UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA
Dipartimento di Informatica Sistemistica e Comunicazione
Facolt di Scienze Naturali Fisiche e Naturali
Dottorato di Ricerca in Informatica - XXIII Ciclo
Anno Accademico 2009-2010

Hypernets: a Class of Hierarchical Petri Nets

Marco Mascheroni
Ph.D. Thesis

Thesis Advisors: Prof.ssa Lucia Pomello
Thesis Coadvisors: Dott. Luca Bernardinello
Thesis Tutor: Prof. Flavio De Paoli
Ph.D. Program Coordinator: Stefania Bandini

Acknowledgements

First and foremost, I would like to thank my advisors Prof. Lucia Pomello and Prof. Luca Bernardinello for their guidance, encouragement and patience. They help me in these years and in particular, encourage my scientific interests. It was a real pleasure to work with them.

I am also grateful to Prof Daniel Moldt, and Prof. Rüdiger Valk for their hospitality at the TGI department of the university of Hamburg, and for all the interesting ideas they gave me during my period of sojurn in Hamburg.

I am particularly grateful to all my colleagues: Elisabetta from the MC3 lab, Thomas, Jos and Lars from the TGI depratment of Hamburg, and Fabio from the physics department.

I would also like to express acknowledgement to all my friends, particularly to Erisa, "house 119" flatmates, and the "friends of the train": with them I spent a lot of nice moments in these years.

Finally, I would thank my family, for their continual support, encouragement and for giving me the freedom to pursue my own interests. I cannot be grateful enough for their untiring support and unconditional belief in me.

Last but not the least, I could not have done any of this without Michela that gave me hope whenever I was down, strength when I felt weak and unquestioning love all along.

And I want to thank all the others I probably forget in this list...

Abstract

The interest of this thesis is on modeling systems of mobile agents, systems composed of several open and autonomous components which can interact and move inside one or more environments.

Several proposal for modeling mobility have been introduced. They can be roughly divided in two categories: Petri net based formalisms, and process algebra based formalisms. In this thesis, Petri net formalisms will mostly be considered, with particular care to formalisms which use the nets-within-nets paradigm. In models compliant to this paradigm the tokens of a Petri net can be nets themselves. Since systems of mobile agents exhibit a nesting structure, it seems natural to use this paradigm.

In particular, the focus of this dissertation is on the hypernet model, whose main characteristic is that it has a limited state space, characteristic that make it suitable to be analyzed using well known Petri net techniques.

The thesis topics range from theoretical aspects of the model to more practical issues.

From a theoretical point of view an extension of the model is introduced. It is proved that this extension preserves all the good properties of the basic model. It is also studied how to apply the well known unfolding technique to this model.

From a practical point of view is is shown how the nets-within nets paradigm can be used to model systems based on the Grid infrastructure. Moreover, a tool which allow to draw and to analyze an hypernet is discussed.

Contents

List of Figures

Chapter 1

Introduction

Systems of mobile agents have attracted a lot of attention in the computing community. They are systems composed of several *open* and *autonomous* components, called *agents*, embedded in a *local* environment, called *location*, in a hierarchical way. Each agent is open in the sense that it can interact with the environment and modify its behavior depending on the past interactions. Autonomy means that an agent is also able to make autonomous choices, and to decide its own fate. On the other hand the environment interacts with the agents by offering some services, and at the same time by restricting the behavior of these agents denying (or not providing) other services. An important point of mobility is that there can be many locations and agents can move from one location to another one. In many models for systems of mobile agents there exists a unique environment, the one of the higher level at the hierarchy, which contains all the other agents and environments.

The huge number of potential interactions and possible behaviors that the components of a system of mobile agents can exhibit make them difficult to understand and difficult to implement. Therefore, the designers of systems of mobile agents often need a formal model of the system to better understand it. The use of a formal model is also useful because it allows the use of *automatic* techniques to verify the system.

A natural setting to model systems of mobile agents is to view them as a set of autonomous entities working concurrently. In this perspective, models able to handle concurrency seem a natural pick.

Several formal models of concurrent computation have been proposed so far. Among the most widely used there are process algebras and Petri nets. The former provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent processes by means of algebraic laws. Process descriptions can be manipulated and analyzed. The latter are a mathematical tool with a graphical representation for describing and studying information processing systems characterized as being concurrent, asynchronous, distributed, nondeterministic.

Two of the first process algebras which introduced concepts for mobility are π-calculus [44], and Ambient calculus [8]. Other approaches are, for example, the kernel language for agent interactions and mobility (KLAIM, see [15]), and Seal calculus [9]. π-calculus is a calculus of communicating systems in which one can naturally express the mobility of processes. The component agents of a system are arbitrarily linked, and the communication between neighbors may carry information which changes that linkage. Ambient calculus identifies ambients as spheres of computation. They are properly nested and this determines locality. Capabilities are provided for entering, leaving and dissolving ambients. Movement across ambient boundaries can be subjective (the process in the ambient decides to employ the capability) or objective (the process outside the ambient dictates the move). Seal calculus identifies seals as agents or mobile computations. Here, seal boundaries are the main protection mechanism and seal communication is restricted to a single level in the hierarchy. Mobility is not under the control of a seal but of its parent, thus subjective moves of the ambient calculus are not supported. Finally, in KLAIM mechanisms that permit one to statically detect violations of security properties related to capabilities and access control have been developed.

In all of these approaches the state of the system is a basic notions while in Petri nets the global state is derived from their local counterparts. Moreover, Petri nets are one of relatively few formalisms admitting the true concurrency semantics, i.e., they can model concurrent execution of several actions directly, in contrast to the interleaving semantics of concurrency, where such an execution is modelled by a set of sequential runs, each of which is a permutation of these actions.

Among Petri net approaches, particularly promising for modeling mobility are the formalisms which use the nets-within-nets paradigm, which was introduced by Valk [49], based on the former work on task-flow nets [48]. In formalisms compliant to this paradigm the tokens of a net can be Petri nets themselves. Three different semantics have been proposed for the nets-within-nets paradigm: a reference semantics (where tokens in the system net are references to common object nets), a value semantics (where tokens in the system net are distinct object nets), and a history process semantics (where tokens in the system net are object net processes) [50].

Taking this as a view point, it is possible to model hierarchical structures, as systems of mobile agents. The environment at the higher level is modeled as a Petri net, called system net. The locations contained in this environment are modeled as places of the system net. The agents contained in the system net locations are modeled as Petri nets again. An interaction between an agent and its environment is modeled as a transition, as well as an action which moves an agent from one location to another one. Without the viewpoint of nets as tokens, the modeler would have to encode the agent differently, e.g. as a data-type. This has the disadvantage that the inner actions cannot be modelled directly, so, they have to be lifted to the system net, which seems quite unnatural. By using nets-within-nets it is possible to investigate the concurrency of the system and of the agent in one model without losing the abstraction needed.

Some nets-within-nets approaches for modeling system of mobile agents which will be discussed in the thesis are nested nets [39], modular Petri nets for mobility [38], and object nets for mobility [31, 32]. Moreover, in [32] the reference semantics (as supported by the Renew tool) has been used to model mobile agent systems. Nested nets and modular Petri nets for mobility use value semantics. This is a natural choice since each agent is dipped in only one environment in every moment. However, an interesting approach are object nets for mobility, where a global namespace is considered. In this case the use of the reference semantics makes sense because different environments can concurrently access and modify an agent. One drawback of nested nets, and object nets for mobility is that the hierarchy obtained by analyzing the containment relation between agents is static. Agents can move from one location to another one if they are in the same agent, but movements from locations belonging to different environment are not possible. Modular Petri nets for mobility solve this problem by using peculiar vacate/occupy transitions which clear/set the marking contained in one location.

One of the main advantages of using formal methods to model concurrent systems in general, and systems of mobile agents in particular, is that it is possible to apply automatic analysis techniques to analyze the system. These techniques can be divided into two categories: static techniques, and dynamic techniques. The former are able to determine properties satisfied by the system, by analyzing the structure of the net modeling the system. Place invariants, transition invariants belong to this category. The latter methods explore every reachable state of the system, and verify if certain specific properties of interest are satisfied. The model checking technique belongs to this category.

Static methods are obviously faster than the dynamic counterpart, but they also are more restrictive: it is possible to miss interesting properties. For example, there exist nets which do not have any invariant. Dynamic techniques are able to check a wider range of properties. For example, with model checking, a property is specified using a suitable temporal logic, and it is checked if every state of the system satisfies that property. If the property is not satisfied, a counterexample is provided, something that is not possible with static methods. The drawback of using dynamic methods is that they need to explore the whole state space, and concurrent systems suffer from the state space explosion problem, i.e.: even small systems yield very large state spaces. To alleviate this problem several proposals have been brought forward. They can roughly be classified as aiming at an implicit compact representation of the full state space of a system (e.g., in the form of a binary decision diagram, see [7]), or at an explicit generation of its reduced representation (e.g., abstraction [12] and partial order reduction [53] techniques). Among them, a prominent technique is McMillan's Petri net unfolding prefix generation [22, 43]. It relies on the partial order view of concurrent computation, and represents system states implicitly, using an acyclic net.

Unfortunately, sometimes the higher expressive power of a formalism comes at a price: some behavioral properties which are decidable in the basic Petri net model, become undecidable in high level formalisms. This is the case of some nets-within-nets models used for modeling mobility. For example, in [33] it was shown that *boundedness* is undecidable even for a class of object nets with a nesting level depth of two, while in [40] it has been shown that boundedness and reachability are undecidable properties for nested nets. Therefore, these formalisms are inadequate if the designer of the system needs to perform some kind of analyses which require these properties.

The main subject of this thesis is the hypernet model, which was introduced in [2] as a nets-within-nets framework for modelling systems of mobile agents. A hypernet is a collection of nets, called agents. Each agent is situated in some location. Locations are places in other agents. This, determines a hierarchy of agents. A peculiar characteristic of hypernet is that agents can exchange tokens with their sub- or super-agents, and thereby change the hierarchy arisen from the containment relation between agents. Modularity plays an important role in the hypernet model: 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. Agents cannot be created, nor destroyed.

In the hypernet model the number of reachable states is finite, therefore no decidability issues are present because it is always possible to generate the whole reachability graph. In [3] it was shown the existence of a morphism from hypernets to 1-safe Petri nets, which ensures that all the techniques available for 1-safe Petri nets are also available for the hypernet model. On the other hand, the main drawback of the hypernet model is the presence of some constraints in the definition 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.

The question of how to generalize the hypernet model without losing any of its important properties (like the possibility to apply techniques available for 1-safe nets, and other structural properties) is one of the main problems addressed in this thesis. A generalization of the basic model is introduced under the name *generalized hypernets*. Some structural constraints, like for example the fact that modules are state machines, have been removed. As we will see by means of examples, this gives the modeler a more flexible framework in which to use the hypernet model. As it has been done for hypernets, it is proved that it is possible to associate a 1-safe net to a generalized hypernet in such a way that their behaviors are equivalent. As for basic hypernets, this result guarantees that all properties which are applicable to 1-safe nets are also applicable to generalized hypernets.

However, from a practical point of view the computation of the 1-safe net is an expensive procedure. Therefore it is important to define techniques for computing properties of a hypernet directly on the model, so that actually building the 1-safe net is not needed. In particular, the problem of using the unfolding technique directly on the hypernet model is faced in this thesis. The concept of finite complete prefix of the unfolding of a generalized hypernet is defined, and a theorem showing that all the reachable markings of the hypernet are present as cuts in its corresponding prefix of the unfolding is proved. This is an important result because it allows the use of all the dynamic analysis techniques which require an unfolding as an input, like, for example, model checking [20]

The thesis also covers two more concrete aspects. From an application point of view it is discussed how the nets-within-nets paradigm has been used to model a Grid tool for High Energy Physics data analysis. The interactions between jobs which need to be executed on the Grid infrastructure, and the software components of the tool were modeled explicitly and in a natural way using nets-within-nets. Thanks to this, the developers of the system were able to find some bugs in the implementation of the tool.

The last aspect discussed in the thesis is the implementation of a tool which allows the use of hypernets to model and analyze systems. A plugin for a tool based on the nets-within-nets paradigm (Renew) has been developed. The main feature of this plugin is that it is able to convert the drawn hypernet in the corresponding 1-safe net, and to use external tools to analyze it. In particular features for computing place invariants, and CTL model checking are available.

To summarize, the results of the thesis are: the definition of a new class of hypernets, called generalized hypernets, more flexible than the original one; the definition of a procedure to build a 1-safe net from a generalized hypernet, in such a way that they have an equivalent behavior; the definition of the notion of finite prefix of the unfolding for a generalized hypernet; the use of the nets-within-nets paradigm to model a Grid tool for High Energy Physics data analysis; the implementation of a tool for drawing and analyzing hypernets.

The thesis is organized as follow:

Chapter 2 recalls the basic notions concerning Petri nets and partial orders which are needed in the thesis.

Chapter 3 discusses the basic hypernet model. The airport example is discussed and the basic definitions are given.

Chapter 4 introduces the generalization of the basic hypernet model. It is shown that the 1-safe nets semantics is preserved, as long as the tree-like structure of the hierarchy arising from the containment relation between agents.

Chapter 5 discusses some examples in order to illustrates the generalized hypernet model, and in which sense it is more flexible than basic hypernets. First it discusses how it is possible to model the behavior of a bartender and a client who go to a bar, and the behavior of a cell respiration process. Then it shows how it is possible to model a class of P-Systems, a computational model based upon the architecture of a biological cell.

Chapter 6 shows how it is possible to apply the unfolding technique to the hypernet model. First, it introduces the concept of branching process of a hypernet, both in an inductive and in an axiomatic way. Then, it shows that the two definitions are equivalent. Finally, it discusses how to build a finite prefix of the unfolding. The main result of the chapter is the proof of a theorem which relates the configurations of the prefix and the reachable markings of the hypernet.

Chapter 7 gives an overview of the main hierarchical Petri net models, with particular attention to models which use the nets-within-nets paradigm for modeling systems of mobile agents. A comparison with the hypernet model is discussed.

Chapter 8 discusses how the nets-within-nets paradigm has been used to model a tool for High Energy Physics data analysis named CRAB. A use case of this tool has been modeled using Renew, and interactions between different component of the system have been explicitly modeled as interactions between nets in the hierarchy.

Chapter 9 is all about the implementation of a Renew hypernet plugin which incorporates features for computing S-invariants, and features for model checking a hypernet.

Chapter 2

Basic Notions

2.1 Petri Nets

Petri nets are a graphical and mathematical modeling tool applicable devised for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, and/or nondeterministic.

Petri nets is a generic name for a whole class of net-based models. Common fundamental concepts that usually Petri net models respect are the following: a system is seen as a collection of local states, local transitions (between local states), and a relation between local states and local transitions which identify the neighbors of an element. The global state of a system is the collection of all local states that currently and concurrently hold. The result of a change caused by a local transition is restricted to the neighborhood of that transition. In the graphical representation of a Petri net places are drawn as circles, transitions are drawn as boxes. The neighborhood relation is represented by drawing arrows between places and transitions, and viceversa.

There can be distinguished in three classes of Petri nets. The firs class are 1-safe nets. A place here models a condition. If the condition holds, then a black token is put inside the place corresponding to the the condition. Therefore the presence of a token in a place represents the holding of the associated condition. When a transition is executed (*fires*), some condition in its neighborhood are disabled, and other conditions of the neighborhood are enabled. Place transition nets (P/T nets), folds some repetitive features of 1safe nets in order to get a more compact representation. Places no longer model conditions, but are more similar to counter which are decreased/increased when a transition fires. Finally, in high level nets formalisms more "compact nets" are obtained by structuring tokens, and annotating arcs and transition. For example, in coloured Petri nets tokens are data structures of a programming language, arcs are are annotated with variables, and transitions are annotated with guards which must be satisfied when the transition fires.

