Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 154 (2006) 121–138
www.elsevier.com/locate/entcs

Modeling the Exogenous Coordination of Mobile Channel-based Systems with Petri Nets

Juan Guillen-Scholten[a,1] Farhad Arbab[a,1] Frank de Boer[a,1]
Marcello Bonsangue[b,2,3]

[a] *CWI, Amsterdam, The Netherlands*
[b] *LIACS, Leiden University, The Netherlands*

Abstract

In this paper, we discuss how to *model* systems that communicate through and are coordinated by mobile channels. Mainly, we focus on modeling the exogenous coordination behavior imposed by these channels. We use Petri Nets as our modeling language, for they provide a graphically and mathematically founded modeling formalism. We give Petri Nets for a set of mobile channel types. This allows us to construct models of applications, by taking the Petri Net of each component and each mobile channel, and composing them together. For this purpose, we define a special Petri Net composition function. We also discuss analysis and simulation of these models and their exogenous coordination behavior.

Keywords: Distributed Mobile Channels, Petri nets, Coordination, Composition

1 Introduction

In the MoCha Framework [6] components and processes are coordinated by *mobile channels*. A mobile channel is a coordination primitive that allows anonymous point-to-point communication, enables dynamic reconfiguration of channel connections in a system, and provides exogenous coordination.

[1] Email: {juan, farhad, frb}@cwi.nl
[2] Email: marcello@liacs.nl
[3] The research of Dr. Bonsangue has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences.

doi:10.1016/j.entcs.2005.12.036

Mobile channels are interesting for all kinds of entities that need to be coordinated, but they are specially interesting for Component Based Software. As we show in [6], they provide a highly expressive data-flow architecture for the construction of complex coordination schemes, independent of the computation parts of components.

The implementation of the MoCha Framework, the MoCha middleware, is suitable for any centralized or decentralized distributed network where we exogenously coordinate the components by means of mobile channels. For example, in [7] we show the benefits of using MoCha for P2P networks.

The purpose of this paper is to give the means for modeling systems that communicate through and are coordinated by these mobile channels. Our main goal is to model, analyze, and simulate the exogenous coordination of these systems. Therefore, we need a *modeling language* with the following features: (1) The language is widely used in both the academic world and industry. (2) It contains well-defined semantics with clear theoretical foundation. (3) It provides analysis of models, like all modeling languages do. (4) It provides model simulation. (5) The language is easy to understand as well as the models that it produces. (6) And last, but not least, there is enough tool-support for this language.

A modeling language that fulfills above requirements is *Petri Nets*. *Petri Nets*, named after their creator Petri [12], provide a graphically and mathematically founded modeling formalism for the concurrent behavior of systems. They offer precise semantics and a theoretical foundation [15].

By providing mobile channels specified in the *Petri Nets* model formalism, we can model systems that use our channels with this formalism. This means that, besides being able to model systems, we automatically get the following advantages: extensive theoretical support, ease of usage, model analysis, simulation of the models, immediate application in different areas, and extensive tool support. Furthermore, while it is not the main objective of this paper, since Petri Nets models have clear and precise semantics, they also automatically give semantics to our mobile channels. The interested reader can compare the mobile channel semantics given in [8] which concentrate on process *interaction*, with the semantics given in this paper which concentrate on *concurrency*.

In section 2, we give a brief overview of the MoCha Framework. In section 3, we give a short introduction to Petri Nets. In section 4, we show how to model systems that use our mobile channels. Here we give the Petri Net models for a set of mobile channel types, discuss the minimal behavior that components need to implement in Petri Nets to use these channels, and give a composition function for constructing systems. In section 5, we give an

example of such a composed system, and discuss analysis and simulation. In section 6, we discuss the complexity of our approach and the need for tools. We conclude with section 7.

2 MoCha

A channel in MoCha (see figure 1) consists of a pair of two distinct ends: usually (*source*, *sink*) for most common channel-types, but also (*source*, *source*) and (*sink*, *sink*) for special types. These channel-ends are available to the components of an application. Components can *write* by inserting values into the source-end, and *take* by removing values from the sink-end of a channel; the data-flow is locally *one way*: from a component into a channel or from a channel into a component.

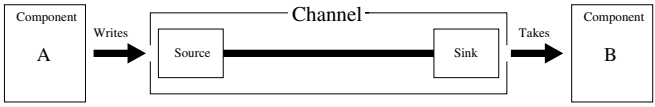


Fig. 1. General View of a Channel.

Channels are *point-to-point*, they provide a directed virtual path between the (remote) components involved in the connection. Therefore, using channels to express the communication carried out within an application is *architecturally very expressive*, because it is easy to see which components (potentially) exchange data with each other. This makes it easier to apply tools for the analysis of dependencies and data-flow analysis in an application.

Channels provide *anonymous communication*. This enables components to exchange messages with other components without having to know *where* in the network those other components reside, *who* produces or consumes the exchanged messages, and *when* a particular message was produced or will be consumed. Since the components do not know each other, it is easy to update or exchange any one of them without the knowledge of the components at the other side of the channels it is connected to. This provides a simple mechanism for composition of components that are decoupled in space and time.

