_i$ *is the* path map *that associates a triple place-transition-place to each path in such a way that at least one of the two places in the triple is local:* $\phi_i(g) = (p, t, q) \Rightarrow \neg(p \in V_i \wedge q \in V_i)$*.*

Example 1. In figure 3 a simple agent with two transitions, three local places and two virtual places is depicted. Unfortunately, if the graphical representation of the agent is done in a classical Petri nets style, there is no way to know which paths constitute the agent.

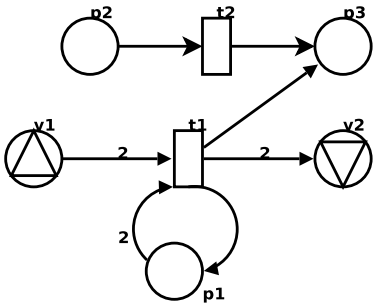


Fig. 3. Agent A_1

Thus, each input arc and each output arc of a transition is labelled with a set of label in such a way that the labels of input arcs are pairwise disjoint and the union of the labels of the input arcs is equal to the union of the labels of the output arcs. If a transition has only one input arc and one output arc then labels are not depicted. In figure 4 all possible path combinations of figure 3 are represented with this graphical notation. There is only one net without a path where both input and ouput places that are vitual. This is the only net that it is an agent according to Definition 1.

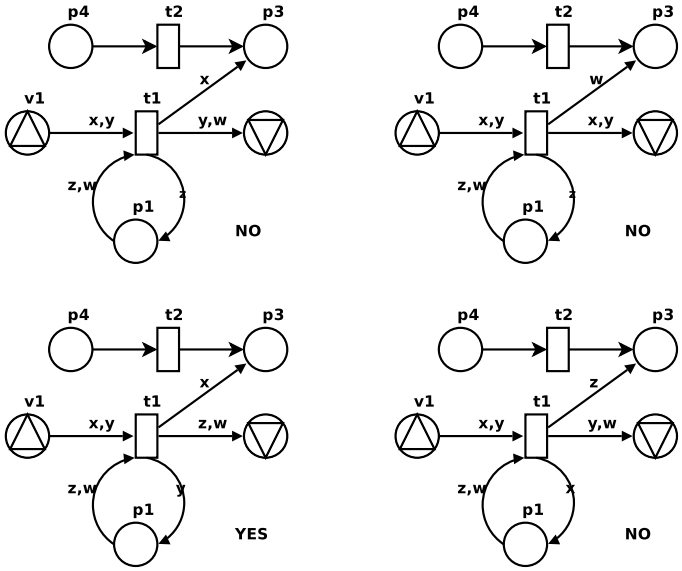


Fig. 4. Notation used to represent possible paths of net in figure 3

Given a path $g \in G_i$, by ${}^\bullet g$ the *input place* of path g is denoted, ie: $p \in P_i : \phi_i(g) = (p, t, q)$ with t and q free. In the same way the *output place* of the path is defined: $g^\bullet = q \in P_i : \phi_i(g) = (p, t, q)$ with t and p free. Notice that each path has only one input place and one output place. Therefore, this notation is slighty different from the one used from classical Petri nets because it returns a single place instead of a set of places.

Given a set of agents $X = \{A_1, A_2, ..., A_n\}$, the following notation is used:

$$P_X = \bigcup_{A_i \in X} P_i, \quad L_X = \bigcup_{A_i \in X} L_i, \quad V_X = \bigcup_{A_i \in X} V_i, \quad T_X = \bigcup_{A_i \in X} T_i,$$

In a similar way V_X^{Up} and V_X^{Down} are defined.

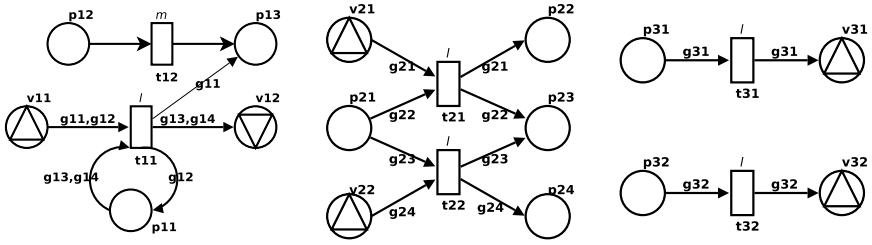
Definition 2. *Let $\mathcal{N} = \{A_1, A_2, \ldots, A_n\}$ be a family of agents, Σ a finite set of sorts, and Λ a finite set of transition labels.*

A hypernet *is a tuple $H = (\mathcal{N}, \sigma_\mathcal{N}, \sigma_G, \lambda)$, where*

- *The agents in $\mathcal{N}$ have disjoint sets of places, paths, and transitions, i.e.: $\forall A_i, A_j \in \mathcal{N}, i \neq j \Longrightarrow P_i \cap P_j = \emptyset \wedge G_i \cap G_j = \emptyset \wedge T_i \cap T_j = \emptyset$; agents that represent usntructured tokens are also allowed empty;*
- *$\sigma_\mathcal{N} : \mathcal{N} \to 2^\Sigma$ is a function that describes the sorts of each agent;*
- *$\sigma_G : G_\mathcal{N} \to \Sigma$ is a function that assigns a sort to each path;*
- *λ: $T_\mathcal{N} \to \Lambda$ is a function that assigns to each transition a label.*

Example 2. Let A_1 be the agent shown in figure 5(a), A_2 the agent shown in figure 5(b), A_3 the agent shown in figure 5(c), and A_4 and A_5 two empty agents. Let $\mathcal{N} = \{A_1, A_2, A_3, A_4, A_5\}$ be a set of agents, $G_\mathcal{N}$ be the set containing all the paths of the agents in $\mathcal{N}$, $\Sigma = \{\alpha\}$ be the set containing the only sort α, and $\Lambda = \{l, m\}$ be a set of labels. Moreover, let $\sigma_\mathcal{N} : \mathcal{N} \to 2^\Sigma$ be a function that assign to each element of the domain the element $\{\alpha\}$, $\sigma_G : G_\mathcal{N} \to \Sigma$ be a similar function, and $\lambda\langle t_{11}, t_{12}, t_{21}, t_{22}, t_{31}, t_{32}\rangle = \langle l, m, l, l, l, l\rangle$ be a function that assigns to each transition the label l with the exception of transition t_{12}.

The tuple $H = (\mathcal{N}, \sigma_\mathcal{N}, \sigma_G, \lambda)$ is a hypernet.



(a) Agent A_1 (b) Agent A_2 (c) Agent A_3

Fig. 5. A set of agents that toegether are an hypernet

Definition 3. *Let $\mathcal{N} = \{A_1, A_2, \ldots, A_n\}$ be a family of agents. A map $\mathcal{M} : \{A_2, \ldots, A_n\} \longrightarrow L_{\mathcal{N}}$, assigning to each agent different from A_1 the local place where it is located, is a* hypermarking *of $\mathcal{N}$ iff, considering the relation $\uparrow_{\mathcal{M}} \subseteq \mathcal{N} \times \mathcal{N}$ defined by : $A_i \uparrow_{\mathcal{M}} A_j \Leftrightarrow \mathcal{M}(A_i) \in L_j$, then the* marking graph *(i.e.: $\langle \mathcal{N}, \uparrow_{\mathcal{M}} \rangle$) is a tree with root A_1.*

Definition 4. *A* marked hypernet *is a pair $(H, \mathcal{M})$ where H is a hypernet and $\mathcal{M}$ is a hypermarking defining the initial configuration.*

Example 3. Consider the hypernet of Example 2. Three of the possible maps from the agents to the local places of that hypernet are the following:

$$\mathcal{M}_1\langle A_2, A_3, A_4, A_5\rangle = \langle p_{11}, p_{21}, p_{31}, p_{32}\rangle$$
$$\mathcal{M}_2\langle A_2, A_3, A_4, A_5\rangle = \langle p_{11}, p_{23}, p_{31}, p_{24}\rangle$$
$$\mathcal{M}_3\langle A_2, A_3, A_4, A_5\rangle = \langle p_{11}, p_{31}, p_{31}, p_{32}\rangle$$

The three graphs in figure 6 correspond to the marking graphs $\langle \mathcal{N}, \uparrow_{\mathcal{M}_1}\rangle$, $\langle \mathcal{N}, \uparrow_{\mathcal{M}_2}\rangle$, $\langle \mathcal{N}, \uparrow_{\mathcal{M}_3}\rangle$ respectively.

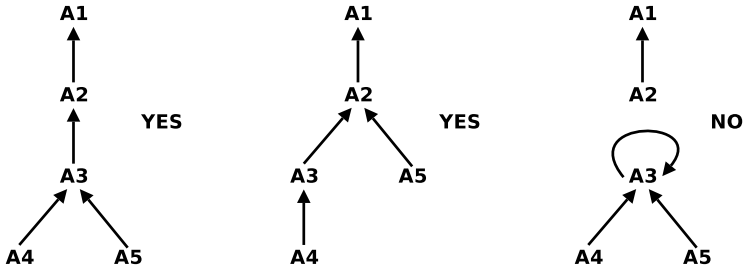


Fig. 6. Possible marking graphs of the hypernet of Example 2

Since the latter marking graph is not a tree with root A_1 only the two pairs $(H, \mathcal{M}_1)$, $(H, \mathcal{M}_2)$ are marked hypernets.

3.3 Behavior

In order to discuss the dynamics of a hypernet, it is convenient to identify some sets of paths which will be used in the formal definitions.

$$
\begin{aligned}
G^{Local} &= \{g \in G_{\mathcal{N}} : \phi_{\mathcal{N}}(g) = (p,t,q) \wedge p \in L_{\mathcal{N}} \wedge q \in L_{\mathcal{N}}\} \\
G^{Out} &= \{g \in G_{\mathcal{N}} : \phi_{\mathcal{N}}(g) = (p,t,v) \wedge v \in V_{\mathcal{N}} \wedge p \in L_{\mathcal{N}}\} \\
G^{In} &= \{g \in G_{\mathcal{N}} : \phi_{\mathcal{N}}(g) = (v,t,p) \wedge v \in V_{\mathcal{N}} \wedge p \in L_{\mathcal{N}}\} \\
G^{Up} &= \{g \in G_{\mathcal{N}} : \phi_{\mathcal{N}}(g) = (p,t,q) \wedge (p \in V_{\mathcal{N}}^{Up} \vee q \in V_{\mathcal{N}}^{Up})\} \\
G^{Down} &= \{g \in G_{\mathcal{N}} : \phi_{\mathcal{N}}(g) = (p,t,q) \wedge (p \in V_{\mathcal{N}}^{Down} \vee q \in V_{\mathcal{N}}^{Down})\} \\
G^{t} &= \{g \in G_{\mathcal{N}} : \phi_{\mathcal{N}}(g) = (p,t,q)\}
\end{aligned}
$$

Notice that in the first five definitions t is free, and in the last one t is fixed.

Let $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$, with $\mathcal{N} = \{A_1, A_2, \ldots, A_n\}$, be a hypernet.

A consortium is a set of interconnected *active* agents, cooperating in performing a set of transitions with the same label l, and moving other *passive* agents along the paths containing those transitions.

A notion of consistency between paths is introduced with the aim of identifying pairs of paths belonging to different agents, that could be associated to exchange tokens. Two paths are consistent if they have the same sort and the same direction. As it can be seen in the formal definition this notion is not a simmetric relation because the first path belongs to the agent which sends the token and the second path belongs to the agents which receive the token.

Definition 5. *Two paths* $g_i \in G_i$ *and* $g_j \in G_j$ *are* consistent *(denoted by* $cons(g_i, g_j)$*)* $\iff$ $A_i \neq A_j \wedge \sigma_G(g_i) = \sigma_G(g_j) \wedge ((g_i \in G^{Up} \cap G^{Out} \iff g_j \in G^{Up} \cap G^{In}) \vee (g_i \in G^{Down} \cap G^{Out} \iff g_j \in G^{Down} \cap G^{In}))$

Notice that the nature of the virtual places (up or down) of the two paths involved in the token exchanging reflects the direction taken by the token.

Example 4. In figure 7 all the possible combinations of paths and the corresponding values of the *cons* predicate are shown. It is supposed that all paths have the same sort. If not, they are not consistent.

Definition 6. *A* consortium *is a tuple* $\Gamma = (l, \tau, C, \delta, \gamma)$ *where:*

1. $l \in \Lambda$ *is the* name *of the consortium,*
2. $\tau = \{t_0, ..., t_m\}$ *is the set of transitions that will be fired. They must belong to different agents and must have the same label, i.e.:* $\forall t_i, t_j \in \tau, i \neq j \Rightarrow (t_i \in T_z \Rightarrow t_j \notin T_z) \wedge \lambda(t_i) = \lambda(t_j) = l$. *With* $G_\tau = \bigcup_{t_k \in \tau} G^{t_k}$ *the set of paths involved in the consortium (i.e.: paths that contain transitions that are in* τ*) are denoted*
3. $C \subseteq \mathcal{N}$ *is the set of* passive agents
4. $\delta : G^{Out} \cap G_\tau \to G^{In} \cap G_\tau$ *is a* bijective *correspondence such that:* $\delta(g) = g' \Rightarrow \text{cons}\ (g, g')$
5. $\gamma : C \longrightarrow G_\tau \backslash G^{In}$ *is a* bijective *correspondence such that:* $\gamma(A) = g \Rightarrow \sigma_G(g) \in \sigma_{\mathcal{N}}(A)$

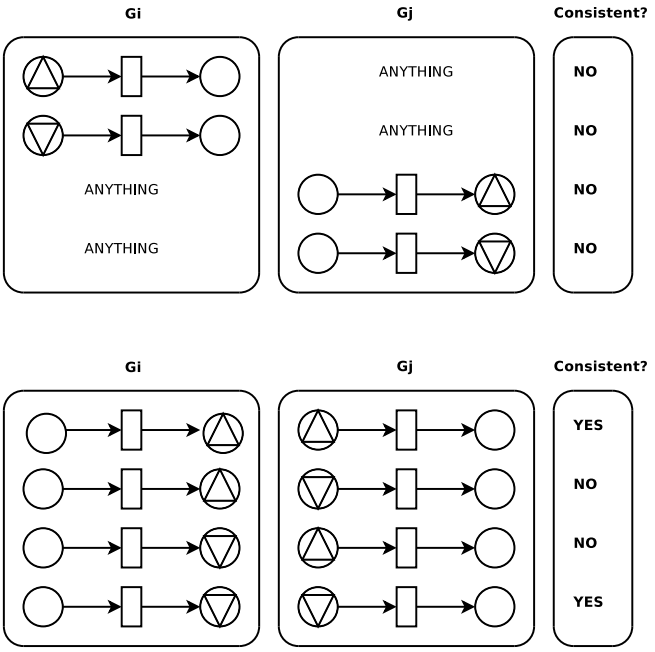


Fig. 7. Possible combinations of paths and their consistency

6. *Agents that receive tokens must not move in the hierarchy, i.e.:* $\forall A_i \in C : \exists g \in G_i \cap G^{In} \cap G_\tau \Rightarrow \gamma(A_i) \notin G^{Out}$
7. *If an agent* A_1 *sends a token to another agent* A_2 *then either the contained agent is a passive agent or the outer agent has no local paths containing the transition which is being fired (see figure 8). Let* $g_1 = (p_1, t_1, v_1) \in A_1$ *and* $g_2 = (v_2, t_2, p_2) \in A_2$ *then*

$$\begin{aligned} \delta(g_1) = g_2 \wedge g_1 \in G^{Up} \quad &\Longrightarrow \quad A_1 \in C \vee (G^{t_2} \cap G^{Local} = \emptyset) \\ \delta(g_1) = g_2 \wedge g_1 \in G^{Down} \quad &\Longrightarrow \quad A_2 \in C \vee (G^{t_1} \cap G^{Local} = \emptyset) \end{aligned}$$

8. *the set of active agents is a minimal one, in the sense that they must be interconnected through the interaction* l*, i.e.:*
   the undirected graph $\mathcal{G} = (\tau, E)$ *is connected,*
   where $E = \{(t_i, t_j) : t_i \in A_i, t_j \in A_j \wedge \exists\, g \in G^{t_i} : \delta(g) \in G^{t_j}\}$

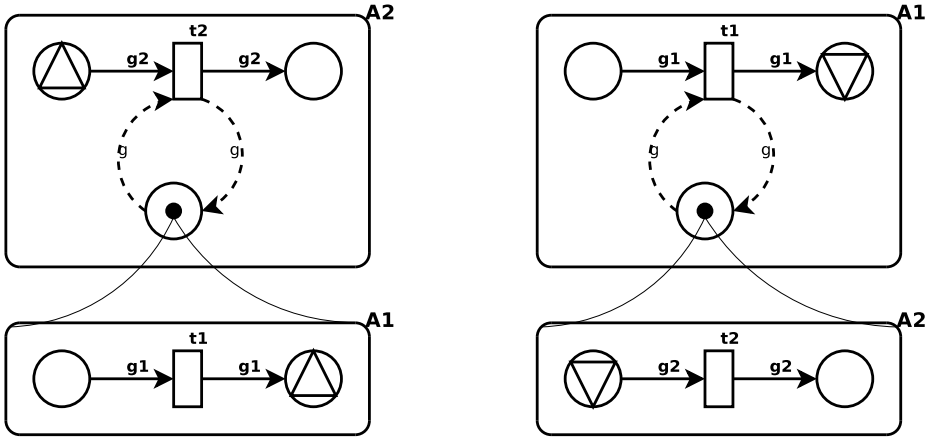


Fig. 8. If the path g exists then the inner agent has to belong to the set of passive agents

Example 5. In the hypernet described in Example 2 with the marking $\mathcal{M}_1$ there are four consortia associated to the label l, namely:

$$\Gamma_1 = (l, \{t_{21}, t_{31}\}, \{A_4\}, \delta\langle g_{21}\rangle = \langle g_{31}\rangle, \gamma\langle A_3, A_4, A_5\rangle = \langle g_{22}, g_{31}, g_{32}\rangle)$$
$$\Gamma_2 = (l, \{t_{21}, t_{32}\}, \{A_5\}, \delta\langle g_{21}\rangle = \langle g_{32}\rangle, \gamma\langle A_3, A_4, A_5\rangle = \langle g_{22}, g_{31}, g_{32}\rangle)$$
$$\Gamma_3 = (l, \{t_{22}, t_{31}\}, \{A_4\}, \delta\langle g_{24}\rangle = \langle g_{31}\rangle, \gamma\langle A_3, A_4, A_5\rangle = \langle g_{22}, g_{31}, g_{32}\rangle)$$
$$\Gamma_4 = (l, \{t_{22}, t_{32}\}, \{A_5\}, \delta\langle g_{24}\rangle = \langle g_{32}\rangle, \gamma\langle A_3, A_4, A_5\rangle = \langle g_{22}, g_{31}, g_{32}\rangle)$$

The set of all consortia of a hypernet H is denoted by $CONS(H)$.

Definition 7. *Let $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$ be a hypernet and $\mathcal{M}$ be a hypermarking.*

A consortium $\Gamma = (l, \tau, C, \delta, \gamma)$ is enabled *in $\mathcal{M}$, denoted $\mathcal{M}[\Gamma\rangle$, iff the following three conditions hold*

$$\forall A \in C, {}^\bullet\gamma(A) = p \Rightarrow \mathcal{M}(A) = p \tag{1}$$
$$\forall g \in G_i \cap G^{Up} : \delta(g) \in G_j, A_i \notin C \Rightarrow A_i \uparrow_{\mathcal{M}} A_j \tag{2}$$
$$\forall g \in G_j \cap G^{Down} : \delta(g) \in G_j, A_i \notin C \Rightarrow A_i \uparrow_{\mathcal{M}} A_j \tag{3}$$

where $\uparrow_{\mathcal{M}}$ was defined in Definition 3.

Definition 8. *If $\mathcal{M}[\Gamma\rangle$, then the* occurrence *of Γ leads to the new hypermarking $\mathcal{M}'$, denoted $\mathcal{M}[\Gamma\rangle\mathcal{M}'$, such that $\forall A \in \mathcal{N}$:*

$$\mathcal{M}'(A) = \begin{cases} \mathcal{M}(A) & \text{if } A \notin C \\ q & \text{if } \gamma(A) \in G^{Out} \text{ and } (\delta(\gamma(A)))^\bullet = q, \\ q' & \text{if } \gamma(A) \notin G^{Out} \text{ and } \gamma(A)^\bullet = q' \end{cases}$$

Definition 9. *Given a marked hypernet $(H, \mathcal{M})$, a hypermarking $\mathcal{M}_n$ is reachable iff $\exists \Gamma_1, \Gamma_2, \ \ldots\ , \Gamma_n \in CONS(H) : \mathcal{M}_0[\Gamma_1\rangle\mathcal{M}_1[\Gamma_2\rangle\mathcal{M}_2, \ \ldots\ , \mathcal{M}_{n-1}\ [\Gamma_n\rangle\ \mathcal{M}_n$*

4 Preservation of the Tree-Like Structure of the Hypermarking

When a consortium is fired it modifies the marking graph, the graph induced by the containment relation between agents (Definition 3). The initial containment relation between agents forms a tree (i.e., $\langle \mathcal{N}, \uparrow_{\mathcal{M}} \rangle$ is a tree); is the containment relation between agents a tree in all possible subsequent configurations of the hypernet? The aim of this section is to prove that firing a consortium preserves the tree structure of a hypermarking

This proof is not trivial because a consortium could correspond in several contemporary movement of agents. Thus, some preliminary notions on tree must be introduced. The first one is a notion of tree which has been adapted for the purposes of this paper from standard definitions in the graph theory (which can be found in [6] for instance).

Definition 10. *A* tree *is a pair* $A = (V, p)$ *where* V *is the set of verteces of the tree and* $p : V \backslash v_0 \to V$ *is the* father *function that associates the father to each vertex with the exception of the root of the tree which is* v_0*.* p *must satisfies the following property:*

$$\forall v \in V \backslash v_0 \ \exists k \text{ such that } p^k(v) = v_0 \text{ with } v_0 \in V \tag{4}$$

Notice that a tree $A = (V, p)$ can be transformed in a graph $G = (V, E)$ with the same set of vertices V and a set of arcs $E = \{(v, v') : p(v) = v'\}$. It is easy to prove that any two vertices are linked by a unique path in G, which is equivalent to say that G is a tree as in theorem 1.5.1 of [6].

As in graph theory, the height of a tree is the length of the longest path from a leaf to the root.

Definition 11. *The* height *of a tree* $A = (V, p)$ *is the minimum number* h *such that* $\forall v \in V : p(v)^k = v_0 \Rightarrow k \leq h$

The following definitions introduces the concept of *move*, *distinct set of moves*, and *application of a distinct set of moves* to a tree. These concepts are used to represent the changes made by a consortium to the graph induced by the containment relation between agents. In particular, these changes can be represented by the application of a distinct set of moves which are *one-level* and *non-cascading*, as it is shown later in this chapter.

Definition 12. *Let* $A = (V, p)$ *be a tree. An* A-move m *is a pair* $(v\ ,\ v')$ *such that* $v \in V \backslash v_0$ *and* $v' \in V$*.* v *is called the* pivot *of the move.*

A move $(v\ ,\ v')$ represents the atomic unit of transformation of a tree. It simply changes the father of the pivot node v from a generic node x to v' (see figure 9), like a consortium can move an agent in such a way that its father change (i.e.: the place in which it is marked belongs to another distinct agent after executing the consortium). However, the effect of a consortium to the marking graph is not represented by a single move, but by a set of moves which have distinct pivots. Thus, sets of *distinct* moves are defined:

Fig. 9. A move with pivot v

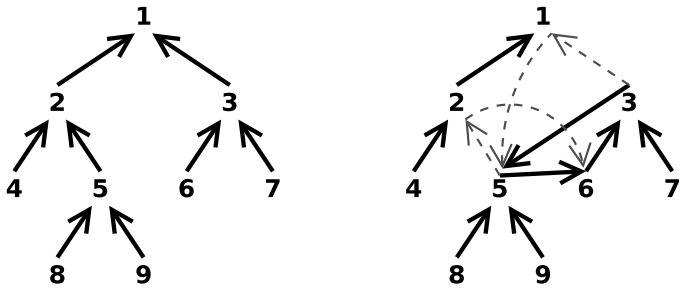
Definition 13. *A set of A-moves* $M = \{(v_0\ ,\ v'_0),\ (v_1\ ,\ v'_1),\ ...,\ (v_n\ ,\ v'_n)\}$ *is called* distinct *iff the moves have pairwise different pivots, i.e.:* $v_0 \neq v_1 \neq ... \neq v_n$. *With* $PVT(M) = \{v_0, v_1, ..., v_n\}$ *the* pivot set *of the distinct set of moves* M *is denoted.*

A set of distinct moves can be applied to a tree. They transform the tree moving some of the edges that connect a vertex to its predecessor in the path toward the root.

Definition 14. *The application of a set of distinct A-moves* $M = \{(v_0\ ,\ v'_0),\ (v_1\ ,\ v'_1),\ ...,(v_n\ ,\ v'_n)\}$ *to the tree* $A = (V,\ p)$ *is a graph* $G = (V, p')$, *i.e.:* $M(A) = G$, *such that:*

$$p'(v) = \begin{cases} v' & \text{if } (v, v') \in M, \\ p(v) & \text{if } v \notin PVT(M), \end{cases}$$

The application of a set of distinct moves to a tree may not be a tree anymore. For example, the application of the distinct set of moves $M = \{(5, 6), (3, 5)\}$ to the tree in figure 10(a) produces a graph that is not a tree anymore (see figure 10(b)).



(a) A tree on which moves $M = \{(5, 6), (3, 5)\}$ is being applied

(b) The graph is not a tree anymore

Fig. 10. Example of moves application

The next step is to recognize the type of moves a consortium results in, and to prove a theorem that says that the application of such moves preserves the tree structure of a graph. In particular, an agent can only move one level per consortium, and the receiving agent must not move in the hierarchy.

Definition 15. *A set of distinct A-moves* $M = \{(v_0\ ,\ v'_0),\quad (v_1\ ,\ v'_1),\quad ...,\ (v_n\ ,\ v'_n)\}$ *is* one-level *iff* $\forall v_i \in \{v_0, v_1, ..., v_n\}, (p^2(v_i) = v'_i) \vee (p(v_i) = p(v'_i))$

A one level move changes the father of the pivot v either to the grand-father of v or to a sibling of v. Moves depicted in figure 10 are not 1-level, whereas the moves $\{(9, 8), (6, 1)\}$ are 1-level.

Definition 16. *A set of distinct A-moves* $M = \{(v_0\ ,\ v'_0),\quad (v_1\ ,\ v'_1),\quad ...,\ (v_n\ ,\ v'_n)\}$ *is* non cascading *iff* $\{v_0, v_1, ..., v_n\} \cap \{v'_0, v'_1, ..., v'_n\} = \emptyset$

A set of moves M is not cascading if the new father of each pivot is not a pivot itself in M.