In this Section we recall basic definitions related to P/T nets and 1-safe nets. A P/T net is a particular kind of directed graph, together with an initial state called the *initial marking*. The underlying graph N of a P/T net is a directed, weighted and bipartite graph consisting of two types of nodes: *places* and *tran-*

sitions. Arcs are either from a place to a transition, or from a transition to a place, and represent the neighborhood relationship. As we already mentioned, in the graphical representation of a Petri net places are drawn as circles, transitions are drawn as boxes, and arcs are drown as arrows from transitions to places and viceversa. Each arc has a weight which is drawn as a number near the arc if it is greater than one.

A *marking* (*state*), assigns to each place p a nonnegative integer k. We say that p is marked with k tokens. Tokens are graphically represented by black dots, and a marking is represented by a distribution of tokens in places. A transition has a certain number of *output* and *input* places, which are places connected to the transition by an arc, and places the transition is connected to by an arc respectively. Tokens can be interpreted as resources, conditions or signals. The formal definition of a P/T net is the following:

Definition 1. *A* net *is a triple* (P, T, F, W) *such that* P *and* T *are disjoint sets of* places *and* transitions *respectively,* $F \subseteq (P \times T) \cup (T \times P)$ *is the* flow relation*, and* $W : F \to N$ *is the* weight function*. Transitions and places must be two disjoint sets, i.e.:* $P \cap T = \emptyset$*.*

Since weight are associated to arcs, the flow relation can be treated as a function $F : (P \times T) \cup (T \times P) \to \mathbb{N}$, and weights can be omitted. The *preset* of a node $x \in P \cup T$, denoted ${}^\bullet x$, is the set containing the elements that immediately procede x in the net, i.e.: ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$. In the same way the *postset* of a node, denoted $x^\bullet$, can be defined. Given a set $S \subseteq P \cup T$, the notion of preset/postset can be extended to S: $S^\bullet = \{y \mid x \in S, y \in x^\bullet\}$. A *subnet* of a net (P, T, F) is a net (P', T', F') where P', T', are subsets of P, T, and F' is the restriction of F to $(P' \times T') \cup (T' \times P')$. With F^+, the transitive closure of the relation F is denoted. A *net system* is a pair $\Sigma = (N, M_0)$ comprising a finite net $N = (P, T, F)$, and an *initial marking* M_0, which is a function from places to the set of the natural numbers, i.e.: $M_0 \colon P \to \mathbb{N}$.

2.1.1 The Firing Rule

The behavior of many systems can be described in terms of system states and their changes. In order to simulate the dynamic behavior of a system, a state or marking in a Petri net is changed according to the following *transition (firing) rule*:

- A transition t is said to be *enabled* if each input place p of t has at least $F(p, t)$ tokens.
- An enabled transition may or may not fire (depending on whether or not the event actually takes place)
- A firing of an enabled transition t removes $F(p, t)$ tokens from each input place p of t, and adds $F(t, p)$ tokens to each output place p of t.

The marking M_0 of the net is updated to the marking M_1 using the *firing rule*: $M'(p) = M(p) - F(p, t) + F(t, p)$. We denote this by $M[t\rangle M'$

Figure 2.1 illustrates a Petri net before and after the firing of a transition. The left image shows the marking before firing the transition. The right image shows the marking reached after firing t

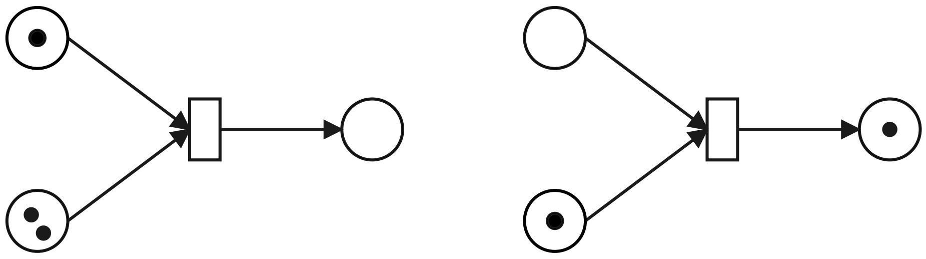


Figure 2.1: An illustration of the firing rule of a Petri net

After modeling a system with a Petri net an interesting question is “What can we do with the model?”. A major strength of Petri nets is their support for analysis of many properties and problems associated with concurrent systems. Typical problems considered in Petri nets are:

- *Reachability*: given a marking M is there a sequence of transitions that transform the initial marking M_0 to M?
- *Boundedness*: is there a natural number k such that the number of tokens in places of the net does not exceed k in any marking reachable from the initial marking?
- *Liveness*: are all the transitions of the net firable from any reachable marking?

The definition of 1-safe nets can be seen as a particular case of the definition of P/T nets where all the arcs have a weight of 1, and there is at most one token in each place in every reachable marking. Even though P/T nets and 1-safe nets have similar graphical and mathematical representations are rather different; for instance, finite P/T nets can have an infinite state space, but finite 1-safe nets cannot.

2.2 Partially Ordered Sets

A partially ordered set (or *poset*) consists of a set together with a binary relation that indicates that, for certain pairs of elements in the set, one of the elements precedes the other. Thus, partial orders generalize the more familiar total orders, in which every pair is related.

A partial order is a binary relation $\leq$ over a set A which is reflexive, antisymmetric, and transitive, i.e.: for all $a, b,$ and $c \in A$:

- $a \leq a$ (reflexivity)
- if $a \leq b,$ and $b \leq a$ then $a = b$ (antisymmetry)
- if $a \leq b,$ and $b \leq c$ then $a = c$ (transitivity)

Let $(A, \preceq)$ a partially ordered set. We call $(A, \preceq)$ a **well founded set** if and only if every non empty subset of A contains at least one minimal element m with respect to the order relation $\preceq$. This notion will be used in Chapter 6.

An **infinite descending chain** S in a partially ordered set $(A, \preceq)$ is a totally ordered subset of A without minimal element.

The principle of induction over partially ordered set, called Noetherian Induction, will also be used in Chapter 6. The following Proposition and the following Theorem are taken from [13].

Proposition 1. *Let $(A, \preceq)$ a partially ordered set. Then the following two statements are equivalent:*

1. *$(A, \preceq)$ is a well founded set.*
2. *There does not exist an infinite descending chain in A.*

Theorem 1. *(The principle of Noetherian induction). Let $(A, \preceq)$ a well founded set. To prove that a property $P(x)$ is true for all elements x in A it is sufficient to prove the following two properties:*

1. ***Induction basis**: $P(x)$ is true for all minimal elements of A.*
2. ***Induction step**: For each non-minimal x in A, if $P(y)$ is true for all $y \preceq x$, then $P(x)$ is true.*

Chapter 3

Basic Hypernets

3.1 Informal Introduction

Petri hypernets have been introduced in [2] as a model specifically designed to cope with systems of mobile agents using the nets-within-nets paradigm. A hypernet is a collection of nets (called agents), together with an assignment of mobile agents as tokens to places of other mobile agents (called hypermarking). One peculiar characteristic of the formalism is that it allows a hierarchical and modular description of reality: there is a natural hierarchy of agents that corresponds to the assumption that any mobile agent should be higher in the hierarchy than any of the tokens it manipulates, and each open agent is a synchronous composition of modules, each one responsible for manipulation of mobile objects traveling along a fixed channel. Another peculiar characteristic of hypernet is that the hierarchy of nets is dynamic, and can change during the dynamic evolution of the system.

As a simple example, consider an airport where plane can refuel, land, and take off and passengers can board, and deplane. In our model, the airport, the passengers, and the planes are represented by agents of a hypernet.

The agent in Figure 3.2 models the behavior of the airport. It has three modules, one for handling passengers, one for handling planes and one for synchronization purposes. Transition *board* belongs to both module *passenger* and module *plane*, and can only be executed synchronously. The same applies for transitions *deplane* and *to_rf*. The dashed half circles are communication places. They can either be *up-communication places*, used for communicating with the net at the level immediately above in the hierarchy (such as the two communicating places of the module plane in the airport agent), or *down-communication places*, used to communicate with an agent located in another module of the current net (such as the communication places in the synch and passenger modules of the airport). In the latter case, the name of a module is provided. In this module there must be an agent ready to provide the *traveling* token which will be moved in the hierarchy, otherwise the transition is not enabled.

For example, transition *deplane* of the passenger module in Figure 3.2 has an input communication place which indicates that a token is expected. Since this communication place is marked with the *plane* annotation, the traveling token which is being moved to place l must be provided by a plane agent. This plane

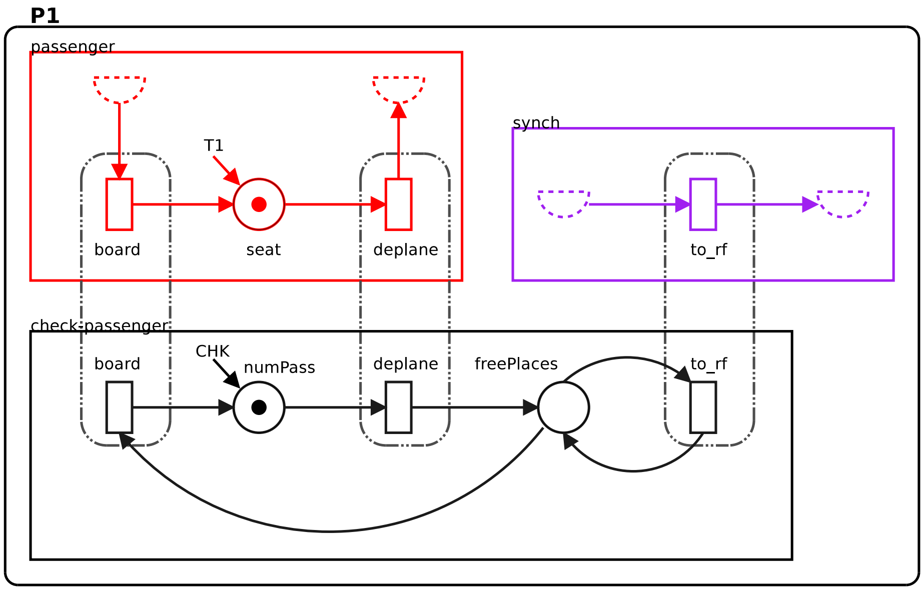


Figure 3.1: The plane agent

agent must be located in the input place of transition deplane in module plane of the airport, namely lg. In the example the only agent which can provide a token is $P1$.

Several works about Petri hypernets have been developed in the literature. A central result shows the existence of a morphism from hypernet to 1-safe Petri nets [3]. This translation not only shows that hypernets are well rooted inside the theory of Petri nets, it also allows us to reinterpret all the properties of the model one can derive on the 1-safe net by means of the technique developed in the literature for this basic net model on hypernets. Another result concerns the definition of a class of transition systems, denominated Agent Aware Transition Systems, defined in order to describe the behavior of hypernets (see [1]). In [4] a logical language for reasoning about systems representable with Petri hypernets were proposed. The language combines two families of modal operators: one family to cope with the temporal, the other to deal with the spatial (or structural) dimension. The problem of model checking properties of a class of the logic on Petri hypernets is shown to be PSPACE-complete.

3.2 Formal Definition

In this Section the formal definition of a hypernet is given. All the definitions are taken from [3].

3.2.1 The Static Structure of a Hypernet

A hypernet is a set of agents. Each agent is composed of modules. Each agent and to each module have a sort which determines which agents can be contained in places of a specific module.

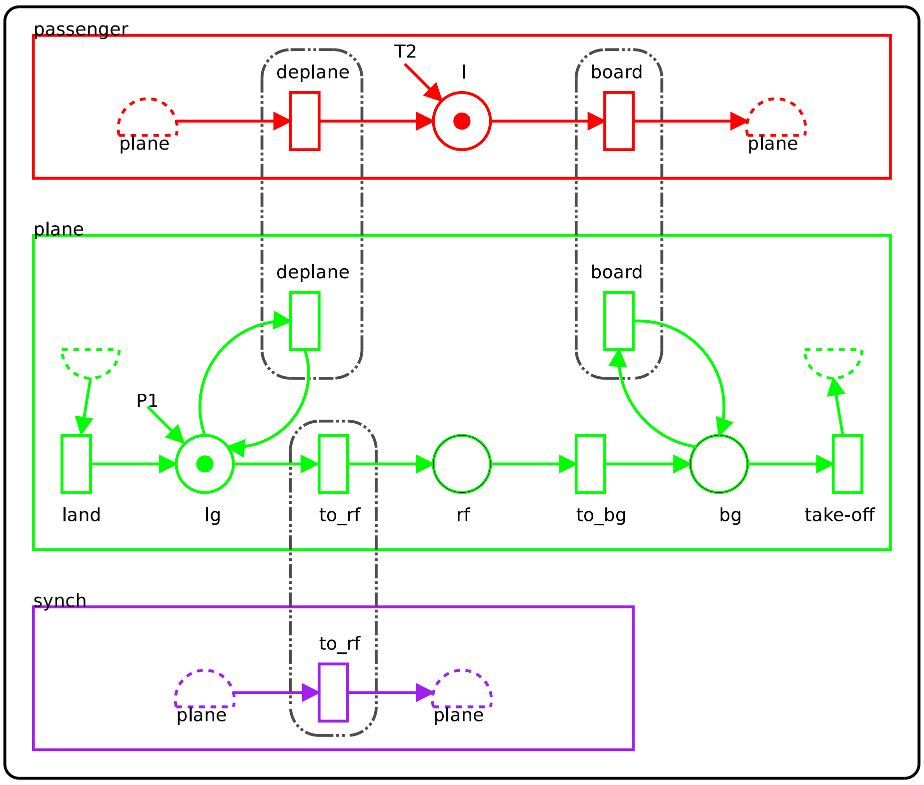


Figure 3.2: The airport agent

Definition 2. *Let Σ be a finite set of sorts. A* module *of sort α (α-module) is a finite Petri net $N = (P_N \cup I_N \cup O_N, T_N, F_N)$, where P_N is the set of places of the module, $I_N \subseteq \{?\} \cup \{?\beta \mid \beta \in \Sigma, \beta \neq \alpha\}$, $O_N \subseteq \{!\} \cup \{!\beta \mid \beta \in \Sigma, \beta \neq \alpha\}$ are the set of virtual communication places (input and output respectively), T_N is the set of transitions and $F_N \subseteq ((P_N \cup I_N) \times T_N) \cup (T_N \times (P_N \cup O_N))$ is the flow relation. It is assumed that $P_N \cap I_N = \emptyset \ \wedge \ P_N \cap O_N = \emptyset$, and moreover $|{}^{\bullet}t| = 1 = |t^{\bullet}|$.*

Definition 3. *A* (Mobile) Agent *is a set A of indexed modules, $A = \{N^\alpha \mid \alpha \in \Sigma\}$, where N^α is an α-module, $P^\alpha_A \cap P^\kappa_A = \emptyset$ when $\alpha \neq \kappa$, and the following consistency conditions are satisfied:*

$$F^\alpha_A(?\beta, t) \Rightarrow \exists\ p \in P^\beta_A \mid F^\beta_A(p, t) \tag{3.1}$$
$$F^\alpha_A(t, !\beta) \Rightarrow \exists\ p \in P^\beta_A \mid F^\beta_A(t, p) \tag{3.2}$$

where P^α_A,T^α_A,F^α_A respectively denotes places, transitions, and flux relation of the α-module of A.

If an agent is empty, then it will be a non structured token like in a standard Petri net. Communication ports ? and ! are used to communicate with the agent situated one level above in the hierarchy (master). In the form $?\beta$ and $!\beta$ they indicate the necessity to communicate with a β-agent one level down in the hierarchy (slave). The two forms are complementary, an agent can communicate with his master only if his master is ready to communicate with a slave of that sort, and viceversa. This idea is captured by the notion of *consortium*.