The ends of a channel are *mobile*. We introduce here two definitions of mobility: physical and logical. The first is defined as physically moving a channel-end from one location to another location in a distributed system, where location is a *logical address space* wherein components execute. The second, logical mobility, is typically defined in the π-calculus as the ability of passing channel(-end) identities through channels themselves to other components in the application; i.e., spreading the knowledge of channel(-ends) references by means of channels. This is possible in MoCha. However, in this

paper we define logical mobility as the changing of channel connections among components in a system by means of *connect* and *disconnect* operations. Both physical and logical mobility are supported by MoCha.

Mobility allows dynamic reconfiguration of channel connections among the components in an application, a property that is very useful and even crucial in systems where components are mobile. A component is called mobile when, in a distributed system, it can move from one location (where its code is executing) to another. Because the communication via channels is also *anonymous*, when a channel-end moves, the components at the other side of the channel are not aware nor affected by this movement.

Channels provide transparent *exogenous coordination*. Channels allow several different types of connections among components without them knowing which channel types they deal with. Only the creator of the connection knows the type of the channel. This makes it possible to coordinate components from the outside (exogenous), and thus, change an application's behavior without having to change the code of its components.

3 A Short Introduction into Petri Nets

Petri Net is actually a generic name for a whole class of net-based models which can be divided into three main layers [17]. The first layer is the most fundamental and is especially well suited for a thorough investigation of foundational issues of concurrent systems. The basic model here is that of Elementary Net Systems [16], or *EN systems*. The second layer is an "intermediate" model where one folds some repetitive features of EN systems in order to get more compact representations. The basic model here is Place/Transition Systems [2], or *P/T systems*. Finally, the third layer is that of high-level nets, where one uses essentially algebraic and logical tools to yield "compact nets" that are suited for real-life applications. Predicate/Transition Nets [5] and Colored Petri Nets [10] are the best known high-level models.

Any Petri Net of the three layers above is suitable to model a system. Moreover, any Petri Net of any layer can be transformed/translated into a Petri Net of another layer [4]. Examples of translation are given in the work of Engelfriet [3] and Jensen [11]. We specified all the channel types of the MoCha Framework using the Place/Transition Petri Nets. In contrast with the high-level Petri Nets, these kind of Petri Nets are at the right level of abstraction with clear non-changeable semantic rules and constructs. However, in this paper, for simplicity, we use the Elementary Net Systems Petri Nets. This last kind of Petri Nets are easier to use, for their theory is a little bit more simpler. Furthermore, the topology of the synchronous channel types of both models

are structural equivalent. For the asynchronous types that we introduce in this paper, it doesn't pay off to use the Place/Transition Petri Nets for all channel types.

3.1 Elementary Net Systems

We give a short introduction of *Elementary Net Systems* (*EN system* in short). We restrict ourselves to the definitions that we need in this paper. For an extensive introduction that also covers several properties of EN systems, equivalences, and EN analysis we refer to the tutorial given in [17]. A net is the most basic definition of all Petri Nets:

Definition 3.1 A *net* is a triple $N = (P, T, F)$ where
(1)P and T are finite sets with $P \cap T = \emptyset$,
(2)$F \subseteq (P \times T) \cup (T \times P)$,
(3) for every $t \in T$ there exist $p, q \in P$ such that $(p, t), (t, q) \in F$, and
(4) for every $t \in T$ and $p, q \in P$, if $(p, t), (t, q) \in F$, then $p \neq q$.

The elements of P are called *places*, the elements of T are called *transitions*, elements of $X = P \cup T$ are called *elements* (of N), and F is called the *flow relation* (of N).

Each place $p \in P$ can be viewed as representing a possible local state of a system. At each moment in time a set of local states (places) participate in the global state of the system. We call such a set of places a *configuration*. Graphically, we denote the places that are part of a configuration with a token; a small black filled circle.

Definition 3.2 A *configuration* C of a net $N = (P, T, F)$ is a subset of P.

Thus, a *configuration* C of a net is a subset of P where each place contains a token. We now define a elementary net system as given in [17]:

Definition 3.3 Definition: An EN system is a quadruple
$M = (P, T, F, C_{in})$ where:
(1)(P, T, F) is a net and
(2)$C_{in} \subseteq P$ is the *initial configuration*

Every transition in an EN system can perform an action called *fire*. This action takes a token from all the input places and places a token to each output place of the transition. This action represents a sequential step of a system. However, for this to happen all the input places of the transition must have a token and its output places must be empty, since a place can have at most only one token at the same time.

Definition 3.4 Let $M = (P, T, F, C_{in})$ be an EN system and let $t \in T$.
(1) ${}^\bullet t$ are the input places of t, and $t^\bullet$ the output places of t.
(2) Let $C \subseteq P$ be a configuration. Then t *has concession* in C (or t can be fired in C) if ${}^\bullet t \subseteq C$ and $t^\bullet \cap C = \emptyset$, written as t **con** C.
(3) Let C,$D \subseteq P$. Then t *fires from* C *to* D *if* t **con** C and $D = (C -^\bullet t) \cup t^\bullet$, written as $C[t > D$; t is also called a sequential step from C to D.

4 Modeling Mobile Channel based Systems

In order to model systems that use mobile channels, we need to model the channels in Petri Nets, from now on PN, first. We, then, take the PN of the components and compose them together with the PN of our mobile channels.

Next, we give the interface of these PN mobile channels. Afterward, we give the interface and minimal behavior that components need to implement in order to use our channels. We, then, specify a PN composition function σ. Besides composing, this function implicitly models the *connect* and *disconnect* channel operation (by using the inverse function as well). Finally, we give the EN systems for a representative set of mobile channel types.