Definition 17. *Let* $A = (V, p)$ *be a tree and let* $M = \{(v_0\ ,\ v'_0),\quad (v_1\ ,\ v'_1),\quad ...,\ (v_n\ ,\ v'_n)\}$ *be a set of moves.* $M_{[k,k']} = \{(v_i, v'_i) \in M : p^j(v_i) = v_0 \wedge k \leq j \leq k'\}$ *denotes the moves whose pivot* v_i *is at depth* $j : k \leq j \leq k'$*. If* $k = k'$ *notation* M_k *is used instead of* $M_{[k,k']}$*.*

Finally, the following theorem can be proved:

Theorem 1. *Let* $A = (V, p)$ *be a tree of height* h*, and let* $M = \{(v_0\ ,\ v'_0),\ (v_1\ ,\ v'_1),\quad ...,\quad (v_n\ ,\ v'_n)\}$ *be a set of distinct, non-cascading, one-level A-moves.*

$$M(A) = M_{[0,h]}(A) \text{ is a tree} \tag{5}$$

Proof. The proof is by induction on the height of the tree starting from the bottom.
BASE CASE: $M_{[h,h]}(A)$ *is a tree.*
$M_{[h,h]}$ *are the moves whose pivots are the leaves of the tree. Since all the moves are one-level, applying these kind of moves to the tree means that, after the move, each leaf in* $PVT(M_{[h,h]})$ *will be either connected to its grandfather or to another leaf, a sibling. In the former case the tree structure is obviously preserved; in the latter case the non-cascading property of the moves guarantees that the path that connects the new father of each pivot to the root does not change after the application of the moves* $M_{[h,h]}(A)$*.*
INDUCTIVE STEP: $0 < i \leq h,\ M_{[i,h]}(A)$ *is a tree* $\Rightarrow M_{[i-1,h]}(A)$ *is a tree.*

The inductive hypothesis written in an extended form says that $M_{[i,h]}(A) = (V, p_i)$ *is a tree, thus:*

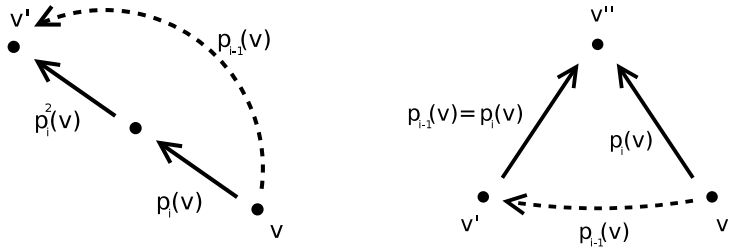
$$p_i(v) = \begin{cases} v' & \text{if } \exists (v, v') \in M_{[i,h]}, \\ p(v) & \text{if } v \notin PVT(M_{[i,h]}) \end{cases}$$

is such that condition 4 holds for p_i.
The inductive thesis says that $M_{[i-1,h]} = M_{i-1}(M_{[i,h]}(A)) = (V, p_{i-1})$ is a tree, thus:

$$p_{i-1}(v) = \begin{cases} v' & \text{if } \exists(v, v') \in M_{i-1}, \\ p_i(v) & \text{if } v \notin PVT(M_{i-1}) \end{cases}$$

is such that condition 4 holds for p_{i-1}.
Case a: $\exists(v, v') \in M_{i-1}$

Since the moves are one level, then either $v' = p_{i-1}(v) = p_i^2(v)$ (up-move: figure 11(a)) or $v' = p_{i-1}(v)$ and since the moves are non-cascading $v'' = p_i(v') = p_{i-1}(v')$ which implies $p_{i-1}^2(V) = p_i(v)$ (down-move figure 11(b)). As nodes v' for the first case and v'' for the latter are far $i-3$ and $i-2$ step from the root respectively, it is possible to say that all nodes between them and the root are not in $PVT(M_{i-1})$, thus the inductive hypothesis can be applyied to say that they are connected to the root.



(a) Up-Move (b) Down-move

Fig. 11. Type of possible moves

Case b: $v \notin PVT(M_{i-1})$
if $v \notin PVT(M_{i-1})$ there are two cases. If the vertex v does not encounter a vertex in $PVT(M_{i-1})$ in the path toward the root, then the inductive hypothesis applies directly. Otherwise the inductive hypothesis applies until the node in $PVT(M_{i-1})$, and then the previous part of the demonstration holds.

As a direct consequence of theorem 1 the following theorem can be proved:

Theorem 2. *Let $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$ be a hypernet, Γ be a consortium and $\mathcal{M}_0$ a hypermarking.*

$$\mathcal{M}_0[\Gamma\rangle\mathcal{M}_1 \Rightarrow \mathcal{M}_1 \textit{is a hypermarking}$$

Proof. A set of moves can be associated to Γ for each passive agent A which moves in the hierarchy, i.e.: $\gamma(A) \in G^{Out}$. From the sixth condition of definition 6, which means that the moves associated to a consortium are non-cascading, and from structure of the function δ, which guarantees that the moves are 1-level, it is immediate to prove that the marking graph $\langle \mathcal{N}, \uparrow_{\mathcal{M}_1} \rangle$ is a tree. Thus, $\mathcal{M}_1$ is a hypermarking.

5 1-Safe Nets Semantics

In this section it is shown how it is possible, starting from a hypernet, to build a 1-safe Petri nets whose behavior is equivalent to the behavior of the hypernet. The basic idea underlying this construction is to associate to each agent A and to each place p of agents in $\mathcal{N} \backslash A$ a place $\langle A, p \rangle$ which represents the presence of agent A in the place p. A token in this place means that the agent A is located at place p in the hypernet. Moreover, for each pair of agents A_j, A_j with $i \neq j$, a place $\langle A_i @ A_j \rangle$ is added in order to reflect the hierarchy structure of the agents of the hypernet. Place $\langle A_i @ A_j \rangle$ is marked if agent A_i is marked in a place of agent A_j.

Definition 18. *Given a hypernet $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$, its associated 1-safe net is the net $1S(H) = (B, E, F)$, such that:*

- $B = \{\langle A, p \rangle : p \in P_{\mathcal{N}}, A \in \mathcal{N}, p \notin P_A\} \cup \{A_i @ A_j : A_i, A_j \in \mathcal{N} \wedge i \neq j\}$
- $E = \{t_\Gamma : \Gamma \in CONS(H)\}$

Given a consortium $\Gamma = (l, \tau, C, \delta, \gamma)$ and its corresponding transition t_Γ

- $(\langle A, p \rangle, t_\Gamma) \in F \iff A \in C \wedge {}^\bullet\gamma(A) = p$
- $(t_\Gamma, \langle A, p \rangle) \in F \iff \gamma(A) \in G^{Out} \wedge (\delta(\gamma(A)))^\bullet = p)$
- $(t_\Gamma, \langle A, p \rangle) \in F \iff \gamma(A) \notin G^{Out} \wedge \gamma(A)^\bullet = p$

Moreover, a loop from consortium Γ to place $\langle A_i @ A_j \rangle$ is added if agent A_i sends/receive a token to/from agent A_j without being also a passive agent (see figure 12):

- $(\langle A_i @ A_j \rangle, t_\Gamma) \in F \wedge (t_\Gamma, \langle A_i @ A_j \rangle) \in F \iff \exists g_j \in G_j \cap G^{Down} : A_i \notin C \wedge \delta(g_j) \in G_i$
- $(\langle A_i @ A_j \rangle, t_\Gamma) \in F \wedge (t_\Gamma, \langle A_i @ A_j \rangle) \in F \iff \exists g_i \in G_i \cap G^{Up} : A_i \notin C \wedge \delta(g_i) \in G_j$

Finally, the following arcs are added to update places $\langle A_i @ A_j \rangle$ when a passive agent moves in the hierarchy:

- $(\langle A_i @ A_j \rangle, t_\Gamma) \in F \iff A_i \in C \wedge \gamma(A_i) \in G^{Out} \wedge \gamma(A_i) \in G_j$
- $(t_\Gamma, \langle A_i @ A_j \rangle) \in F \iff A_i \in C \wedge \gamma(A_i) \in G^{Out} \wedge \delta(\gamma(A_i)) \in G_j$

Now, an association between hypermarking and marking of the 1-safe net is defined:

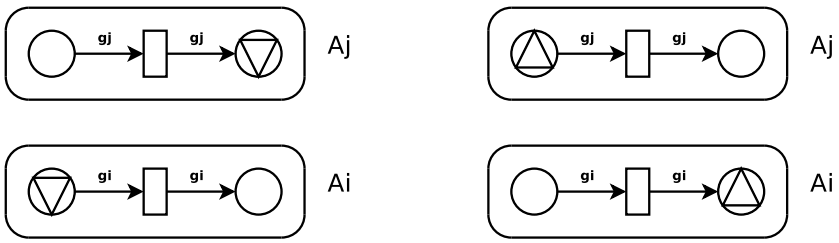


Fig. 12. Situations where a loop is added

Definition 19. *A marking $\mathcal{M}$ in a hypernet $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$ has a corresponding marking $m = 1S_H(\mathcal{M})$, that is:*

$$m(\langle A, p\rangle) = \begin{cases} 1 & \text{if } \mathcal{M}(A) = p \\ 0 & \text{otherwise} \end{cases}$$

$$m(\langle A_i @ A_j\rangle) = \begin{cases} 1 & \text{if } A_i \uparrow_{\mathcal{M}} A_j \\ 0 & \text{otherwise} \end{cases}$$

Notice that function $1S_H$ from reachable hypermarking of H to reachable marking of $1S(H)$ is injective, and the function between consortia of the hypernet and corresponding transition of the 1-safe net is bijectivechee. In the following theorem it is proved that the behavior of the 1-safe net simulates the behavior of the associated hypernet.

Theorem 3. *Let $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$ be a hypernet, $\mathcal{M}$ be a hypermarking and $\Gamma = (l, \tau, C, \delta, \gamma) \in CONS(H)$ be a consortium*

$$\mathcal{M}[\Gamma\rangle\mathcal{M}' \text{ in hypernet } H \Longrightarrow 1S_H(\mathcal{M})[t_\Gamma\rangle 1S_H(\mathcal{M}') \text{ in } 1S(H)$$
$$1S_H(\mathcal{M})[t_\Gamma\rangle m \text{ in } 1S(H) \Longrightarrow \mathcal{M}[\Gamma\rangle 1S^{-1}(m) \text{ in hypernet } H$$

Proof. *1. $\mathcal{M}[\Gamma\rangle\mathcal{M}'$ in hypernet $H \Longrightarrow 1S_H(\mathcal{M})[t_\Gamma\rangle 1S_H(\mathcal{M}')$ in $1S(H)$*
First, it is shown that if consortium Γ is enabled in a hypermarking $1S_H(\mathcal{M})$, then the corresponding transition t_Γ is enabled in the marking of the 1-safe net $1S_H(\mathcal{M})$:
$\mathcal{M}[\Gamma\rangle \Rightarrow 1S_H(\mathcal{M})[t_\Gamma\rangle$
Condition 1, condition 2, and condition 3 of Definition 7 hold for hypothesis. It has to be shown that each input place of t_Γ is marked in $1S(H)$.
There are two kind of input places of t_Γ: places which have the $\langle A, p\rangle$ form, and places which have the $\langle A_i @ A_j\rangle$ form (Definition 18).
The first kind of places are added when there is a passive agent A that is mapped via γ in a path which has an input place p. More precisely, there is an arc between a place $\langle A, p\rangle$ and transition t_Γ only when condition $A \in C \wedge {}^\bullet\gamma(A) = p$ is true. Thus, if this condition hold, place $\langle A, p\rangle$ must be

marked, i.e.: $\mathcal{M}(A) = p$ *(Definition 19). But, this is true for hypothesis because condition 1 of Definition 7 is true.*
An arc between the $\langle A_i@A_j\rangle$ *kind of places and* t_Γ *are added in the followind three situations:*
(a) $\exists g_i \in G_i \cap G^{Up} : A_i \notin C \wedge \delta(g_i) \in G_j$
(b) $\exists g_j \in G_j \cap G^{Down} : A_i \notin C \wedge \delta(g_j) \in G_i$
(c) $A_i \in C \wedge \gamma(A_i) \in G^{Out} \wedge \gamma(A_i) \in G_j$
Therefore, if these conditions hold the place $\langle A_i@A_j\rangle$ *must be marked, i.e.:* $A_i \uparrow_{\mathcal{M}} A_j$ *(Definition 19). It is easy to see that if the first holds, then for hypothesis (condition 2)* $A_i \uparrow_{\mathcal{M}} A_j$ *also holds. The same is true for the second case and condition 3. In the third case there exists a place* $p \in A_j$ *such that* ${}^\bullet\gamma(A_i) = p$*. For hypothesis* $\mathcal{M}(A_i) = p$ *(condition 1), which really means that* $A_i \uparrow_{\mathcal{M}} A_j$*.*
Then the first point of the theorem is proved:
$\mathcal{M}[\Gamma\rangle\mathcal{M}' \Rightarrow 1S_H(\mathcal{M})[t_\Gamma\rangle 1S_H(\mathcal{M}')$
It must be shown that $1S_H(\mathcal{M}') = 1S_H(\mathcal{M})\backslash {}^\bullet t \cup t^\bullet$*, that is the result of firing the consortium in the hypernet and firing the transition in the 1-safe net is the same. Definition 8 distinguishes between passive agents and other agents. If a non passive agent A is located in place p (i.e.:* $\mathcal{M}(A) = p$*) before the firing of the consortium in the hypernet, then it is also located in p after the firing of* Γ *(first condition of Definition 8), as long as place* $\langle A, p\rangle$ *stay marked in the 1-safe net (there is not an arc between* $\langle A, p\rangle$ *and* t_Γ *because A is not passive, i.e.:* $A \notin C$*).*
Passive agents are either marked in $(\delta(\gamma(A)))^\bullet$ *or* $\gamma(A)^\bullet$ *after the execution of consotrium* Γ*, depending if* $\gamma(A) \in G^{Out}$ *or not. In the same way, the corresponding places are marked in the 1-safe net because they are post-condition of transition* t_Γ *(see the fourth and fifth point of Definition 18)* .
The sixth and seventh conditions of Definition 18 are now analized. Loops for places of type $\langle A_i@A_j\rangle$ *leaves the place marked after the execution of transition* t_Γ*. In the same way in the hypernet, since* A_i *is not passive it is marked in the same place of agent* A_j *both before and after the execution of consortium* Γ *(Definition 8).*
The last two conditions says that if agent A_i *is passive (*$A_i \in C$*) and move in the hierarchy (*$\gamma(A_i) \in G^{Out}$*) then place* $\langle A_i@A_j\rangle$ *will not be marked anymore if* $\gamma(A_i) \in G_j$*, but place* $\langle A_i@A_k\rangle$ *is marked instead, where* $\delta(\gamma(A_i)) \in G_k$*. By analyzing the second condition of definition 8 it can be inferred that agent A will be marked in the agent containing the place* $\delta(\gamma(A_i))^\bullet$ *after the execution of* Γ*.*

2. $1S_H(\mathcal{M})[t_\Gamma\rangle m$ *in* $1S(H) \implies \mathcal{M}[\Gamma\rangle 1S_H^{-1}(m)$ *in hypernet H The proof of the secon part of the theorem can be build observing that the correspondence between the consortia of H and the transitions of* $1S(H)$ *is bijective, whereas the correspondence between the hypermarkings of H and the markings of* $1S(H)$ *is injective. Since they are both invertible marking* $\mathcal{M}$ *and consortium* Γ *can be retrieved. Analyzing the way the 1-safe net is built (Definition 18) it can be seen that marking m is itself the image of a hypermarking* $\mathcal{M}'$*.*

Indeed, places $\langle A, p\rangle$ which are preconditions of $1S(H)$ will not be marked after the firing of transition t_Γ because, by construction, the place marked is $\langle A, p'\rangle$ where p' is the place where agent A is located after firing consotrium Γ. In the same way places $\langle A_i@A_j\rangle$ are updated in such a way that the hierarchical structure of the hypernet is reflected.

As a consequences of theorem 3 and of the fact that each agent is marked in one place in every hypermarking, then it is possible to say that given a hypernet H the corresponding net $1S(H)$ is 1-safe.

6 Conclusions

Hypernets were introduced in [2] for modeling systems of mobile agents, where agents are structured in a containment hierarchy which can dynamically change. In this paper an extension that relaxes some constraints has been considered. It has been shown that the containment relation induces a tree-structure in the agent's hierarchy in any reachable configuration. Moreover it was shown that it is possible to translate a hypernet into a 1-safe net with an equivalent behavior.

Hypernets have a "law of conservation of tokens" which prevents creation and destruction of agents. From one point of view this is useful because it is guaranteed that the number of tokens is limited. On the other hand, in certain contexts, allowing the creation of new tokens can be useful. So, an interesting research topic regards the question of how to add to the Hypernet model a mechanism to create tokens maintaining a semantics expressed in terms of basic Petri nets.

Another important research topic pertains to the definition of techniques of behavioral analysis of systems modeled with hypernets. The translation into the 1-safe net and Theorem 3 guarantee that all analysis techniques studied for the class of 1-safe nets can be redefined for the class of hypernets defined in this paper. Invariant calculus and model checking are two of those techniques. To this purpose algorithms for generating in an "optimal" way the corresponding 1-safe net should be developed.

Acknowledgements. I wish to thank Luca Bernardinello and Lucia Pomello for their guidance and ideas.

References

1. M Bednarczyk, L Bernardinello, W Pawłowski, and L Pomello. Modelling and analysing systems of agents by agent-aware transition systems. In F. Fogelman-Soulie, editor, *Mining Massive Data Sets for Security: Advances in Data Mining, Search, Social Networks and Text Mining, and their Applications to Security*, volume 19, pages 103–112. IOS Press, 2008.
2. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. Modelling mobility with Petri Hypernets. In *Recent Trends in Algebraic Development Techniques*, volume 3423/2005 of *Lecture Notes in Computer Science*, pages 28–44. Springer Berlin / Heidelberg, 2005.

3. Marek A. Bednarczyk, Luca Bernardinello, Wiesław Pawłowski, and Lucia Pomello. From Petri hypernets to 1-safe nets. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, MOCA'06, Bericht 272, FBI-HH-B-272/06, 2006*, pages 23–43, Jun, 06.
4. Luca Bernardinello, Nicola Bonzanni, Marco Mascheroni, and Lucia Pomello. Modeling symport/antiport p systems with a class of hierarchical Petri nets. In *Membrane Computing*, volume Volume 4860/2007 of *Lecture Notes in Computer Science*, pages 124–137. Springer Berlin / Heidelberg, 2007.
5. Jordi Cortadella and Wolfgang Reisig, editors. *Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*, volume 3099 of *Lecture Notes in Computer Science*. Springer, 2004.
6. Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2005.
7. Michael Köhler and Berndt Farwer. Object nets for mobility. In Jetty Kleijn and Alexandre Yakovlev, editors, *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2007.
8. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling mobility and mobile agents using nets within nets. In Wil M. P. van der Aalst and Eike Best, editors, *ICATPN*, volume 2679 of *Lecture Notes in Computer Science*, pages 121–139. Springer, 2003.
9. Michael Köhler and Heiko Rölke. Properties of object Petri nets. In Cortadella and Reisig [5], pages 278–297.
10. Michael Köhler-Bußmeier and Frank Heitmann. On the expressiveness of communication channels for object nets. *Fundamenta Informaticae*, 93(1-3):205–219, 2009.
11. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In Cortadella and Reisig [5], pages 484–493.
12. Irina A. Lomazova. Nested Petri nets - a formalism for specification and verification of multi-agent distributed systems. *Fundam. Inform.*, 43(1-4):195–214, 2000.
13. Rüdiger Valk. Nets in computer organization. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume Volume 255/1987 of *Lecture Notes in Computer Science*, pages 218–233. Springer Berlin / Heidelberg, 1987.
14. Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *ICATPN*, volume 1420 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 1998.
15. Rüdiger Valk. Concurrency in communicating object Petri nets. In Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 164–195. Springer, 2001.
16. Rüdiger Valk. Object Petri nets: Using the nets-within-nets paradigm. In *Lectures on Concurrency and Petri Nets*, volume 3098/2004 of *Lecture Notes in Computer Science*, pages 819–848. Springer Berlin / Heidelberg, 2004.

Nets-in-nets with SNAKES

Franck Pommereau

LACL. University Paris East
61 avenue du général de Gaulle
94010 Créteil. France
pommereau@univ-paris12.fr

Abstract. This paper presents the toolkit SNAKES, focusing on the ability to model Petri nets whose tokens are Petri nets (so called *nets-in-nets*). SNAKES is a general Petri net library that allows to model and execute Python-coloured Petri nets: tokens are Python objects and net inscriptions are Python expressions. Since SNAKES itself is programmed in Python, Petri net inscriptions can handle Petri net objects as data values, for instance as tokens.

1 Introduction

SNAKES is a general Petri net library designed with quick prototyping in mind. For this aim, SNAKES offers a flexible architecture based on a *core library*, that defines a basic Petri net structure, complemented with a variety of *extension modules* (*i.e.*, *plugins*), that introduce additional features. The core library provides a general model of Python-coloured Petri nets: tokens are Python objects, transitions guards are Python expressions and arcs can be labelled with Python expressions. The core model of SNAKES is very similar to Jensen's well known ML-coloured Petri nets [12] but using Python instead of ML as colour domain.

In this paper, we focus on the fact that, because SNAKES itself is programmed in Python, it is possible to exploit the features of SNAKES within a Petri net inscriptions or tokens. In particular, we will show how to define Petri nets whose tokens are Petri nets, *i.e.*, nets-in-nets [27, 28, 13, 14, 25]. It is also possible to define interactions between the nets at various levels: we will show how to synchronise the firing of transitions in Petri nets used as tokens with the firing of the transition that consumes or produces these token nets. Moreover, we will show how to transform token nets at firing time and how to program transitions that modify the structure of the nets when fired, similarly to what happens in reconfigurable object nets [4, 11]. Our aim is to show the feasibility of these features through very simple example, a real implementation would require to design and program dedicated plugins in order to provide more general and reusable features, which is out of the scope of the present paper.

In order to demonstrate these possibilities, the rest of the paper will focus on concrete examples. The features and architecture of SNAKES will be introduced when needed in the examples. A general presentation of SNAKES can be found in [22, 23] and a tutorial is available at the SNAKES homepage [20]. Key features

of Python will be explained as needed and Python programs are very readable, so programmers usually feel comfortable with simple Python code (like in this paper). Thus, prior knowledge of Python or another imperative object-oriented programming language is desirable to read the source code in this paper. A good Python tutorial is available at the Python homepage [24]. The complete source code of the examples is provided as an appendix at the end of the paper. We assume that the reader is familiar with coloured Petri nets and nets-in-nets models.

1.1 Related works

The tool Renew [26] implements reference nets [16, 17], which can be seen as a variant of nets-in-nets where places carry references to Petri nets instead of Petri nets objects. To this respect, the techniques presented in this paper allow to implement reference nets since the programmer can choose to handle Petri net objects by reference of to copy them (see Section 3). Renew also implements asynchronous communication through fusion places and synchronous communication through channels. Both these aspects are left out of the scope of this paper. However, it is likely that they can be implemented using SNAKES: fusion places may be implemented by sharing place objects among several nets, and channels can be seen as a generalisation of the much simpler synchronisation mechanism presented in Section 5. To the best of our knowledge, Renew does not support Petri nets transformations like those presented in Sections 4 and 6. Renew is programmed in Java, it is actively developed and released as open source software with a specific licence.

JFern [19] is another Java tool to model Petri nets in which tokens are Java object and annotations are Java bytecode. This should allow to implement the features demonstrated in this paper, possibly with less flexibility since Python is more dynamic than Java. Unfortunately, to the best of our knowledge, JFern is almost not documented yet. The latest release (3.0.1) of JFern is dated of January 2006 but the project looks still active and patches are regularly contributed. Like SNAKES, JFern is free software released under the GNU LGPL [10].

Another similar project is PNtalk [15] that uses Smalltalk as colour domain. The latest version of PNtalk has been released in December 2008 and this is also free software distributed under the MIT licence [18].

Reconfigurable object nets, or RONs [11], have been implemented as an Eclipse extension presented in [4]. Using this tool, net transformation are defined graphically as local graph transformation rules associated to dedicated transitions. When such a transition fires, the attached rules are applied to the disjoint union of the Petri nets that the transitions consumes as tokens, producing a new Petri net to the post-places. In comparison, the approach presented in Section 4 allows to program directly in Python arbitrary transformations and to attach them to arbitrary transitions, which generalises the approach of RONs. Moreover, as shown in Section 6, transformations can also be applied to the Petri net the fired transition belongs to (and not only to the Petri nets used as

tokens). It can even be envisaged that the firing of a transition at any level of nesting can transform a Petri net at any other level, including an upper level.

Finally, we may consider COOPNBuilder [7] that implements the COOPN formalism [6] in which algebraic Petri nets are encapsulated into objects, the latter being then integrated into a context layer to specify their compositions. With respect to nets-in-nets, no nesting of Petri nets is allowed and three different classes of objects are involved (Petri nets objects, abstract data types and contexts). So, although COOPN has many similarities with the nets-in-nets variants addressed in this paper, COOPN is quite a distinct model. In particular, the context level involves coordination techniques that are completely out of the scope of SNAKES. The latest version of COOPNBuilder (1.0.7-β) has been released in March 2006 (but source code for version 1.0.11-β is available since July 2009). COOPNBuilder is free software distributed under the GNU GPL [9].

2 SNAKES overview

To start with, we define a very simple Petri net, which allows to introduce SNAKES by demonstrating how the well-known colored Petri nets concepts are implemented in SNAKES. This section thus shows how to:

- load SNAKES with plugins;
- build a Petri net using SNAKES;
- draw a Petri net using a plugin;
- fire transitions;
- change the marking of a net;
- compute the whole state space of a Petri net.

2.1 Loading SNAKES

First, we load SNAKES. If no plugin is needed, its enough to import module snakes.nets that exposes the core model of SNAKES and whose structure is depicted at the top of Figure 1. However, we would like to be able to draw Petri nets so we need to load a plugin called gv that uses GraphViz [1] in order to layout nets and print pictures of them.

Listing 1. `basic.py` — loading plugins.

```
1 import snakes.plugins
2 snakes.plugins.load("gv", "snakes.nets", "nets")
3 from nets import *
```

Line 1 loads module snakes.plugins in order to allow to call its function load in Line 2. (Notice that Python statements extend until the end of the line, with no semi-colon or other terminator.) This function load expects three arguments:

1. The name of the plugin to load, or a list of plugin names. Here, we load a single plugin called gv.

2. The name of the module being extended. This is almost always snakes.nets since it provides all the core library.
3. The name of the module created by extending snakes.nets with plugin gv. Here we call nets this freshly created module.

Thanks to this function call, a module called nets is now loaded and Line 3 allows to access directly to its content from our program.

Essentially, a plugin is implemented as a Python module with a function that can create sub-classes of classes from the extended module (here snakes.nets), or define completely new classes. Function snakes.plugins.load takes care to load the required modules, call the function to build the new classes, create a module with all the API in; it also avoids loading a plugin twice and manages plugins dependencies. This is illustrated in Figure 1 where module snakes.nets is extended by snakes.plugins.clusters, yielding a new module that is itself extended by snakes.plugins.gv. Plugin cluster is automatically involved by snakes.plugins.load since plugin gv depends on it. The module nets created in the above listing includes all the classes from the three modules depicted in Figure 1, taking into account the down-most version of classes PetriNet and StateGraph. Notice that this could not be achieved through standard Python import mechanism because the class hierarchy is defined dynamically, depending on which plugins are loaded by the program.