By convention, $F^{\alpha}_{A}(?\beta, t)$ and $F^{\alpha}_{A}(t, ?\beta)$ will be drawn as half dashed circles with the β sort inscribed inside the half circle, connected with an arc to the transition t in the α module of agent A.

For example, the consistency conditions (3.1) and (3.2) guarantee that if such a link between modules α and β exists, then $pre^{\beta}_{A}(t)$ or $post^{\beta}_{A}(t)$ will be local places (not communication ports), where $pre^{\beta}_{A}(t)$ and $post^{\beta}_{A}(t)$ denotes the input and the output places of transition t in the module β. This places are unique because of the state machine constraint for modules. They can be communication ports.

Figure 3.3 shows how the links described by the consistency conditions appear.

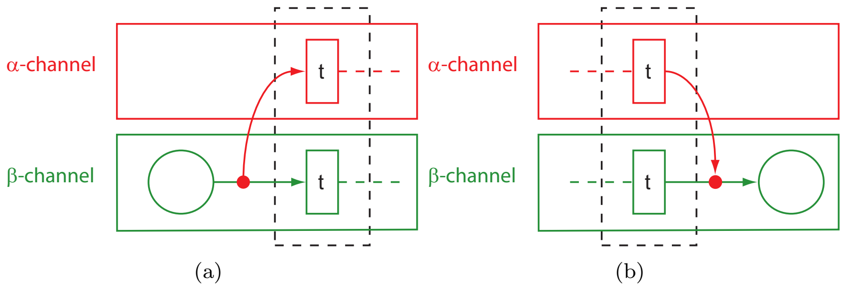


Figure 3.3: Consistency conditions

Definition 4. *A* hypernet *is a finite family of agents $\mathcal{N} = (A_i)_{i\in I}$, each with* one sort*, i.e.: a function $\sigma : \mathcal{N} \to \Sigma$ which assign to each agent its sort.*

We assume agents in $\mathcal{N}$ have disjoint sets of local places. With $P_A \hat{=} \bigcup\{P^{\alpha}_{A} \mid \alpha \in \Sigma\}$ and $T_A \hat{=} \bigcup\{T^{\alpha}_{A} \mid \alpha \in \Sigma\}$ we denote the local places and the transitions of a particular agent $A \in \mathcal{N}$.

With the symbol $P_{\mathcal{N}} \hat{=} \bigcup\{P_A \mid A \in \mathcal{N}\}$ we denote the set of all places of an hypernet, and with $T_{\mathcal{N}} \hat{=} \bigcup\{T_A \mid A \in \mathcal{N}\}$ we denote the set of all transition of an hypernet. Each local place has one sort: $\sigma(p) = \alpha$ where α is the only sort of place p..

3.2.2 The Dynamic of a Hypernet

It is necessary to explain what a hypernet transition is. An agent A can execute a specific transition t only if each of its modules is ready to execute it. An intra module synchronization is assumed.

Let us focus on an α-module. If all input and output places of t in the module α are local, the in order to execute t it is sufficient a token in $pre^{\alpha}_{A}(t)$. This token is moved in $post^{\alpha}_{A}(t)$.

If communication placed are involved, a group of agents (called consortium) can cooperate to execute the transition t. It is necessary to establish an inter agent synchronization.

In particular, there is an α-*flow* from an agent A to an agent A', i.e., $A \triangleright_\alpha A'$, when an agent A is ready to send an agent of sort α from its α-module, to the α-module of the agent A'. To create an α-flow the two agents must be *adjacent*, one of the two must be a token of the other one. Moreover, if one of the two agents is ready to send/receive a token, the other must be ready to receive/send a token. Figure 3.4 shows this concept. In both cases there is an α-*flow* from A to A' ($A \triangleright_\alpha A'$).

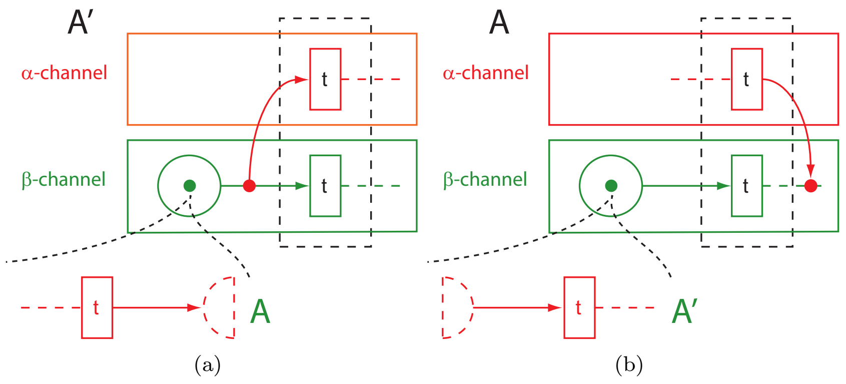


Figure 3.4: Example of creation of an *alpha*-flow from A to A' ($A \triangleright A'$)

We now formally define the notion of consortium. Consider a triple $\Gamma = (t, \tau, \xi)$, where $t \in T_{\mathcal{N}}$ is a transition, $\tau \subseteq \{A \in \mathcal{N} \mid t \in T_A\}$ is a set of *active agents* and $\xi : In(\Gamma) \to \mathcal{N}$ is an injective function which associates *active agents* (token) to input places of the instances of t in τ. The definition of these places is: $In(\Gamma) \hat{=} P_{\mathcal{N}} \cap \bigcup_{\alpha \in \Sigma} \{pre^\alpha_A(t) \mid A \in \tau\}$; in the same way it is possible to define output places $Out(\Gamma)$.

Consider two agents $A, A' \in \tau$. As explained by Figure 3.4 there is an α-*flow* from A to A' ($A \triangleright_\alpha A'$) if one of the two following conditions are satisfied:

$$post^\alpha_A(t) =! \wedge \xi(pre^\beta_{A'}(t)) = A \wedge pre^\alpha_{A'}(t) =?\beta \tag{3.3}$$

$$post^\alpha_A(t) =!\beta \wedge \xi(pre^\beta_A(t)) = A' \wedge pre^\alpha_{A'}(t) =? \tag{3.4}$$

The set of *passive agents* is $\tau_\xi = \{\xi(p) \mid p \in In(\Gamma)\}$. Define $\uparrow: \tau_\xi \to \tau = A' \uparrow = A \iff \xi(p) = A'$ for some $p \in P_A$,. It is now possible to define the notion of *consortium*:

Definition 5. *A triple Γ defined as above is a consortium if the following conditions hold:*

1. $\tau \neq \emptyset$.
2. $A \in \tau \wedge F^\alpha_A(?, t) \Rightarrow A \in \tau_\xi \wedge A \uparrow \triangleright_\alpha A$.
3. $A \in \tau \wedge F^\alpha_A(t, !) \Rightarrow A \in \tau_\xi \wedge A \triangleright_\alpha A \uparrow$.
4. $A \in \tau \wedge F^\alpha_A(?\beta, t) \Rightarrow \xi(p) \in \tau \wedge \xi(p) \triangleright_\alpha A$, *where* $p = pre^\beta_A(t)$.
5. $A \in \tau \wedge F^\alpha_A(t, !\beta) \Rightarrow \xi(p) \in \tau \wedge A \triangleright_\alpha \xi(p)$, *where* $p = pre^\beta_A(t)$.

6. *The undirected graph* $(\tau, \bigcup_{\alpha \in \Sigma} \triangleright_\alpha)$ *is connected.*

The state of a hypernet is a partial function $\mu : \mathcal{N} \to P_{\mathcal{N}}$ which preserves sorts. A consortium Γ is enabled in a hypermarking μ (i.e.: $\mu|\Gamma\rangle$) if $\forall p \in In(\Gamma) \Rightarrow \mu(\xi(p)) = p$.

A consortium Γ enabled in μ can be fired leading to a new hypermarking μ' defined as:

$$\mu'(A) = \begin{cases} \mu(A) & \text{se } A \notin \tau_\xi; \\ trg(\mu(A)) & \text{se } A \in \tau_\xi \end{cases}$$

In particular, the new assignment of open nets as tokens which results from the consortium execution, is a well founded hypermarking: it preserves the tree like structure of the hypermarking [3].

Chapter 4

Generalized Hypernets

Basic 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. In this section, a generalization of the basic model which relaxes these constraints is defined under the name of *generalized hypernets*. As it will be shown, the 1-safe nets semantics is preserved, as long as the tree-like structure of the hierarchy arising from the containment relation between agents. The state machine module structure constraint is replaced with the local notion of paths (defined in Section 4.1.1), which are themselves of a certain sort. The need of a generalization of the hypernet model arose in [5], when it was not possible to model with basic hypernets a class of membrane systems, computational models based upon the architecture of a biological cell.

4.1 Concepts

As for basic hypernets, the term *agent* is used 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.

4.1.1 Paths

As it has already benn mentioned when the basic model of hypernets has been introduced, the first difference between a hypernet and a Petri net 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 tokens 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 Petri net 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 Figure 4.1 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 4.1(b)), or could be separated and moved one to $p3$ and one to $p4$ (Figure 4.1(c)).

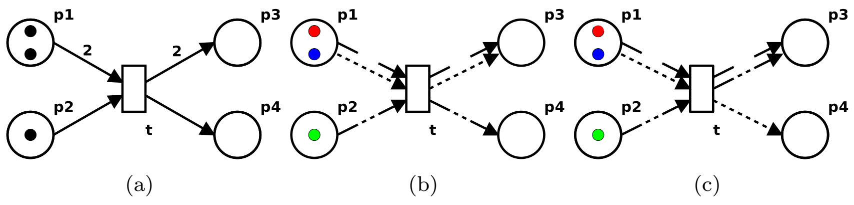


Figure 4.1: Paths are used in the firing rule of the generalize hypernet model

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 11) and where each token will be placed after a transition is fired. In each of the two Figures 4.1(b) and 4.1(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 4.1(b), and paths $(p_1, t, p_3), (p_1, t, p_4), (p_2, t, p_3)$ are present in Figure 4.1(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 4.4, 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 machine. 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 [5] 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 easy modeled if the formalism allows the use of weighted arcs, whereas with synchronized state machines the model is more constrained.

4.1.2 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 2, is always a tree. Virtual places are introduced to manage the passage of tokens between *close* agents. 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 4.2(a), or down if token B is located in a place of token A 4.2(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.

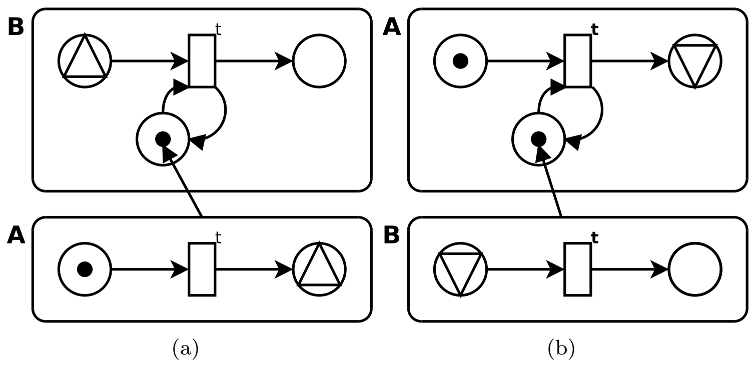


Figure 4.2: Virtual *up* places, and virtual *down* places

4.2 Formal Definition

4.2.1 Static Structure

Definition 6. *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 4.3 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.*

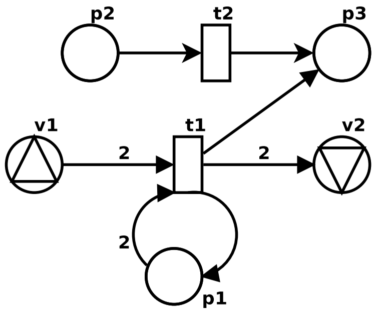


Figure 4.3: A simple agent A_1

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 4.4 all possible path combinations of figure 4.3 are represented with this graphical notation. There is only one net without a path where both input and ouput places that are virtual. This is the only net that is an agent according to Definition 6.

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 7. *Let* $\mathcal{N} = \{A_1, A_2, \dots, A_n\}$ *be a family of (possibly empty) 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*

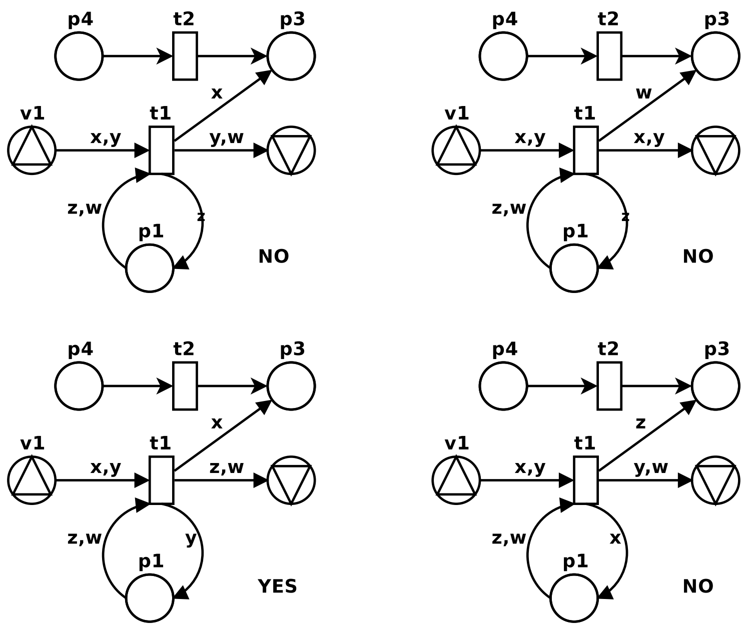


Figure 4.4: Notation used to represent possible paths of net in figure 4.3

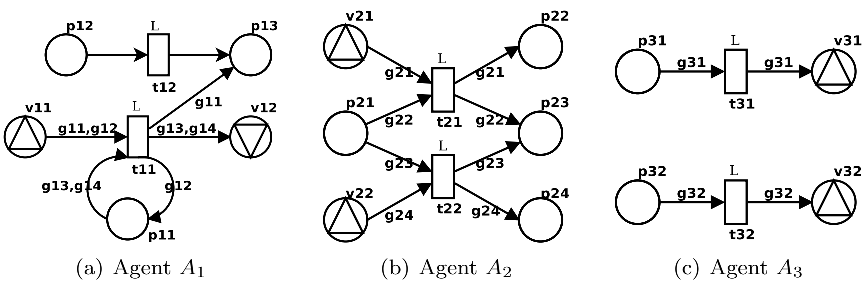
- *The agents in $\mathcal{N}$ have disjoint sets of places, paths, and transitions, i.e.:* $\forall A_i, A_j \in \mathcal{N}, i \neq j \implies (P_i \cup T_i) \cap (P_j \cup T_j)$*;*
- $\sigma_{\mathcal{N}} : \mathcal{N} \to 2^{\Sigma}$ *is a function that describes the sorts of each agent;*
- $\sigma_G : G_{\mathcal{N}} \to \Sigma$ *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 4.5(a), A_2 the agent shown in figure 4.5(b), A_3 the agent shown in figure 4.5(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 assigns to each element of the domain the element $\{\alpha\}$, let $\sigma_G : G_{\mathcal{N}} \to \Sigma$ be a function which assigns to each path the sort α, 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.

Definition 8. *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 (the choice of A_1 is by convention)*

Definition 9. *A* marked hypernet *is a pair $(H, \mathcal{M})$ where H is a hypernet and $\mathcal{M}$ is a hypermarking defining the initial configuration.*



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

Figure 4.5: A set of agents that together form an hypernet

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 4.6 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.*

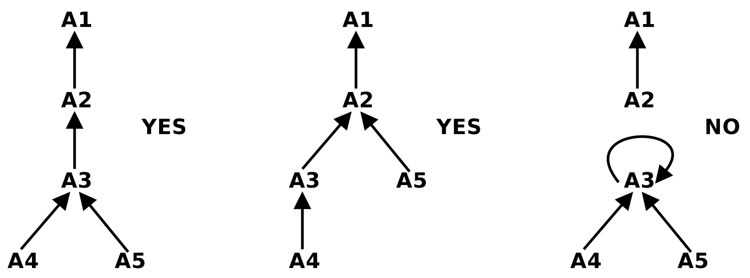


Figure 4.6: Possible marking graphs of the hypernet of Example 2

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.*

4.2.2 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.