4.1 Mobile Channel Interface

The EN and P/T-net mobile channel systems that we present in this section have all the *same interface* from the point of view of the components of a system. We give this interface in figure 2. Each channel-net has an internal part that is determined by it's type, and an interface that is common to all channel types consisting of four *interface places*, two for each channel-end. We graphically denote these places by marking an extra symbol, *I*, on the outside of the circle. The *interface places* are part of a protocol to ensure that all *write* and *take* operations are *blocking*; i.e. an active entity performing such an operation blocks until the operation succeeds and terminates.

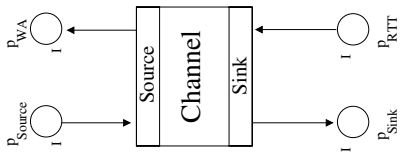


Fig. 2. PN Mobile Channel Interface

The places p_{Source} and p_{WA} constitute the interface of the source channel-end. A component that wants to perform a *write operation* on this end, puts a token into place p_{Source}. This token represents the fact that a data element is available but has not yet been accepted by the channel. In other words, the

write operation is pending between the component and the source channel-end. When the token is removed from this place by the channel, it means that the channel is processing the *write operation*. Upon completion of the operation, the channel puts a token in the interface place p_{WA}; a write acknowledgment.

The places p_{Sink} and p_{RTT} constitute the interface of the sink channel-end. A component that wants to perform a *take operation* on this end, puts a token in place p_{RTT} (Ready To Take). This token reveals the desire and willingness of the component to take a data element from the channel. However, the channel knows that there is a component waiting (and wanting) to *take* an element only when the token in p_{RTT} successfully "enters" the channel due to a fire action. The channel terminates the *take operation* by putting a token into the p_{Sink} interface place. The component, then, can take the token from this place.

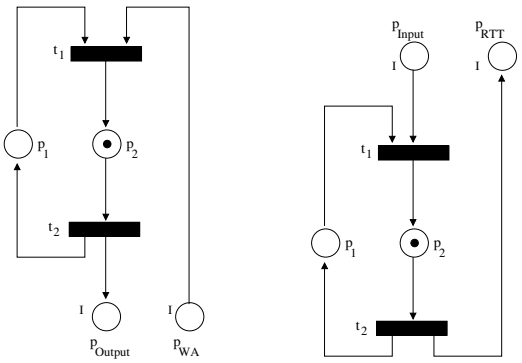
We don't explicitly model a source- and a sink-end in the mobile channel nets. A channel-end is implicitly modeled by its two interface places and the internal transitions where these places are either input or output of. Observe, that the semantics of the *write* and *take* operations are analogous with the ones defined in [8].

4.2 Component Interaction

The components of a system interact with the mobile channels thought the interface that we defined above. From the point of view of the channels, a component consists of one or more active entities (threads or processes) that perform *write* and *take* operations. In figure 3 we give the EN systems of two single entity components. They represent the minimal behavior that components need to implement regarding the *write* and *take* operations toward channels; i.e. they implement their side of the *blocking* protocol as described in section 4.1.

Figure 3(a) shows the PN of a simple writer. This net has two interface places that are meant for composition with channels:$\{p_{Output}, p_{WA}\}$. The initial configuration of the net is $\{p_2\}$. The writer starts the *write operation* by executing $\{p_2\}[t_2 > \{p_{Output}, p_1\}$. At this point it is blocked for it must wait until it receives a write acknowledgment; i.e. a token is placed in p_{WA}. If the writer is interacting with a source channel-end, at the time that it receives the acknowledgment the token in place p_{Output} is already gone. Therefore, we end up with the configuration $\{p_1, p_{WA}\}$. The writer ends the operation by performing the sequential step $\{p_1, p_{WA}\}[t_1 > C_{in}$. At this point, it may start writing again.

Figure 3(b) shows the PN of a simple taker. This net has also two interface places that are meant for composition with channels: $\{p_{Input}, p_{RTT}\}$. The



(a) A Writer (b) A Taker

Fig. 3. A Writer and a Taker EN System

initial configuration of this net is $\{p_2\}$. The taker starts the *take operation* by executing $\{p_2\}[t_2 > \{p_1, p_{RTT}\}$. At some point in time the channel it is interacting with takes the token of p_{RTT}, and later on, it puts a token back in place p_{Input}. The resulting configuration is $\{p_1, p_{Input}\}$. The taker ends the operation by performing $\{p_1, p_{Input}\}[t_1 > C_{in}$. At this point, it may start taking again.