2.2 Building a Petri net

The next step is to create a Petri net as an instance of class PetriNet (Line 4), and add nodes (Lines 5–7) and arcs to it (Lines 8–9):

Listing 2. `basic.py` — building the Petri net.

```
4 n = PetriNet("mynet")
5 n.add_place(Place("p1", [dot]))
6 n.add_place(Place("p2"))
7 n.add_transition(Transition("t"))
8 n.add_input("p1", "t", Value(dot))
9 n.add_output("p2", "t", Value(dot))
```

The created net is given the name `"mynet"`. Then, each place is created as an instance of class Place whose constructor expects a name and an optional collection of tokens. Notice that, in this context, the name of a place (or of a transition), plays the role of an identifier that must be unique within a given Petri net. So, the first place is called `"p1"` and initially marked by a single black token (given as a list with only one dot item); the second place is called `"p2"` and is not initially marked. The black token value is available as name dot in SNAKES. Each place is added to the net using the dedicated method add_place. Line 7, a transition called `"t"` is added to the net. As usual with coloured Petri nets, a transition may be equipped with a guard that stands for a firing condition; if none is provided, like in this example, True is assumed. Arcs are divided into *input* arcs, *i.e.*, from a place toward a transition, and *output* arcs, *i.e.*, from a

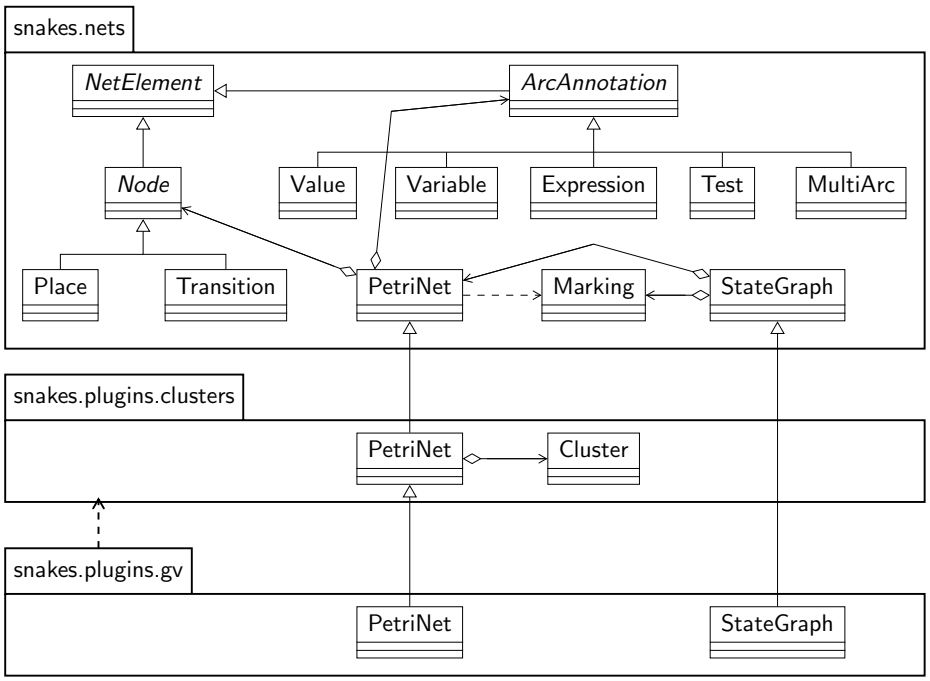


Fig. 1. Loading plugin gv, that depends on plugin clusters, on the top of module snakes.nets. (Picture from [22].)

transition toward a place. Line 8, an input arc is added from `"p1"` to `"t"` and labelled by Value(dot); this means that the arc can consume (because it is an input arc) a single token whose value is the black token dot. Similarly, an output arc is added in order to produce a black token into `"p2"` when `"t"` fires.

Thanks to plugin gv, a PetriNet instance is equipped with a method draw that allows to layout and draw the net. So, a picture of our net can be produced with a single statement:

Listing 3. `basic.py` — drawing the net.

```
10 n.draw("mynet.ps")
```

The result is shown on the left of Figure 2. Here, PostScript output has been requested by providing file extension "`.ps`", but many other formats are supported by GraphViz (in particular, PNG and JPEG).

2.3 Firing transitions

The firing of a transition can be decomposed in two steps:

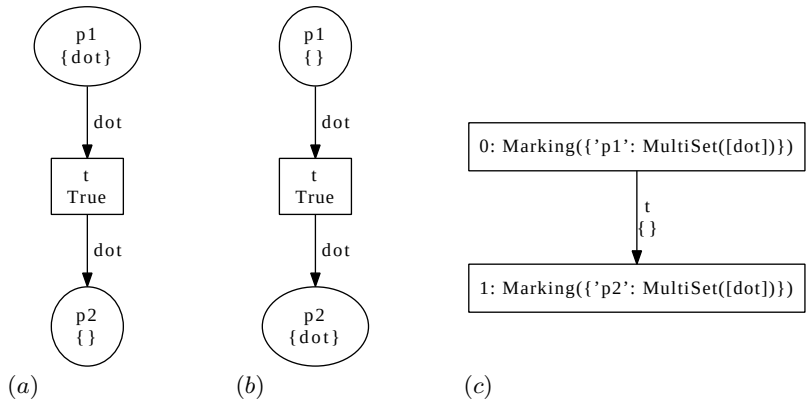


Fig. 2. (*a*) our first Petri net as drawn by GraphViz; (*b*) the same net after its transition has fired; (*c*) the corresponding marking graph. Each place is labelled by its name and marking (in set-like notation); each transition is labelled by its name and guard; each state in the marking graph is labelled by its number and marking; each edge in the marking graph is labelled by a transition name and a mode that correspond to the firing.

1. Possible *modes* are computed: a mode is binding of the variables in the transition guard and the annotations of its adjacent arcs that enables the transition.
2. A mode is chosen in order to actually fire the transition, *i.e.*, consume and produce tokens according to input and output arcs.

This is reflected by the following code:

Listing 4. `basic.py` — firing a transition.

```
11 t = n.transition("t")
12 m = t.modes()
13 t.fire(m.pop())
```

Line 11, the transition object for `"t"` is fetched from the net and stored in variable `t` for an easier access to it. Then, its modes are computed and stored into variable `m`. Since the transition involves no variable, only one mode is possible to fire it and this is the empty binding; so, `m` holds a single value. Last, the transition is fired with a mode picked from `m` (`m.pop()` removes and returns a value from `m`, that is here the unique value). In general, any mode from `m` could be chosen for firing and `m` is just a regular Python list. In particular, if `t` has no mode, then `m` is the empty list `[]`, resulting in an exception on `m.pop()`. If the net is drawn again, we get the picture displayed in the middle of Figure 2. By iterating over all the transitions of a Petri net, one may collect all the possible firings that may occur at a given marking.

2.4 Computing the state space

In order to reset the marking, we can use the following:

Listing 5. `basic.py` — resetting the marking.

```
15 n.set_marking(Marking(p1=MultiSet([dot]), p2=MultiSet([])))
```

Method set_marking expects a Marking instance as its argument, which is constructed by providing for each place the multiset of its tokens.

Then, instead of manually computing modes and firing transitions, we can use class StateGraph in order to compute all the reachable states from the provided initial marking. The resulting graph can be drawn thanks to plugin gv. For instance:

Listing 6. `basic.py` — building and drawing the state space.

```
16 g = StateGraph(n)
17 for i in g :
18     g.net.draw("mynet-%s.ps" % i)
19 g.draw("mynet-states.ps")
```

The constructor of class StateGraph expects the marked net whose state space has to be computed. Lines 17–18, a loop iterates over the states of g, numbered from zero. The body of the loop just contains Line 18 (Python defines block through indentation) that draws the net in each reached state: i is the state number and operator % allows to substitute the value of i inside the template string `"mynet-%s.ps"` (similarly to printf in C) with an automatic coercion of integer i to a string. Notice that g.net.draw is called, and not n.draw as one could expect; indeed, g uses a copy of n so that n is not affected when the marking of the copy is modified as the state space is explored. The resulting pictures are exactly the same as in Figure 2. Finally, the state space itself is drawn, which results in the picture shown on the right of Figure 2. Notice that the state space is constructed on-the-fly, *i.e.*, the successors of each state are computed only when the state is iterated over. This is useful, for instance, to display a progression of state space computation or to check a property on-the-fly.

3 Petri nets as tokens

In this section, we use instances of class PetriNet as tokens. For simplicity, we consider two levels: the token level where Petri nets use dot as their tokens, and an upper level that uses the nets of the token level as tokens. So, we call *token nets* the Petri nets used as tokens, and *container nets* the Petri nets whose places carry token nets. In general, any depth of nesting could be used by applying exactly the same technique. In order to produce the token nets, we define a function token_net:

Listing 7. `toknets.py` — defining a token net factory.

```
5 def token_net (name) :
```

```
6       a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
7       net = PetriNet(name)
8       net.add_place(Place(a, [dot]))
9       net.add_place(Place(b))
10      net.add_transition(Transition(t))
11      net.add_input(a, t, Value(dot))
12      net.add_output(b, t, Value(dot))
13      return net
```

This function (whose body, as usual, is delimited by indentation) expects as its sole argument a string that is used to define nodes names and that becomes the name of the returned Petri net. For instance, if `name="foo"` then Line 6 results in `a="afoo"`, `b="bfoo"` and `t="tfoo"`.

Then, a container Petri net is defined with a similar structure:

Listing 8. `toknets.py` — building the container net.

```
15  n = PetriNet("container")
16  n.add_place(Place("p1", [token_net("Toknet")]))
17  n.add_place(Place("p2"))
18  n.add_transition(Transition("t"))
19  n.add_input("p1", "t", Variable("x"))
20  n.add_output("p2", "t", Expression("x.copy()"))
```

Line 16, a new token net is created for the marking of place `"p1"`. Line 19, the input arc is labelled by a variable called `"x"`; this allows to consume one token from the input place while binding its name to a variable `x`. Line 20, the output arc is labelled by an expression; this allows to compute a new token to be produced in the output place. Here, we simply copy the token net bound to `x` by calling its method `copy`.

Copying net tokens may be important in order to avoid unwanted side effects. For instance: suppose that we add an output arc from the transition toward `"p1"` in order to reproduce the consumed token net, and suppose that we do not copy token nets (*e.g.*, by using `"x"` instead of `"x.copy()"` on the output arcs). Then, after firing the container's transition, the two places of the container net would hold each a reference to the same token net. Firing the transition of one of these token nets would thus modify the marking in both places of the container net. This may be desirable or not, and this can be controlled by choosing to produce either copies of token nets or references to them. For instance, to implement reference nets [16, 17], it may be chosen to produce references to Petri net objects instead of copies.

The rest of the program builds the marking graph, drawing the container net in each state as well as all the token nets in each place. This is made by iterating over every state and then over every token in every place at each given state. The resulting pictures are displayed in Figure 3.

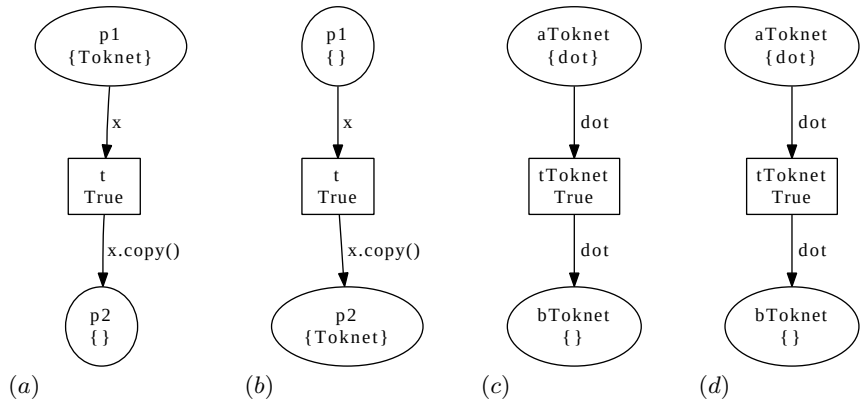


Fig. 3. (*a*) the initial state of the container net; (*b*) the state of the container net after firing; (*c*) the token net from place `"p1"` in (*a*); (*d*) the token net from place `"p2"` in (*b*), which is a copy of that in (*c*).

4 Transforming token nets

In this section, we elaborate on the previous example and consider a more complex case where two token nets are composed in order to produce a new one. This is one particular case of net transformation and, more generally, arbitrary transformations may be used.

For this example, we make use of Petri net compositions defined in the *Petri Box Calculus* (PBC) and its variants [2, 3, 8, 21] and implemented in SNAKES. These composition operations have been chosen here because they are already implemented in SNAKES, which is not the case of, *e.g.*, those from BPMN or BPEL [29, 5]. In a nutshell, each place in PBC is labelled with a *status*, allowing to distinguish *entry* places, *internal* places and *exit* places. All together, these places form the *control flow* of a Petri net and PBC defines binary operators to compose nets with respect to their control flow. If N_1 and N_2 are two PBC nets:

- the *sequential composition* $N_1 \,\&\, N_2$ allows to execute N_1 once, followed by one execution of N_2;
- the *choice* $N_1 + N_2$ allows to execute once either N_1 or N_2;
- the *iteration* $N_1 * N_2$ allows to execute repeatedly N_1, followed by one execution N_2;
- the *parallel composition* $N_1 \mid N_2$ allows to execute both N_1 and N_2 concurrently. This latter operation simply consists of a disjoint union of the two composed nets.

Notice that these operators are denoted above as implemented in SNAKES, instead of as originally defined in PBC (SNAKES uses Python operators).

We now consider token nets similar to the previous ones, with input places (the `a`$\cdots$ ones) being labelled as entry places, and output places (the `b`$\cdots$ ones)

labelled as exit places. The container net consists of one transition with two input places holding one token net each, and one output place to receive the parallel composition of the two token nets consumed upon firing.

The program starts with the loading a SNAKES extended with plugins gv (to draw nets) and ops (to support PBC operations). Then, function token_net is defined just as before except that it assigns statuses to the places (Lines 8–9):

Listing 9. `compose.py` — loading plugins and defining token nets factory.

```
1  import snakes.plugins
2  snakes.plugins.load(["gv", "ops"], "snakes.nets", "nets")
3  from nets import *
4
5  def token_net (name) :
6      a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
7      net = PetriNet(name)
8      net.add_place(Place(a, [dot], status=entry))
9      net.add_place(Place(b, status=exit))
10     net.add_transition(Transition(t))
11     net.add_input(a, t, Value(dot))
12     net.add_output(b, t, Value(dot))
13     return net
```

Then, the container net, called `"composer"`, is created as explained above:

Listing 10. `compose.py` — building the container net.

```
15 n = PetriNet("composer")
16 n.add_place(Place("left", [token_net("foo")]))
17 n.add_place(Place("right", [token_net("bar")]))
18 n.add_place(Place("parall"))
19 n.add_transition(Transition("t"))
20 n.add_input("left", "t", Variable("x"))
21 n.add_input("right", "t", Variable("y"))
22 n.add_output("parall", "t", Expression("x|y"))
```

Lines 20–21, notice the arcs labelled by variables allowing to bind the two token nets to different names x and y. Line 22, the parallel composition of these two token nets is computed by simply providing expression "x|y" on the output arc.

The rest of the program builds the state space and draws all the container and token nets at each state as previously. This results in the pictures shown in Figure 4. It may be noted that node names in a compound net are systematically obtained from the node names of its components; for instance, "$[x|]$" denotes a new node obtained from a node called x in the left operand of the parallel composition (read this name as "x" on the left of "|", with "$[\cdots]$" around to handle nested compositions). Similarly, the name of the compound net "foo|bar" is derived from that of its components. This renaming is made systematically by plugin ops to ensure that only disjoint nets are composed.

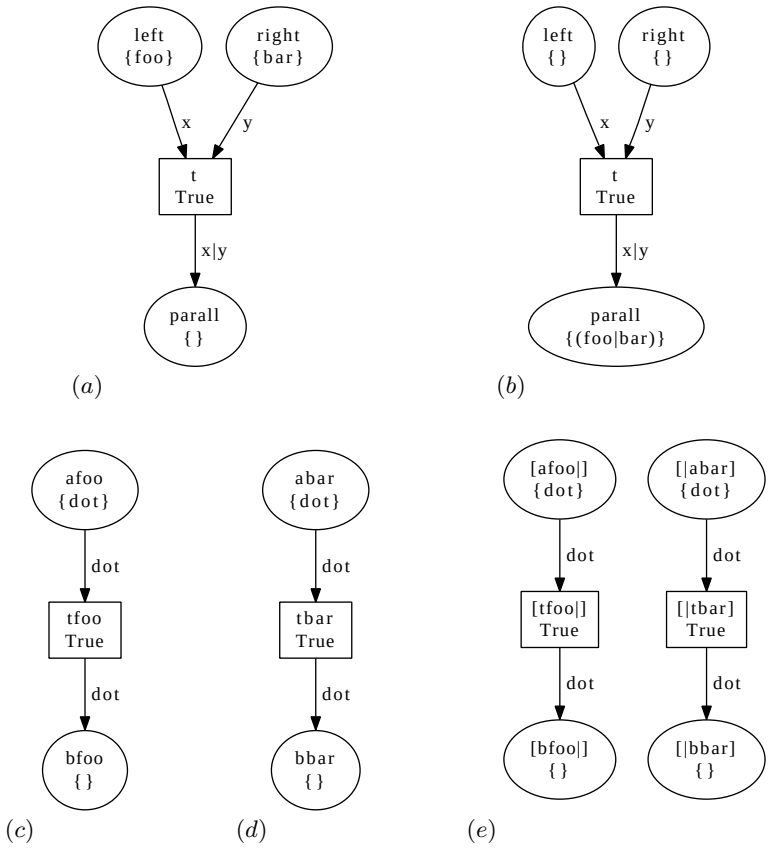


Fig. 4. (*a*) the initial state of the container net; (*b*) the container net after firing; (*c*) the token net from place `"left"` in (*a*); (*d*) the token net from place `"right"` in (*a*); (*e*) the token net from place `"parall"` in (*b*), which is the parallel composition of the token nets in (*c*) and (*d*).

5 Synchronising firing

Let us consider again the example from Section 3. In this section, we elaborate on it to enforce synchronised firing between transitions at the two levels.

The beginning of this program is exactly as in Listing 7 (see Listing 17). Then, after the definition of function token_net, we also define a function ready to check whether transitions from a token net are ready to fire. This function consists of iterating over given transitions and for each, test if it has no mode and if so, returns False. If none of the provided transition is inactive, then function ready returns True. A second helper function called synchro is needed to produce a copy of a net in which specified transitions have been fired. This function does not check whether firing the given transitions is possible (if not, an exception will occur) and we assume that function ready has been used appropriately for this purpose. Indeed, this is checked in the guard of the transition in the container net:

Listing 11. `synchro.py` — building the container net.

```
29 n = PetriNet("synchroniser")
30 n.globals["ready"] = ready
31 n.globals["synchro"] = synchro
32 n.add_place(Place("p1", [token_net("Toknet")]))
33 n.add_place(Place("p2"))
34 n.add_transition(Transition("t", Expression("ready(x,'tToknet')")))
35 n.add_input("p1", "t", Variable("x"))
36 n.add_output("p2", "t", Expression("synchro(x,'tToknet')"))
```

Lines 30–31, the defined functions are declared in the execution environment of the container net. Function ready is used Line 34 in the guard of the transition (provided by an Expression object as the second argument of the constructor of class Transition). Function synchro is used on the output arc in order to produce a copy of the net taken from the input place (as x) in which the transition `"tToknet"` has fired. We known that this firing is possible since transition `"t"` is guarded by ready(x,'tToknet') that exactly checks this. Notice that Python accepts both single and double quotes to delimit strings, which is very convenient to embed in a string Python expressions that involve string literals (like in Lines 34 and 33 above).

The rest of the program draws states as previously in order to produce the pictures shown in Figure 5.

6 Self-modifying nets

As a last example, we consider a Petri net whose structure is modified by the firing of its transitions. We start with a net that has the same structure as that of Figure 2(a). When its transition `"t"` fires, it creates a new transition (called `"back"`) and arcs in order to reset the marking. When this transition `"back"` fires, it removes itself from the net.

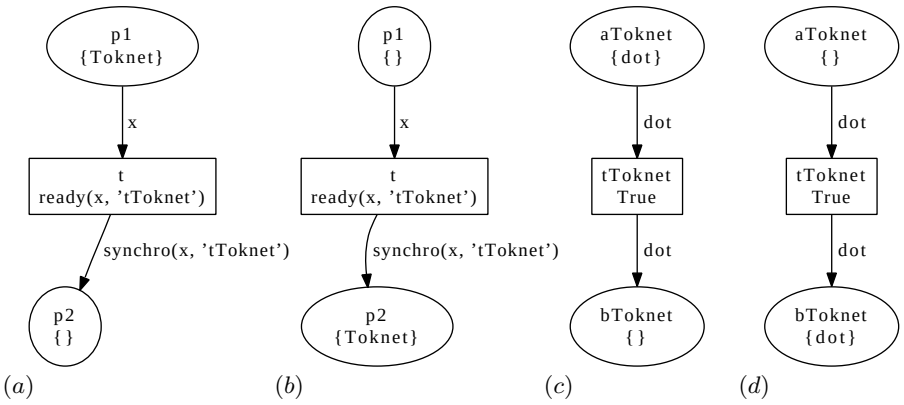


Fig. 5. (a) the initial state of the container net; (b) the state of the container net after firing; (c) the token net from place `"p1"` in (a); (d) the token net from place `"p2"` in (b), which is exactly that from (c) after the firing of its transition. With respect to Figure 3, notice in (d) that the transition as fired.

In order to achieve such an effect, we extend twice class Transition in order to redefine its method fire: a first class BackwardTransition is used to implement the transition added to the net; a second class ForwardTransition is used to implement the adding of a BackwardTransition.

Listing 12. `modify.py` — defining transition classes for net transformation.

```
5  class BackwardTransition (Transition) :
6      def fire (self, binding) :
7          Transition.fire(self, binding)
8          self.net.remove_transition(self.name)
9
10 class ForwardTransition (Transition) :
11     def fire (self, binding) :
12         Transition.fire(self, binding)
13         self.net.add_transition(BackwardTransition("back"))
14         self.net.add_input("p2", "back", Value(dot))
15         self.net.add_output("p1", "back", Value(dot))
```

The method fire of class Transition or one of its sub-classes is passed two arguments: self is the instance whose method has been called (this is an explicit argument in Python and corresponds to this in Java) and binding is the mode under which the transition is fired. Line 7, method fire from the super-class Transition is called in order to achieve the actual firing. Line 8, the transition removes itself from the Petri net that is available as attribute net. Notice that removing a transition also removes all its arcs. Class ForwardTransition is defined

similarly, the only difference is that it adds a BackwardTransition and arcs to the net instead of removing them.

The rest of the code is dedicated to build the net and draw the two states of the net, resulting in the pictures shown in Figure 6.

Listing 13. `modify.py` — building the self-modifying net and firing a transition.

```
17 n = PetriNet("modifier")
18 n.add_place(Place("p1", [dot]))
19 n.add_place(Place("p2"))
20 n.add_transition(ForwardTransition("t"))
21 n.add_input("p1", "t", Value(dot))
22 n.add_output("p2", "t", Value(dot))
23
24 n.draw("modifier-0.ps")
25 n.transition("t").fire(n.transition("t").modes().pop())
26 n.draw("modifier-1.ps")
```

Notice Line 20 that we use ForwardTransition instead of Transition.

To build a meaningful state space of such a net, it is necessary to update the notion of state. Indeed, a marking is not enough to know the possible successor states but we need to store in each state the full Petri net, including its structure and marking.

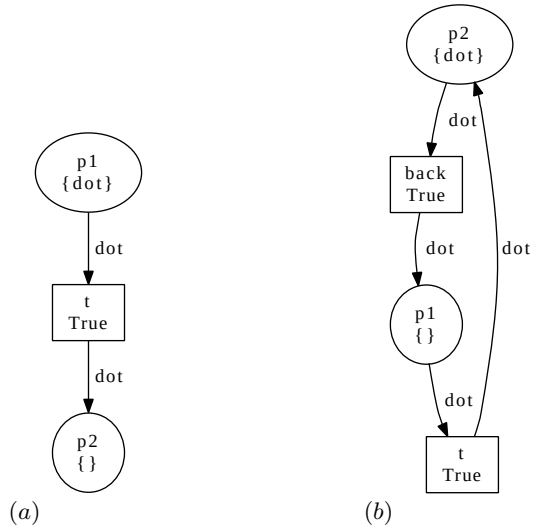


Fig. 6. (*a*) the initial state of the self-modifying net; (*b*) the state of the self-modifying net after firing the forward transition. Firing the backward transition from (*b*) would result in net (*a*) again.

7 Conclusion

In this paper, we have presented the toolkit SNAKES, focusing on its ability to represent and execute Petri whose tokens are Petri nets. In particular, we have shown that it is easy to perform transformations on nets at different levels of nesting at firing time, and to synchronise the firing of the transitions at different levels.

The code presented throughout the paper should be considered as a proof of concept rather than as a guideline about how to implement these features in a real tool. For such a use case, it would be better to carefully design and program a plugin for SNAKES in order to provide a solution that would be both more general and easier to use. The design phase in particular seems crucial for both generality and robustness, it's likely that it requires a good background about nets-in-nets. Any contribution to SNAKES or collaboration proposal toward adding such a plugin will be welcomed.

Future works will be dedicated to model more complex and realistic examples from the domain of objects, components, agents and services. This will allow in particular to draw comparison with existing tools in terms of ease of use, expressive power and performances. Moreover, we implemented here only primitive features, not directly relating them to any particular model. It will be interesting to consider a selection of well-established and theoretically founded models and implement them using SNAKES. Considering existing models will allow to clearly fix the requirements for the various features envisaged (*e.g.*, information exchange between synchronised transitions is formalised as channels in reference nets [16]). For these future works also, any collaboration proposal will be welcomed.

References

1. AT&T Research. GraphViz, graph visualization software. ⟨`http://www.graphviz.org`⟩.
2. E. Best, R. Devillers, and J. Hall. The Petri box calculus: a new causal algebra with multilabel communication. In *Advances in Petri Nets 1992*, volume 609 of *LNCS*. Springer, 1992.
3. E. Best, R. Devillers, and M. Koutny. *Petri net algebra*. Springer, 2001.
4. E. Biermann, C. Ermel, F. Hermann, and T. Modica. A visual editor for reconfigurable object nets based on the ECLIPSE graphical editor framework. In G. Juhas and J. Desel, editors, *AWPN'07*, Universität Koblenz-Landau, Germany, 2007.
5. Business process execution language for Web services (BPEL), version 1.1. ⟨`http://www.ibm.com/developerworks/library/ws-bpel`⟩.
6. D. Buchs and N. Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Trans. Software Eng.*, 26(7), 2000.
7. COOPNBuilder. ⟨`http://smv.unige.ch/research-projects/co-opn`⟩.
8. R. Devillers, H. Klaudel, M. Koutny, and F. Pommereau. Asynchonous box calculus. *Fundamenta Informaticae*, 54(1), 2003.
9. F. S. Foundation. GNU general public license. ⟨`http://www.fsf.org/licensing/licenses/gpl.html`⟩.