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

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 virtual paths are consistent if they have the same sort and the same direction (the virtual places of the paths are both up or down places). As it can be seen in the formal definition this notion is not a symmetric relation because the first path belongs to the agent which sends the token and the second path belongs to the agent which receives the token.

Definition 10. *Two paths $g_i \in G_i$ and $g_j \in G_j$ are* consistent *(denoted by $cons(g_i, g_j)$) if:*

- $i \neq j$
- $\sigma_G(g_i) = \sigma_G(g_j)$
- $(g_i \in G^{Up} \cap G^{Out} \wedge g_j \in G^{Up} \cap G^{In}) \vee (g_i \in G^{Down} \cap G^{Out} \wedge g_j \in G^{Down} \cap G^{In})$

$\wedge \wedge ()$

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 4.7 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 11. *A* consortium *is a tuple $\Gamma = (l, \tau, \text{PASS}, \delta, \gamma)$ 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) is denoted*

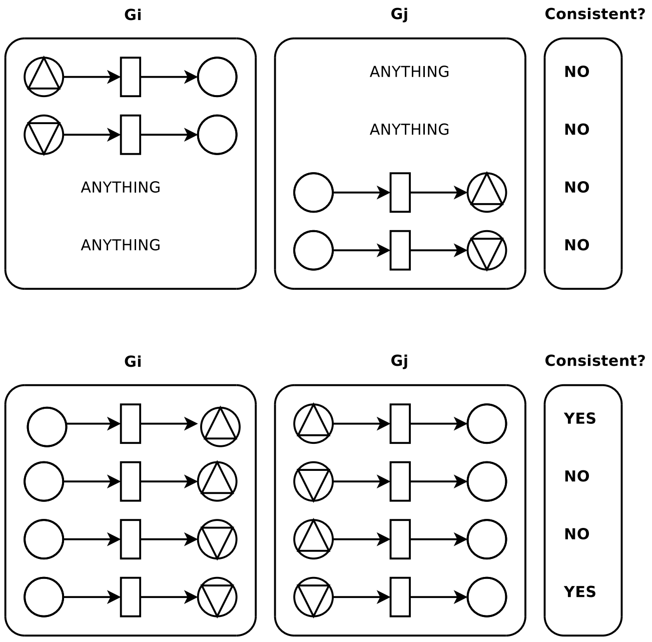


Figure 4.7: Possible combinations of paths and their consistency

3. PASS $\subseteq \mathcal{N}$ *is the set of* passive agents

4. $\delta : G^{Out} \cap G_\tau \rightarrow G^{In} \cap G_\tau$ *is a* bijective *correspondence such that:* $\delta(g) = g' \Rightarrow$ cons (g, g')

5. γ : PASS $\longrightarrow G_\tau \backslash G^{In}$ *is a* bijective *correspondence such that:* $\gamma(A) = g \Rightarrow \sigma_G(g) \in \sigma_{\mathcal{N}}(A)$

6. *Agents that receive tokens must not move in the hierarchy, i.e.:* $\forall A_i \in$ PASS $: \exists g \in G_i \cap G^{In} \cap G_\tau \Rightarrow \gamma(A_i) \notin G^{Out}$

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 4.8). 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*

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

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 $E = \{(t_i, t_j) : t_i \in A_i, t_j \in A_j \wedge \exists\, g \in G^{t_i} : \delta(g) \in G^{t_j}\}$

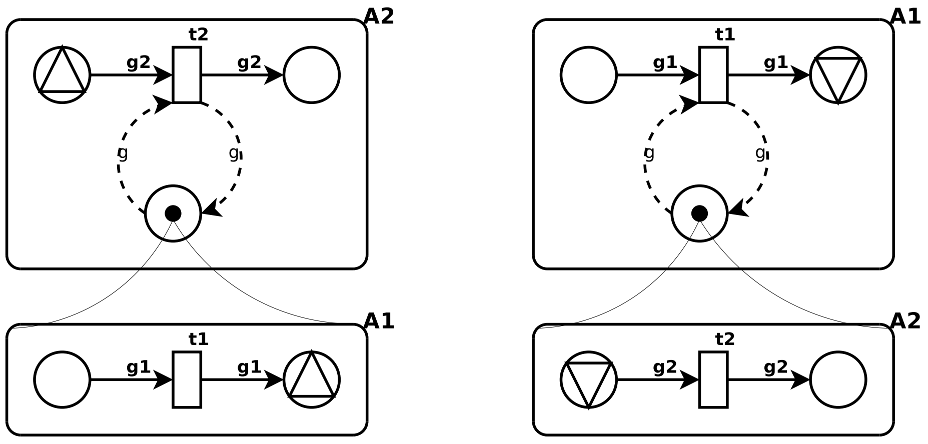


Figure 4.8: If the path g exists then the inner agent must be passive agents

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

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

$$\forall A \in \text{PASS}, {}^\bullet\gamma(A) = p \;\Rightarrow\; \mathcal{M}(A) = p \tag{4.1}$$
$$\forall g \in G_i \cap G^{Up} : \delta(g) \in G_j, A_i \notin \text{PASS} \Rightarrow A_i \uparrow_{\mathcal{M}} A_j \tag{4.2}$$
$$\forall g \in G_j \cap G^{Down} : \delta(g) \in G_j, A_i \notin \text{PASS} \Rightarrow A_i \uparrow_{\mathcal{M}} A_j \tag{4.3}$$

where $\uparrow_{\mathcal{M}}$ was defined in Definition 8.

Definition 13. *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 $\forall A \in \mathcal{N}$:*

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

Definition 14. *Given a marked hypernet $(H, \mathcal{M})$, a hypermarking $\mathcal{M}_n$ is reachable iff $\exists \Gamma_1, \Gamma_2, \ldots, \Gamma_n \in CONS(H) : \mathcal{M}_0[\Gamma_1\rangle\mathcal{M}_1[\Gamma_2\rangle\mathcal{M}_2, \ldots, \mathcal{M}_{n-1}\ [\Gamma_n\rangle\ \mathcal{M}_n$*

4.3 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 8). 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 simultaneous movements 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 [16] for instance).

Definition 15. *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:*

$$\forall v \in V \backslash v_0 \; \exists k \textit{ such that } p^k(v) = v_0 \textit{ with } v_0 \in V \tag{4.4}$$

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 [16].

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

Definition 16. *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 17. *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.*

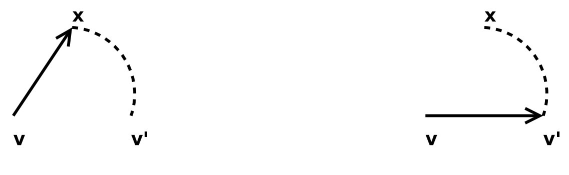


Figure 4.9: A move with pivot v

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 4.9), 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:

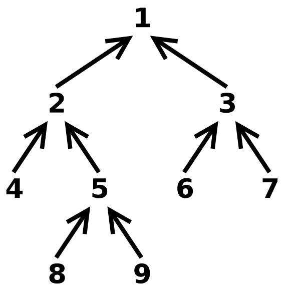
Definition 18. *A set of A -moves $M = \{(v_0\ ,\ v'_0),\ (v_1\ ,\ v'_1),\ ...,$ $(v_n\ ,\ v'_n)\}$ is called* distinct *iff the moves have pairwise different pivots, i.e.: $v_0 \neq v_1 \neq ... \neq v_n$. With $PVT(M) = \{v_0, v_1, ..., v_n\}$ the* pivot set *of the distinct set of moves M is denoted.*

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.

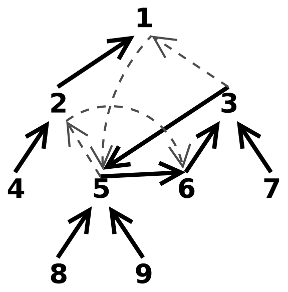
Definition 19. *The application of a set of distinct A -moves $M = \{(v_0\ ,\ v'_0),$ $(v_1\ ,\ v'_1),\ ..., (v_n\ ,\ v'_n)\}$ to the tree $A = (V,\ p)$ is a graph $G = (V, p')$, i.e.: $M(A) = G$, such that:*

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

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 4.10(a) produces a graph that is not a tree anymore (see figure 4.10(b)).



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



(b) The graph is not a tree anymore

Figure 4.10: 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 20. *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 4.10 are not 1-level, whereas the moves $\{(9, 8), (6, 1)\}$ are 1-level.

Definition 21. *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 22. *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 2. *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.*

$$M(A) = M_{[0,h]}(A) \text{ is a tree} \tag{4.5}$$

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 4.4 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 4.4 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 4.11(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 4.11(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.

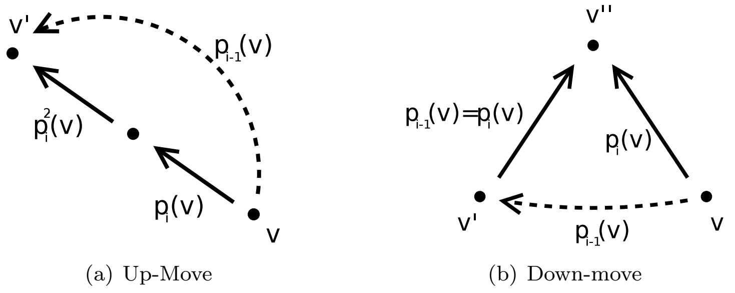


Figure 4.11: Type of possible moves

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 2 the following theorem can be proved:

Theorem 3. *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 11, 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. □

4.4 1-Safe Net 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 23. *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 $\Gamma = (l, \tau, \text{PASS}, \delta, \gamma)$ *and its corresponding transition* t_Γ

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

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 4.12):*

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

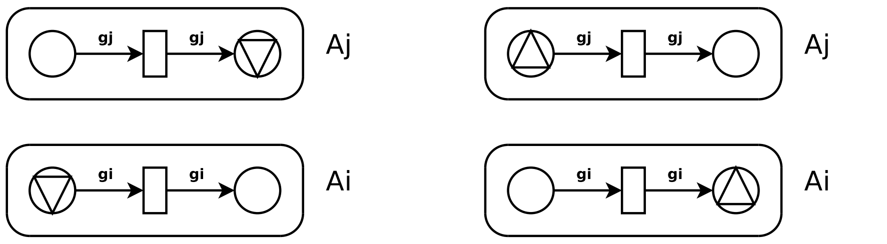


Figure 4.12: Situations where a loop is added

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

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

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

Definition 24. *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:*

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

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

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 4. *Let $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$ be a hypernet, $\mathcal{M}$ be a hypermarking and $\Gamma = (l, \tau, \mathrm{PASS}, \delta, \gamma) \in CONS(H)$ 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 4.1, condition 4.2, and condition 4.3 of Definition 12 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 23).

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 $A \in \mathrm{PASS} \wedge {}^\bullet\gamma(A) = p$ is true. Thus, if this condition hold, place $\langle A, p\rangle$ must be marked, i.e.: $\mathcal{M}(A) = p$ (Definition 24). But, this is true for hypothesis because condition 4.1 of Definition 12 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) $\exists g_i \in G_i \cap G^{Up} : A_i \notin \mathrm{PASS} \wedge \delta(g_i) \in G_j$
(b) $\exists g_j \in G_j \cap G^{Down} : A_i \notin \mathrm{PASS} \wedge \delta(g_j) \in G_i$
(c) $A_i \in \mathrm{PASS} \wedge \gamma(A_i) \in G^{Out} \wedge \gamma(A_i) \in G_j$

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 24). It is easy to see that if the first holds, then for hypothesis (condition 4.2) $A_i \uparrow_{\mathcal{M}} A_j$ also holds. The same is true for the second case and condition 4.3. 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 4.1), 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 13 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 13),

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.: $A \notin$ PASS).

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 23) .

The sixth and seventh conditions of Definition 23 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 13).

The last two conditions says that if agent A_i is passive ($A_i \in$ PASS) 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 13 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 23) 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 4 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.

Chapter 5

Examples

In this Chapter the generalized hypernet model is illustrated by means of examples. First it is discussed how it is possible to model the behavior of a bartender and a client who go to a bar, and the behavior of a cell respiration process. Then it shows how it is possible to model a class of P-Systems, a computational model based upon the architecture of a biological cell.

5.1 The Bar Example

The aim of this Section is to describe a situation in which the agents involved are a bartender and a client which can go to the bar and drink something, or which can go to their houses. As we will see, the use sorted paths, and multi-sorted agents is crucial. It will be discussed how they generalize the idea of module of the basic hypernets, adding flexibility to the model.

The characteristics of this example are the following. The client need the presence of the bartender if he wants to order a drink, otherwise he can just sit down and wait. After the client has drunk he can exit the bar and go to the street. The bartender can go from the bar to the street too. From the street they can go to their own houses, which are modeled as agents. In their houses the client and the bartender can do some houseworks, or they can sleep. Whenever they want, they can also exit the house and go again to the street.

There also is a maid which works for both the bartender and the client. She can access their houses, she can do several houseworks, and then she can exit the house and go to the street. The maid does not like bars, and she never goes to any bar.

Bartender, client, and maid are modeled as empty nets. The client house, and the bartender house are modeled with the structured nets in Figure 5.1, and Figure 5.1 respectively. The bar is the agent depicted in Figure 5.1, and the outside world, which is the system net, is shown in Figure 5.1.

Before starting a detailed description of each one of this nets, it is important to explain the role of types. In the example there are some constraints that agents (bartender, client and maid) must obey. The maid cannot access the bar, but she can access the houses of the other two agents. The bartender can access his house, and the bar as bartender. The client can access his house, and the bar as client. To enforce this constraints, a hierarchy of types has been used.

The sorts used by the client to go to his house are of type *CLIENT-HOUSE* (see transition t_2 of Figure 5.1 and transition t_{17} of Figure 5.1), a type owned by the maid and the client. In this way both the client and the maid can go to the client house. To go to the bar as client you need the sort *CLIENT* (transitions t_1 and t_6). The only agent who owns that sort is the client. In the same way the access to the bartender house, and the access to the bar as a bartender have been restricted. Finally there is a basic type used for transition which every agent can do (provided it is able to enter the appropriate environment).

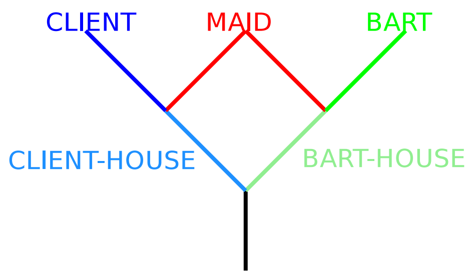


Figure 5.1: The sort hierarchy of the example

Figure 5.1 summarize the types owned by agents. The client agent owns the types *CLIENT* and *CLIENT-HOUSE*. The bartender agent owns the types *BARTENDER* and *BARTENDER-HOUSE*. The maid agent owns the types *BARTENDER-HOUSE* and *CLIENT-HOUSE*. In other words, client, maid and bartender have each one its own type. The client and the bartender have the type which allow them to enter their house, while the maid have both the types to enter their houses.

The types of paths in the whole example are recognizable by looking at the color of the arrows which connect places and transitions, or by looking at the comments indicated by the dashed lines.

The World

The net world is shown in Figure 5.1, and it is the system net.

It contains two unconnected places, *bars* and *houses*, which contain the structured agents modeling the houses and the agent modeling the bar. The other place of the net is the place named *street*, which is a temporary place where client, bartender and maid can use to move from one structured agent to another one.