4.3 PN Composition of Components and Channels

We have introduced the interface of channels and the interface of components toward channels. We now need to compose components and channels together. There are several construction strategies possible. Our major requirement is that such strategy is *compositional*. One should be able to distinguish the individual components and channels in the composed system, and it must be easy to decompose and rearrange the system; i.e. updating and replacing components and channels without having to change the rest of the system. Therefore, for example, we cannot do composition and optimize the resulting PN for it may not be possible to decompose after many composition steps anymore. Our strategy is, then, to do composition on the *interface places*. This way, we don't have to change the internals of components and channels. It is easy to recognize the individual parts. And, decomposition is also clear and easy to do. For this purpose we define a composition function that mergers, or concatenates, *interface places*:

Definition 4.1 We define the composition function $\sigma : (X_1, P_1, X_2, P_2) \longrightarrow Y$ where,
(1) X_1, X_2 and Y are EN systems
(2) P_1 and P_2 are ordered finite set of places, with $P_1 \subseteq P_{X_1}$, $P_2 \subseteq P_{X_2}$ and

$|P_1| = |P_2|$. Typical elements of these sets are $p_1 \in P_1$ and $p_2 \in P_2$.
We construct the new Petri Net EN System Y as follow,
(3) We rename the places, transitions and flow relations of nets X_1 and X_2 that have the same name.
(4) $P_Y = (P_{X_1} \setminus P_1) \cup (P_{X_2} \setminus P_2) \cup P_{new}$, where
$\forall(pairs(p_{1-i}, p_{2-i}))\ \exists p_{new-i} \in P_{new}$,
with i as an index from 1 to $|P_1| = |P_2|$, and $|P_1| = |P_2| = |P_{new}|$,
(5) $T_Y = T_{X_1} \cup T_{X_2}$,
(6) $F_Y = (F_{X_1} \cup F_{X_2} \cup F_I) \setminus F_{Rem}$, where
$\forall(i \in 1\,to\,|P_k|)$ if $(p_{k-i}, t) \in F_{X_k}$ then $(p_{k-i}, t) \in F_{Rem} \wedge (p_{new-i}, t) \in F_I$,
$\forall(i \in 1\,to\,|P_k|)$ if $(t, p_{k-i}) \in F_{X_k}$ then $(t, p_{k-i}) \in F_{Rem} \wedge (t, p_{new-i}) \in F_I$,
with $k = \{1, 2\}$, $p_{k-i} \in P_k$ and $p_{new-i} \in P_{new}$, both with index i.
(7) $C_{in-Y} = (C_{in-X_1} \setminus P_1) \cup (C_{in-X_2} \setminus P_2) \cup C_{in-new}$, where
$(\forall i \in 1\,to\,|P_{new}|)$if $p_{k-i} \in P_k \wedge p_{k-i} \in C_{in-X_k}$ then $p_{new-i} \in C_{in-new}$,
with $k = \{1, 2\}$.

The function σ takes EN Systems as parameters, X_1 and X_2. The function also takes two set of places, P_1 and P_2, that correspond to the interface places of respectively X_1 and X_2 that we want to compose. The result of the function is a new EN System Y, that is constructed as follow: (3) We rename all the places, transitions, and relation flows of nets X_1 and X_2 that cause name conflicts due to the composition. (4) Each place of X_1 and X_2 is present in Y, except for the interface places of P_1 and P_2. Each pair $\{p_1, p_2\}$ of these places, that are related to each other for having the same index number, are substitute for a new place, p_{new}, that is inserted in Y. (5) The composition is done on interface places so the transitions of Y are just the union of the ones in X_1 and X_2. (6) Every flow relation present in either X_1 or X_2 is also present in Y. The flow relations that involve the interface places in P_1 and P_2, represented in F_{Rem}, are changed to be involved in the new added places of point 3, represented in F_I. (7) The C_{in} of Y is the union of the ones of X_1 and X_2. However, the places of P_1 and P_2 may also be present at the initial configurations of these two last EN systems. Since these places do not exist anymore in Y, we add their corresponding new places from P_{new} into the initial configuration.

We can now compose components and channels using our function σ. For example, we obtain the PN-system $Comp$ of figure 4, by applying the σ function to the writer-, taker component and a channel (which we defined previously): $Comp = \sigma(Taker, \{p_{Input}, p_{RTT}\}, Tmp, \{p_{Sink}, p_{RTT}\})$, $Tmp = \sigma(Writer, \{p_{Output}, p_{WA}\}, Channel, \{p_{Source}, p_{WA}\}$. In this figure we take the general channel definition. Later on, we give several PN-systems for different

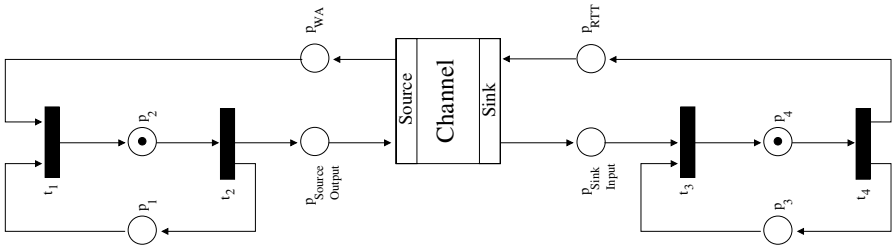
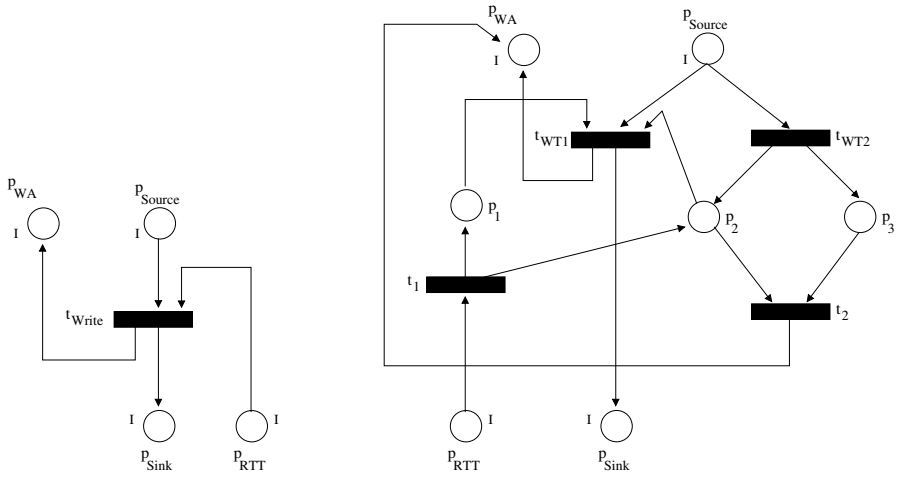