10. F. S. Foundation. GNU lesser general public license. ⟨http://www.fsf.org/licensing/licenses/lgpl.html⟩.
11. K. Hoffmann, T. Mossakowski, and H. Ehrig. High-level nets with nets and rules as tokens. In *ICATPN'05*, volume 3536 of *LNCS*. Springer, 2005.
12. K. Jensen and L. M. Kristensen. *Coloured Petri Nets: modelling and validation of concurrent systems*. Springer, 2009.
13. M. Köhler and H. Rölke. Concurrency for mobile object-net systems. *Fundamenta Informaticae*, 54(2–3), 2003.
14. M. Köhler and H. Rölke. Properties of object Petri nets. In *ICATPN'04*, volume 3099 of *LNCS*. Springer, 2004.
15. R. Kočí. Towards model-based design with PNtalk. In *MOSMIC'05*, Žilina, Slovakia, 2005.
16. O. Kummer. Simulating synchronous channels and net instances. In J. Desel, P. Kemper, E. Kindler, and A. Oberweis, editors, *5. Workshop Algorithmen und Werkzeuge für Petrinetze*. Universität Dortmund, Fachbereich Informatik, 1998.
17. O. Kummer. Tight integration of Java and Petri nets. In J. Desel and A. Oberweis, editors, *6. Workshop Algorithmen und Werkzeuge für Petrinetze*. Goethe-Universität, Institut für Wirtschaftinformatik, Frankfurt am Main, Fachbereich Informatik, 1999.
18. MIT licence. ⟨http://www.opensource.org/licenses/mit-license.php⟩.
19. M. Nowostawski. JFern, Java-based Petri net framework. ⟨http://sourceforge.net/projects/jfern⟩.
20. F. Pommereau. SNAKES is the net algebra kit for editors and simulators. ⟨http://lacl.univ-paris12.fr/pommereau/soft/snakes⟩.
21. F. Pommereau. Versatile boxes: a multi-purpose algebra of high-level Petri nets. In *DASDS'04*. SCS/ACM, 2004.
22. F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, September 2008.
23. F. Pommereau. Quickly prototyping Petri nets tools with SNAKES. In *PNTAP'08*, ACM Digital Library. ACM, 2008.
24. Python Software Foundation. Python programming language. ⟨http://www.python.org⟩.
25. R. Sánchez-Herrera, N. Villanueva-Paredes, and E. López-Mellado. High-level modelling of cooperative mobile robot systems. In *Distributed Autonomous Robotic Systems 6*. Springer, 2007.
26. Theoretical Foundations Group, Department for Informatics, University of Hamburg. Renew: The reference net workshop. ⟨http://www.renew.de⟩.
27. R. Valk. Petri nets as token objects: an introduction to elementary object nets. In *ICATPN'98*. Springer-Verlag, 1998.
28. R. Valk. Object Petri nets: Using the nets-within-nets paradigm. In *Advanced course on Petri nets*, volume 3098 of *LNCS*, pages 819–848. Springer, 2003.
29. S. White. Business process modeling notation (BPMN), version 1.0. ⟨http://www.bpmi.org⟩.

A Complete source code of the examples

A.1 Basic example

Listing 14. `basic.py`

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *
n = PetriNet("mynet")
n.add_place(Place("p1", [dot]))
n.add_place(Place("p2"))
n.add_transition(Transition("t"))
n.add_input("p1", "t", Value(dot))
n.add_output("p2", "t", Value(dot))
n.draw("mynet.ps")
t = n.transition("t")
m = t.modes()
t.fire(m.pop())
n.draw("mynet-bis.ps")
n.set_marking(Marking(p1=MultiSet([dot]), p2=MultiSet([])))
g = StateGraph(n)
for i in g:
    g.net.draw("mynet-%s.ps" % i)
g.draw("mynet-states.ps")
```

A.2 Nets as tokens

Listing 15. `toknets.py`

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *

def token_net(name):
    a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
    net = PetriNet(name)
    net.add_place(Place(a, [dot]))
    net.add_place(Place(b))
    net.add_transition(Transition(t))
    net.add_input(a, t, Value(dot))
    net.add_output(b, t, Value(dot))
    return net

n = PetriNet("container")
n.add_place(Place("p1", [token_net("Toknet")]))
n.add_place(Place("p2"))
```

```
n.add_transition(Transition("t"))
n.add_input("p1", "t", Variable("x"))
n.add_output("p2", "t", Expression("x.copy()"))

g = StateGraph(n)
for i in g :
    g.net.draw("%s-state%s.ps" % (g.net.name, i))
    for place in g.net.place() :
        for j, tok in enumerate(place.tokens) :
            tok.draw("%s-state%s-place%s-token%s.ps"
                     % (g.net.name, i, place.name, j))
```

A.3 Composing token nets

Listing 16. `compose.py`

```
import snakes.plugins
snakes.plugins.load(["gv", "ops"], "snakes.nets", "nets")
from nets import *

def token_net (name) :
    a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
    net = PetriNet(name)
    net.add_place(Place(a, [dot], status=entry))
    net.add_place(Place(b, status=exit))
    net.add_transition(Transition(t))
    net.add_input(a, t, Value(dot))
    net.add_output(b, t, Value(dot))
    return net

n = PetriNet("composer")
n.add_place(Place("left", [token_net("foo")]))
n.add_place(Place("right", [token_net("bar")]))
n.add_place(Place("parall"))
n.add_transition(Transition("t"))
n.add_input("left", "t", Variable("x"))
n.add_input("right", "t", Variable("y"))
n.add_output("parall", "t", Expression("x|y"))

g = StateGraph(n)
for i in g :
    g.net.draw("%s-state%s.ps" % (g.net.name, i))
    for place in g.net.place() :
        for j, tok in enumerate(place.tokens) :
            tok.draw("%s-state%s-place%s-token%s.ps"
                     % (g.net.name, i, place.name, j))
```

A.4 Synchronised firing

Listing 17. `synchro.py`

```
import snakes.plugins
snakes.plugins.load("gv", "snakes.nets", "nets")
from nets import *

def token_net (name) :
    a, b, t = "a%s" % name, "b%s" % name, "t%s" % name
    net = PetriNet(name)
    net.add_place(Place(a, [dot]))
    net.add_place(Place(b))
    net.add_transition(Transition(t))
    net.add_input(a, t, Value(dot))
    net.add_output(b, t, Value(dot))
    return net

def ready (net, *names) :
    for n in names :
        if not net.transition(n).modes() :
            return False
    return True

def synchro (net, *names) :
    net = net.copy()
    for n in names :
        t = net.transition(n)
        m = t.modes()
        t.fire(m.pop())
    return net

n = PetriNet("synchroniser")
n.globals["ready"] = ready
n.globals["synchro"] = synchro
n.add_place(Place("p1", [token_net("Toknet")]))
n.add_place(Place("p2"))
n.add_transition(Transition("t", Expression("ready(x,'tToknet')")))
n.add_input("p1", "t", Variable("x"))
n.add_output("p2", "t", Expression("synchro(x,'tToknet')"))

g = StateGraph(n)
for i in g :
    g.net.draw("%s-state%s.ps" % (g.net.name, i))
    for place in g.net.place() :
        for j, tok in enumerate(place.tokens) :
            tok.draw("%s-state%s-place%s-token%s.ps"
```

```
44                                  % (g.net.name, i, place.name, j))
```

A.5 Self-modifying net

Listing 18. `modify.py`

```
1  import snakes.plugins
2  snakes.plugins.load("gv", "snakes.nets", "nets")
3  from nets import *
4
5  class BackwardTransition (Transition) :
6      def fire (self, binding) :
7          Transition.fire(self, binding)
8          self.net.remove_transition(self.name)
9
10 class ForwardTransition (Transition) :
11     def fire (self, binding) :
12         Transition.fire(self, binding)
13         self.net.add_transition(BackwardTransition("back"))
14         self.net.add_input("p2", "back", Value(dot))
15         self.net.add_output("p1", "back", Value(dot))
16
17 n = PetriNet("modifier")
18 n.add_place(Place("p1", [dot]))
19 n.add_place(Place("p2"))
20 n.add_transition(ForwardTransition("t"))
21 n.add_input("p1", "t", Value(dot))
22 n.add_output("p2", "t", Value(dot))
23
24 n.draw("modifier-0.ps")
25 n.transition("t").fire(n.transition("t").modes().pop())
26 n.draw("modifier-1.ps")
```

Part III

Short Presentations

From Service-Oriented Architecture via Coloured Petri Nets to Java Code

Zheng Liu and Kees M. van Hee

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
z.liu3@tue.nl, k.m.v.hee@tue.nl

Abstract. In software engineering we have many methods to design systems at different levels of abstraction. One of the main problems is to transform a design at one level to a design at another level. In this paper we consider system design at two levels: system modelling and code design. For the first level we use coloured Petri nets as a modelling language and Service Oriented Architecture as a style of modelling. For the second level we use Java as a language and the object-oriented style of programming. We show how one can transform system models into programmes.

Keywords: Code generation; SOA; Coloured Petri nets; Java

1 Introduction

At a high level, software systems are often described by their architecture consisting of *components*, also called modules, services or even agents, and their relationships. Service-Oriented Architecture (SOA) is a popular style of architecture, which is underlied by different technologies, i.e., SOAP [2], BPEL [3], WSDL [4] and etc. SOA has at least five important characteristics: (1) asynchronous communication, (2) a clear separation of functional specification and internal structure, (3) service discovery, (4) dynamic service binding, and (5) use of XML-based standards. When specifying, modelling, and analysing a complex system, we need a language in which we express the static and dynamic properties of the system. Petri nets, in particular coloured Petri nets ([18], [19], and [20]), are widely used for this purpose. Petri nets have the advantage over other process languages that they have clear semantics. There are many methods to analyse models, and there are software tools (i.e., CPN Tools [1]) to facilitate the modelling and analysis including the animation and simulation of the designed system. However, a system model has to be transformed into programme code for deployment in real life. This is a non-trivial task because the styles of modelling and programming are often different. There are many papers on code generation from Petri nets, [12] provides an overview. In this paper we use the SOA style for modelling with coloured Petri nets and the Object-Oriented (OO) style of programming using Java.

Today the OO style has been widely used in programming. There is, however, a gap between the SOA style, Petri net modelling, and object-oriented programming. Object-oriented programming languages and Petri nets are based upon different concepts. On the one hand, some concepts of the object-oriented paradigm such as encapsulation, inheritance, and etc are widely used in system modelling. On the other hand, Petri nets do not fully support all the major concepts of the object-oriented paradigm (i.e., for inheritance in Petri nets see [9]). How to integrate Petri nets with object-oriented concepts is still a rich research domain [12].

The goal of this paper is to bridge the gap between system modelling at the architectural level using coloured Petri nets and software design using an object oriented language. We provide a systematic approach for the component-based modelling with coloured Petri nets and the transformation from the component models into Java code. The model-to-code transformation is achieved in this paper manually, and is illustrated by an example. In principle, the code generation can be done (partly) automatically, but that is out of scope of this paper.

The rest of the paper is organised as follows. A background upon coloured Petri nets and Java language is provided in Section 2. How to model components with coloured Petri nets is discussed in Section 3. In Section 4 we introduce the rules to transform component-based coloured Petri nets to Java code. We present other related work in Section 5. Finally, Section 6 concludes the paper.

2 Background

We provide a short introduction to coloured Petri nets and the Java programming language. More information can be found in [18], [19], [20] for Petri nets, and in [17], [25] for Java programming. The classical Producer-Consumer model is our leading example.

2.1 Coloured Petri nets

Petri nets are mathematically precise models of discrete event systems. A Petri net is a graph consisting of two types of nodes: *places* and *transitions*, connected by directed arcs only between nodes of different types. The places neighbouring a transition are called input or output places depending upon the arc direction. The state, called *marking*, of a Petri net is a distribution of *tokens* over places. A place may contain zero or more tokens. Tokens either are indistinguishable objects (in *classical* Petri nets) or have a data value (in *coloured* Petri nets). The value of a token belongs to a certain type, called *colour set*, associated to the place where the token resides. A transition may cause a discrete event, called a *firing*, which changes the state. In coloured Petri nets the arcs are associated with variables and formulae, called *arc inscriptions*. The variables are bound to values by the proposed input tokens. The produced tokens by firing transitions obtain a value by the inscriptions on the arcs to the output places. In addition a transition can have a *guard*, which is a predicate and only if the predicate

evaluates to true, using the binding of the variables provided by the input tokens, the transition may fire.

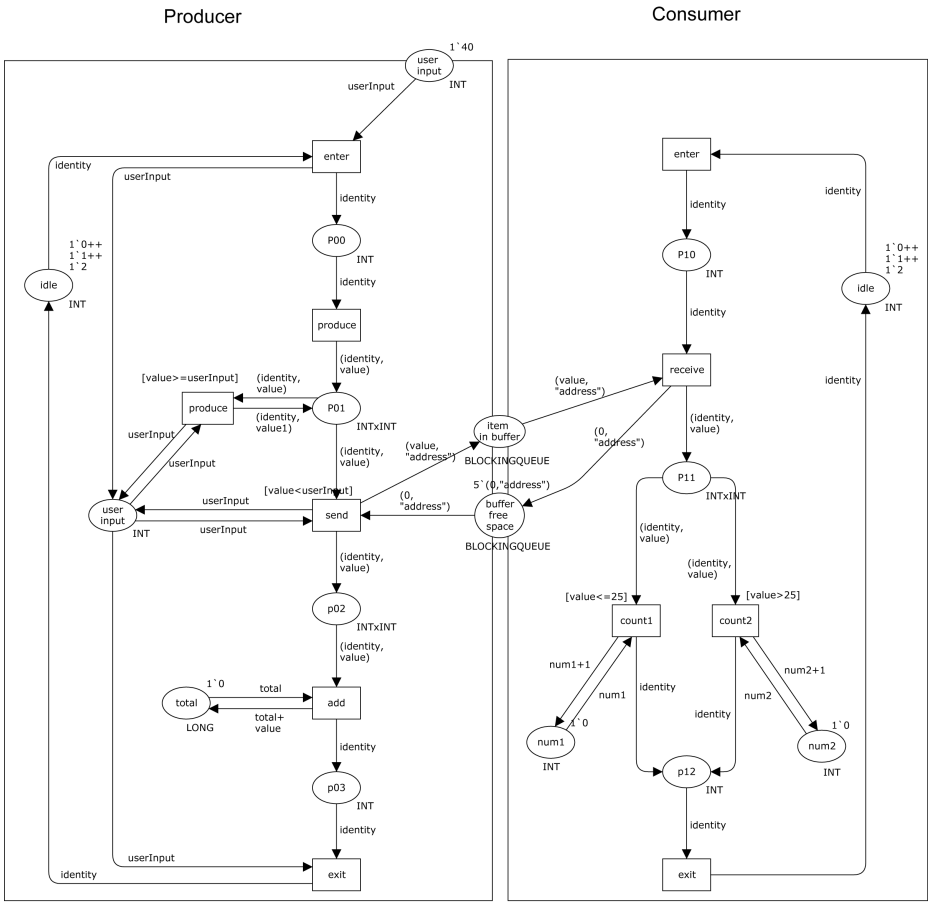
Coloured Petri nets, used in CPN Tools, have data values for tokens, as well as time stamps for tokens and hierarchy. The arc inscriptions are specified in the functional programming language ML [21]. Hierarchy in Petri nets allows one to refine a transition to a separate component, called *subnet* or in CPN-Tools, *subpage*. This allows one to build a model in either a top-down or a bottom-up manner.

Fig. 1 depicts our Producer-Consumer example. This example consists of two components for a producer and a consumer, respectively. Each component has a place called *idle*, it holds the identity of component instances. The Producer receives an external user input value and stores it in the place *userInput*. The Producer keeps generating a random value till the value generated is less than the user input. Then the Producer sends the last generated value to the place *item in buffer*. The place *item in buffer* and the place *buffer free space* together represent a buffer, where the place *item in buffer* holds the elements stored in the buffer, and the place *buffer free space* holds the available space in the buffer. The Producer calculates the total value of the production, and stores the value in the place *total*. Finally, the Producer returns the value of its identity to the place *idle*. The Consumer retrieves data from the buffer, so the available buffer space is increased after each retrieval. If the retrieved data value is less than or equal to a predetermined value (25 in this example), then the place *num1* is increased by 1, otherwise the place *num2* is increased by 1. Finally, the Consumer returns the value of its identity to the place *idle*. The Producer and the Consumer interact through exchange of tokens via a pair of places: *item in buffer* and *buffer free space*.

2.2 Java programming language

Java is a high-level general-purpose object-oriented programming language. The main idea of object orientation is to design software around *objects* it manipulates, rather than the actions it performs. A *class* is a blueprint describing the state and behaviour of the created *instances*. A class can create one or more instances, or called objects. A class has a set of *static fields* and *static methods* merely associated with the class, and shared by all its objects. In contrast, each object of a class owns a set of *instance fields* and *instance methods*, separated among the objects. A class has one or more *constructors* to initialise the instance fields of its objects. *Encapsulation* provides the basis for modularity by hiding information from external access and attaching the information to the methods that need access to it only. Therefore, an object has an internal structure hidden from the outside world, and a well defined behaviour.

Java provides a mechanism for better hierarchical organisation of a programme, called *class nesting*. In class nesting, one class (called *nested class*) is defined within another class (called *enclosing class*). Class nesting represents a *composition relationship*, as a nested class is a member of its enclosing class.




```
colset STRING=string;                    var identity:INT;
colset INT=int with 0..99;               var userInput,value,value1,num1, num2:INT;
colset LONG=int;                         var total:LONG;
colset INTxINT=product INT*INT;
colset BLOCKINGQUEUE=product INT*STRING;
```

Fig. 1. A Prodcuer-Consumer Example

Multithreading allows two or more units of the same programme to run concurrently on multiple processors which can be simulated on one processor. Such a programme unit is called a *thread*. To create a thread, a class has to implement Java *Runnable* interface (or inherit Java *Thread* class), which has a single method called *run()*. The code belonging to the thread is placed into the *run()* method. When a thread is started, the *run()* method is executed. A thread terminates normally when its *run()* method terminates. When a thread is interrupted, the most common response is to terminate the *run()* method. Threads communicate primarily by sharing access to a *BlockingQueue*, which is a Java public interface with various implementation classes since Java Development Kit (JDK) 1.5.

3 Modelling a Service Oriented Architecture with Coloured Petri Nets

The architecture of a system is a model consisting of components and their relationships. Relationships denote either hierarchy (one component is part of another) or they denote communication between two components. Components can be atomic or composed by other components. Our component model is expressed using coloured Petri nets. A good introduction regarding SOA is provided in [10]. In this section we briefly summarise the necessary concepts for our component modelling based on [7]. Also, we will introduce a systematic approach to model components.

3.1 SOA concepts

The component model presents an abstract view upon the components of a system as well as the interactions between components by message exchange. In the SOA paradigm a component delivers services to other components while it makes use of services of other components. A component can be treated as either a *black* box or a *white* box. For a black box, the internal knowledge of a component is hidden, and is viewed in terms of its input, output and transfer characteristics. For a white box, the architecture of a component is used as a blueprint for the component development. Components are differentiated between *atomic components* and *composite components*. An atomic component is composed of a *process* (which is a set of *activities*), *data variables*, *methods*, and *ports*. A composite component describes a hierarchical relationship between components.

The interfaces between a component and other components in its environment are represented by ports. A port can specify either the services *requested* (called a *buy-side* port) or the services *provided* (called a *sell-side* port).

An *activity* in one component may exchange *messages* via a port with another component. An activity can access the data variables defined in the same atomic component as the activity itself and it can transform them. An activity is considered to be atomic, so execution could not be interrupted. There are two types of data variables, *persistent* and *volatile*. The persistent variables are for

the component itself, whereas the volatile variables are for an instance of the component. Each component has one process which describes a set of activities and determines the order of execution. So the process takes care of the *orchestration* (also called *workflow*) of the component. This process also describes the relation between activities and data variables. We assume that a component can be *instantiated* multiple times. Once a component has been *initialised*, the process and the variables belonging to it are initialised as well. The component afterwards can be instantiated to create new instances by a dedicated *enter* activity; each instance is identified by a unique instance *identity*. Note that we assume one process and possibly many instances. A process instance can access (volatile) data variables of its instance as well as the (persistent) data variables of the component. When an instance finishes by the *exit* activity, it is destroyed. On the other hand, a component only ends when the component itself is deactivated.

When a *message* is exchanged between instances, the instance identity should be either an explicit part of the message or can be derived from it. When a message arrives, there could be two possibilities. Firstly, if there is exactly one instance which can handle the request, then it is handled immediately or at a later stage. Secondly, if there are multiple instances which can handle the request, then it is delivered to one of them, i.e., non-deterministically.

Another issue is the *integration* of different components. There are two kinds of connections between two components: *horizontal* connections between two components at the same level of the hierarchy and *vertical* connections between a component and one of its sub components. Vertical integration means fusing ports of a subcomponent and a super component, while horizontal integration is achieved by fusing ports of two components at the same hierarchical level. Note that we only consider asynchronous communication.

3.2 Component modelling in coloured Petri nets

In this section, we introduce how to model components and their interactions in coloured Petri nets. We assume each component to be one subnet with a special structure, called *workflow net*. A workflow net here is a Petri net with one initial transition called *enter*, one final transition called *exit*, and any internal place or transition in the net is on a path from the initial to the final transition. More details are explained in Section 3.3.

Each *transition* in the workflow net of a component models an *activity* in the component. Transitions are triggered by receiving *tokens*. A token can be either a *message* or an internal control flow token. Each net starts with an *enter* transition and ends in an *exit* transition. The enter transition creates a new instance with a unique identity. The exit transition destroys an instance and does a clean-up job if necessary (we explain this in Section 3.3).

Each *place* can be a *data store* place, an *interface* place, or a *control flow* place. Firstly, an internal place can be a data store place which represents a *data variable*, its *colour set* is the *data type*, and its token stores the current value of the data variable. These places are connected to transitions with two arcs:

an input as well as an output arc. So the token is consumed and (re-)produced in the same transition, which mimics the update of the data variable. Secondly, a set of interface places can be a *port* for communication by message exchange with one another component. Thirdly, the remaining internal places are control flow places, they determine the order of execution of transitions. The *marking* of all places together models the *state* of a component.

Each instance of a component is also an instance of the workflow net of the component. An instance can be represented by one or more tokens in a net. For example, if there is a concurrent behaviour, then an instance is represented by multiple tokens. Every instance has a unique identity, so the tokens belonging to the same instance have the same identity. Only the tokens having the same identity (with the exception of message tokens) can be consumed and produced by the transitions that consume multiple tokens at the same time. An instance may have volatile data variables. Any volatile data variables are created (destroyed) by the enter (exit) transition and have the same identity as all other tokens of the instance. All instances can access the persistent data variables. Persistent data variables are created at the initiation of the component and are the only token in their place.

Since we only consider asynchronous communication, our workflow nets have input and output places forming ports for communication. These so-called *open Petri nets* [8] can be *glued* (or *composed*) by fusing input/output places with the same name. Composability must be assured before any integration of the nets. Two nets are considered to be composable if they do not share any internal places, input places and output places. Horizontal connection between two components at the same level is modelled by fusing (may be after renaming of places) input places of one component with output places having the same name of the other component. Vertical connection between a component and one of its sub components is modelled by fusing (may be after renaming of places) input places of one component with input places having the same name of its sub component as well as output places of the component with output places with the same name of the sub component. It is important that the composed components are able to terminate in a proper way (see e.g. [8]); we consider this problem in Section 3.3.

3.3 Workflow nets

A *workflow* net in this paper is a slight variation of the *standard workflow net* [5]. The main difference is that we fused the *initial* and *final* places into a place called *idle* and we allow more than one initial token. A workflow net (see Fig. 2) has *internal* places and *interface* places (internal places could be either control flow places or data store places). A workflow net has one *enter* transition and one *exit* transition, as well as one place called *idle* that is the only internal input place for the enter transition and the only internal output place for the exit transition. The place idle contains the number of free (idle) instances of the component. The enter (exit) transition may also have an input (output) place in the interface. Further each other (internal) place or transition is on a directed

path from enter to exit. The workflow net takes care of the orchestration of the component. We assume the workflow nets are *safe*, i.e., if started with only one token in *idle* in each reachable state there will be at most one token per place.

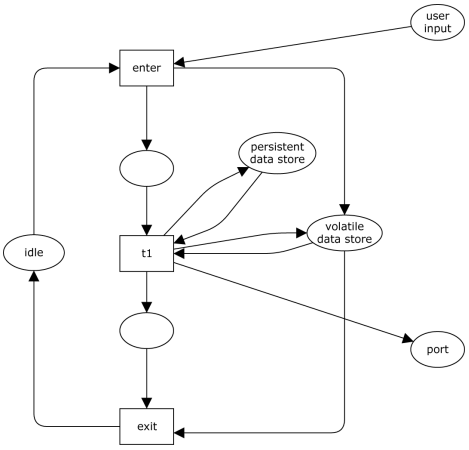


Fig. 2. A workflow net

In this paper, we restrict our workflow nets to be *well-structured* only. The class of well-structured workflow nets is the same as the class of models that can be expressed in BPEL (see [15]). Hence, on the one hand, it is sufficiently expressive. On the other hand, it enforces the designer to work in a structured way, i.e., by stepwise refinement principles. There are several (equivalent) ways to introduce a well-structured workflow net, we recommend [11], [15], and [24]. We restrict ourselves to refine transitions only. A single transition can be refined by one of the four rules: *sequence*, *iteration*, *parallel*, and *condition*. They are depicted in Fig. 3. We may apply the rules recursively, starting with a particular net called *(1)* displayed in Fig. 4. This net has three transitions(*enter*, *t1*, and *exit*), only *t1* can be refined. In this case, we refine *t1* with the *Parallel rule*, and then we get *(2)* in Fig. 4. Any net constructed in this way is called a well-structured net. Note that the dummy transitions are only used for control flow, and they are not refined. After generating a well-structured workflow net, data store places and interface places may be added. As in *(3)* in Fig. 4, two data store places, *persistent data store* and *volatile data store*, are added, *persistent data store* and *volatile data store* model a component variable and an instance variable, respectively. Also an output interface place, *port*, is added as well to model a port for sending messages.

In order to generate component models in a structured way, there are three steps involved. Firstly, describe the internal workflow of each component using well-structured workflow net. The workflow net is generated by means of stepwise refinement, applying the refinement rules to non-dummy transitions only.

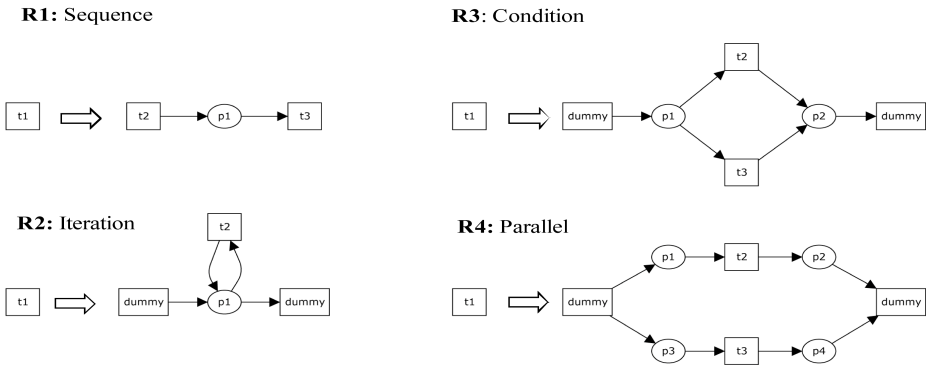


Fig. 3. Refinement rules

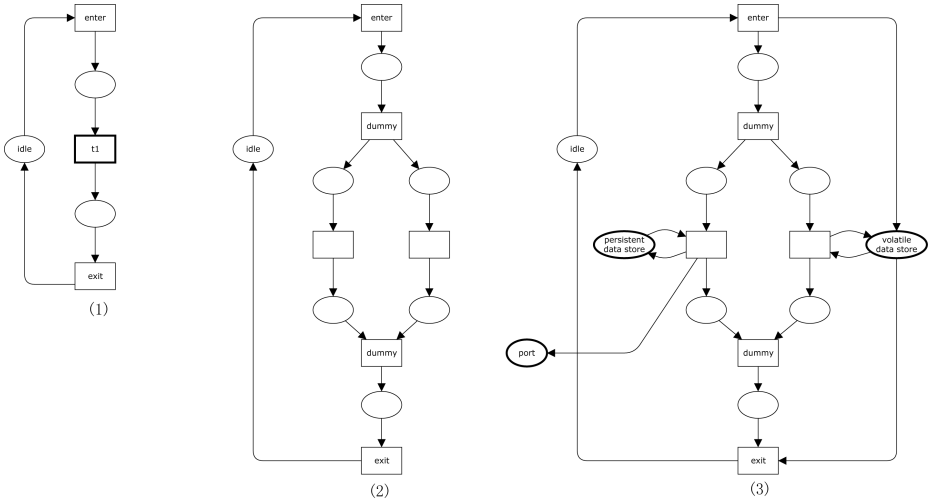


Fig. 4. Generation of a well-structured workflow net by refinement rules

Secondly, for each component, add data store places for data variables, one place for one variable, as well as a set of interface places for a port for communicating with another component. Thirdly, connect ports of different components for integration.