The five transitions of the net are all used for communication purposes. *enterClient* and *enterClient* let the client and the bartender to enter the bar respectively. They are all used to transfer down the empty tokens, and as discussed before, the types of the paths plays an important role in defining who can enter what. The transition *exit* is used to transfer the client, the bartender and the maid, from houses/bar to the street.

The Bar

The net which model the agent bar is shown in Figure 5.1.

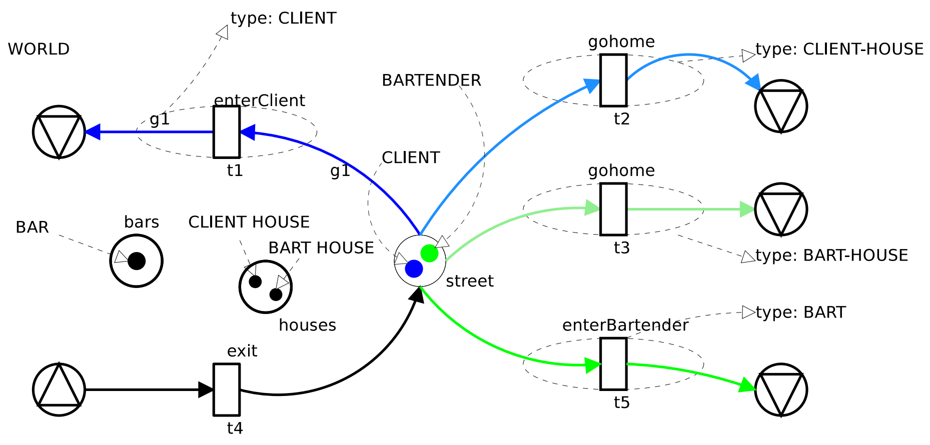


Figure 5.2: The system net of the Bar example

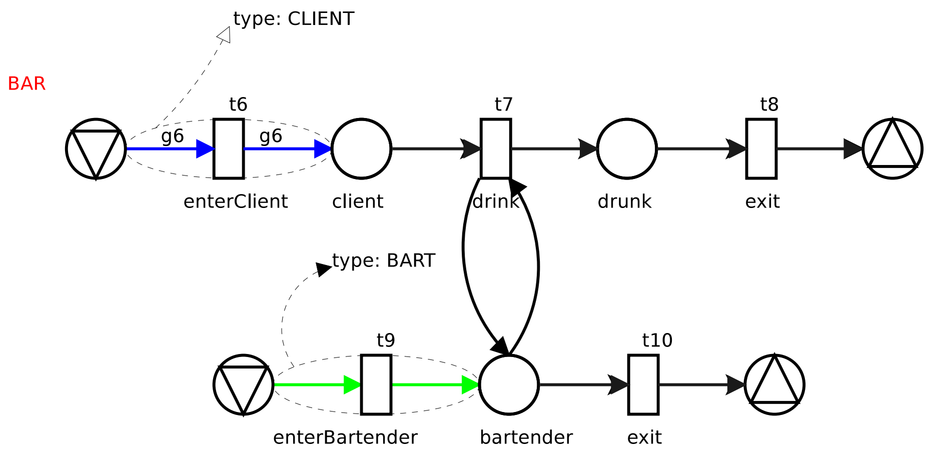


Figure 5.3: The net modleing the bar agent

There are two transitions which let the client and the bartender go inside the bar. Transition *enterClient* allows the client to go inside the bar. The path which contain the transition is of sort *CLIENT*, and so the only token which can use it is the client itself. In the same way, *enterBartender* is used by the bartender to go into the bar. Once the client and the bartender are in the bar they can synchronize to make the transition *drink* enabled. The bartender stays in the place *bartender* and it is ready to serve another client. The client will move to the place *drunk*, and then it can exit the bar and go to the street. When the bartender has finished its work it can also exit the bar.

The Houses

The two nets which model the client and the bartender houses have the same shape. The only thing that change is the type of the transition which let the agent enter the house. Let us start analyze the bartender house shown in Figure 5.1

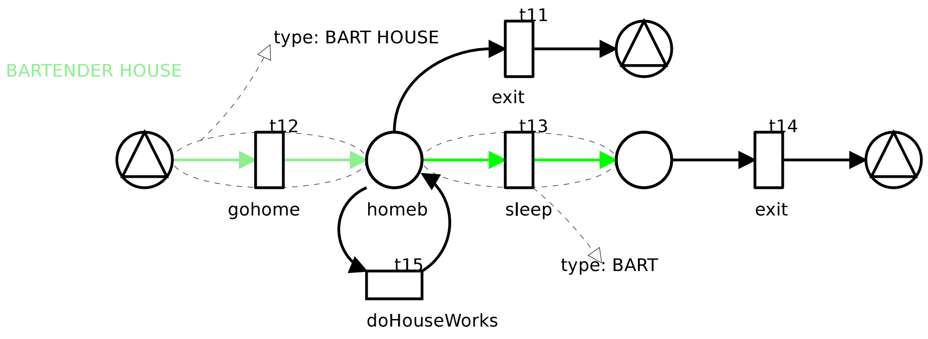


Figure 5.4: The net modeling the bartender house

The maid and the bartender can enter the house by firing the transition *gohome*. There (when they are in the place *homeb*), they can do the houseworks, or they can exit the house and return to the street. Only the bartender is allowed to sleep in its house, therefore the path containing the transition sleep is of sort *BARTENDER*.

The net modeling the client house is very similar, but the path containing the transition sleep is of type *CLIENT*, and the path which allows agents to enter the house is of sort *CLIENT-HOUSE*. This net is shown in Figure 5.1.

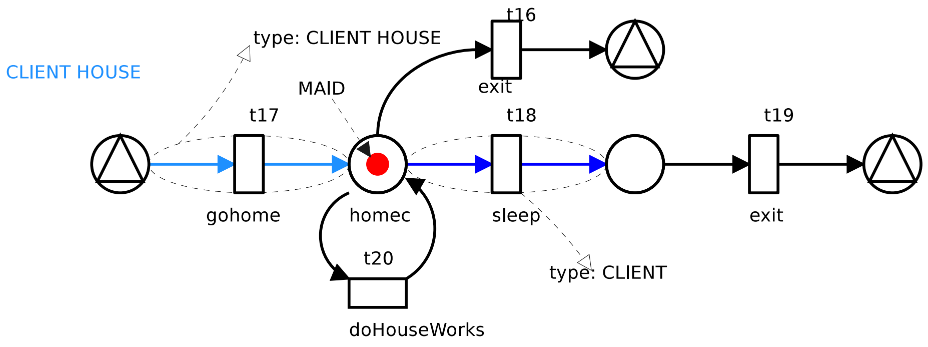


Figure 5.5: The net modeling the client house

Final Considerations

This example models a simple case of study where three agents - a maid, a client, and a bartender - can move in three buildings - two houses and a bar. The features added by the generalization which allow to model more easily a situation like this, are the use of typed paths instead of modules, and the use of multi-typed agents. The strict module subdivision of the basic hypernet model implies that the *street* place must be splitted if it have to contain agent of different types. The alternative is to assign the same sort to the three moving agents. Therefore, the access restriction applied to the agents, which allow them to access only some buildings, must be modeled in some more complex way. For example, a possibility is to structure the empty moving agents as net, and encode the information of where it can go in some way inside the net.

The too strict module subdivision of the basic hypernet model is also evident if we consider a variation of the example where there are more than one bar, and more than one client. Suppose a bartender wants to go in another bar B as a client. With generalized hypernet it is sufficient to add to this bartender the sort corresponding to the sort which allows client to enter the bar B. Again, in the basic hypernet model there are no meaning to do this in a simple and elegant way. In particular, the reason of allowing multi-sorted agents is clear if we look at this example.

The next example shows how paths allows the modeling of the cellular respiration process in an easy way.

5.2 The Cellular Respiration Process Example

The aim of this section is to represent the cellular respiration process using the model of generalized hypernets. In particular the modeled aspect is the transport of oxygen and carbon dioxide from lungs to cell, which is done by red blood cells. In each red blood cell there are heme groups, which are able to bind four molecules of oxygen. A red blood cell binds oxygen in the lungs, and transport it to the tissues of the body. In the cell the oxygen is used in several reactions for producing energy. The most important one is probably the aerobic respiration which happens in mitochondria, membrane-enclosed organelle found in the cytoplasm of each cell of the human body. The simplified form of this reaction takes one molecule of sugar ($C_6H_{12}O_6$) and six molecules of oxygen (O_2), and transform them in six molecules of carbon dioxide (CO_2), plus six molecules of water (H_2O), which are expelled from the cell by osmosis.

The system is modeled using five nets. The first one is called *WORLD* (Figure 5.2), and it is the system net. Inside this net there is another net, called *HUMAN1*, which model the human metabolism. This net is is shown in Figure 5.2. As it can be seen it contains other two structured agents which model the behavior of a cell (*CELL* net), and of a red blood cell (*RBC* net). These two nets are shown in Figure 5.2, and in Figure 5.2 respectively.

In the following subsections a detailed description of all these net is given.

The world

The world net is the system net and it is shown in Figure 5.2. The net simply contains a human, and it models with two transitions the actions of inspire and expire.

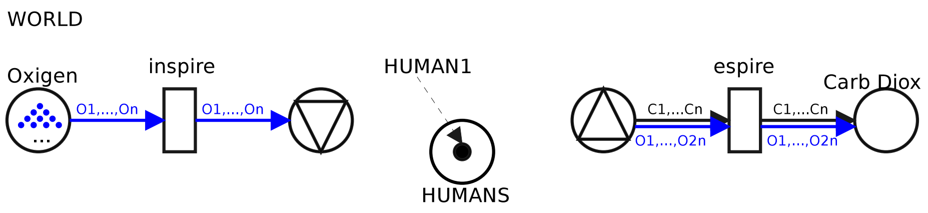


Figure 5.6: The system net of the Blood example

The *inspire* transition binds n oxygen molecules and transfer them inside the human body. The n paths which contain the transition *inspire* are not shown in the figure, nut they are graphically represented by the inscription $O_1, ..., O_n$ associated to the arc connecting the *Oxygen* place and the *inspire* transition. In the same way, the transition expire takes carbon dioxide molecules from the human body and transfer them in the air outside the world.

The Human Metabolism

The net modeling the respiration cycle, where oxygen is transported from lungs to tissues by red blood cell is shown in Figure 5.2.

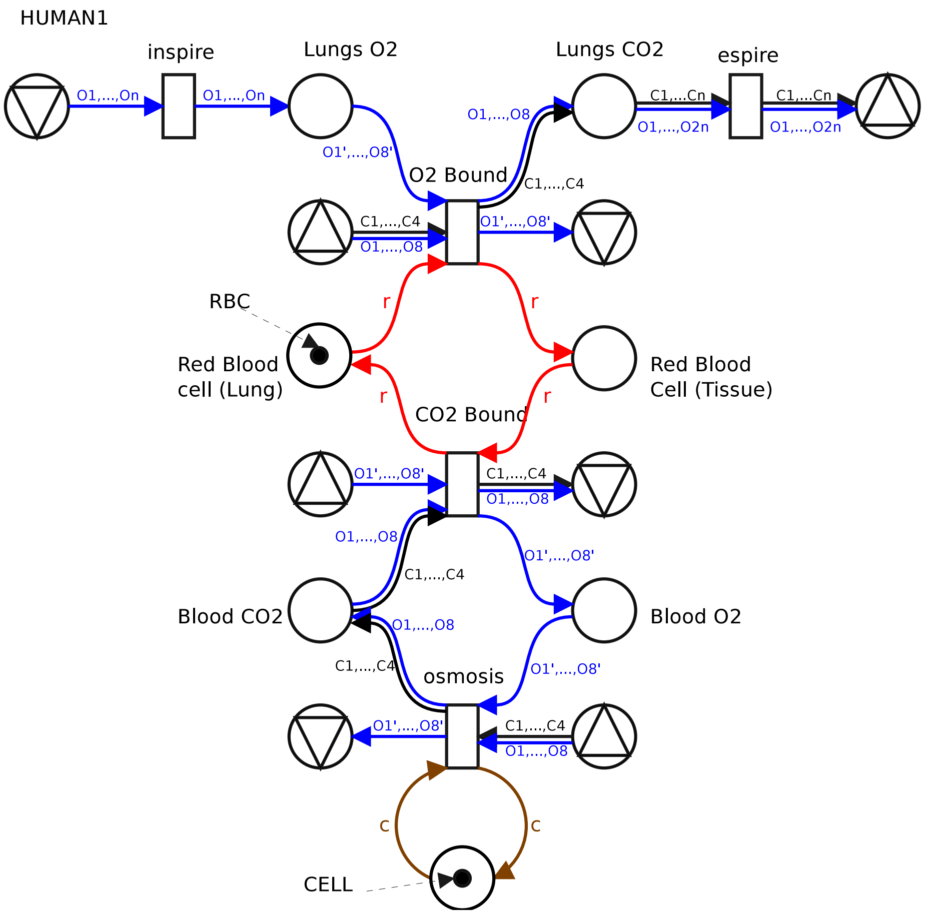


Figure 5.7: The net modeling the metabolism of the body

Transitions *inspire*, and transition *expire* are the entwined to the transitions with the same name in the net world. They transport oxygen from the world to the body, and carbon dioxide from the body to the world. Places *Red Blood Cell (Lung)*, and *Red Blood Cell (Blood)* contain single red blood cells. For simplicity, in the example there is only one red blood cell. The former place indicate that the cell is in the lung, and it is ready to give away carbon dioxide and to take oxygen. Place *Red Blood Cell (Tissue)* indicates that the red blood cell is near a tissue and it is ready to exchange oxygen for carbon dioxide. The oxygen taken from the cell goes to the *Blood O2* place, while the carbon dioxide is taken from the place *Blood CO2*. As the name suggests, transition *osmosis* models the transfer oxygen from the blood to a cell by means of osmosis, and the transfer of carbon dioxide from a cell to the blood (again by osmosis).

The Cell

The behavior of the cell is modeled as the net shown in Figure 5.2. The transition *cellular respiration* models the fundamental chemical reaction which creates energy from sugar and six oxygen molecules: $C_6H_{12}O_6 + 6O_2 \rightarrow 6H_2O + 6CO_2$.

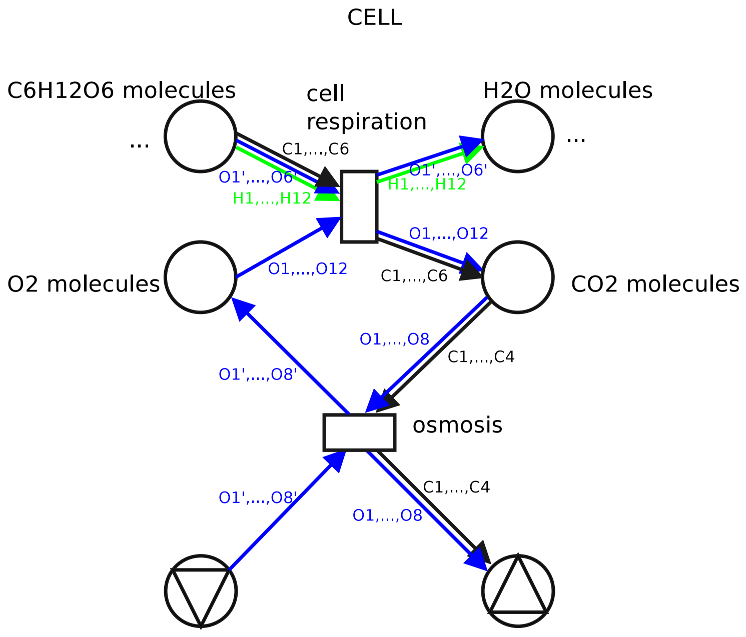


Figure 5.8: The net modeling the behavior of a cell

Again, the paths can be derived by looking at the inscriptions on arcs. The three arcs connecting the place which contains the sugar, and the transition which models the reaction represent twenty-four paths: six for atoms of carbon, six for atoms of oxygen, and twelve for atoms of hydrogen. All these atoms correspond to a molecule of sugar ($C_6H_{12}O_6$). In the same way twelve atoms of oxygen are selected, and when the transition fires all these atoms are put in such a way that the six molecules of water and the six molecules of carbon dioxide are represented correctly.