Fig. 4. Composing a Writer and a Taker with Mobile Channels



(a) Synchronous Channel

(b) Lossy Synchronous Channel

Fig. 5. The Synchronous and Lossy Synchronous Channel EN Systems

mobile channel types.

The composition function σ implicitly models the operation *connect*. Our components don't issue a *connect* request, and we don't keep any administration concerning connect. However, if a component is composed with a channel by σ we regard this component connected to the appropriate channel-end. Analogous to *connect*, we implicitly model *disconnect* with the function σ^{-1}; the inverse of σ.

4.4 A Set of EN and P/T-net Mobile Channels Systems

We now take a representative set of mobile channel types and give an EN system for each of them.

4.4.1 The Synchronous Channel Type

With a synchronous channel the I/O operations of both ends are synchronized; the I/O operations atomically succeed. Figure 5(a) shows the EN system of this channel. The internals of this channel type is just a transition t_{Write} that synchronizes the four interface places as defined in section 4.1. The places p_{Source} and p_{RTT} are input places of transition t_{Write}. Therefore, only when both the writing and the taking components have each inserted a token in these places, the I/O operations atomically succeeds (at the same time); each component inserts a token to its corresponding place as described in section 4.2. We give the sequential firing step: $\{p_{Source}, p_{RTT}\}[t_{Write} > \{p_{Sink}, p_{WA}\}$. At the end a token is inserted in the places p_{Sink} and p_{WA}. This indicates the end of the I/O operations, from the point of view of the channel.

4.4.2 The Lossy Synchronous Channel Type

With the lossy synchronous channel, if there is no I/O operation performed on the sink channel-end while writing a value to the source-end, the *write operation* always succeeds but the value gets lost. In all other cases, the channel behaves like a normal synchronous type.

Figure 5(b) gives the EN system for this channel type. There are two paths for a successful *write operation*. One, is determined by the t_{WT1} transition and exhibits the behavior of a synchronous channel. The other, is determined by the t_{WT2} transition and exhibits the lossy behavior. The choice between the first or the second path depends whether there is a component waiting to take a value or not, this is symbolized by the presence of a token in place p_{RTT}. However, the channel is not aware of this intention yet. Only when firing transition t_1 does the channel know that a component is ready to accept a value: $\{p_{RTT}\}[t_1 > \{p_1, p_2\}$.

If there is a token in place p_{Source}, there is a component trying to write, transition t_{WT1} fires when there is a token in places p_1 and p_2; there is a component waiting to take. At the same time, transition t_{WT2} is blocked because of the token in place p_2. Therefore, the written value synchronously flows from p_{Source} to p_{Sink}: $\{p_{Source}, p_1, p_2\}[t_{WT1} > \{p_{Sink}, p_{WA}\}$.

However, if there are no tokens in places p_1 and p_2, there is no component to take, transition t_{WT2} fires and the value gets lost while the *write operation* succeeds. Observe that, the transition t_{WT1} cannot fire due to the lack of a token in place p_1. There is no need to model a garbage collector to delete the token value since the firing of transition t_2 already takes care of this. We give the sequential steps of the lossy path: $\{p_{Source}\}[t_{WT2} > \{p_2, p_3\}[t_2 > \{p_{WA}\}$.

4.4.3 The FIFO and FIFO n Channel Type

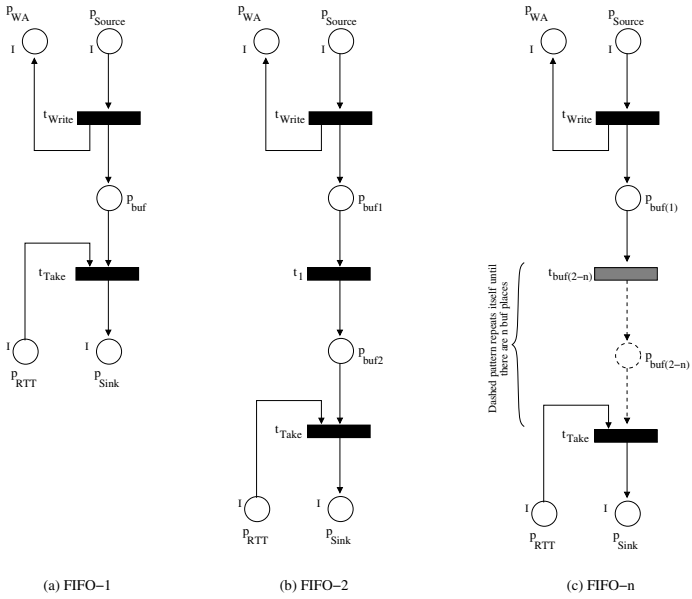
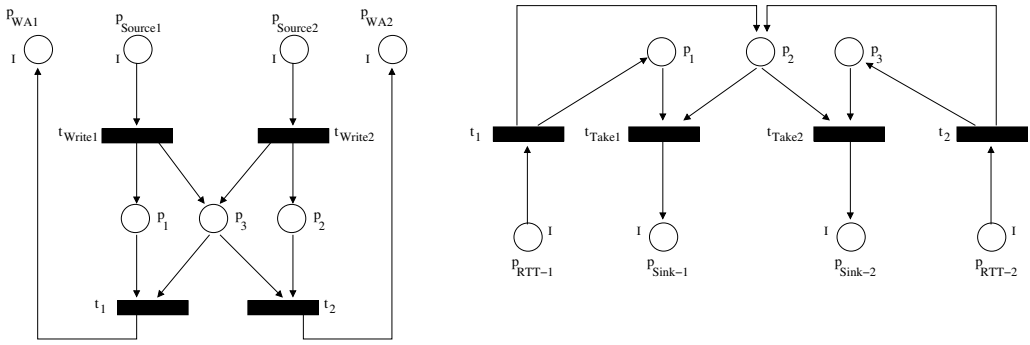