An important property of workflow nets is *proper termination*. This means that when starting in the initial state with only one token in the idle place, in each reachable state (marking) of the workflow net, it must be possible to reach the final state, having only one token in the idle place. Proper termination for the original notion of workflow nets is introduced in [6] and is part of the *soundness* property. For the situation where there are more instances, the notion of *generalised soundness* applies (see [16]), which states that if in the initial state there are k tokens, then it is always possible to end in a state with exactly k tokens in the final place, for each k. For well-structured nets it is proven in [15] that they have the generalised soundness property, so our components have the proper termination property and therefore we do not have to worry about garbage collection at the end of execution (garbage collection takes care of tokens in places other than the place *idle*). Note that the results on (generalised) soundness hold for classical Petri nets, so colour is discarded. Colouring could destroy the proper termination property, however (generalised) soundness can be considered as a general sanity check for workflow nets.

In Appendix A, we show how we could construct the Producer-Consumer example depicted in Fig. 1 using the refinement rules.

4 Transform Coloured Petri Net Models to Java Code

Based upon the modelling techniques introduced in Section 3, once a model is ready it should be transformed into code. In this section we will present a rule-based approach to transform such a net into Java code. We will illustrate our approach using the Producer-Consumer example in Appendix B. We give the transformation only for well-structured workflow nets. It is also possible to transform other workflow nets into Java (see Section 5), which is out of scope of this paper.

We define the model-to-code transformation in 6 steps.

- *Step 1*
  Each component at the highest level runs on a thread to enable concurrency between components. We create a class that implements the *Runnable* interface (or extends the *Thread* class) for each component at the highest level. Any components which are not contained in other components are at the highest level. Concurrency is also allowed at low level. Recall that the parallel workflow pattern is defined in Fig. 3. Each concurrent path in the parallel workflow pattern runs on a separate thread to enable concurrency between methods. The parallel workflow pattern is the only case that concurrency is allowed at low level (see *Step 5* for more details).

– *Step 2*
A subcomponent is converted into a nested class. We use class nesting to convert a hierarchical structure, where the super component is transformed into the enclosing class and the subcomponent is transformed into the nested class. In a hierarchical net, a transition (and its surrounding arcs) in the super net can be replaced with a subnet providing the same services, so a method in the enclosing class can be replaced with a nested class. An enclosing class and its nested class run on the same thread. Hence the workflow of a nested class is implemented in its constructor. When an object of the nested class is created, an instance of its workflow is started. Fig. 5 describes a hierarchical structure in Java. In this figure, the *SuperNet* class runs on a thread, its workflow starts in the thread's *run()* method. The *SuperNet* class has a nested class called *SubNet*, which is created in the *methodReplacedWithSubnet()* method of the *SuperNet* class and can replace this method. The enclosing class *SuperNet* and the nested class *SubNet* run on the same thread, the workflow of the *SubNet* class is implemented in its constructor.

```
public class SuperNet implements Runnable {
   ......
   public SuperNet() {
      ......
      new Thread(this).start();
   }
   /* workflow of SuperNet*/
   public void run() {
      ......
      methodReplacedWithSubnet();
      ......
   }
   ......
   public void methodReplacedWithSubnet() {
      SubNet subnet = this.new SubNet();
      ......
   }
   private class SubNet {
      ......
      public SubNet() {
        // workflow of SubNet
      }
      ......
   }
   ......
}
```

Fig. 5. Convert hierarchy to Java code

– *Step 3*
Data store places are converted into fields of a class. Data store places holding persistent data variables are converted into static fields (prefixed with the keyword *static*). Data store places holding volatile data variables are

converted into instance fields. The colour set of each place is converted into the data type of each field. The label of each place is converted into the name of each field. For each instance of any component, we give it a unique identity. This identity is converted into an instance field called *identity*.

– *Step 4*
Each transition, unless it is a control flow dummy, is converted into one method. Control flow dummies do not correspond to any methods. The *enter* transition is converted into a constructor. All instance fields are initialised in a constructor (static fields are initialised when they are converted in step 3). In a constructor, besides initialisation of instance fields, the thread on which the class code will runn is also started by the statement *new Thread(this).start()*. As introduced in Section 2.2, the code for a thread to run is placed in the *run()* method, a *run()* method is provided for each class running on a thread (Note that this *run()* method is not modelled in a net, as it does not belong to a class, it belongs to the environment running the class instead). The workflow net of a component is explicitly implemented in the *run()* method (see Fig. 6). The advantage of this explicit implementation of a component workflow net is that the control flow is clear, so it is easy to detect any structural problems of the component. Therefore, when an instance is created, its constructor is invoked, the constructor starts the thread by calling the *start()* method, which subsequently invokes the *run()* method, consequently an instance of the component workflow is started. The *exit* transition, the last activity of a component in its lifecycle, does the general purpose clean-up. However, the orchestration process is sound by construction, so this is not necessary.

```
public class MyComponent implements Runnable
{
   ......
   public MyComponent() // constructor
   {
      ......
      new Thread(this).start(); // start a thread, run() method is called
      ......
   }
   ......
   public void run()
   {
      // workflow is specified in the run() method
   }
   ......
}
```

Fig. 6. Run a component on a thread

– *Step 5*
The four basic patterns of the workflow (*sequence*, *condition*, *iteration*, and *parallel* depicted in Fig. 3) are translated into standard programming constructs, also called *patterns*. We employ the same stepwise refinement approach that we introduced in Section 3.3. We always replace a method by a piece of code realising one of the patterns. A *sequence* pattern is converted into two methods separated and followed by a semicolon *;*. They are executed sequentially. A *condition* pattern is converted into *if...else...* statements. If the condition is true, then the control flow executes one method, if false the method of the alternative path is executed. If there are two or more alternative paths in a model, *if...else...* statements are replaced with *if...elseif...else...* statements or *switch* statements. An *iteration* pattern is converted into a *while* statement. The control flow loops through the method, while a specified condition is true. A *parallel* pattern is converted into *multithreading*. A thread is created for each concurrent path in a model. Once a thread has been started, the *isAlive()* method (from Java API) is employed to test whether the thread is still alive. If all threads are dead, the concurrent paths join again. Fig. 7 presents the code for different workflow patterns.

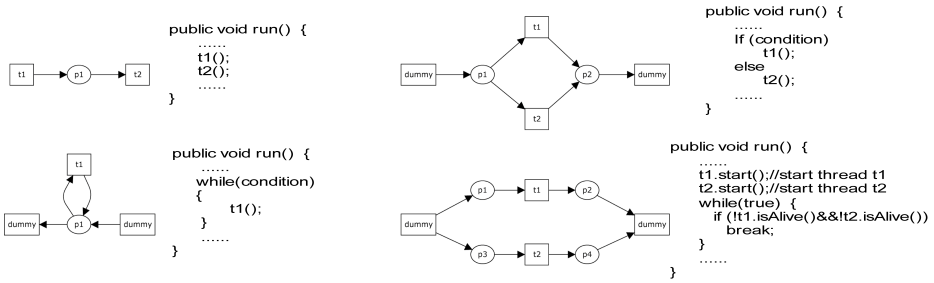


Fig. 7. Code for workflow patterns

– *Step 6*
Each interface place is converted into one instance of a *BlockingQueue*. In the model of a system, nets communicate asynchronously by means of the interface places in the ports. In Java we model this by means of *glue code*, i.e. we use the Java interface *BlockingQueue* (with its implementation classes provided by Java) and we create for each interface place (of all components together) one instance of a BlockingQueue as a static field. In each class modelling a component we refer to these instances by the *take()* and *put()* methods of the BlockingQueue.

Table 1 summarises the mapping of concepts among SOA, CPN, and Java. In Appendix B we show how our Producer-Consumer example is transformed into Java code.

Service Oriented Architecture	**Coloured Petri Nets**	**Java**
Atomic component	Subnet	Nested class
Composite component	Composite coloured Petri net	Enclosing class
Orchestration process	Workflow net	Constructor/run() method
Hierarchy	Super nets and sub nets	Class nesting
Activity	Transition (with guard)	Method (with *ifelse* statement)
Port	Set of interface places	Instance of the *BlockingQueue*
Asynchronous communication	Place fusion	Put() or take() methods of the *BlockingQueue*
Exception handling	Transition	Exception handler
Data variable	Data store place	Data variable
Component variable	Without instance identity	Static element
Instance variable	With instance identity	Instance element

Table 1. Concepts mapping among SOA, CPN, and Java

5 Related work

In [22] and [23], three approaches of generating code from a Petri net have been discussed. One approach makes use of the structure of a net. This is what we did: we used well structured subnets and we transformed them into code fragments. Another approach employs the reachability graph of a Petri net and runs this as a state machine. This is only possible for any net with a finite reachability graph. The last approach called simulation uses a Petri net simulator to determine in each step which transition is enabled. The first approach, which we adopted, gives good readable Java code that is understandable without knowing the transformation made. While the second and third approaches are inefficient and require a kind of simulation engine in the Java code, which typically depends on modelling approach. Still [23] advocates the simulation approach because it is so flexible that it can handle all Petri nets, without structural restrictions. We are in favour of the structural approach since it is a good practice to model in a structured way and the SOA approach implies that the orchestration of components should be done according to this approach (if the BPEL language is used then automatically the same structuring is obtained). Another difference with [23] is that we do not use Petri nets to model atomic methods, while [23] uses Petri nets for that purpose specifically.

In [13] and [14], an approach similar to ours is presented. They let components communicate by connecting input places of one component to output transitions of another, but we fuse interface places of the two components. They use Petri nets at the interface level only and do not consider the orchestration of components.

In Chapter 21 of [12] an approach for code generation is also presented. It mainly focuses on the identification of components that have a state machine character. To this end they use place invariant techniques. The difference with our approach is that we start with modelling components, so we do not have

to discover them. Their approach is not intended to generate efficient code but to generate a prototype of the intended system for analysis purpose and not for execution.

6 Conclusion

In this paper we introduced an approach to model systems in coloured Petri nets using the SOA style, and we presented a way to transform these Petri net models into Java code. In modelling, we require the component models are well-structured workflow nets. To generate such workflow nets, we introduced a set of rules for the stepwise refinement of transitions by four basic workflow patterns. For representing hierarchy we used nested classes and for asynchronous communication between components we used a queuing mechanism offered by Java. We illustrated the approach by a variant of the well-known Producer-Consumer example. As future work we plan to make the transformation from CPN to Java as much as possible automatically.

References

1. CPN Tool Version 2.2.0, 2006. URL: http://wiki.daimi.au.dk/cpntools/cpntools.wiki [Accessed on 25/08/2009].
2. SOAP Version 1.2, 2007. URL: http://www.w3.org/TR/soap12-part1/ [Accessed on 25/08/2009].
3. Web Service Business Process Execution Language Version 2.0, 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf [Accessed on 25/08/2009].
4. Web Services Description Language (WSDL) Version 2.0, 2007. URL: http://www.w3.org/TR/wsdl20/ [Accessd on 25/08/2009].
5. W. M. P. van der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, Toulouse, France, 1997.
6. W. M. P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 2001.
7. W. M. P. van der Aalst, M. Beisiegel, K. M. van Hee, D. Konig, and C. Stahl. An soa-based architecture framework. *International Journal of Business Process Integration and Management*, 2(2):91–101, 2007.
8. W. M. P. van der Aalst, K. M. van Hee, P. Massuthe, N. Sidorova, and J. M. van der Werf. Compositional service tree. In *Proceedings of 30th International Conference, PETRI NET 2009*, Paris, France, 2009.
9. T. Basten and W. M. P. van der Aalst. Inheritance of dynamic behavior: Development of a groupware editor. In *Concurrent Object-Oriented Programming and Petri Nets*, pages 391–405, 2001.
10. M. Bell. *Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*. John Wiley & Sons, 2008.
11. P. Chrzastowski-Wachtel, B. Benatallah, R. Hamadi, M. ODell, and A. Susanto. A top-down petri net-based approach for dynamic workflow modeling. In *Business Process Management, Lecture Notes in Computer Science*, volume 2678, pages 336–353. Springer-Verlag, 2003.

12. C. Girault and R. Valk, editors. *Petri Nets for System Engineering: A Guide to Modelling, Verification, and Applications.* Springer-Verlag, 2002.
13. N. Hagge and B. Wagner. Mapping reusable control components to java language constructs. In *Proceedings of 2nd IEEE International Conference on Industrial Informatics (INDIN)*, Berlin, Germany, 2004.
14. N. Hagge and B. Wagner. Java code patterns for petri net based behavioral models. In *Proceedings of 3rd IEEE International Conference on Industrial Informatics (INDIN)*, Perth, Australia, 2005.
15. K. M. van Hee, J. Hidders, G. Houben, J. Paredaens, and P. Thiran. On the relationship between workflow models and document types. *Information Systems*, 2008.
16. K. M. van Hee, N. Sidorova, and M. Voorhoeve. Generalised soundness of workflow nets is decidable. In *Proceedings of the 25th International Conference, ICATPN'04*, Bologna, Italy, 2004.
17. C. Hortsmann and G. Cornell. *Core Java Volume I Fundamentals.* Prentice Hall, 8 edition, 2007.
18. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use (Basic Concepts, Monographs in Theoretical Computer Science)*, volume 1. Springer-Verlag, 1997.
19. K. Jensen. An introduction to the practical use of coloured petri nets. In *Lectures on Petri Nets II: Applications, Lecture Notes in Computer Science*, volume 1492, pages 237–292. Springer-Verlag, 1998.
20. L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, pages 98–132, 1998.
21. G. Michaelson. *Elementary Standard ML.* UCL Press, 1995.
22. S. Philippi. Seamless object-oriented software development on a formal base. In *DAIMI PB: Workshop Proceedings Software Engineering and Petri Nets*, Aarhus, Denmark, 2000.
23. S. Philippi. Automatic code generation from high-level petri-nets for model driven systems engineering. *Journal of Systems and Software*, 79:1444–1455, 2006.
24. H. A. Reijers. Design and control of workflow processes: Business process management for the service industry. In *Lecture Notes in Computer Science*, volume 2617. Springer-Verlag, 2003.
25. R. Simmons. *Hardcore Java.* O'Reilly, 2004.

Appendix

A Example of component modelling

We reconstruct the Producer model and the Consumer model (in Fig. 1) using stepwise refinement with the four rules. Fig. A1 depicts the stepwise refinement for generating the Producer component model.

We start with the workflow net as described in net (1). Transition *t1* is refined by using the *Sequence Rule*, then we get *produce*, *t2*, *send*, *add* transitions in net (2). Subsequently, transition *t2* is refined by using *Iteration Rule*, then we get *produce* transition and two dummy transitions (which are only for the control flow) in net (3). Then we reduce the net by removing the dummy transitions, and

fusing the input/output places of the dummy transitions. Finally, we add a data store place *total* for a persistent data variable, and a data store place *userInput* for a volatile data variable, an input interface place *user input*. In addition, we add an input interface place *buffer free space* and an output interface place *item in buffer* for a port, then we get net (4). The model generated is a well-structured workflow net starting with the *enter* transition and ending in the *exit* transition. Every transition models an atomic activity. The place *total* holds persistent data, the place *userInput* holds volatile data. The places *buffer free space* and *item in buffer* model a buffer for asynchronous communication.

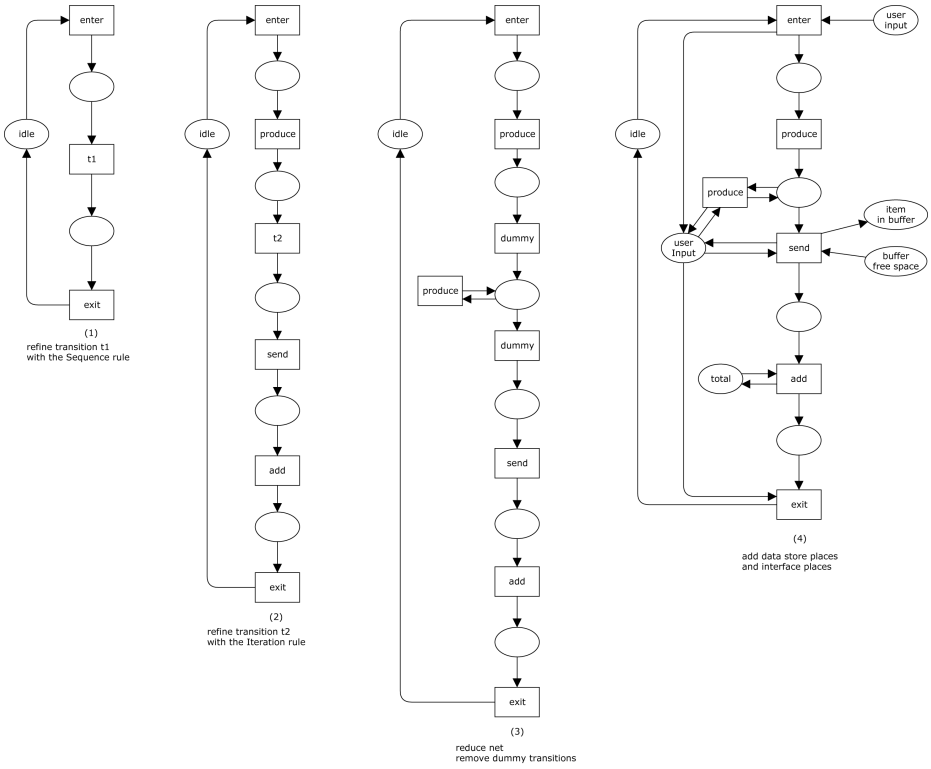


Fig. A1. Stepwise refinement for Producer component model

Fig. A2 illustrates how to generate the workflow net for the Consumer. In order to glue components, the *item in buffer* place in the Producer model is fused with the *item in buffer* place in the Consumer model, and the two *buffer free space* places in both component models are fused as well.

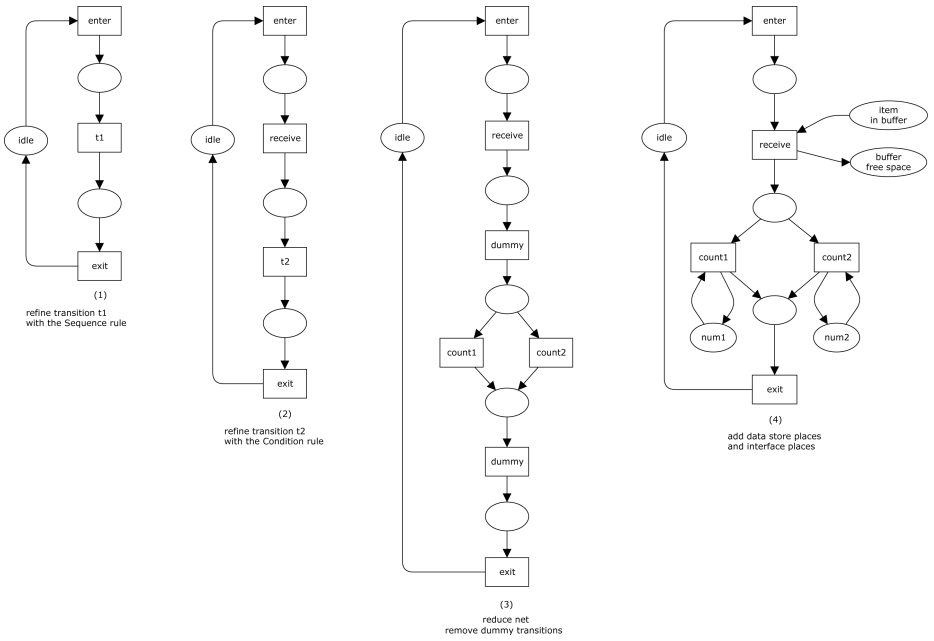


Fig. A2. Stepwise refinement for Consumer component model

B Example of code generation

In this section, we use our Producer-Consumer example to illustrate how to transform a component model to Java code. Fig. B1 and Fig. B2 present the code for the Producer model and the code for the Consumer model, respectively. The code transformation for the Producer component is discussed in detail.

We generate code according to the 6 steps defined in Section 4. Step 1, we realise a class called *Producer* that implements the *Runnable* interface to create a thread. The Producer will run on this thread. We skip Step 2, as the Producer itself is an atomic component and does not have any subcomponents. Step 3, we convert the data store places into code. We declare a static field *total* and an instance field *userInput*. The data type of *total* is *long* and the data type of *userInput* is *int* in Java. We also declare an instance field *identity* with the data type of *int*, as an *identity* is required for each object. Step 4, we convert all non-dummy transitions into code. The *enter* transition is converted into a constructor. In this constructor we initialise all the instance fields declared in Step 3, and start the thread by calling the *start()* method using the following statement *new Thread(this).start()*. As the *start()* method invokes the *run()* method (see Section 2.2), we provide a *run()* method for the Producer. Subsequently, we transform the *produce* transition into the method *produce*, the *send* transition into the method *send*, and the *add* transition into the method *add*. Because the orchestration process is sound by construction, so clean-up is not

necessary, and the transition *exit* is ignored. Step 5, in the *run()* method, we realise the orchestration process, and transform the basic workflow patterns into code based on Fig. 7. For example, we can identify an iteration pattern in the Producer. We use a *while* block to implement the iteration pattern. In this block, the *produce()* method is executed repeatedly untill the loop condition becomes false. Step 6, we use the *BlockingQueue* to enable asynchronous communication. As a pair of the *item in buffer* place and the *buffer free space* place models a buffer, we transform such a pair into an object of one of the BlockingQueue implementation classes provided by Java (Java implements five BlockingQueue classes, namely *ArrayBlockingQueue*, *DelayQueue*, *LinkedBlockingQueue*, *PriorityBlockingQueue*, *SynchronousQueue*, respectively. See [17] for more details). We declare an object of a BlockingQueue implementation class (which is *ArrayBlockingQueue*) called *buffer* as a static field in the Producer. Then we use its *put()* method to put a data into the buffer.

```
import java.util.Random;
import java.io.*;
import java.util.concurrent.*;

class Producer implements Runnable
{
   private static BlockingQueue buffer; //buffer modelled by the places buffer element and free buffer space
   private static long total; //modelled by the total place
   private int userInput; //modelled by the userInput place
   private int identity; //required for each object

   /*
    * The constructor is modelled by the enter transition
    */
   public Producer(BlockingQueue buffer, int userInput, int identity)
   {
            this.identity = identity;
            this.buffer = buffer;
            this.userInput = userInput;
            new Thread(this).start(); //start the thread invoking run()
   }

   /*
    * The run() method realises the workflow of the component
    */
   public void run()
   {
            try
            {
                     int value;
                     value = produce();
                     //keep producing a random value till the condition is met
                     while(value >= userInput)
                     {
                              value = produce();
                     }
                     send(value);
                     add(value);
            }
            catch (InterruptedException ex)
            {
                     System.out.println(" INTERRUPTED");
            }
   }

   /*
    * The produce() method is modelled by the produce transition
    */
   public int produce()
   {
            Random rand = new Random();
            return(rand.nextInt(100));
   }

   /*
    * The send() method is modelled by the send transition
    */
   public void send(int value) throws InterruptedException
   {
            buffer.put(new Value(value));
   }

   /*
    * The add() method is modelled by the add transition
    */
   public void add(int value)
   {
            total = total + value;
   }
}
```

Fig. B1. Java code for Producer

```
import java.util.concurrent.BlockingQueue;

class Consumer implements Runnable
{
   private static BlockingQueue buffer; //buffer modelled by the places buffer element and free buffer space
   private static int num1; //modelled by the num1 place
   private static int num2; //modelled by the num2 place
   private int identity; //required for each object

   /*
    * The constructor is modelled by the enter transition
    */
   Consumer(BlockingQueue buffer, int identity)
   {
            this.identity=identity;
            this.buffer = buffer;
            new Thread(this).start();//start the thread invoking run()
   }

   /*
    * The run() method realises the workflow of the component
    */
   public void run()
   {
            try
            {
                     int value = receive();
                     if (value < 25)
                              count1();
                     else
                              count2();
            }
            catch (InterruptedException ex)
            {
                     System.out.println("CONSUMER INTERRUPTED");
            }
   }

   /*
    * The count1() method is modelled by the count1 transition
    */
   public void count1()
   {
            num1++;
   }

   /*
    * The count2() method is modelled by the count2 transition
    */
   public void count2()
   {
            num2++;
   }

   /*
    * The receive() method is modelled by the receive transition
    */
   public int receive() throws InterruptedException
   {
            return ((Value)buffer.take()).getValue();
   }
}
```

Fig. B2. Java code for Consumer

-ACTAS- ADAPTIVE COMPOSITION AND TRADING BASED ON AGENTS

Reinhold Kloos[1,2], Rainer Unland[2], and Cherif Branki[3]

[1] Department of Computing, SOI, City University London
Northampton Square, London, EC1V, United Kingdom
r.kloos@soi.city.ac.uk
[2] DAWIS, Institute for Computer Science and Business Information Systems
University of Essen-Duisburg, Schützenbahn 70, 45117 Essen, Germany
(Kloos, UnlandR)@cs.uni-due.de
[3] School of Computing, University of Paisley, Paisley PA1 2BE, Scotland
cherif.branki@uws.ac.uk

Abstract. Challenges for approaches, dealing with services, Service Composition, and Service Coordination, are the complex aspects of a (composite) service and its domain specific constrains. Ideas of services and policies of different domains are often incompatible. Service Grounding and Service Deployment as well as the observation of non-functional service characteristics are additional reasons for the complexity of service composition in practice. For a general understanding, the different aspects of a service are discussed. In order to take advantage of well-established methods, the paper proposes a framework based on agents, called ACTAS, for the pre-selection of Service Providers on the basis of principally compatible and available services. Compatibility and services are described with semantic characteristics in a declarative way. The adaptability of the composition is created through the behavioural semantic of the Service Properties defined in the context of the characteristics. The behavioural semantic allows the use of well-established methods for dealing with relevant Service Properties. We suggest ontological repositories, containing service description components instead of complete Service Descriptions.

1 Introduction

Daily, we discover, compose, and consume services performed by agents with different backgrounds and skills. We do not need a complete knowledge about the services. We rely and trust on the established interfaces, policies, and methods. The agents often do negotiation and planning of services on our behalf. However, the automatic Service Discovery and Service Composition of electronic services (e-services) originated in different domains can be a challenge.

Languages like Interface Description Language (IDL) for CORBA or Web Service Description Language (WSDL) for Web Services, which are used for the description of interfaces of Remote Procedure Calls (RPC) or Message Exchange, are examples for functional descriptions of services. The research of Semantic

Web Services introduces semantic descriptions for an improved compatibility description. One example are IOPE-capability descriptions of OWL-S [1]: (Input, Output, Precondition, and Effect), which allow a workflow like composition. Other approaches determine compatibility of services through testing if their communication protocols can be coordinated [2]. Service composition approaches can be categorized in AI planning based [3–5], workflow based [6, 7], ontology based [6,8], multi-agent based [9,10], or based on miscellaneous concepts [11,12].