Finally, the transition *osmosis* models an exchange of oxygen and carbon dioxide with the blood. Carbon dioxide molecules are sent to the red blood cells present in the blood, and oxygen molecules are absorbed from the the blood.

The Red Blood Cell

The last net we are going to discuss represent the behavior of a red blood cell.

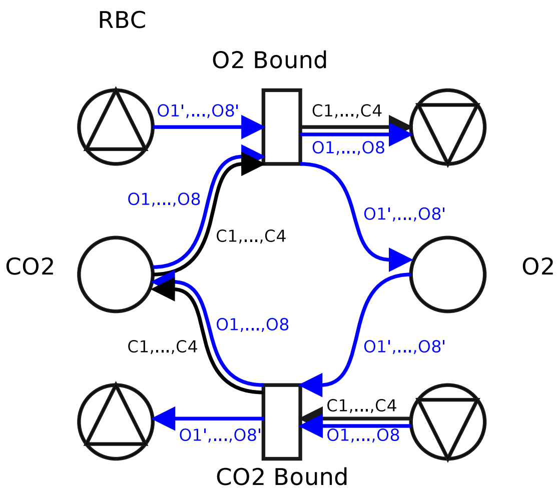


Figure 5.9: The net modeling the behavior of a red blood cell

There are two transitions: *O2 Bound* and *CO2 Bound*. The former is used to give away carbon dioxide, and at the same time to take oxygen. The latter transition does the opposite operation: it gives away carbon oxygen to a cell, and in exchange it takes carbon dioxide.

Molecules of carbon dioxide are stored in the place named *CO2*. The way molecules are represented is the same as in the cell net. A molecule of carbon dioxide is represented as an atom of carbon (an empty agent), and two atoms of oxygen (again empty agents) located in the place *CO2*.

When more than one molecule is present in this place all the atoms are represented as empty agents. The drawback of this solution is that the identify of each molecule of carbon dioxide is lost. This is not a bid deal because every molecule is equivalent to the others. If the designer wants to distinguish between them, a solution can be to represent a molecule of carbon dioxide as a structured agent which contain an atom of carbon and to atoms of oxygen.

5.3 Modeling a Class of Membrane System

In this example it is shown how it is possible to model a class of membrane systems using generalized hypernets. Membrane systems are computational model inspired by the behavior of the living cell. They are composed by a set of nested membranes, organized in a tree like structure, which may contain molecules. Each membrane has a set of rule used to modify the content of the membrane, or to import/export molecules to the outer membrane. Rules are applied until there are no rules enabled. Then, the result of the computation are the molecules found in the outer membrane. The characteristics of the class of membrane systems which has been chosen for this example are explained in Section 5.3.1.

The hierarchy of agents in a hypernet resembles the hierarchy of membranes in a P system, and the mechanism of consortia can be seen as a way to exchange molecules across a membrane. The main idea of this translation is quite simple:

each membrane and each individual molecule in the P system is represented by an agent in the hypernet. Molecule agents are unstructured, that is, they are simple tokens, like in usual nets, and can only be passively moved by the active components. Membrane agents, viceversa, are nets, with places that can contain molecules agents, and places that can contain other membrane agents. Consortia correspond to rules of the P system, whereby molecules can be exchanged across a membrane.

It should be noted that hypernets would allow, in themselves, movement of membrane agents, so that the hierarchical structure of membranes could change. This capability is not exploited here, since we deal with P systems where only molecules move around, but might be useful in modelling more general kinds of systems.

The focus of this example is not in the computational aspects of the theory of P systems, but rather on modelling aspects. Consequently, we compare the two models on the basis of their reachable configurations.

5.3.1 P systems with symport/antiport rules

Many kinds of membrane systems have been investigated during the last years. One of the most studied variant of the general model of P systems was introduced in [45] under the name of systems with symport/antiport rules. Those terms came from two membrane transport mechanisms. Whereas the term symport stands for the biological process by which two molecules pass together across a membrane, when the two molecules pass simultaneously, but in the opposite direction, the process is called antiport.

The class of membrane systems with symport/antiport rules is a class of purely communicating P systems, where the objects involved in the computation only pass through membranes. This means that the objects involved never change and a sort of conservation law for objects is observed during the entire evolution of the system.

Many results on this kind of P systems, especially about their computational power, can be found in [46],[41],[42],[23]. Here we provide a simplified version of the definition of P system with symport/antiport rules supplied by Păun [47].

A Formal definition

Formally, we define a P system with symport/antiport rules (of degree m), as a construct of the form

$$\Pi = (O, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m),$$

where:

- O is the (finite and non empty) alphabet of objects
- the membrane structure $\mu = (N, E, i)$ is a *rooted tree* underlying Π, where $N = \{1, 2, \ldots, m\}$ is the set of nodes and each node in N defines a membrane of Π. The set $E \subseteq N \times N$ defines the edges. For each node $j \in N$, the membrane associated to the node j contains all the membranes associated to the children of j. i is the root of the tree and hence the skin membrane (the outermost membrane of the system)

- $w_1, w_2, \ldots, w_m$ are multisets over O representing the objects present in the regions $1, 2, \ldots, m$ of the membrane structure μ in the initial configuration of the system

- $R_1, R_2, \ldots, R_m$ are finite sets of evolution rules associated with the membranes of μ. Moreover we impose $R_i = \emptyset$, where i is the skin of the membrane structure. This clause ensures that the external membrane is impermeable and hence the total number of objects involved in the computation is finite (and constant); this is required if we want to build hypernets with a finite number of agents

The term molecule is sometimes used in the rest of the paper when referring to the objects in the membranes of the P system.

As said above, each rule governs the communication through a specific membrane and can be of two kinds, symport rule or antiport rule. A symport rule is of the form (u, in) or (u, out), where u is a multiset over O, stating that all the objects of u pass together through a membrane, entering in the former case and exiting in the latter. For example, after the application of the symport rule (u, in) (contained in a membrane i) the multiset associated to this membrane (denoted by w_i) will contain all the objects previously present plus the objects present in u. The multiset associated to the membrane that contains i, will contain all the object previously present minus the object present in u. Similarly, an antiport rule is of the form $(u, out; v, in)$, where u and v are multisets over O, stating that when u exits, at the same time, a multiset v must enter the membrane.

The P system described above evolves from configuration to configuration by the application of a multiset of rules in each membrane. Formally, a configuration is a tuple $C = (v_1, v_2, \ldots, v_m)$ and $C \overset{\hat{R}}{\Rightarrow} C'$ denote that C evolves into C' due to the application of $\hat{R}$, where $\hat{R} = (\bar{R}_1, \bar{R}_2, \ldots, \bar{R}_m)$ is a multi-rules vector applicable to C and $\bar{R}_j$ is a multiset over R_j.

The evolution of the system is non-deterministic and maximally parallel; this means that in each step we look to the set of rules, and we try to find a multiset of rules by choosing their multiplicities non-deterministically. The multiset of rules must be applicable to the multiset of objects available in the respective regions and must be maximal, that is, no further rule can be added to it.

Figure 5.10 shows a fragment of a P system with symport rules. The system depicted here consists of two nested membranes: j, the inner membrane, and i, the outer, which we assume to be a membrane contained in a larger membrane structure. The set of rules associated to i is $R_i = \{(b, out), (a^2, out)\}$ and the set of rules of j is $R_j = \{(ab^2, in), (a, out)\}$. In the same way we define the initial multisets of objects $w_i = ab^3$ and $w_j = a^2$.

In this configuration the rules r_1, r_3, r_4 are enabled and a multi-rules vector can be built with this rules in a maximally parallel manner, i.e.: the multi-rules vector $\hat{R} = (\{r_1\}, \{r_3, r_4, r_4\})$ is applicable to the initial configuration. Note that other multi-rules vectors can be applied to the same configuration. The application of $\hat{R}$ leads to a new state where the objects in the membrane i are a^2 and the objects in the membrane j are ab^2.

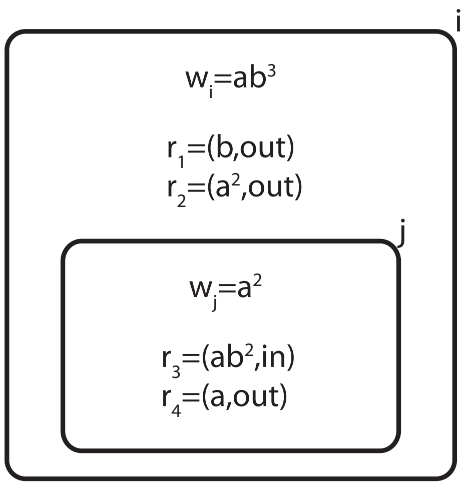


Figure 5.10: Fragment of a symport/antiport P system

The Hypernet Translation

A P system with symport/antiport rules and an impermeable external membrane can be modeled as a hypernet. The idea is to associate to each membrane a structured agent which contains a transition for each rule belonging to this membrane and to adjacent membranes. Each molecule of the P system is modeled as an empty token located in a place of the hypernet. For each type of molecule a place is present in the agent modeling the membrane. This place will contain all the molecules of this type present in the P system. Each transition modeling a rule is connected to the places with as many arcs as the number of required molecules.

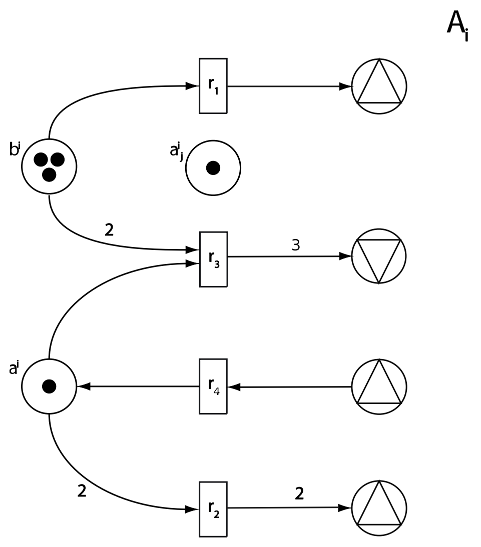


Figure 5.11: The hypernet corresponding to membrane i

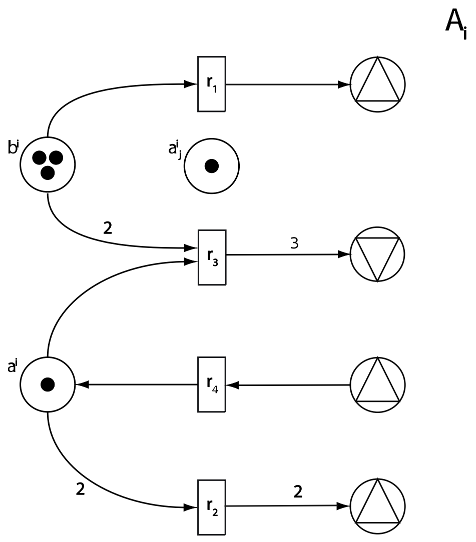


Figure 5.12: The hypernet corresponding to membrane j

For example, Figure 5.11 and Figure 5.12 show the fragment of the hypernet corresponding to the P system of Figure 5.10. The two membranes i and j are modelled by the agents A_i (Figure 5.11) and A_j (Figure 5.12). The local place $a^i_j \in P_i$, which contains (as token) the agent A_j, reflects the fact that the membrane j is nested inside i, while the local places a^i, b^i represent the presence of molecules a and b respectively, inside the agent A_i (this is also true for a^j, b^j and the membrane j). Then $\{r_1, r_2, r_3, r_4\} \subseteq T_{\mathcal{N}}$ are transitions built from the evolution rules of the membrane system. The initial hypermarking matches the initial configuration of the P system.

Note that for simplicity reasons in this example, we have just used symport rules, but antiport rules can also be modelled, as shown above in this Section.

It is possible to prove that the reachable configurations associated to a P system with symport/antiport rules correspond to the reachable hypermarkings of the hypernet which models the P system. For detailed definitions, and proofs see [5].

Chapter 6

Unfoldings

The importance of designing and analyzing concurrent systems has already been advocated in this thesis. One of the problem which make the analysis of systems so difficult is the *state space explosion* problem, i.e., even small systems may exhibit a huge number of possible behaviors which exponentially grow when the number of concurrent components of the system grow.

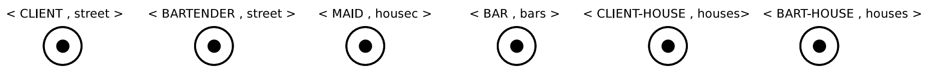
Answering questions about the system using behavioral techniques (like model checking) implies the exploration of all these behaviors. However, a common situations in concurrent systems is that several different behaviors lead to the same result, because they represent different ordering of the same set of concurrent actions, i.e., they are different interleaving. In practice, n independent actions can occur in $n!$ different orders leading to the same result.

Unfoldings represent one way to exploit this observation. They are a mathematical structure which explicitly represent concurrency and casual dependence between actions, but hide information about all the possible interleavings of concurrent actions. Therefore, they are exponentially smaller than naive mathematical structures which represents all the possible interleavings, like case graphs. Obviously, behavioral techniques which explore all the possible states of a concurrent system, like model checking, will be more efficient if the input data structure which represents the behavior of the system is small.

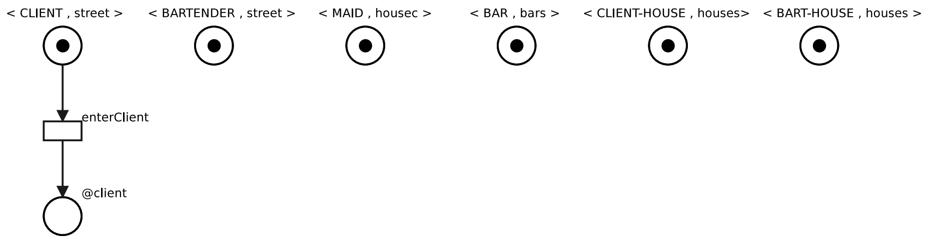
The aim of this Chapter is to find a way to apply the unfolding technique directly on the hypernet model, without computing the expensive 1-safe expansion. Section 6.1 informally introduces the concept of process of a hypernet by means of an example. In Section 6.2 the preliminary definitions needed in the rest of the chapter are introduced. Section 6.3 contains an axiomatic definition, and an inductive one of processes of a hypernet. Finally, Section 6.4 contains the definition of the notion of unfolding of a hypernet.

6.1 An Introductory Example

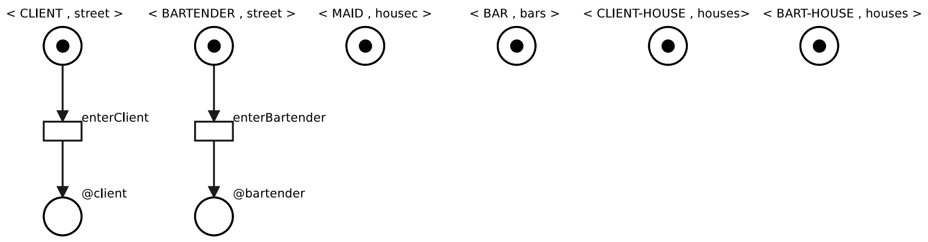
How is it possible to represent the behavior of a concurrent system without explicating all the possible interleavings of concurrent actions? The answer is to represent them by using a suitable mathematical structure which is able to handle concurrency. Traditionally, the way to do that in the Petri net theory is by using another labelled Petri nets with some peculiar characteristics, like the absence of conflicts, and the absence of loops. These labelled Petri nets are called *processes* of the original system. In the literature there is a difference between processes which contain information about several run of the system, and processes which only contain information about a single run of the system . The formers are called *branching* processes (see [17]), and the Petri nets which represent these processes contain forward conflicts. The latter are simply called processes (see [6]), and the Petri nets which represent these processes cannot contain conflicts. The unfolding of a Petri net N is a particular, and usually infinite branching process which contains all the possible behaviors of N.