Fig. 6. The FIFO 1, 2 and n Channel EN Systems

With an asynchronous FIFO channel type the I/O operations that are performed on both channel-ends are done in an asynchronous way. Values written into the source channel-end are internally stored in a buffer until taken from the sink-end. Figure 6(a) shows the EN system of a FIFO-1 channel. As one could expect, the internal buffer of capacity one is modeled by place p_{buf}. We write a value into the channel by performing the sequential step $\{p_{Source}\}[t_{write} > \{p_{buf}, p_{WA}\}$, and we take a value by performing $\{p_{buf}, p_{RTT}\}[t_{Take} > \{p_{Sink}\}$. In figure 6(b) we give a FIFO-2 EN system channel. Naturally, it contains two buffer places. Figure 6(c) gives the general scheme of a FIFO EN system channel with buffer capacity n. Observe, that if n is *unlimited*, the unbounded FIFO channel type, we get an EN system with infinite places. To avoid this, we work with a P/T-net system where we have a single buffer place with unlimited capacity. However, for space saving reasons, we omitted this kind of PN in this paper.

4.4.4 The Asynchronous Drain Channel Type

The asynchronous drain channel type has two source channel-ends. Furthermore, the I/O operations performed on the ends of this channel succeed one at a time exclusively. So the *write* operations on its two ends *never succeed simultaneously*. This is reflected in the net we give in figure 7(a). Place p_3



(a) Asynchronous Drain

(b) Asynchronous Spout

Fig. 7. The Asynchronous Drain- and Spout EN Channel Systems

makes sure that either transition t_{Write1} or transition t_{Write2} fires, but not both at the same time.

Let's assume that there are two simultaneous writes available; the net configuration is $\{p_{source1}, p_{source2}\}$. Then, we can perform the *write operation* on the left source-end first: $\{p_{source1}, p_{source2}\}[t_{Write1} > \{p_{source2}, p_1, p_3\}$, at this configuration transition t_{Write2} is blocked, $\{p_{source2}, p_1, p_3\}[t_1 > \{p_{source2}, p_{WA1}\}$. Or, we can perform the *write operation* on the source-end at the right first: $\{p_{source1}, p_{source2}\}[t_{Write1} > \{p_{source1}, p_2, p_3\}$, at this configuration transition t_{Write1} is blocked, $\{p_{source1}, p_2, p_3\}[t_1 > \{p_{source1}, p_{WA2}\}$. However, we can never perform both *write operations* at the same time, since we cannot fire transitions t_{Write1} and t_{Write2} concurrently.

4.4.5 The Asynchronous Spout Channel Type

The asynchronous spout channel type has two sink channel-ends. Furthermore, the I/O operations performed on the ends of this channel succeed one at a time exclusively. So the *take operations* on its two ends never succeed simultaneously. This is reflected in the net we give in figure 7(b). Place p_2 makes sure that either transition t_{Take1} or transition t_{Take2} fires, but not both at the same time.

Let's assume that there are two simultaneous takes available; the net configuration is $\{p_{RTT1}, p_{RTT2}\}$. Then, we can perform the *take operation* on the left sink-end first: $\{p_{RTT1}, p_{RTT2}\}[t_1 > \{p_{RTT2}, p_1, p_2\}$, at this configuration the token in place p_2 blocks the firing of transition t_2, $\{p_{RTT2}, p_1, p_2\}[t_{Take1} > \{p_{RTT2}, p_{Sink1}\}$. Or, we can perform the *take operation* on the sink-end at the right first: $\{p_{RTT1}, p_{RTT2}\}[t_2 > \{p_{RTT1}, p_2, p_3\}$, at this configuration the token in place p_2 blocks the firing of transition t_1, $\{p_{RTT1}, p_2, p_3\}[t_{Take2} >$

$\{p_{RTT1}, p_{Sink2}\}$. However, we can never perform both *take operations* at the same time, since we cannot fire transitions t_{Take1} and t_{Take2} concurrently.