Approaches, which discover and compose services of different domains, should take advantage of well-established methods. Our approach is a Service Discovery framework built on a multi-agent system (MAS) using existing solutions offered through a Facility Agent (FA) by the Service Provider. We called the approach ACTAS - Adaptive Composition and Trading based on Agents. ACTAS uses simple semantic characteristics for the pre-selection of available and principally compatible services. Two services are principally compatible when they hold the same semantic characteristics. ACTAS distinguishes between the Service Model (SModel) and the Composition Model (CModel). The SModel is for the description of the service and its Service Modes. The CModel defines the principal compatibility and the solution of constraints for the composition. These constraints are partly described with methods, which are defined together with Service Properties in the context of a semantic characteristic. The methods create semantic behaviours for Service Properties, their merge and exchange of values. The introduced semantic behaviour enables ACTAS to be adaptable. For instance, methods for dealing with IOPE-capability descriptions or WSDL descriptions could be included in the Service Descriptions through Service Properties classes holding such descriptions.

ACTAS distinguishes between Service Templates (ST) and Service Offers (SO) managed by the FA. The later published through a Service Offer Export Record (SOER) only exists, when the service is available. A SO is based on a ST and enumerates the available Service Modes and their current values of the Service Properties. Thus, depending on the (resource) management abilities of the FAs, the SModel of ACTAS supports information about the availability of the service (a non-functional Service Property). The Composition-Model (CModel) uses the Service Offers (SO) for solving the constraints. The SO contains objects with the mentioned methods. Trading Agents (TrA) and Composition Agents (CoA), which could have their own policies, allow their application.

The wrapping of several Service Modes in one Service Description (ST, SO), the direct support of availability aspects, and the adaptability through the semantic behaviour in the Service Description is unique to our best of knowledge. ACTAS suggests to manage ontological repositories for the components of the SModel ((semantic) characteristics and classes of methods) instead of having (only) repositories for complete Service Descriptions. In this way, Service Descriptions become comparable and adaptable through standardized entities. In many approaches based on MAS, the services themselves are represented through software agents, which try to find direct compositions of the services through negotiation [9, 10]. In [13, 14], the authors follow a similar concept. They show

that service-oriented computing can benefit from MAS technology by adopting the coordination mechanisms, interaction protocols, and decision making tools designed for MAS.

The paper discusses several aspects of services, which lead to the unavoidable complexity of Service Descriptions. The introduction of our approach follows. The application of the introduced models and future research concludes the paper.

2 Service

Definition 1 (Service).
*A **Service** consists of actions performed by an entity on behalf of another. It is an asset with an inherent value. The consumption of a service involves the transfer of value and (mostly) the generation of cost with the consequent need of its settlement. A Service has functional and non-functional properties. A Service (**Composite Service**) can be composed of several **Component Services**.*

The existence of a common service definition and unique characteristics could simplify the design of general service composition environments. Therefore, Justin O'Sullivan, David Edmondy and Arthur H. M. ter Hofstede [15] called for an accurate service description, in order to reduce the gap between manual and electronic services. Definition 1 gives a general definition of a service. For a clear distinction of a service from the production of a commodity or good, many researchers name the following characteristics of Services as unique: (a) intangibility, (b) heterogeneity, (c) inseparability of production and consumption, (d) perishability (cannot be inventoried), and (e) the ownership is unchangeable. However, Prof. Lovelock [16] shows that these characteristics do not hold in all cases. For example the production and consumption can be separated in Example 2.

Different research domains of Computer Science developed their own ideas and policies for services. A distinction between Business Services and Technical Services became apparent. Software Engineering sees services as a new software paradigm and distribution model. The electronic processing of services enables on one hand the processing and collection of information about services, and on the other hand the support of communication, cooperation, and composition of services. The standardization of Web Services made the construction of Distributed Information Systems (DIS) more feasible. An intensive research in the area of Semantic Web Services (SWS) (OWL-S, WSMO) [17–19] and dynamic Service Oriented Architectures (SOA) [20], often based on multi-agent systems, show the urge for an improvement of description and handling of e-services with extended logic (similar Enterprise Application Integration (EAI) in the context of DIS). In the following sections we have a closer look at aspects of a service, which makes their handling complex.

2.1 1^{st} aspect of service: Service Description depend on view

The description of a service depends on the view on a service. On one hand, the view changes with the application domain. On the other hand, a Service Provider has another view on a service than a Service Requester.

A service in a technical domain (Technical Service) can be distinguished from a service in a mainly business oriented domain (Business Service). Additionally, various definitions of the term service and specific constrains exist in the different domains of business (cf. [16, 21–24]).

Example 1 shows that a (composite) Business Service can consist of several dynamically selected component services. The component services have directed server-client relationships among each other. The server or client role of a component service is not fixed and depends on the relationship.

Example 2 introduces a simple Technical Service for audio communication using a gateway service. From one point of view, a Technical Service can be treated as a Business Service, since it has to be paid for the use of a telephone. From another point of view, the Technical Service consists also of several component services for the realisation of the technical function. The relationships between the component services of a Technical Service are non-directed, when the component services have a peer-to-peer like relationship, which is for instance the case for the realisation of the H.323 communication between the service of the IP-phone and the service of the gateway in Example 2.

In order to reflect the different views of Service Provider and Service Requester, Business Services distinguish between Business to Customer (B2C) and Business to Business (B2B) relationships. A similar distinction is made for Technical Services since the technical side of a Technical Service is transparent for the customer. The Service Provider (in Example 2 the telecommunication provider), has the responsibility for the acceptance, transmission and delivery of the information. The example of the Technical Service also shows that a composite service might have several Service Clients in the service consumption phase. The main research in the context of (Semantic) Web Services is in the area of Business Services. An approach for using Web Services in the area of Technical Services is UWS [17].

*Example 1 (**Booking of a travel**).* The Composite Service "Booking a travel" Fig. 1 consists of different Component Services: Inquiry, Booking (When inquiry was successful), Billing (the Travel Agency is now the client), paying (The Bank is a completely new Service Provider).

*Example 2 (**Telecommunication with gateway**).* Fig. 2 illustrates an example of a Technical Service including the service of a gateway or gatekeeper for the conversion of signals and protocols with different telecommunication standards. The standardized telecommunication protocol H.323 [25] allows the voice over IP communication. The composed Technical Service consists of the Component Services of a mobile, gateway, and IP-phone. The use of the gateway is transparent for the two Service Clients (named A and B in the figure). It is likely that

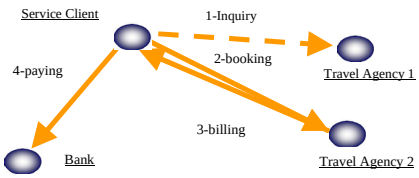


Fig. 1. - simple Business Service

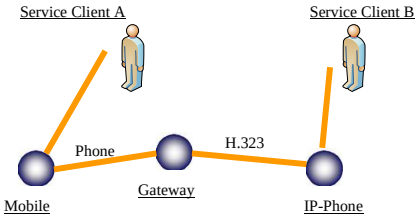


Fig. 2. - simple Technical Service

only one Service Requester (A, B, or an external agent) requested the service. A planning program for the scheduling of conversations could be such an external agent.

2.2 2^{nd} aspect of service: An entity for Software Engineering

Software Engineering has its own views on services. It looks at Software Services as a distribution model [26] and as a software paradigm [27]. Due to the loose coupling, Software Services as a software paradigm can be easier reused than objects or components, although their granularity is rather coarse. As a distribution model, the applying software can subscribe to the whole functionality of a software service, or it restricts its subscription to parts of the functionality. The distribution model leads to Service Oriented Architecture (SOA). SOA provides a standard programming model that allows software components, residing on any network, to be published, discovered, and invoked by each other as services. Therefore, Service Traders or Service Registries provide a vital aspect of a SOA. An intensive research for intelligent service discovery frameworks took place from the beginning of the nineties [28–38]. There is an ongoing research interest in Service Trading with agents [39, 40].

Software Services as e-services need an agent for their realisation and consumption. Even hardware and manual services are represented through an agent, in order to enable their planning through e-services. The agent sends and receives messages, while the Service Description (e.g. WSDL in case of a Web Service) is the resource. The realising agent can be an autonomous software agent, with its features of autonomy, re-activeness, pro-activeness, and communicativeness. In this way, (group) policies, planning, negotiating, and learning can be integrated in the distribution model of Software Services.

With the standardization of Web Services, Distributed Information Systems (DIS), which traditionally dealt with e-business services, became more feasible (cf. [41]). A main step in the development of DIS was the development of the vision of Enterprise Application Integration (EAI). In EAI, the middleware approach was extended with business-logic (e.g. workflow) and basic interoperability support (e.g. message brokering), in order to allow the integration of various applications. A result was the development of Application Servers and

SOAs with integrated layers for business logic. With a similar motivation, the research in the area of Semantic Web Services allowed an improved support for Business Services through the combination of semantic descriptions with methods of workflow management and AI [3, 6, 7].

2.3 3^{rd} aspect of a service: Service Life Cycle

On one hand, the service Life Cycle can be described in several processes from the perspective of the Service Provider (Service Definition, Property Provision, Service Delivery) as well as from the perspective of the Service Requester (Provider Discovery, Property Discovery, Service Call). The Property Discovery and Property Provision are negotiating processes for the refinement of the Service Properties. Service Call and Service Delivery are corresponding processes for the Service Consumption.

On the other hand, the Life Cycle can also be described in six phases: Phase 1 - Service Management, Phase 2 - Service Trading and Service Planning, Phase 3 - Service Discovery, Phase 4 - Checking of Service Properties and Constraints, Phase 5 - Service Grounding, and Phase 6 - Service Consumption and Feedback.

In Phase 1 (Service Management) the process of Service Definition is done. The SOA enables the publishing, planning, and trading with the Service Definitions in Phase 2. Service Trading and Service Planning are in principal independent from a concrete Service Request. The Service Request is the starting point of Phase 3. When no fitting Service Provider is known, the Service Requester will use the SOA for the Provider Discovery process, in order to find a Service Provider offering a service, which matches the Service Request. The phases 4 and 5 are necessary for the refinement of Service Properties, when the Service Requester does not have enough information for the Service Call process, in order to start the Service Consumption phase (phase 6).

Phase 4 is dealing with semantic and functional constraints. Service Grounding (phase 5) is solving the system specific constraints for the Service Delivery like e.g. resource management. Due to the inherent complexity of a service (5^{th} aspect of a service) the Service Grounding can involve the coordination and scheduling of (new) Component Services. The phases 3 to 5 must be repeated, if (component) services do not fulfil some constraints turning up during the refinement of the Service Properties. A feedback for learning and keeping up the Service Quality might be provided at the end of phase 6.

The service description and the constraints become more detailed and more system/domain specific with the progress of the mostly domain specific refinement process. Therefore, the exchange between service descriptions and concepts gets more complicated in the later phases of the Life Cycle.

2.4 4^{th} aspect: Non-Functional Properties

Non-functional properties are considered [15] to be constraints over the functionality of the service. Non-functional properties like availability of a service have to be considered, in order to ensure that the component services are still callable

in phase 6. Temporal and spatial representation, channels (usually a channel describes the way how a service is delivered), trust, and security are non-functional properties, which are important for serveral phases of the Life Cycle of a service. For instance in Example 2, it would be unacceptable for Service Discovery, if the IP-phone was not accessible for spatial or security reasons. The dealing with non-functional properties like Service Quality, Settlement, or Warranty can involve new Component Services. Therefore, the support of non-functional properties increases the inherent complexity of a service. The observation of Service Quality may lead to a stop of the Service Delivery (phase 6), when the detected values fall under a given threshold. A possible reaction to an interruption is the Service Substitution. In the worst case of substitution, the phases 3 to 6 have to be repeated, in order to discover and deliver a new (component) service. The non-functional property availability describes a domain independent concept for every service.

2.5 5^{th} aspect: The inherent complexity of a service

The inherent complexity of a service is tackled through Service Composition (phase 3, phase 4, and phase 5) and Service Coordination (phase 5), i.e. that a service can consist of several component services, which have to coordinate their message exchange in order to achieve the goal state of the service.

A choreography, which is also called a coordination protocol [41,42] is a model of the global sequence of operations, states, and conditions that control the interactions involved in the participating services. The interaction prescribed by a choreography results in the completion of some useful common business goal. Example 1 shows a simple choreography. A choreography may be described through a choreography description language (e.g. WS-CDL [43,44]). A choreography description language permits the description of how Web services can be composed, how service roles and associations in Web services can be established, and how the state, if any, of composed services is to be managed.

A choreography can be distinguished from an orchestration, which also takes place in phase 5. An orchestration describes from the local perspective of the calling service how different services are composed into a coherent whole. The orchestration specifies the order, in which services are invoked, and the conditions under which a certain service may or may not be invoked; in particular, it defines the sequence and conditions in which one Web service invokes other Web services in order to realize some useful local business goal. The Business Process Modelling Language (for Web Services) (BPEL, BPEL4WS) [45–47] is an example for an orchestration language.

The Service Compositions of phase 5 (choreography and orchestration) is often transparent for the Service Client/Requester due to e.g. company policies or security reasons. For the design of generic domain independent concepts, it is an advantage to support the description of compatibility in the earlier phase 3 of the Life Cycle.

3 ACTAS Approach

The discussion of the various aspects of a service showed that the lack of general descriptive service information is a result of several factors:

1. The heterogeneous nature of services in different domains
2. The different views on a service
3. The lack of generic domain independent concepts for describing services
4. The inherent complexity of services
5. The system-specific challenges of Service Grounding
6. Non-functional service properties including Service Quality and trust
7. Intentional behaviour on behalf of some Service Providers to limit a Service Requestor's ability to compare services due to company policy, patent-protection, or security

The weight of these factors increases in the later phases of the Life Cycle of a (composite) service. The reflection also showed that the goal of ongoing research is the improvement of Service Discovery and Service Composition with methods and descriptions beyond a purely functional one. Possibilities for dealing with non-functional properties, learning of preferences, and planning of services open up. However, the complexity of the description is also increasing and generates demands for mediation. In this sense, the advantage of the standardisation of Web Services for DIS is declining. On the other hand, perfect, domain-specific solutions became possible keeping up the dream of SOA (cloud computing, Web3.0, ITIL).

Alternatively, the paper proposes a framework environment called ACTAS (Adaptive Composition and Trading based on Agents), which allows a preselection of available, principally compatible services of different domains through semantic characteristics. Reasoning for solving of constraints gains new information about the Composite Service. The next sections introduce ACTAS.

3.1 The Data Models of ACTAS

Definition 2 (Components of SModel).

1. *Service Template, (ST :== (FA-ID, GCh-set, SM-set))*
2. *Service Offer Export Record, (SOER :== (FA-ID, ST-ID, Co-set))*
3. *Service Offer, (SO), like ST with objects*
4. *Service Mode, (SM :== (GCh-set, SP-set))*
5. *Service Port, (SP :== (CCh-set, Option-slots))*
6. *General Characteristic, (GCh :== (Name, GCh-Property-set, Co-set)) (GCh-Property :== (Property-Name, Property-Class))*
7. *Compatibility Characteristic, (CCh :== (Name, CCh-Property-set, Co-set)) (CCh-Property :== (Property-Name, Property-Class, Property-Merge-Class))*
8. *Request Characteristic (RCh) is a Compatibility Characteristic describing a compatibility from the perspective of a Service Requester*

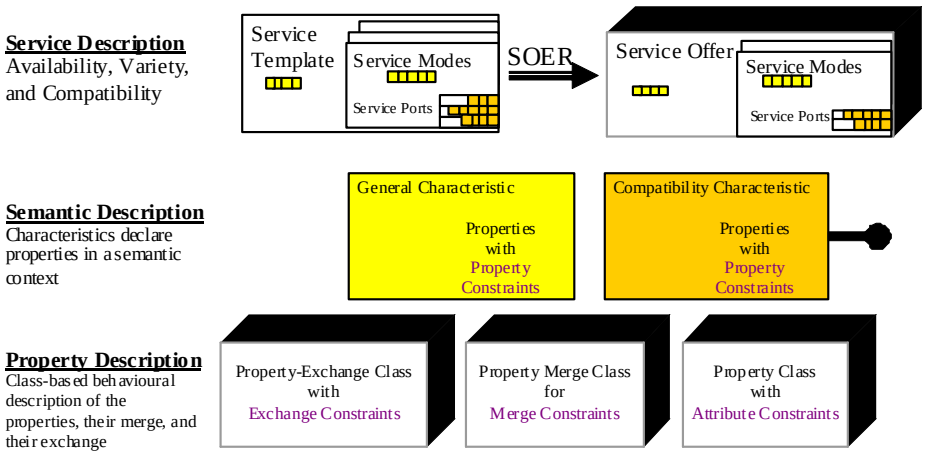


Fig. 3. - ACTAS Data Components for Service Description (SModel)

In Fig. 3 and in Definition 2, the components of the Service-Model (SModel) of ACTAS are listed. ACTAS does not publish complete Service Descriptions in a static repository , but (certified) components for the Service Definition. Semantic Characteristics are the central component. Through the publication in ontological repositories, the semantics of the characteristics is commonly agreed. A characteristic creates a semantic context for the definition of its Service Properties. ACTAS distinguishes between General Characteristics (GCh) and Compatibility Characteristics (CCh). The characteristic also defines sets of constraints (Co-set in Definition 2) for its properties. General Characteristics contain information about the service. Compatibility Characteristics are additionally used for the description of (principal) compatibility between services.

Besides the characteristics, the Service Management of ACTAS publishes classes for the property description. Property Classes are used for the definition of properties in the characteristics (GCh-Property). The methods of the Property Class create a behavioural semantic for the attributes of a property, including value constraints. Unchangable value constraints can directly be implemented in the property class (attribute constraints) or are given in the semantic context of the characteristic (property constraints), when the Service Property is defined through the class. The value constraints can lead to several valid variants of the property.

*Example 3 (**Service Property for keeping of temperatures**).* A property class for keeping temperatures could contain methods for dealing with units of measurement like Celsius or Fahrenheit. Such a property could also deal with value constraints if only information about possible temperature ranges are known.

Example 4 (Possible Characteristics for the examples). Compatibility Characteristics could be introduced for the agency service or banking service in Exam-

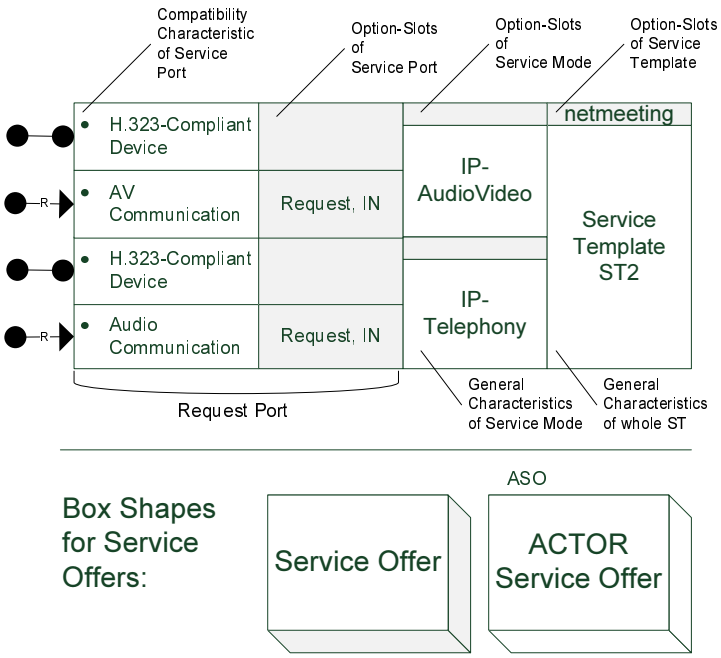


Fig. 4. - Service Symbols

ple 1. Compatibility Characteristics describing the technical component services in Example 2 are further examples. A Compatibility Characteristic for security could wrap properties, which describe what kind of authentification and membership are necessary for using the service. A possible General Characteristic contains standardized information about the Service Provider.

The Property-Merge Class checks constraints for the compatibility of two properties of the same class. The Property-Merge Class is part of the property definition in a Compatibility Characteristic (CCh-Property). The Property-Exchange Classes can be used for the definition of constraints, which concern one or several properties in the resulting composite data structure (Composite Structure). The Composite Structure and the objects of the property classes (Property Class, Property-Merge Class, and Property-Exchange Class) are elements of the CModel of ACTAS.

A Service Template (ST) describes for the different Service Modes (SM) the constraints and compatibilities. For instance, Fig. 4 is an illustration of a Technical Service description, which has two possible Service Modes for the support of IP-telephony (one supports audio-video). Thus, several variants of a service can be kept in one Service Description, which can be useful for the resource management.

Each Service Mode (SM in Definition 2) has at least one Service Port (SP). Each Service Port describes the compatibility to another service through Com-

patibility Characteristics and Option-slots. The Compatibility Characteristics ensure that only services with semantically matching Service Properties can be related. The ***Option-slots*** determine options for the relationship of compatible services. For example the *"direction Option-slot"* of a Service Port declares that the Service Port demands a directed relationship (OUT for the client role, IN for the server role). Thus, in a directed relationship (IN, OUT) or (OUT, IN) are matching pairs for the direction Option-slots. The Service Mode "IP-Telephony" in Fig. 4 offers (direction Option-slot IN) a service for Audio-Communication and needs for its realisation the service of a H.323 device (second non-directed Service Port). The Option-slots are like pre-conditions for the relationships of services. The checking of compatibility is task of the Composition Model (CModel) (cf. sections 3.3, 3.4).

3.2 Phase 1, Phase 2: Framework for Service Model (SModel)

Fig. 5 shows the system architecture/framework of ACTAS. In phase 1 (Service Management), administrators design the components of ACTAS and export them in ontological repositories (1 in Fig. 5). The Service Provider uses these components, in order to design Service Templates (ST) (2 in Fig. 5). ACTAS assumes that a Service Provider knows the Service Grounding and Service Deployment of the provided services. ACTAS enables the Service Provider to offer a ST of a service through a Facility Agent (FA)(FA-ID in ST, cf. Definition 2). The Service Template (ST) enumerates the different Service Modes (SM). For the Service Discovery and Service Composition, each SM offers the service through its Service Ports (SP) with some selected Characteristics.

In order to keep track of the availability, ACTAS distinguishes between Service Templates (ST) and Service Offers (SO). A SO is build from a ST with a Service Offer Export Record (SOER). A SOER (cf. Definition 2) declares in its constraints-set (Co-set), which Service Modes of a Service Template are currently valid and which value constraints for the Service Properties given in the ST shall be applied. Service Offers contain objects for the classes of Service Properties.

In phase 2 (Service Trading and planning) the Trader Agents (TrA) get in touch with the Facility Agents (FA). As autonomous software agents, the Trader Agents can comply with trading policies. They can do their trading with Service Templates for planning or Service Offers (SO) (3 in Fig. 5). A Trader Agent (TrA) can support automatic service composition through composing of new Service Offers. In this case, the Trader Agent (TrA) would be the Facility Agent (FA) for a new built Service Offer.

3.3 ACTAS phase3: Principally compatible services (CModel)

Definition 3 (Components of CModel).

1. *Service Request, SRe :== (ReA-ID, ClRe-set)*
2. *Client Request, ClRe :== (PA-ID, ReP-set)*
3. *Request Port, ReP :== (RCh-set, Option-slots)*

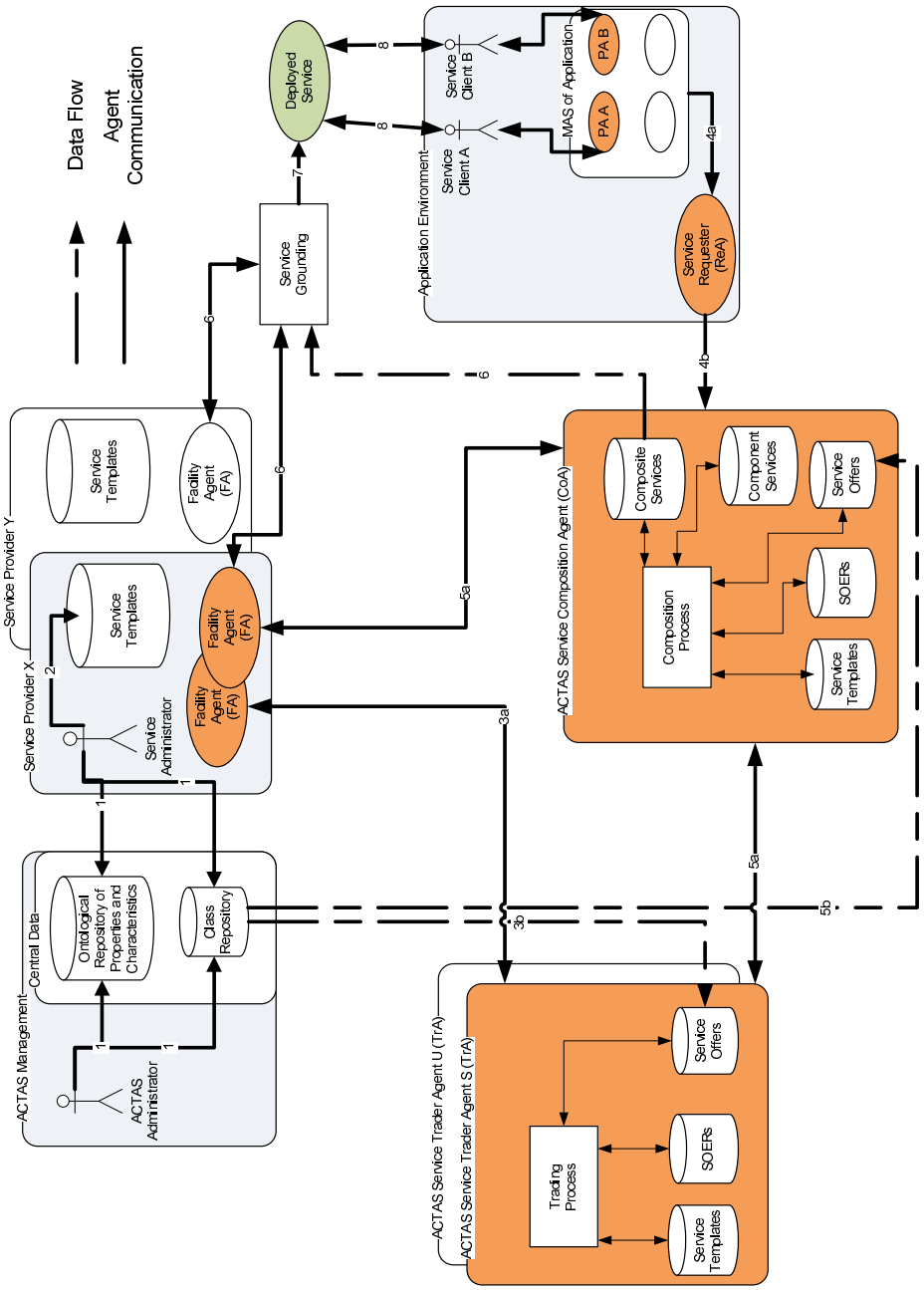


Fig. 5. - ACTAS system architecture

4. *Composite Structure :== (Actors, SM-set, MP-set, Me-Co-set, Ex-Co-set)*
5. *merged Service Ports, (MP :== (SP_a, SP_b))*
6. *Merge Constraint, (Me-Co :== (P-$Object_a$, P-$Object_b$, Property-Merge-Object))*
7. *Exchange Constraint,*
   (Ex-Co :== (Pre-Condition, Ex-Rule, Property-Exchange-Object))

In the third phase, the Request Agent (ReA) is contacted through the application environment. The Request Agent is integrated in the applying environment. The ReA creates the Composition Agent (CoA) and the Service Request (SRe) on behalf of the Service Client(s) (4 in Fig. 5). The Service Request is used for the initialisation of the Composite Structure (cf. Definition 3). The CoA finds the candidates for Component Services with the help of the Facility Agents and the Trader Agents (5 in Fig. 5).