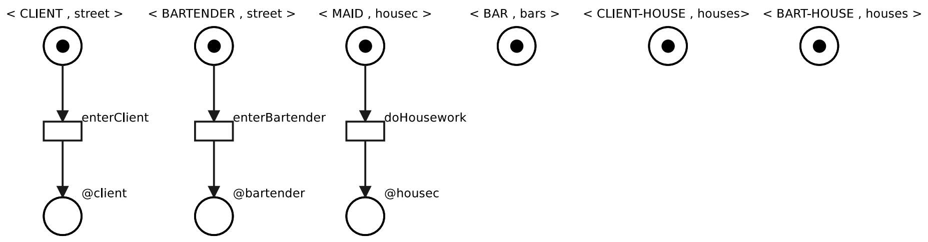
(a) The process representing the initial marking. Real names of places (transitions) of the canonical net are not shown. Labels of the nodes of the process are drawn instead



(b) The extension of the process in Figure 6.1(a) with the consortium corresponding to transition *enterClient*. If the agent is clear from the context labels of new places are shortened to the form @place. @client is a shortened version of the label $\langle CLIENT, client\rangle$

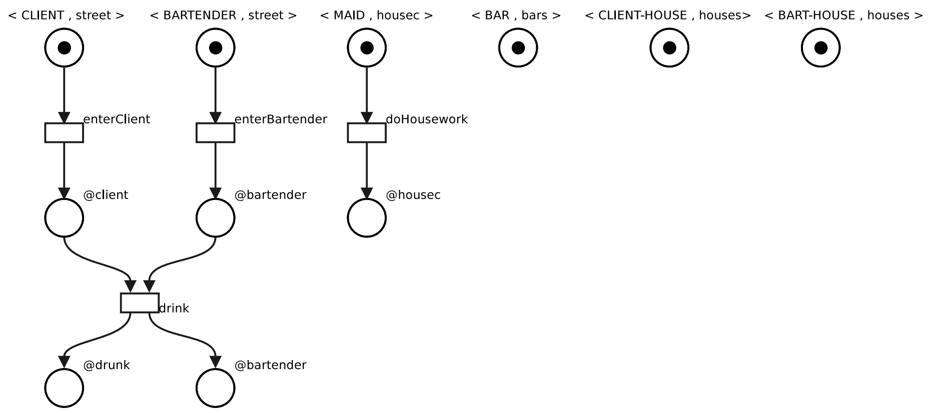


(c) The extension of the process in Figure 6.1(b) with the consortium corresponding to transition *enterBartender*

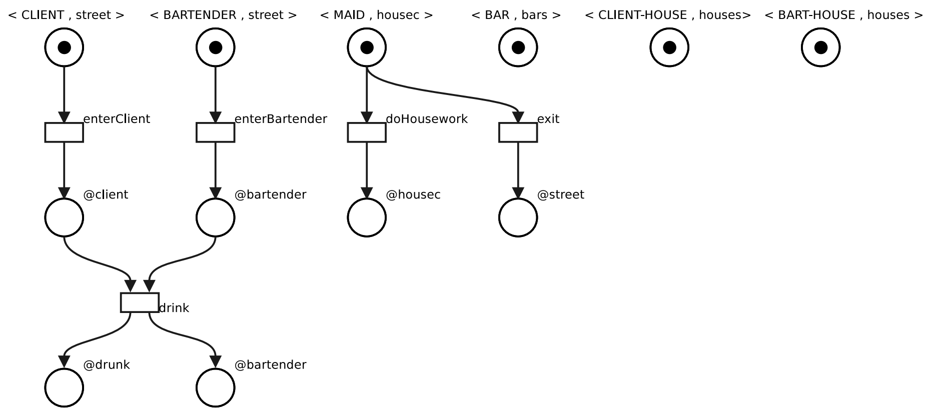


(d) The extension of the process in Figure 6.1(c) with the consortium corresponding to transition *doHousework*. The figure continues in the next page.

McMillan proposed in [43] an algorithm for the construction of a finite initial



(e) The extension of the process in Figure 6.1(d) with the consortium corresponding to transition *drink*



(f) The extension of the process in Figure 6.1(e) with the consortium corresponding to transition *exit*

Figure 6.1: A branching process of the Bar example (Section 5)

part of the unfolding which contains full information about the reachable states. The initial part of the unfolding satisfying this property is called *finite complete prefix*. Esparza, Romer, and Voegler showed in [22] that McMillan finite complete prefix of the unfolding is sometimes larger than necessary (exponentially larger in the worst case). They proposed a refinement of the McMillan algorithm which overcome this problem. Khomenko, Koutny, and Voegler developed a general technique for truncating net unfoldings, parameterized according to the level of information about the original unfolding one wants to preserve in [29]. Moreover, they proposed a new notion of completeness of a truncated unfolding. From an application point of view the unfolding technique has been successfully applied to verification problems, like the check of relevant properties of speed independant circuits [30], or unfolding-based model checking algorithms [19, 18, 21].

In the rest of this introductive section it will be informally explained how it is possible to build a branching process of a hypernet.

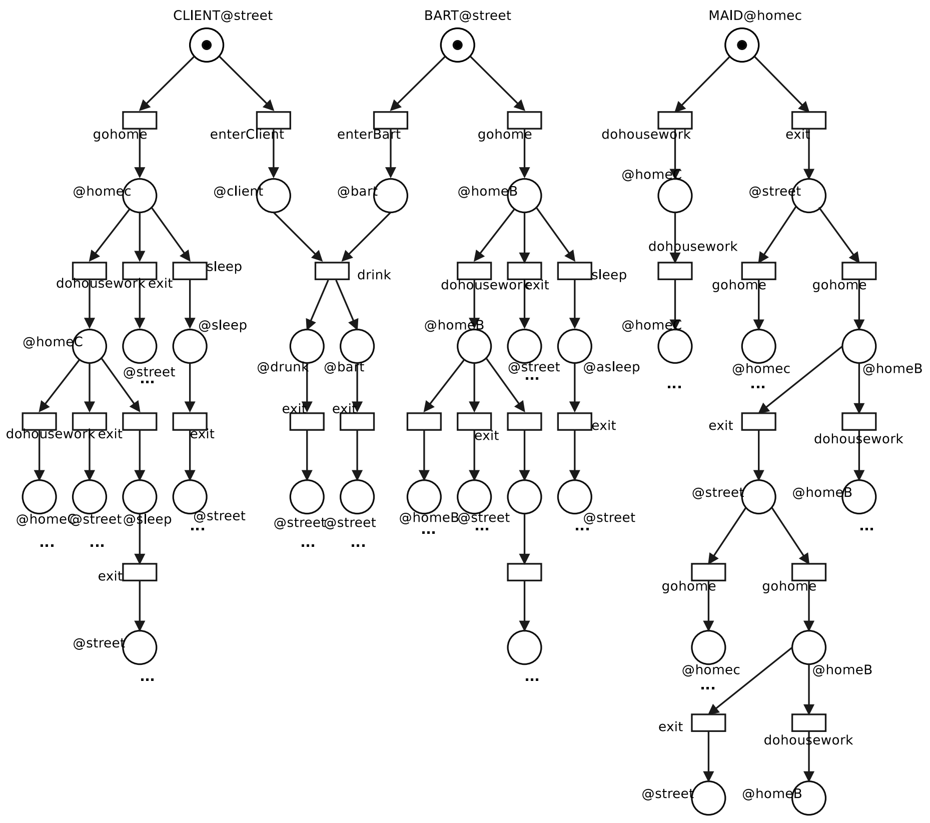


Figure 6.2: A fragment of the unfolding of the Bar example (Section 5)

Consider the hypernet modeling the Bar example of Section 5. Figure 6.1 shows how it is possible to incrementally build a process of the Bar example. For each agent but the system net a place $\langle A, p\rangle$ is added (Figure 6.1(a)). This place represents the fact that the agent A is located at place p in the initial marking. A label representing this fact is added too. After that, a consortium enabled in the initial marking is chosen and added, together with a set of places representing the hypermarking reached after the firing of the consortium.

For example, in Figure 6.1(b) the consortium which models a client entering the bar has been added. For simplicity, in all the figures the transitions are not labelled with the name of the consortia, but with a significant label. This name, together with input/output places, can be used to reconstruct the consortium. The real names of the two transitions labelled *enterClient* in the agents $WORLD$ and BAR of the Bar example are t_1, t_6 respectively, and assuming they insist on two paths named g_1 and g_2 respectively, the consortium modeled by the transition added in Figure 6.1(b) is $(enterClient, \{t_1, t_6\}, \{CLIENT\}, \delta\langle g_6\rangle = \langle g_1\rangle, \gamma\langle CLIENT\rangle = \langle g_1\rangle)$. Finally, a place $\langle CLIENT, client\rangle$ representing the new location of agent $CLIENT$ is added too. It is only labelled @*client* because instead of using labels with the form $\langle agent, place\rangle$, labels with the form @*place* are employed if the name of the agent is clear from the figure.

In the same way the consortia corresponding to the transitions labelled *enterBartender*, *doHousework*, *drink*, *exit* are respectively added in Figures 6.1(c), 6.1(d), 6.1(e), 6.1(f).

Note that new transitions, and new places are always added even if the current Petri net already contains transitions and places carrying the same labels. For instance, when going from 6.1(c) to 6.1(d) by adding a new transition labelled *doHousework*, a new place labelled $\langle MAID, housec\rangle$ is added even though 6.1(c) already contains a place with this label.

A branching process is usually a *partial* view of the behavior of a hypernet. However, as for Petri nets, there is a branching process (which is usually infinite) in which every possible run of the hypernet is represented. This maximal branching process will also be called *unfolding*. For example, a fragment of the unfolding of the hypernet representing the Bar example is shown in Figure 6.2. Starting from the set of places representing the initial hypermarking, each enabled consortium is added to the unfolding in the same way discussed before. Then, starting from each set of places which represents a hypermarking $\mathcal{M}$ in the unfolding, each consortium enabled in $\mathcal{M}$ is recursively added to the unfolding. For obvious reasons, the figure has been truncated at some point, and three dots has been added to indicate a cut of the unfolding. The formal definition of unfolding of a hypernet will be given in Section 6.4.

The questions which are discussed in the next sections are the following: which are the fundamental characteristics a labelled Petri net should have to be a process of a hypernet? Is there an inductive procedure which can be used to build all the possible processes of a hypernet? Is there a way to represent all the reachable hypermarking in a *finite complete prefix* of the unfolding?

6.2 Preliminary Definitions

In this Section some notions given in the literature and concerning branching processes are introduced. They will be used in the following sections. The definitions and results discussed in this Section are mostly taken from [20, 22, 29, 17].

Intuitively, a branching process will either be a Petri net containing no events (like the process in Figure 6.1(a)), or the result of extending a branching process with an event, or the union of a set of branching processes. Clearly, unions of Petri nets play a central role in the formal definition of branching processes. They are formally defined component-wise [20]:

Definition 25. *The union $\bigcup N$ of a (finite or infinite) set N of Petri nets is defined as the Petri net:*

$$\bigcup N = (\bigcup_{(P,T,F,M_0)\in N} P, \bigcup_{(P,T,F,M_0)\in N} T, \bigcup_{(P,T,F,M_0)\in N} F, \bigcup_{(P,T,F,M_0)\in N} M_0)$$

Unions, however, must be handled with some care. Unless the names of the nodes are well chosen, they can generate "wrong" nets that do not correspond to the intuition behind the notion of branching process. For example, the union of two copies of the net in figure 6.1(b), in which the labeling of the nodes is the same, but their names are different, is a net with fourteen places and two transitions.

The use of canonical nets, as proposed in [20]. solves this problem. In a canonical net, to a node with a label x it is assigned a name in the form (x, X), where X is a set containing the names of the input places/transitions of the node x.

Definition 26. *Let Σ be an alphabet, and let $\mathcal{C}$ be the smallest set satisfying: if $x \in \Sigma$ and X is a finite subset of $\mathcal{C}$, then $(x, X) \in \mathcal{C}$. A $\mathcal{C}$-*net *is a net (B, E, F) such that:*

- *$B \subseteq \mathcal{C}$,*
- *$E \subseteq \mathcal{C}$,*
- *$F \subseteq (B \times E) \cup (E \times B)$ is such that $\forall\ (\hat{x}, \hat{y}) \in F,\ \hat{x} = (x, X) \in B$ (or E) $\wedge$ $\hat{y} = (y, Y) \in E$ (or B) $\Rightarrow (x, X) \in Y$*
- *if $(x, X) \in B \cup E$, then $X = {}^{\bullet}(x, X)$*

In a canonical net each node is given a unique name which contains information about its past. In this way, names and labels are in some sense tied, and if two nets which have the same labeling are fused together by a union, then no redundant places are added.

Example 6. *Consider the net in Figure 6.1(a), and let H be the hypernet of the Bar example. We choose the alphabet Σ as the union of all consortia, and the union of all the pair $\langle$ agent, place $\rangle$ (as defined later at the beginning of Section 6.3.1). Then, the places of the net in Figure 6.1(a) are given the names $(\langle CLIENT,\ street\rangle, \emptyset), (\langle BARTENDER, street\rangle, \emptyset), \ldots$. In Figure 6.1(b) a transition named $(\Gamma,\ \{(\langle CLIENT,\ street\rangle, \emptyset)\})$ is added, where Γ is the consortium corresponding to transition enterClient (remember that in the figures we do not use the name of the consortia). Finally, a place named $(\langle CLIENT, client\rangle, \{(\Gamma, \{(\langle CLIENT,\ street\rangle,\ \emptyset)\})\})$ is added too.*

The reader may think that canonical names are a useless complication, and may be worried about their length. But they are just a mathematical tool which allow the definition of infinite branching processes. From a practical point of view there is no need to use them and they will not be considered in the examples. However, from a theoretical point of view they are essential to avoid the issue related to union of nets discussed before.

As it has been discussed in the introduction of this chapter, not every Petri net can be used to represent a process of a concurrent system. For example, the acyclicity of the net, and the absence of backward conflicts are two prerequisites of a net which represent a process. The following Definition introduce the concept of *occurrence net*, Petri nets with constraints which make them suitable to be used to represent the behavior of concurrent systems. This definition, and the notations that will be introduced later in this chapter, are taken from [17, 22, 29, 21].

Definition 27. *An* occurrence net *is a $\mathcal{C}$-net $O = (B, E, F)$ such that:*

1. *$\forall b \in B,\ |{}^{\bullet}b| \leq 1$,*
2. *O is* acyclic*, or equally $(B \cup E, F^{+})$ is a partially ordered set,*

3. *O is* finitely preceeded, *i.e.:* $\forall x \in B \cup E, \ |\{y \mid yF^+x\}| < \infty$,

4. *$Min(O) \subseteq B$, where $Min(O)$ is the set of nodes, which are minimal w.r.t. F^+.*

Clearly, all the nets of Figure 6.1 satisfy these requirements and are occurrence nets. Now, the notions of causality, conflicts, and concurrency typical of Petri nets are introduced. Let $x, y \in B \cup E, \ x \neq y$. The following notations will be used in the paper:

- x *causally precedes* y, denoted $x < y$, iff there exists a non-empty oriented path from x to y,

- x, y are *in conflict*, denoted $x\#y$, iff there exist two non-empty paths $st_1...x$ and $st_2...y$ such that $s \in B$, and $t_1 \neq t_2$

- x, y are *concurrent*, denoted $x \ co \ y$, iff not $x < y$ nor $y < x$ nor $x\#y$.