5 Analysis and Simulation

We now know how to model systems that use our mobile channels by means of Petri Nets. In this section we discuss the analysis and simulation of these models. Figure 8 models a system that consists of a write-, and a take component as defined in section 4.2. These two components interact through a *lossy synchronous* channel, as defined in section 4.4. We get the system by composing the Petri Nets of each separate entity by means of the function σ: $\sigma(Taker, \{p_{Input}, p_{RTT}\}, Tmp, \{p_{Sink}, p_{RTT}\})$, $Tmp = \sigma(Writer, \{p_{Output}, p_{WA}\}$,-$LossySynchronous, \{p_{Source}, p_{WA}\}$.

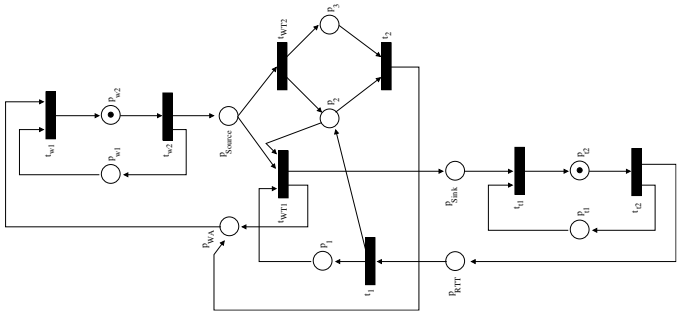


Fig. 8. An Example of Composition

We can *simulate* the system by playing the "token game". This game consists of firing transitions, when possible, to get the system from one state into the other. If we cover all possible firing sequences, we get all the possible states of the system, and thus all the possible *exogenous coordination* of this system. In figure 9 we give the sequential configuration graph of figure 8. For a precise definition of (sequential) configurations graphs, see [17].

Besides simulation, we can also *analyze* the *exogenous coordination* behavior of the models for desired, or undesired, properties and features. Petri Nets offer extensive analysis of its models. The most common analysis features include *causality*, *concurrency*, *conflicts*, *confusions*, *deadlocks*, and *equivalence*. The first, studies the *causality* between the events of a system. The second, analyzes which system events are *concurrent* at the same moment in time. The third, analyzes which events are *conflicting* at the same moment in time. Sophistic interaction between concurrency and conflicts can lead to, the fourth, *confusions*; events that are concurrent can become conflict and vice-versa. The first four features make it possible to reason about, the fifth,

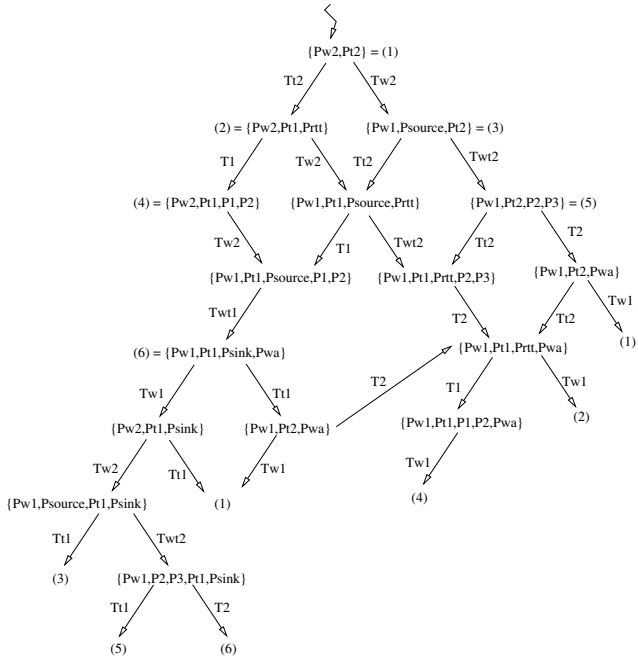


Fig. 9. The Sequential Configuration Graph of Figure 8

deadlocks in a system. And finally, Petri Net analysis include *equivalence*. Usually, similarity of systems is build upon the notion of morphism.

For example, we can analyze the *exogenous behavior* of our example model to find *concurrent steps*. For this purpose, we can look at its sequential configuration graph, given in figure 9. Basically, every diamond shape in the figure represents a concurrent step. The first possible concurrent step is $\{p_{w2}, p_{t2}\}[\{t_{t2}, t_{w2}\} > \{p_{w1}, p_{t1}, p_{source}, p_{rtt}\}$; we can arrive from configuration $\{p_{w2}, p_{t2}\}$ to configuration $\{p_{w1}, p_{t1}, p_{source}, p_{rtt}\}$, by first firing transition t_{w2} and then transition t_{t2}, or vice-versa. For the precise definition of *concurrency* and other analysis we refer to [17,2].

6 Complexity and the Need for Tools

The system we modeled in figure 8 is quite small and simple. Analyzing and simulating this system "by hand" is possible, but already not a pleasant task. For example, look at the size of figure 9. A real application consists of many components and many mobile channels. Therefore, modeling such an application quickly results in a big Petri Net model that is not human-

tractable anymore.

One of the reasons is that the construction strategy we chose is compositional (see section 4.3). Therefore, we cannot optimize the PN systems we obtain, because then we could not recognize its constituent parts anymore. However, the main reason for this explosion is that EN and P/T systems do not offer high-level constructs such as constrains. This obliges us to explicitly implement everything we need. For example, we explicitly implemented a protocol for modeling *blocking* operations (see section 4). If our goal is to produce human-readable models, then, the approach we take in this paper using EN and P/T systems is not really suitable. A better choice, for example, is a higher-level Petri Net like Colored Petri Nets [10], or, to use the MoCha-pi calculus given in [8].