The Service Request (SRe) consists of one Client-Request (ClRe) for each Service Client. Every Client Request holds Request Ports (ReP), which are Service Ports that contain only Request Characteristics (RCh, cf. Definition 2). A Request Characteristic (RCh) is a special kind of Compatibility Characteristic, which is used for the description of services from the view of a Service Requester. In this way, ACTAS distinguishes between the Service Requester (B2C relationship) and Service Provider (B2B relationship) view. A Request Port describes automatically a directed relationship through the direction Option-slot OUT indicating the role of a client. The Client Request is not a complete Service Description, but allows the pre-selection of services fulfilling the given characteristics.

Each Client Request (ClRe) is used for the creation of a Actor Service Offer (ASO) (Service Client is seen as actor). ASO is the dual element to a Service Offer (SO) and the starting point of the Composite Structure for a Composite Service candidate. An Actor Service Offer (ASO) could be created from an Actor Service Template (AST), which collects (learnt) preferences of the Service Client as "Request Modes". The learning of preferences is part of our future research.

In the end, Definition 4 specifies the *"principal compatibility of services"* with the principal compatibility of Service Ports and the selection of the Service Modes, to which the Service Ports belong. Principally compatible services in ACTAS simply held the same sets of Compatibility Characteristics and matching sets of Option-slots.

A Service Port, which is not merged with a principally compatible Service Port in the Composite Structure, is called an *"Open Port"*. The positive result of phase 3 is a Composite Structure with no Open Ports, holding the Actors (the Service Clients in the current version of ACTAS), the selected Service Modes, a set of merged Service Ports (MP), and sets of constraints, which are solved in the next phase of the Life Cycle (cf. section 3.4).

Definition 4 (Principal Compatibility). *Two services (a and b) are principally compatible (pcompatible(a,b)) iff their Service Offers (SO_a and SO_b) are principally compatible (pcompatible(SO_a, SO_b)).*

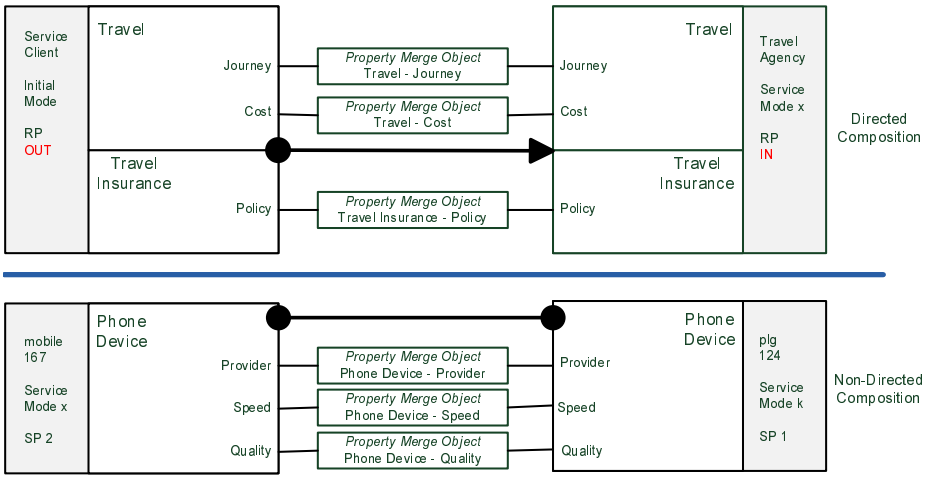


Fig. 6. - ACTAS Compatibility

Two Service Offers SO_a *:== (FA-ID*$_a$*, GCh-set*$_a$*, SM-set*$_a$*) and* SO_b *:== (FA-ID*$_b$*, GCh-set*$_b$*, SM-set*$_b$*) are principally compatible, iff they have two principally compatible Service Modes* $SM_a \in$*SM-set*$_a$ *and* $SM_b \in$*SM-set*$_b$*:*
pcompatible(SO_a*,* SO_b*)* $:\Longleftrightarrow$ *pcompatible(*SM_a*,* SM_b*))*

Two Service Modes SM_a *:== (GCh-set*$_a$*, SP-set*$_a$*) and* SM_b *:== (GCh-set*$_b$*, SP-set*$_b$*) are principally compatible, iff they have two principally compatible Service Ports* $SP_a \in$*SP-set*$_a$ *and* $SP_b \in$*SP-set*$_b$ *:*
pcompatible(SM_a*,* SM_b*)* $:\Longleftrightarrow$ *pcompatible(*SP_a*,* SP_b*)).*

Principally compatible Service Ports (SP_a *:== (CCh-set*$_a$*, Option-slots*$_a$*),*
SP_b *:== (CCh-set*$_b$*, Option-slots*$_b$*)) have matching sets of Compatibility Characteristics and Option-slots.*
Two sets of Compatibility Characteristics (CCh-set$_a$*, CCh-set*$_b$*,*
$\#$*CCh-set*$_a$ $=$ $\#$*CCh-set*$_b$*) match, iff for every Compatibility Characteristic in one set (CCh*$_a$ *:== (Name*$_a$*, CCh-Property-set*$_a$*, Co-set*$_a$*)* $\in$ *CCh-set*$_a$*) exists a matching Compatibility Characteristic in the other set (CCh*$_b$*:== (Name*$_b$*, CCh-Property-set*$_b$*, Co-set*$_b$*)* $\in$ *CCh-set*$_b$*). Only Compatibility Characteristics with the same kind/name match (Name*$_a$ $=$ *Name*$_b$*).*
Two sets of Option-slots (Option-slots$_a$ *and Option-slots*$_b$*) are compatible iff no incompatible pairs of Option-slots of the same kind can be build.*

3.4 ACTAS phase4: Solving constraints (CModel)

In phase 4 (Checking Constraints), the Composition Agent checks the constraints with the declarative Composite Structure of the Composite Service. Possibly, the Facility Agents or Personal Agents are involved. The Facility Agents can be used

for the reservation of resources. The Personal Agent is like a Facility Agent for the ASO of the Service Client. During checking of constraints, the Composition Agent can get in touch with the Personal Agent and/or contact the human being behind the Personal Agent, in order to involve them in the negotiations of the Property Discovery process. This policy of a CoA can be adapted through the behavioural semantics of the property description (Fig. 3).

ACTAS distinguishes between Merge-Constraints (Me-Co) and Exchange-Constraints (Ex-Co). Merge Constraints "merge" the Service Properties (P-Object$_a$, P-Object$_b$) of matching Compatibility Characteristics (cf. Fig. 6) of principally compatible Service Ports (SP$_a$, SP$_b$). The merge process of a Service Property is defined through the merge method of the Property Merge Object (merge(P-Object$_a^{old}$, P-Object$_b^{old}$,P-Object$_a^{new}$, P-Object$_b^{new}$,Variant)). The result of a successful merge are two new objects, which are possibly identical. For the support of backtracking in the reasoning, the merge process can determine possibly existing variants for the application of the constraints.

A merge of two properties depends on the semantic context and the direction of the relationship. Since the Property-Merge-Class of the Property-Merge-Object is given with the Compatibility Characteristic, the sematic context is observed. The eventually existing direction Option-Slots of the principally compatible Service Ports determine the direction (cf. section 3.2). In the case of the property in Example 3, the merge method will try to find a common value (range) for the temperature, which complies to the value constraints of both objects. In a directed relationship one property object contains the requested values and the other one the offered values. A directed relationship could be used for the merge of properties containing IOPE-capability descriptions (Semantic Web Service descriptions, OWL-S (cf. 2.5)). The merge method would determine if the capability descriptions are compatible.

The Me-Co-set with its merged Service Property objects is the "information container" of the Composite Structure. The Exchange Constraints (Ex-Co) in the Ex-Co-set describe constraints upon these Service Properties. In the rest of phase 4, ACTAS tries to solve the Exchange Constraints. The next Exchange Constraint for solving is selected with the Pre-Condition and performed with the Ex-Rule. Property-Exchange Object(s) can be used for solving of more complicated Exchange Constraints since the Service Properties are objects themselves. Exchange Constraints are described in the Service Definition of a Service Template. The Pre-Condition could ensure that the involved Service Properties have a fixed/set value or that they fulfill given value constraints. The gained new objects of the Ex-Rule will be checked again, if they still comply with its Merge Constraint. An Exchange Constraint is applied only once. Phase 4 ends when all Exchange Constraints with fulfilled Pre-Condition were applied.

The behavioural semantics of the elements of the Property Description Fig. 3 adapts the reasoning behaviour of ACTAS in phase 4 and allow e.g. the inclusion of estabished methods of semantic web. The result of the reasoning of phase 4 is the gain of new information in the Composite Structure.

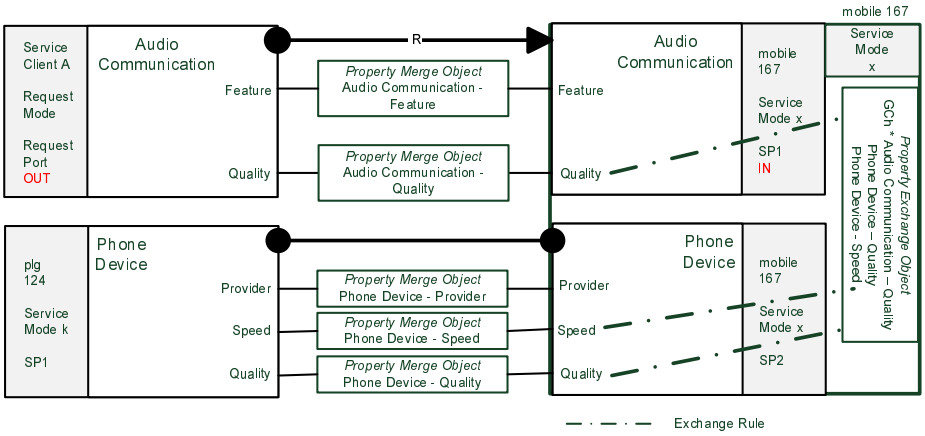


Fig. 7. - ACTAS Exchange Rule

*Example 5 (**Exchange Constraint for temperatures**).* The temperature property of Example 3 could be in a Compatibility Characteristic, which describes the service of a radiator. The Service Requester is not interested in the readiator, he/she asks simply for a "warm" sleeping room or living room. The Exchange Constraint ensures that the temperature description of the Request Characteristic is translated into the right temperature value (range) for the radiators in the specified rooms.

3.5 The framework character of ACTAS

The framework character of ACTAS enables on one hand its reasoning and the pre-selection of Composite Service candidates. On the other hand, it allows the use of established service environments for the subsequent refinement of the Service Properties in the Life Cycle of the service. Fig. 5 gives an overview of the multi-agents system realising the framework character of ACTAS. ACTAS is based on following agents: Facility Agents (FA, Interface to the Service Provider), Trader Agents (TrA), Request Agents (ReA, Interface to the application environment), and Composition Agents (CoA). In an MAS environment of the service requesting application, Personal Agents (PA) and Group Agents often exist, which could support ACTAS. The Personal Agents represent potential Service Clients.

ACTAS assumes that the specific characteristics of an offered service are known to its Service Provider. The Service Provider is responsible for the Resource Management and automatic deployment of the service, even when its consumption is delayed. ACTAS improves the possibilities for a preemptive Resource Management by the Service Provider. Resources for selected Service Modes can be reserved for the Service Grounding process. Simultaneously, the Service Provider can adapt the advertisements of Service Offer accordingly through

exporting new Service Offer Export Records (SOER). For non-functional properties, domain specific Compatibility Characteristics could be introduced and included in the description of a Service Port for Service Composition.

The Service Provider controls the Facility Agents. Facility Agents (FA) export the Service Templates (ST) and Service Offer Export Records (SOER). They also provide the access to the service and initiate the necessary Service Grounding process (phase 5). Service Grounding for ACTAS likely means the transparent call of other service environments. Special Compatibility Characteristic can ensure that services of the right Service Environments are selected. The agents of ACTAS provide (new) information gained through the reasoning process of phase 3 and phase 4. ACTAS could support Service Substitution and the learning of preferences based on the feedback.

In a successful case, the gained information is provided for the Facility Agents, in order to support the subsequently Service Grounding in phase 5 (6, 7 in Fig. 5). The Service Grounding is independent from ACTAS and can be done with the established methods ((Semantic) Web Services, MAS, EAI) and the negotiating support of the Facility Agents.

In phase 6 (Service Consumption), the agents of ACTAS can still be contacted for information or feedback to the application environment.

The pre-selection is (a early) part of the Property Provision process in the Life Cycle of a service (cf. 2.3). It makes the SModel and CModel of ACTAS independent from the challenges of the Service Grounding, which continues the Property Provision process. The framework design of ACTAS, which is based on so-called Facility Agents (FA), relies on the ability of the agents to negotiate for the realisation of the Service Grounding. The Facility Agents are part of the Service Provision. The Service Grounding of ACTAS can be for instance the call of a manual service, or the start of a new Service Composition initiated by the Service Provider. ACTAS discovers (principal) compatible services and enables their providing Facility Agents to get in touch for follow-up negotiation. ACTAS supports this process with gained information about the selected Service Mode and the property values found through reasoning over the constraints. When the Service Grounding does not lead to a successful Service Deployment or the Service Quality is below a threshold, the framework of ACTAS can be asked for a backtracking, in order to look for an alternative Composite Service. However, these decisions are out of the control of ACTAS and have to be initiated by the FAs.

3.6 Application of ACTAS

Fig. 8 and Fig. 9 show schematically service compositions (principal compatibility) of ACTAS for Example 2 and Example 1, respectively. In the case of Technical Service (Example 2), Actor Service Offers (ASO) for the Service Clients A and B are the initialisation of the Composite Structure. Service Client A has currently access to his/her mobile (Service Offer "mobile167"), and Service Client B can use the Cisco IP-phone (Service Offer "c11"). Option-slots could be introduced for checking of non-functional properties (spatial or security), in order

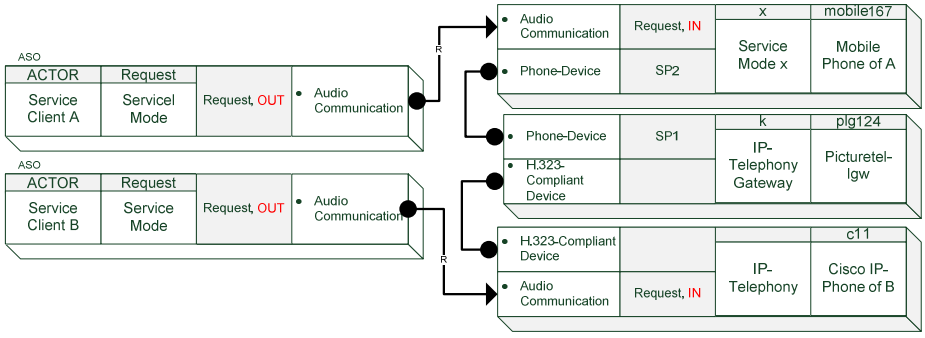


Fig. 8. - ACTAS Technical Service

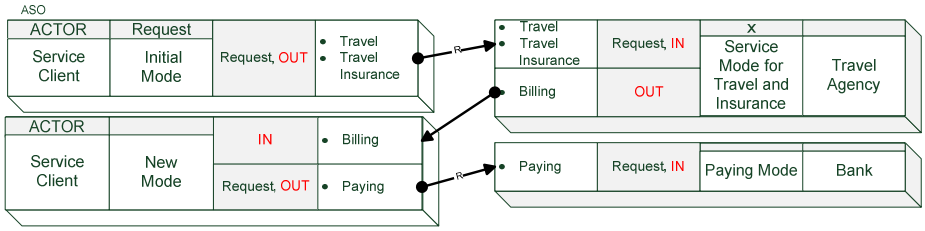


Fig. 9. - ACTAS Business Service with two Service Modes for the ASO

to ensure early that the service is accessible. The same can be achieved with adding of appropiate Compatibility Characteristis, of which the constraints are checked in phase 4 (cf. 3.4). An available H.323-gateway (Service Offer "plg124") completes the Composite Structure. The selected Service Mode of the gateway service supplies the connection service of a H.323-compliant device and the connection service of a phone-device.

In case of the Composite Service for the Business Service (Fig. 9), the Service Request did not only contain a Request Characteristic for a "Travel" service, but additionally the Request Characteristic "Travel Insurance", which semantically could mean (depends on the common ontological repository) that the travel service has to include travel insurance. Another possible way to express this constraint of the request could be the inclusion of appropriate Service Properties for Travel Insurance in the Request Characteristic "Travel".

The Actor Service Offer (ASO) for the Service Client Request was the initialization of the Composite Structure. ACTAS supports through backtracking the inquiry process of Example 1, if the inquiry process is not a separate service. Following the example, the first candidate Component Service would be Travel Agency 1. However the checking of the merge constraints of its Service Properties in phase 4 will reveal that Travel Agency 1 is not an appropiate service. The behavioural semantic of the Property Merge object (or Property Exchange object) could include a consulting with the Personal Agent (PA) of the Service Client,

possibly involving the Service Client himself/herself. The algorithm of ACTAS backtracks to phase 3 and discovers with Travel Agency 2 a new candidate for the Component Service.

The principally compatible Service Mode of the travel agency includes a Service Port with the Compatibility Characteristic for "Billing". A special directive of an Option-slot links back to the Service Client and asks for a new Service Mode of its ASO. An Actor Service Offer (ASO) can be extended with a new Service Mode, possibly with the help of the Personal Agent. This new Service Mode includes a new Service Port with the Request Characteristic "Paying", which is composed through a directed composition with the principally compatible Service Mode of a suitable bank.

4 Conclusion and Future Research

The reflections of the five aspects of a service showed that a general description of a service considering all demands is not possible due to several factors. Alternatively, the paper proposed an environment called ACTAS, which enables a simple pre-selection and composition with commonly agreed semantic characteristics. Directed and non-directed relationships are supported. Characteristics can be defined for a B2C (Request Characteristic) and for a B2B relationship. Through the distinction between Service Offers and Service Templates, the non-functional property availability is integrated. The framework character is created through a multi-agent system (MAS), which allows the integration of existing, domain-specific solutions. The Software agent paradigm supports the use of various policies for the trading and composition. These policies can be adapted through behavioural semantics. In this way, the behaviour of Service Properties, the Service Composition, and the dealing with constraints for the whole Composite Structure are adaptive for the current semantic context. The inclusion of Trading Agents enables the Automatic Service Compostion, i.e. new services are dynamically build and offered. In the end, additional information for the subsequent Service Grounding can be provided.

Future research will be an extension of our declarative testing environment and its integration in multi-agent environments. Automation of the Service Grounding with established Service Composition methods is another goal of our research.

References

1. Owl-s 1.2 release: (pre release), http://www.ai.sri.com/daml/services/owl-s/1.2/
2. Wu, Z., Deng, S., Li, Y., Wu, J.: Computing compatibility in dynamic service composition. Knowledge and Information Systems (2008)
3. Wu, Z., Ranabahu, A., Gomadam, K., Sheth, A.P., Miller, J.A.: Automatic composition of semantic web services using process and data mediation: Technical report, http://knoesis.cs.wright.edu/library/publications/download/SWSChallenge-TR-METEOR-S-Feb2007.pdf (2007)
4. Pistore, M., Bertoli, P., Cusenza, E., Marconi, A., Traverso, P.: Ws-gen: A tool for the automated composition of semantic web services: Iwcs 2004, http://iswc2004.semanticweb.org/demos/26/paper.pdf (2004)
5. Lécué, F., Léger, A.: A formal model for semantic web service composition. The Semantic Web - ISWC 2006 (LNCS) (4273) (2006) 385–398
6. Cabral, L., Domingue, J., Motta, E., Payne, T., Hakimpour, F.: Approaches to semantic web services: an overview and comparisons. The Semantic Web: Research and Applications (LNCS) (3053) (2004) 225–239
7. Kvaloy, T.A., Rongen, E., Tirado-Ramos, A., Sloot, P.: Automatic composition and selection of semantic web services. Lecture Notes in Computer Science (LNCS) **vol 3470/2005** (2005) 184–192
8. Gomez-Perez, A., Gonzalez-Cabero, R., Lama, M.: Ode sws: A framework for designing and composing semantic web services. IEEE Intelligent Systems **19**(4) (2004) 24–31
9. Küngas, P., Matskin, M.: Semantic web service composition through a p2p-based multi-agent environment. Agents and Peer-to-Peer Computing (LNCS) (4118) (2006) 106–119
10. Burstein, M., Bussler, C., Finin, T., Huhns, M.N., Paolucci, M., Williams, S., Zaremba, M., Sheth, A.P.: A semantic web services architecture. Internet Computing, IEEE **9** (2005) 72–81
11. Honavar, V., Basu, S., Lutz, R.: Algorithms and software for interactive discovery and composition of semantic web services: Project summary, http://www.cs.iastate.edu/~honavar/ailab/projects/services.html (06.08.2008)
12. Sattanathan, S., Narendra, N.C., Maamar, Z.: Conwesc - context-based semantic web services composition: Icsoc, 05, http://www-rocq.inria.fr/who/Sattanathan.Subramanian/Sattanathan_ICSoC2005.pdf (2005)
13. Bromuri, S., Urovi, V., Morge, M., Stathis, K., Toni, F.: A multi-agent system for service discovery, selection and negotiation. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems. Volume 2. International Foundation for Autonomous Agents and Multiagent Systems, Budapest, Hungary (2009) 1395–1396
14. Toni, F., Grammatikou, M., Kafetzoglou, S., Lymberopoulos, L., Papavassileiou, S., Gaertner, D., Morge, M., Bromuri, S., McGinnis, J., Stathis, K., Curcin, V., Ghanem, M., Guo, L.: The argugrid platform: An overview. In Hutchison, D., Altmann, J., Fahringer, T., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Neumann, D., Nierstrasz, O., eds.: Grid Economics and Business Models. Springer-11645 /Dig. Serial]. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg (2008) 217–225
15. O'Sullivan, J., Edmond, D., Hofstede, ter, A.H.M.: What's in a service?: Towards accurate description of non-functional service properties: Service description: A

survey ot the general nature of services: Kluwer academic. Distributed and Parallel Databases (DAPD) **12**(2/3) (2002) 117–133
16. Lovelock, C., Gummesson, E.: Whither services marketing? in search of a new paradigm and fresh perspectives. In Parasuraman, A., Lemon, K.N., eds.: Journal of Service Research. Volume Volume 7, No. 1, August 2004. SAGE publications (2004) 20–41
17. Owl-s 1.1 release, `http://www.daml.org/services/owl-s/1.1/`
18. Domingue, J., Roman, D., Stollberg, M., eds.: Web service modeling ontology (wsmo) - an ontology for semantic web services: Position paper at the w3c workshop on frameworks for semantics in web services, june 9-10, 2005, innsbruck, austria, `http://www.w3.org/2005/04/FSWS/Submissions/1/wsmo_position_paper.html` (2005)
19. Kumar, S.: Semantic web service composition. Institution of Electronics and Telecommunication Engineers (Vol. 25, No. 3 (2008)) (2008) 105–122
20. Stoutenburg, S., Obrst, L., Nichols, D., Ken Samuel and Paul Franklin: Applying semantic rules to achieve dynamic service oriented architectures. In Eiter, T., ed.: Second International Conference on Rules and Rule Markup Languages for the Semantic Web, Las Alamitos, Calif., IEEE Computer Society (2006) 75–82
21. Deschrevel, E., Crespi, J.P.: Service definition for next generation networks: Networking, international conference on systems and technologies 23-29 april. ICN/ICONS/MCL (194) (2006) 22
22. Zeithaml, V., Bitner, M.: Services Marketing. McGraw-Hill, New York (1996)
23. Kotler, P.: Marketing Management Analysis, Planning, Implementation, and Control. 6 edn. Prentices-Hall International (1988)
24. Kotler, P.: Marketing Management. 11 edn. Prentices-Hall International, Upper Saddle River, NJ, USA (2003)
25. H.323 (white paper): On-line education: Tutorial, `http://www.iec.org/online/tutorials/h323/index.asp`
26. Elfatatry, P., Layzell, A.: Software as a service negotiation perspective: Computer software and application conference, 26-29 august. COMPSAC Proceedings **26** (2002) 501–506
27. Saeed, M., Jaffar-Ur-Rehmann, M.: Enhancement of software engineering by shifting from software product to software service: Information and communication technology, first international conference 27.-28. august. ICICT Proceedings (2005) 302–308
28. Bearman, M., Raymond, K.: Federating traders: an odp adventure. In de Meer, J., Heymer, V., Roth, R., eds.: Open distributed processing. IFIP Transactions. North-Holland, Amsterdam (1992) 125–143
29. Christoffel, M.: Cooperations and federations of traders in an information market. In Schroeder, M., Stathis, K., eds.: Proceedings of the AISB'01. AISB01, Agents & Cognition. University of York, Heslington, York, England (2001) 51–60
30. Jacob, B.L., Mudge, T.: The trading function in action. In Herbert, A., Tanenbaum, A.S., eds.: Proceedings of the 7th ACM SIGOPS European Workshop. ACM, Connemara, Ireland (1996) 241–247
31. Jacob, B.L., Mudge T.: Support for nomadism in a global environment. In: Workshop on Object Replication and Mobile Computing (ORMC'96). ACM, San Jose, CA, USA (1996)
32. Geihs, K., Farsi, R.: Sevice trading in electronic market. In: International Journal of Electronic Market. Springer (1997) number 3

33. Kostkova, P., Wilkinson, T.: Magnet: A virtual shared tuplespace resource manager. In Paprzychi, M., ed.: International Journal on Parallel and Distributed Computing. Volume 1 of International Journal on Parallel and Distributed Computing. NOVA Science Books, Commack, New York (1998)
34. Marvie, R., Merle, P., Geib, J., Leblanc, S.: 3.3.8 torba: Trading contracts for corba. In: 6th USENIX Conference on Object-Oriented Technologies and Systems. The USENIX Association, San Antonio, Texas, USA (2001)
35. Puder, A., Markwitz, S., Gudermann, F., Geihs, K.: Ai-based trading in open distributed environments. In: International Conference on Open Distributed Processing (ICODP'95). Chapman and Hall (1995)
36. Richman, A., Hoang, D.: Accomplishing distributed traders utilizing the x.500 directory. In: MICC'95. Kluwer Academic Publishers, Malaysia (1995)
37. Vogel, A., Beitz, A., Ianella, R.: Discovery and Access of Services in Globally Distributed Systems. DSTC Symposium. DSTC Pty. Ltd., Brisbane, Australia (1995)
38. Vogel, A., Bearman, M., Beitz, A.: Enabling interworking of traders. In Raymond, K., Amstrong E., eds.: Open Distributed Processing III. Chapman & Hall, London (1995)
39. Preist, C., Bartolini, C., Byde, A.: Agent-based service composition through simultaneous negotiation in forward and reverse auctions. In: Proceedings of the 4th ACM Conference on Electronic Commerce, New York, NY, ACM Press (2003) 55–63
40. Trastour, D., Preist, C., Coleman, D.: Using semantic web technology to enhance current business-to-business integration approaches. In: Proceedings, Los Alamitos, Calif., IEEE Computer Society (2003) 222
41. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Data-Centric Systems and Applications. Springer, Berlin, Heidelberg (2004)
42. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D., eds.: Web services architecture: W3c working group note 11 february 2004, `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/` (08.02.2004)
43. Kang, Z., Wang, H., Hung, P.C.: Ws-cdl+ for web service collaboration. Information Systems Frontiers **9** (2007) 375–389
44. Kavantzas, N., Burdett, D., Ritzinger, G., eds.: Web services choreography description language version 1.0: W3c working draft 27 april 2004, `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/` (27.04.2004)
45. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Bpel v1-1: Business process execution language (bpel4ws), `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf` (06.05.2003)
46. Mandell, D.J., McIlraith, S.A.: Adapting bpel4ws for the semantic web: The bottom-up approach to web service interoperation. In Dieter Fensel, Katia P. Sycara, John Mylopoulos, Fensel, D., eds.: The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003, Proceedings /// The semantic Web. Volume 2870 of Lecture notes in computer science. Springer, Berlin (2003) 227–241
47. Vasiliev, Y.: SOA and WS-BPEL: Composing Service-Oriented Architecture Solutions with PHP and Open-Source ActiveBPEL. Packt Publishing (2007)

On Time in Games

Rustam Tagiew, Heinrich Jasper

Institute for Computer Science of TU Bergakademie Freiberg, Germany

Abstract. Our work considers finite strategic games for needs of AI, i.e. their formal definition as well as their implementation using agent based techniques. This allows for both, the interaction with people as "natural" players as well as gaming in an all artificial environment with artificial agents as players. This paper concentrates on the formal language for the specification of discrete time depending elements of finite strategic games. Based on a Petri Net approach we investigate how to neatly integrate delays, timeouts and sudden events in a general game playing formalism. Upon that we will find a game theoretic solution for timed games. This is the basis for an algorithm for a game server that implements an environment for strategic game playing with arbitrary natural or artificial agents or both. Concrete application examples show the feasibility of our approach.