However, the Petri Net models that our approach produces are very suitable for verification and simulation using tools. This is due to the fact that, EN and P/T systems have clear non-changeable semantic rules and constructs, and, that our Petri Net models explicitly encode low-level technical details that are otherwise not specified. Fortunately, there are many tools available for EN and P/T systems. For an extensive list of these tools we refer to the state-of-the-art work in [13].

We are experimenting with the *Platform Independent Petri Net Editor* (*PIPE*) tool [1]. We chose this tool because, it is *free of charge*, *platform-independent*, offers *simulation* and *analysis* modules, and gives XML support.

7 Conclusions

In this paper, we showed how to *model* distributed systems that communicate through and are coordinated by mobile channels. We focused on modeling the exogenous coordination of these systems. Examples of such systems are Component Based and P2P networks, as explained in [6,7]. The *modeling language* we chose to use is Petri Nets. We discussed the modeling of components, mobile channels and their composition into distributed systems. We discussed the analysis and simulation of these Petri Net models. In particular, the analysis of *causality*, *concurrency*, *conflicts*, *confusions*, *deadlocks*, and *equivalence* of the *exogenous coordination* of these models. We, also, discussed the negative and positive points of the approach we take for mapping MoCha into Petri Nets. On the negative side, the models that we produce quickly become intractable for humans. On the positive side, the low-level semantics of these models make them very suitable for simulation and analysis using one of the many tools that are available for the kind of Petri Nets we use.

Petri Nets have been used in many different application areas. As a result

there is a high degree of expertise in the modeling field. Interesting for this paper is the work on Web Services composition presented in [9]. MoCha channels are suitable for Web Services, and we are intend to use the channel Petri Nets of this paper for this purpose as well. In [14], Petri Nets are used to model distributed algorithms. Although, we did not introduce an explicit notion of location, our channel Petri Nets define a distributed implementation of the MoCha Framework channel types.

References

[1] J.D. Bloom, *Platform Independent Petri-Net Editor (PIPE)*, electronic manual, 2005, available on-line at http://petri-net.sourceforge.net

[2] J. Desel, and W. Reisig, *Place/Transition Petri Nets*, Lecture Notes in Computer Science, Vol. 1491: Lectures on Petri Nets I: Basic Models, pages 122-173. Springer-Verlag, 1998.

[3] J. Engelfriet, *Branching processes of Petri nets*, Acta Informatica Volume 28, Issue 6, pages 575 - 591, Springer-Verlag, 1991.

[4] J. Engelfriet, *Private Correspondence*, LIACS, 2004.

[5] H. J. Genrich, *Predicate/transition nets*, Advances in Petri nets 1986, part I on Petri nets: central models and their properties, pages: 207 - 247, Springer-Verlag, 1987.

[6] J.V. Guillen-Scholten, F. Arbab, F.S. de Boer and M.M. Bonsangue, *A Channel-based Coordination Model for Components*, A. Brogi and J. Jacquet, editors, Proceedings of the 1st International Workshop on Foundations of Coordination Languages and Software Architectures, ENTCS 68.3, Elsevier Science, 2002.

[7] J.V. Guillen-Scholten, F. Arbab, *Coordinated Anonymous Peer-to-Peer Connections with MoCha*, Proc. of the 4th International Workshop on Scientific Engineering of Distributed Java Applications (FIDJI 2004), Luxembourg, 24-25 November 2004, Lecture Notes in Computer Science, Vol. 3409, pp. 68-77, Springer-Verlag, January 2005.

[8] J.V. Guillen-Scholten, F. Arbab, F.S. de Boer, and M.M. Bonsangue, *MoCha-pi, an Exogenous Coordination Calculus based on Mobile Channels* Proceedings of the 2005 ACM Symposium on Applied Computing, Santa Fe, New Mexico, USA, March 13-17, 2005.

[9] R. Hamadi, B. Benatallah, *A Petri net-based model for web service composition*, Proceedings of the Fourteenth Australasian database conference on Database technologies 2003, Volume 17, pages: 191 - 200, Australia, 2003.

[10] K. Jensen, *A Brief Introduction to Coloured Petri Nets*, Tools and Algorithms for the Construction and Analysis of Systems. Proceeding of the TACAS'97 Workshop, Enschede, The Netherlands 1997, Lecture Notes in Computer Science Vol. 1217, Springer-Verlag 1997.

[11] K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997. ISBN: 3-540-60943-1.

[12] C.A. Petri, *Nets, Time and Space.* Theoretical Computer Science, 153:348, 1996.

[13] Petri Nets World, *Petri Nets Tool Database*, Available on-line at http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/, 2005.

[14] W. Reisig, *Elements of Distributed Algorithms: Modeling and Analysis with Petri nets*, Springer-Verlag, 1998.

[15] W. Reisig. G. Rozenberg (Eds.),*Lectures on Petri Nets I: Basic Models*, Advances in Petri Nets, Lecture Notes in Computer Science, vol. 1491, Springer-Verlag, 1998.

[16] G. Rozenberg, *Behaviour of Elementary Net Systems*, Lecture Notes In Computer Science, Vol. 254, pages 60-94, Springer-Verlag, 1986

[17] G. Rozenberg and J. Engelfriet, *Elementary Net Systems*, Lecture Notes in Computer Science, v. 1491, Springer-Verlag, 12-121, 1998.