1 Introduction

Finite strategic games are interactions of a finite amount of agents, where every agent's payoff depends on actions of other agents. The payoff is the quantified value of the achieved result or also the degree of fulfilling the goals. Every agent has a finite amount of actions and there is a finite amount of states. This interaction is called strategic, because agents anticipate goals of other agents and can predict their actions. Strategic reasoning is a nested reasoning about reasoning of other agents. This reasoning is best defined by epistemic logic [1]. A finite strategic game has a least one solution [2]. This solution is called equilibrium in game theory and is a combination of the players's strategies. If all players are rational, none of them benefits if he deviates. In AI, games have at least two research threads - mechanism design and agent design [3, p.632]. For mechanism design, we must invent game rules for a specified behavior. In agent design, we have game rules and must compute adequate behavior. These two tasks can be computed in a theoretical way by running routines over a game tree or in practical way by building an environment where different real players can interact.

For the definition of such games in a formal language as well as for the analysis of actual games carried out in our environment arguing about time is a crucial thing. When specifying the rules for a strategic game one might want to express things like "the next turn has to be carried out within 20 seconds after the previous turn finished". In other games one might want to monitor the timely behavior of natural agents, i.e. how many seconds a person "thinks" on her or his next turn, in order to calculate game theoretic equilibria.

Our ontology of time is based on real or physical time as considered for example in planning or modelling scenarios. Physical time can be characterized as linear, forward branching or parallel. Whereas linear time provides a total ordering on time the others allow divergent non-comparable time occurrences. Although the latter two are necessary for the modeling of non-predictive future worlds or the Einstein Universe we concentrate for simplicity on linear time.

Linear time in turn can be either bounded, one-side unbounded or generally unbounded. We do not consider a minimal or maximal instant of time, thus our model copes with unbounded linear time - of course up to implementation limitations due to data representation in the game server. Furthermore, time can be continuous i.e. isomorphic to real numbers or discrete. Discrete time can be subdivided into time with known minimal resolution (isomorphic to natural numbers) or with arbitrary solution (isomorphic to rational number). For practical reasons in the implementation we use a minimal, system dependent resolution of some milliseconds, called the chronon.

At least we have to define the ontological primitives for the modeling of time. These are time points which are one instant of time with predefined accuracy, that is a interval of "surrounding chronons" which might collapse to one chronon in the case of maximal accuracy. A time point is specified from year to millisecond via the mask "YYYY.MM.DD:hh:mm:ss-msec". The second representational primitive is the time interval, a continuous sequence in time bounded by the lower and upper time point. The third concept is that of duration, for example to model delays or to compute time points or time intervals from other time points or time intervals, respectively. Durations are represented by numbers accompanied with the appropriate unit, e.g. "6 hours" or "5 seconds".

All three time primitives presented above are necessary in finite strategic games. Time points are for example used for specifying the temporal aspect of start and stop events or are part of the description of each players turn. Time intervals are used to describe or to monitor the physical temporal dimensions of each turn in a game carried out by our game management agent. Calculations on physical time need of course durations, for example to calculate the time point for "twenty seconds after the next turn of player a".

2 Related Work

We found two well-known independent approaches in modelling general games. Both cannot be used for the definition of time dependent elements of a game. GALA (game-theoretic analysis for a large class of games) language is the first one [4]. It is logic-based and is considered for general definition and solving of games. In the GALA language, games are represented as branching programs. Every branching node in a game tree is a call of a function and also a logic proposition, which can be satisfied in a couple of ways (branches). The supporting system for GALA is prolog. The GALA system generates a game tree using a definition of a game. The generated game tree is forwarded to GAMBIT[5].

GAMBIT is state-of-art open source game solving system. GALA can also solve games itself using commercial linear programming libraries. GALA can represent finite games of imperfect information. There is no approach to the definition of game server based on GALA.

The second work is general game playing (GGP) [6]. It considers finite games of perfect information, which are called deterministic in game theory. The main idea of GGP is providing an environment for conducting contests between different artificial agents. GGP provides a game model. The game model is a graph consisting of game states connected by actions. Actions are the transitions between states. The game model can include cycles. States are explained to be not monolithic. That means that they consist of a couple of separately changeable items like a database. GGP provides a logic-based language GDL for the definition of the game model. They use a vocabulary of predicates. A transition is performed as an update of the dynamic knowledge base. The GGP environment is based on web services.

We use a Petri Nets (PN) based approach for our work. There exists a huge amount of proposals for the modeling of temporal aspects in Petri Nets, see for example the overview in [7]. None of the approaches uses a time model as elaborated as ours. This is due to the fact that our model has been used in various practical applications and these showed the necessity of such a full-fledged approach for the modeling of "real" time, see for example [8]. Furthermore, within the community of Time Petri Nets the term "interval" is usually used for the specification of some "amount of time" or delays that has to be considered when firing or not-firing a transition. This contradicts to the typical practical experience that intervals and durations must not be intermixed and there should be a clear notation on this as defined in our ontology of time above.

3 Game Examples

In real-life situations, many decisions must be done in a bounded time period. For instance, a car driver noticed a deaf cyclist on his lane and that his brakes fail. He can only turn right or continue ahead. If he turns right, he can probably survive a collision with a tree. If he continues ahead, the cyclist can perhaps survive a collision with the car. The time in this example is not discrete, but it can be modelled to be discrete, if we choose high resolution. We are not interested in searching a correct decision for this situation, but in its formal representation. Fig.1(right) shows this situation as a game in extensive form. Fig.1(left) shows a state transition diagram of this situation. Driving ahead happens, if the driver decides to do that or if he stays undecided till the deadline t_1. He can also repeal his decision by turning back to the road. In real-life, the car can be on infinitely many positions between road and land and might hit both, cyclist and tree. We reduced the number of states to be finite. In result, the driver has only two alternatives - hit the cyclist or hit the tree. And he has a couple of milliseconds to choose his destiny. Game theoretic solutions do not depend on

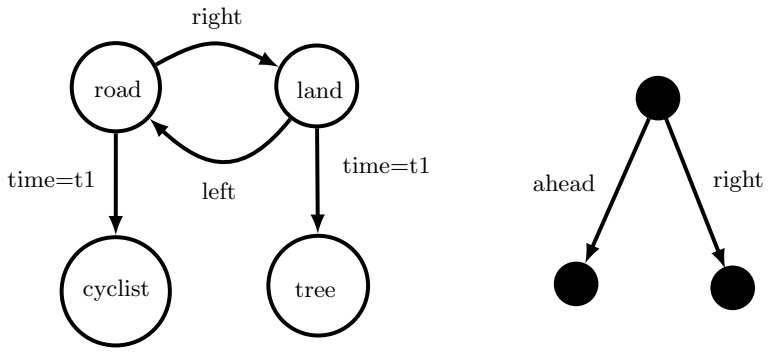


Fig. 1. Failing brakes.

the length of time periods for decisions. But, if one considers real agents and knows something about their reasoning dynamics, one can predict the outcome of a game depending on the length of the time periods given for decisions.

One another example is the well-known muddy children puzzle [9]. There is a couple of children, whose faces have a characteristic property to become sometimes dirty which is not noticed by the children itself. The children are honest and intelligent, but they never speak with each other about the state of their faces. If an elder person asks children with a dirty face to step forward, they do it only, if they are sure. If there are n dirty children, the elder person has to ask them n times till all dirty children step forward. This is a description, as it is used in most of the works considering muddy children puzzle as example for epistemic logic. But, why has the elder person to repeat himself? After the first question, he can also say something like 'Tick!'. Each question of the of the elder person is like an event, which stimulates children to make one further step in nested reasoning. If there is only one dirty child, he sees nobody else besides him and steps forward. If there are two dirty children, every dirty child sees the another dirty child and thinks that this child is perhaps the only one. In the case with two dirty children, the elder person can stay still. If all children are equally intelligent, they need for the first step of reasoning the same period of time. After the first step, both muddy children notices that nobody steps forward. Then they perform the second step and do the right thing. The asking of the elder person is not required, if children have the same reasoning speed and have a common knowledge about it.

4 PNSI

We use transition systems (TS) of PN for modelling games [10]. PNSI[1] is a combination of two elements - PN [11] and SI (strategic interaction) - PNSI =

[1] ['pɛn'zaɪ]

$(P, Q, F, W, M, I, C, N,$
$D, A, O, H, B)$. P - set of places; Q - set of transitions, where $P \cap Q = \oslash$ holds; $F \subseteq (P \times Q) \cup (Q \times P)$ - set of directed arcs; $W : F \to \mathbb{N}_1^+$ - function for weights at the arcs; $M \in \mathbb{N}^{|P|}$ - current assignment of places; I - set of agents, empty element ε stands for environment or also nature; $C \subset (Q^*)^*$ - subset of sequences of transitions, called choice sets; Every transition is a member of only one element of C; $N : C \to \mathbb{N}$ - numbering function, which is not injective; $D : \mathbb{N} \to (\mathbb{R}_0^1)^n$ - function for firing probability distribution in a choice set, where $\sum(D(\_)) = 1$ and n is number of elements of the related choice sets; $O : \mathbb{N} \to I \cup \varepsilon$ denotes ownership; $A : Q \to \mathbb{R}^{|I|}$ - payoff vector of a transition, if it fires; $H : P \to I^*$ provides for every place a subset of agents for which it is hidden. Agents can alter D for own numbers and see all unhidden places; $B : I \to \mathbb{R}$ - current account balance of agents.

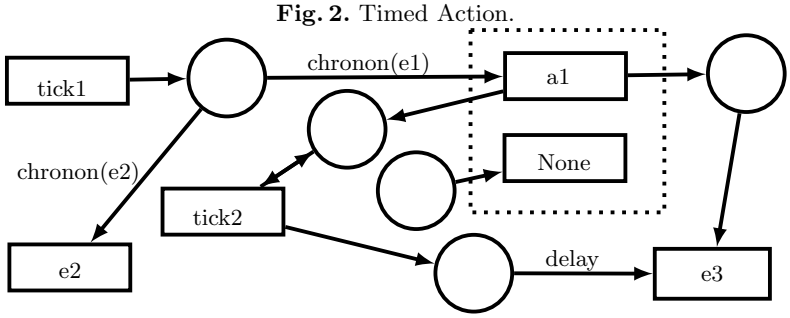
The main idea for PNSI is the concept of numbered choice sets. A choice set is a set of transitions, in which only one transition can be fired exclusively in a step. Every choice set has a number. Multiple choice sets can also have the same number. A number can have an owner and a firing probability distribution over the transitions in corresponding choice sets. If a number has an owner then the owner can alter the distribution of his number. Further, we can calculate the probability for every arc $\to$ in TS - $probability(\overset{t_1 \ldots t_n}{\to}) = \prod_{i=1}^{n} D(N(c))[position(t_i)]$, where $c[position(t_i)] = t_i$ and $n \leq |C|$.

Hiding places is useful for modelling imperfect information. Payoffs can not be hidden – there is no reason, why a player should not know the payoff, he actually gets. To construct a default PNSI structure for a concrete game, one has to create a transition for every action and a place for every state. Every arc of this PN is weighted with 1. States are created as assignments of places. All places of a state have zero tokens, except of the place, which corresponds to this state. Outgoing and incoming arcs for a transition can be derived on the basis of two connected states. But one can easily find a game, where one can construct more than one PNSI structure.

PNSI is able to model time dependent elements of games such as timeouts, delays and sudden events. In games, actions cause events (also effects). For instance, an action $a1$ causes an event $e3$. The event $e3$ must happen in same chronon or after the action $a1$. The events $e1$ and $e2$ are the bounds of the time period in which the action $a1$ can cause the event $e3$. There is no relation between time points of the events $e2$ and $e3$. The action can be also performed by the environment. Summarized that means $chronon(e1) \leq chronon(a1) \leq chronon(e2)$ and $chronon(a1) \leq chronon(e3)$, where $chronon()$ is the time point of the event. Using this model, we can define, what time dependent elements are. $e2$ is a timeout for $a1$, $chronon(e3) - chronon(a1)$ is a delay and if the action belongs to the environment, it is a sudden event. Fig.2 shows a PNSI structure for modeling a timed action. Timepoints for events are coded as weights at the arcs. The dashed box around two transitions is a choice set. A dashed box for a choice set is used only if the choice set has more than one transition. Transitions with prefix 'tick' fire every chronon.

We propose a game server, which iterates Alg.1. One iteration of the algorithm needs exactly one chronon. There are two details of the algorithm which need further discussion. The first is that a transition for a choice set is chosen independently of proving sufficiency of incoming tokens (in lines 15-20). This de-

Fig. 2. Timed Action.



tail enables representing of pause actions like *None* in the timed action example. If a player sets 1 as probability for None, no actions will be performed. The same lines show that for identically numbered choice sets, transitions can be chosen from different positions. This is done to avoid useless operations. Because of practical considerations, the distributions of owned numbers are restricted to be $\in \mathbb{N}_0^1$. That means that every player can only choose the position for his number, i.e. where to set 1. Then, there is no need to run the loop on numbers instead of choice sets. Every position of an owned number gets additionally an alias. An altering command of a player consists of number and action alias.

M, B and D can be changed in a running game. It is possible to stop the game server, save the current state of PNSI, load it again and restart the game server. This makes game management based on PNSI persistent. Further, one can record updates of (M, B, D) as events in the game. It is done by the function *record* in lines 4, 43 and 47. That can be used for analyzing behavior of real agents like human players.

To solve a PNSI declared game, we use a transformation to a GAMBIT acceptable format. GAMBIT accepts two kinds of game representation - the strategic form and the extensive form. The game representation in extensive form used by GAMBIT is a game tree with repeated states. This representation is called EFG. EFG is a tree with three kinds of nodes - chance node, personal node and terminal node. Every node contains an outcome, which is a payoff vector sized according to number of players. Chance nodes contain addionally a vector of probabilities for outgoing nodes. Personal nodes contain the owner and a vector of names for actions. Personal nodes can be connected in case of imperfect information. An already existing algorithm for constructing EFG based on PNSI for a couple of chronons is not provided in this work.

```
Alg.1: Game Server Iteration
Data: PNSI
 1 While not a_chronon_expired {
 2   commands = receive_commands;
 3   PNSI.implement(commands);
 4   record(commands);}
 5 create_set(active);
 6 Foreach t in PNSI.transitions
24 While not fired.empty {
25   tp = fired.remove_first;
26   PNSI.produce_outgoing(tp);
27   changed.add(ta.outgoing);
28   PNSI.produce_payoffs(tp);}
29 Foreach a in PNSI.agents {
```

```
 7   If PNSI.enough_incoming_tokens(t)
 8     active.add(t);
 9 If active.empty
10   complete_game;
11 create_list(tobefired);
12 Foreach c in PNSI.choice_sets {
13   th = c.choose_randomly_transition;
14   If active.contains(th)
15     tobefired.add(th);}
16 create_list(fired);
17 create_set(changed);
18 While not tobefired.empty {
19   ta = tobefired.remove_at_index(random);
20   If PNSI.enough_incoming_tokens(ta)}{
21     PNSI.abolish_incoming_tokens(ta);
22     changed.add(ta.incoming);
23     fired.add(ta);}}
30   Foreach p in changed {
31     If not PNSI.hidden(p, a)
32       add2message(a, p);
33     record(p);}
34   Foreach p in amounts {
35     add2message(a, p);
36     record(p);}
37   send\_message(a);}
```

5 Application

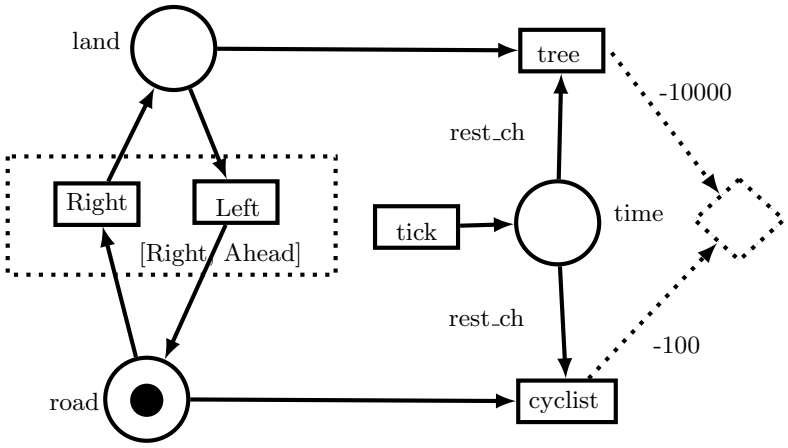


Fig. 3. Failing brakes in PNSI.

Fig.3 shows a model of the failing brakes example in PNSI. You see a choice set owned by the driver, which allows to switch a token between *land* and *road*. The transition *tick* fires every chronon and fills the empty place *time*. At a time point, when there are *rest_ch* tokens in the place *time*, the car hit one of the possible targets and the driver gets his payoff. The dashed diamond is the account of the driver.

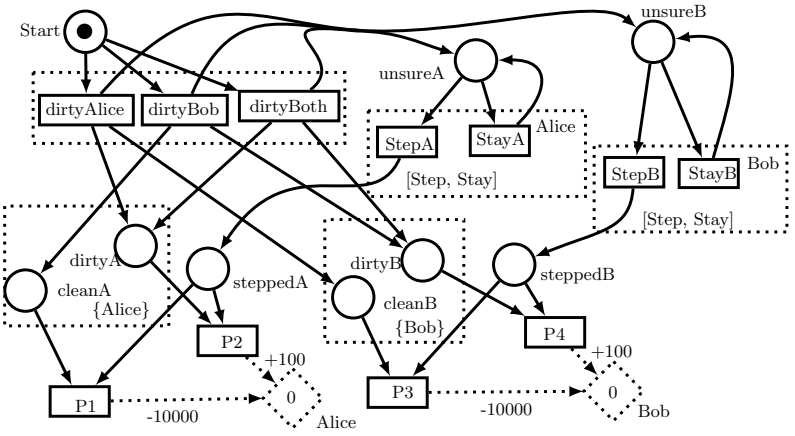


Fig. 4. Muddy Children Puzzle in PNSI.

Fig.4 shows a model of the muddy children puzzle with two children (Alice and Bob) in PNSI. It is modelled as a game, where a correct decision is rewarded and a wrong punished. The place *Start* has one token at start. There are three possible states - 'Alice is dirty', 'Bob is dirty' or 'Both are dirty'. There is a choice set controlled by chance with three transitions for these three possibilities. There are four places for decoding the current state of the world - *cleanA*, *cleanB*, *dirtyA* and *dirtyB*. Places *dirtyA* and *cleanA* are hidden (dashed rectangles around places) for Alice and places *dirtyB* and *cleanB* are hidden for Bob. Unhidden are the places beginning with 'unsure' or 'stepped'. Every player has a choice between *Step* and *Stay*. Transitions *P1* till *P4* defines rewards and punishments. We transformed PNSI for a number of chronons in an EFG, and could find the nash equilibrium using GAMBIT. GAMBIT transforms games in extensive form into games in strategic form, as it is common for solving games [12]. A subset of nash equilibria of the game in strategic form are sequential equilibria in extensive form. Calculated sequential equilibrium of muddy children puzzle is identical to the behavior predicted by epistemic logic.

GAMBIT has an ability to generate a colored image for an EFG. A freely downloadable poster [13] shows the game tree generated over 4 chronons using PNSI for the muddy children puzzle. These 4 chro-nons are needed for calculating the equilibrium in this game. The 4 chronons are the decision of the environment, first iteration for players's decisions, second iteration for players's decisions and the payoff chronon. Due to the fact that one can not use colors in this paper format, we could not include the entire tree in this work. Without colors, the image becomes remarkably harder to conceive. Fig.5 shows the tree for first two chronons (black for players, gray for environment). Annotation 'A:B' at a node means that this node belongs to a set of indistinguishable nodes of the player

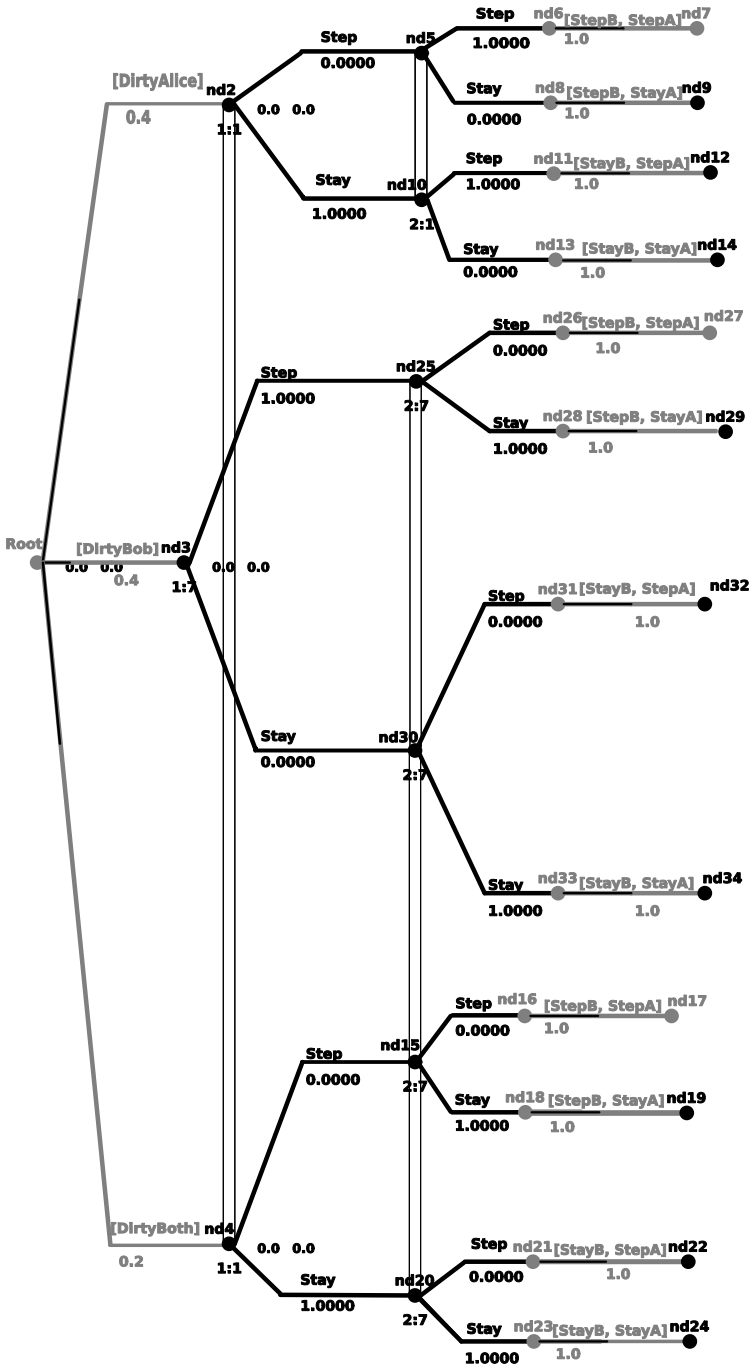


Fig. 5. Muddy Children Puzzle in EFG.

A with the number B. Bob is the first player and Alice is the second. At the second chronon, personal nodes *nd2*, *nd4* and *nd3* belong to Bob and the nodes on the ends of Bob's decisions belong to Alice. Every action of the environment is denoted by transitions, which are to be fired. Every node has a payoff vector (Bob, then Alice). Information sets are depicted by thin parallel lines. Nodes *nd2* and *nd4* are connected, because in these nodes Alice is dirty - these nodes are indistinguishable for Bob. The unique pure strategy equilibrium of this game is depicted on the tree through probabilities at the actions. For instance, the arc from *nd3* to *nd25* has the probability 1. As you also see, the game tree can be continued infinitely, if all players stay at their positions.

6 Conclusion

PNSI has a couple of advantages for modelling timed games. PNSI representation is more compact than EFG. It provides a graphical representation, which is not available using logic-based approaches. It enables modelling of time in a game, what is not available neither in EFG nor in contemporary logic based game description languages. It satisfies both game computing tasks - game solver and game server definition. It provides an ability of game protocoling and persistent game computing.

The only bottleneck is the size of the representation, which is significantly bigger than in logic-based approaches. For instance, if one intends to model chess with a clock, one needs to create 13 places for every cell of the board (12 kinds of pieces plus empty). This makes 832 places only for representing assignment of the board. The other case is representing of payoff matrixes. Every entry in such a matrix needs a transition. But some big payoff matrixes can be summarized by a couple of simple rules. However, PNSI is still useful in practice [14]. As future work, we plan to reproduce our experiences in PNSI for constructing a logic-based representation language for games with time. This new language will have a possibility of graphical representation for understanding of time dependent processes running in a game.

References

1. Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning about Knowledge. The MIT Press, Cambridge, Massachusetts, London, England (1995)
2. Nash, J.: Non-cooperative games. Annals of Mathematics (54) (1951) 286 – 295
3. Russel, S., Norvig, P.: Artificial Intelligence. Pearson Education (2003)
4. Koller, D., Pfeffer, A.: Representations and solutions for game-theoretic problems. Artificial Intelligence **94**(1-2) (1997) 167–215
5. McKelvey, R.D., McLennan, A.M., Turocy, T.L.: Gambit: Software tools for game theory, version 0.2007.01.30. www.gambit-project.org (2007)
6. Genesereth, M.R., Love, N., Pell, B.: General game playing: Overview of the aaai competition. AI Magazine **26**(2) (2005) 62–72

7. Cassez, F., Roux, O.H.: From time petri nets to timed automata. Journal of Systems and Software **79** (2005) 1456–1468
8. Jasper, H., Zukunft, O., Behrends, H.: Time issues in advanced workflow management applications of active databases. In: Active and Real-Time Database Systems, Springer (1995)
9. Meyer, J., van der Hoek, W.: Epistemic Logic for Computer Science and Artificial Intelligence. Cambridge University Press (1995)
10. Tagiew, R.: Multi-agent petri-games. In: CIMCA, IEEE (2008) 130–135
11. Priese, L., Wimmel, H.: Petri-Netze. Springer (2008)
12. McKelvey, R.D., McLennan, A.: Computation of equilibria in finite games. 87–142
13. Tagiew, R., Jasper, H.: Games with time. Poster at IK (2009)
14. Tagiew, R.: Towards a framework for management of strategic interaction. In: ICAART, INSTICC (2009) 587–590