For example, in Figure 6.1(f) the place whose label is $\langle CLIENT, street\rangle$ causally precedes the place whose label is @*client*, places @*housec* and @*street* are in conflict, and places @*client* and @*bartender* are concurrent. The same can be said for transitions: *enterClient* and *drink* are causally related, *enterClient* and *enterBartender* are concurrent, and *doHousework* and *exit* are in conflict.

The notion of *configuration* is important for branching processes. Given a set of events C of an occurrence net, it is interesting to know if it represents a "valid" set of action of the system . With valid we mean that there exists a set of actions of the system which is an occurrence sequence, and which correspond to the labels associated to the set C. The following definition contains the conditions a set of events must satisfy in order to be a configuration [29].

Definition 28. *A* configuration *C of an occurrence net is a set of events such that:*

1. *C is causally closed, i.e.:* $e \in C \Rightarrow \forall e' < e, \ e' \in C$

2. *C does not contain conflicts, i.e.:* $\forall e, e' \in C, \ \neg(e\#e')$

From now on, let $O = (B, E, F)$ be an occurrence net. Given $e \in E$, with $[e]$ the backward causal closure of e is denoted, i.e.: $[e] = \{e' \in E \mid e' < e\}$. Given a configuration C and a set of events E', $C \oplus E'$ denotes the fact that $C \cup E'$ is a configuration, and $C \cap E' = \emptyset$. The set E' is a *suffix* of C, and $C \cup E'$ is an *extension* of C. A set of conditions B' such that $\forall \ b, b' \in B', b \ co \ b'$ is called *co-set.* A *cut* is a maximal co-set.

For example, in Figure 6.1(f) the set of events $S = \{$*enterClient*, *enter-Bartender*$\}$ is a configuration, and the *backward causally closure* of event *drink* contains *drink*, *enterClient*, and *enterBartender*, i.e.: $[drink] = \{drink, enterClient, enterBartender\}$. Moreover, $S \oplus \{enterBartender\}$ holds, infact $S \cup \{$*enter-Bartender*$\} = [drink]$. Therefore, the set $\{enterBartender\}$ is a suffix of S, and the set S is a prefix of $[drink]$. The set of conditions $\{$@*client*, @*bartender*$\}$ is a co-set, and the co-set $cs = \{$@*client*, @*bartender*, $\langle MAID, housec\rangle$, $\langle BAR, bars\rangle$, $\langle CLIENT\text{-}HOUSE, houses\rangle$ $\langle BART\text{-}HOUSE, houses\rangle\}$ is a cut.

Finite configurations and cuts are tightly related. Let C be a configuration of O, it is possible to prove that the co-set Fin(C), defined below, is a cut [17]:

$$\text{Fin}(C) = (Min(O) \cup C^{\bullet}) \setminus {}^{\bullet}C$$

For example, $\text{Fin}(S) = \{\langle CLIENT, \textit{street}\rangle$, $\langle BARTENDER, \textit{street}\rangle$, $\langle MAID, \textit{housec}\rangle$, $\langle BAR, \textit{bars}\rangle$, $\langle CLIENT\text{-}HOUSE, \textit{houses}\rangle$, $\langle BART\text{-}HOUSE, \textit{houses}\rangle\} \setminus \{\langle CLIENT, \textit{street}\rangle, \langle BARTENDER, \textit{street}\rangle\} \cup \{@\textit{client}, @\textit{bartender}\} = cs$

6.3 Branching Processes

In this section two notions of branching process of a hypernet are introduced. The first one is *axiomatic*: we define a set of conditions a pair ⟨net, mapping from net's element to hypernet's elements⟩ must satisfy in order to be a process of the hypernet. The second one is *inductive* and formalizes the steps to follow to build a branching process, like it has been done in Example 6.1: starting from a process which represents the initial hypermarking we identify all the steps which can be done to inductively build a process. Finally, we prove a theorem which states that nets which satisfy the conditions of the first definition are the same of the nets built using the steps defined in the second definition.

First, a preliminary notion concerning a technical question is introduced: when dealing with processes of a hypernet the easier way to represent a hypermarking is to consider a set of pairs ⟨agent/place⟩. Given a hypermarking $\mathcal{M}$, it is possible to associate a set in which there is a pair ⟨agent/place⟩ for each agent but the system net, meaning that an agent is marked in a determined place. The following definition formalizes this concept.

Definition 29. *Given a hypernet $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$, and a hypermarking $\mathcal{M}$, the* hypercase set *associated to $\mathcal{M}$, denoted* $\text{SET}(\mathcal{M})$*, is the set of pairs agent/place such that* $\text{SET}(\mathcal{M}) = \{\langle A, p\rangle \mid \mathcal{M}(A) = p\}$.

In the same way it is interesting to have a way to do the opposite operation, i.e.: to start with a set S which represent a hypercase set, and obtain the corresponding hypermarking. The following proposition identifies necessary and sufficient conditions a set S must satisfies in order to be a hypercase set.

Proposition 2. *A set S is a* hypercase set *iff the following two conditions hold:*

- $|S| = |\mathcal{N}| - 1$
- *$\exists$ a hypermarking $\mathcal{M}$ such that $\forall A_i \in \mathcal{N} \setminus A_0, \exists b \in S$ such that $b = \langle A_i, \mathcal{M}(A_i)\rangle$*

Given a hypercase set S, $\text{HM}(S)$ *denotes the hypermarking $\mathcal{M}$.*

Note that, given a set S, there can be only one hypermarking which satysfy the second condition of the proposition.

Example 7. *Consider the hypermarking $\mathcal{M}\langle CLIENT, BARTENDER, MAID, BAR, CLIENT\text{-}HOUSE, BART\text{-}HOUSE\rangle = \langle \textit{street}, \textit{street}, \textit{housec}, \textit{bars},$*

houses, *houses*$\rangle$ *of the hypernet representing the Bar example. The hypercase set associated to* $\mathcal{M}$ *is:*

$$\begin{aligned}\mathrm{SET}(\mathcal{M}) = \{&\langle CLIENT, street\rangle, \langle BARTENDER, street\rangle, \langle MAID, housec\rangle,\\ &\langle BAR, bars\rangle, \langle CLIENT\text{-}HOUSE, houses\rangle,\\ &\langle BART\text{-}HOUSE, houses\rangle\} = S\end{aligned}$$

Viceversa the hypermarking associated to the set S *(which satisfy the conditions of proposition 2) is* $\mathrm{HM}(S) = \mathcal{M}$.

In the following subsection the axiomatic definition of branching process of a hypernet is given, and a theorem showing that the cuts of a branching process correspond to reachable markings of the hypernet is demonstrated.

6.3.1 Axiomatic Definition

From now on let $H = (\mathcal{N}, \sigma_{\mathcal{N}}, \sigma_G, \lambda)$ be a hypernet, and let $\mathcal{M}_0$ be its initial hypermarking. Moreover, let $\Sigma_1 = \{\langle A, p\rangle \mid p \in L_{\mathcal{N}}$ and $A \in \mathcal{N}\backslash A_0\}$ be a set containing an element for each combination $\langle$agent/place$\rangle$ in H, and $\Sigma_2 = \{CONS(H)\}$ be the set of consortia in H.

.

Definition 30. *A* branching process $\beta = (O, h)$ *of* $(H, \mathcal{M}_0)$ *is an occurrence net* $O = (B, E, F)$ *with a labelling function* $h : B \cup E \to \Sigma_1 \cup \Sigma_2$ *satisfying:*

1. $h(B) \subseteq \Sigma_1$ *and* $h(E) \subseteq \Sigma_2$ *(h preserves the nature of the nodes)*
2. *Let* $h(e) = \Gamma = (l, \tau, \mathrm{PASS}, \delta, \gamma)$ *be the consortium associated to an event* e *through* h. *Then:*
   (a) $h({}^\bullet e) = \{\langle A, p\rangle \mid A \in \mathrm{PASS}$ *and* $p = {}^\bullet\gamma(A)\}$ *(each input place of an event e corresponds to a pair* $\langle$ *passive agent,input place* $\rangle$ *in the consortium* $h(e)$*),*
   (b) $h(e^\bullet) = \{\langle A, p\rangle \mid A \in \mathrm{PASS}$ *and* $(p = trg(A)\}$, *where* $trg(A) = \gamma(A)^\bullet \wedge \gamma(A) \notin G^{Out} \ \vee \ p = \delta(\gamma(A))^\bullet \wedge \gamma(A) \in G^{Out})$. *(each output place of an event e corresponds to a pair* $\langle$*passive agents/output place*$\rangle$ *in the consortium* $h(e)$.
3. $\forall e_1, e_2 \in E$, ${}^\bullet e_1 = {}^\bullet e_2 \wedge h(e_1) = h(e_2) \Rightarrow e_1 = e_2$ *(there are no duplicated events),*
4. *Let* $A \in \mathcal{N}$ *and* $p \in P_{\mathcal{N}}$. $\mathcal{M}_0(A) = p \iff \exists b \in Min(O)$ *and* $h(b) = \langle A, p\rangle$ *(The restriction of h to* $Min(O)$ *corresponds to the initial hypermarking).*
5. $\forall (x, X) \in B \cup E, h((x, X)) = x$

In order to avoid confusion, it is convenient to have two different names for transitions/places of the hypernet, and for transition/places of its processes. Transitions belonging to the process will be called events, and places will be called conditions.

The set containing all the configurations of a branching process β is denoted by $Conf(\beta)$.

In order to prove that the configurations of a branching process correspond to reachable hypermarkings of the hypernet the following lemma is needed.

Lemma 1. *Let C be a finite configuration of O, and $\{e\}$ a suffix of C. Then $^\bullet e \subseteq \mathrm{Fin}(C)$.*

Proof. By contradiction assume $p \in {}^\bullet e$, and $p \notin \mathrm{Fin}(C)$. Since $\mathrm{Fin}(C)$ is a cut, then $\exists q \in \mathrm{Fin}(C), p \neq q$ such that $\neg(p\ co\ q)$. Therefore $q < p \ \vee\ p < q \ \vee\ p\#q$ must be true. First, $p\#q$ is impossible because $C \cup \{e\}$ is conflict free. If $p < q$, then e is in conflict with $^\bullet q$, which is impossible. If $q < p$ then there is at least an event e' between p and q such that $q = {}^\bullet e'$. But that's impossible because $q \in \mathrm{Fin}(C)$. □

The following theorem shows that the image through h of the cut associated to a configuration C, i.e., $h(\mathrm{Fin}(C))$ is a hypercase set. Moreover, it is shown that the hypermarking corresponding to that hypercase set, i.e., $\mathrm{HM}(h(\mathrm{Fin}(C))$ is reachable from the initial hypermarking. With $\mathrm{Mark}(C)$ the hypermarking associated to a configuration C is denoted, i.e.: $\mathrm{Mark}(C) = \mathrm{HM}(h(\mathrm{Fin}(C))$.

Theorem 5. *Let $\beta = (O, h)$ with $O = (B, E, F)$ be a branching process of $(H, \mathcal{M}_0)$.*

$$C \subseteq E \textit{ is a finite configuration of } O \Rightarrow$$
$$h(\mathrm{Fin}(C)) \textit{ is a hypercase set } \wedge \mathrm{Mark}(C) \in [\mathcal{M}_0\rangle$$

Proof. The proof is by induction starting with the empty configuration, and then adding a single event as a suffix of a given configuration.

Induction basis: Let C' be the empty configuration. Then $\mathrm{Fin}(C') = Min(O)$, which is a hypercase set with $\mathrm{Mark}(C) = \mathcal{M}_0$, a reachable hypermarking.

Induction step: Let C be a configuration such that the theorem holds for it, where $\mathcal{M} = \mathrm{Mark}(C)$. Let $C' = C \cup e$ be an extension of C, with $h(e) = \Gamma = (l, \tau, PASS, \delta, \gamma)$. By lemma 1, $^\bullet e \subseteq \mathrm{Fin}(C)$, and by condition 2a of Definition 30 $h(^\bullet e) = \{\langle A, p\rangle \mid A \in PASS \text{ and } p = {}^\bullet\gamma(A)\}$. Therefore, $\forall A \in PASS,\ \mathcal{M}(A) = \gamma(A)$, which implies $\mathcal{M}[\Gamma\rangle$.

Let $\mathcal{M}'$ be such that $\mathcal{M}[\Gamma\rangle\mathcal{M}'$.

$$\begin{aligned}
&\mathrm{SET}(\mathcal{M}') = \mathrm{SET}(\mathcal{M}) \cup \\
&\{\langle A,p\rangle \mid A \in C \text{ and } (p = trg(A)\} \backslash \{\langle A, p\rangle \mid A \in C \text{ and } p = {}^\bullet\gamma(A)\} \\
&\ = h(\mathrm{Fin}(C)) \cup h(e^\bullet) \backslash h(^\bullet e) = h(\mathrm{Fin}(C) \cup e^\bullet \backslash {}^\bullet e) = h(\mathrm{Fin}(C \cup \{e\})) = h(C')
\end{aligned}$$

□

6.3.2 Inductive Definition

Let b be a canonical net over the alphabet Σ. We call the *standard labeling* of b the function which assign its canonical name to each node , i.e.: $can((x, X)) = x$. The set of branching processes of a hypernet can be characterized in an inductive way.

Definition 31. *Let $(H, \mathcal{M}_\prime)$ be a marked hypernet. $BP((H, \mathcal{M}_0))$ is the smallest set of $\mathcal{C}$-nets satisfying:*

1. *Let $S = \{(\langle A_1, \mathcal{M}_0(A_1)\rangle, \emptyset), ..., (\langle A_n, \mathcal{M}_0(A_n)\rangle, \emptyset)\}$ be a set. The $\mathcal{C}$-net, having S as set of places and no events, belongs to $BP((H, \mathcal{M}_0))$.*

2. *Let $\beta \in BP((H, \mathcal{M}_0))$, let m be a cut of β such that $can(m)$ is a hypercase set, let $\Gamma = (l, \tau, \text{PASS}, \delta, \gamma)$ be a consortium enabled in $\text{HM}(can(m))$, and let $S = \{(\langle A, p\rangle \ , X) \in m \mid A \in \text{PASS}$ and $p = {}^\bullet\gamma(A)\}$. The $\mathcal{C}$-net, which is obtained by adding to β the event (Γ, S), and one place $(\langle A, trg(A)\rangle, \{(\Gamma, S)\})$ for each passive agent A in the consortium Γ, belongs to $BP((H, \mathcal{M}_0))$.*

3. *If $Y \subseteq BP((H, \mathcal{M}_0))$, then $\bigcup_{\beta \in Y} \beta \in BP((H, \mathcal{M}_0))$.*

Note that $\{\langle A_1, \mathcal{M}_0(A_1)\rangle, ..., \langle A_n, \mathcal{M}_0(A_n)\rangle\}$ is a hypercase set.

In the following subsection it is shown that the inductive and the axiomatic definitions of branching process of a hypernet, represents the same class of nets.-

6.3.3 Equivalence of Axiomatic and Inductive Definitions

The first lemma demonstrated in this section proves that each net built using the steps of Definition 31 satisfies the conditions of Definition 30.

Theorem 6. *If $\beta \in BP((H, \mathcal{M}_0)) \Rightarrow (\beta, can)$ is a branching process of H.*

Proof. **Induction basis**: We have to prove that the $\mathcal{C}$-net $\beta = (B, E, F)$ such that $B = \{(\langle A_1, \mathcal{M}_0(A_1)\rangle, \emptyset), ..., (\langle A_n, \mathcal{M}_0(A_n)\rangle, \emptyset)\}$, and $E = \emptyset$ satisfies the conditions of definition 30.

Condition 1: The function *can* maps the elements of B, to pairs $\langle A, p\rangle \in \Sigma_1$.

Condition 2a, condition 2b, and condition 3 are true because $E = \emptyset$.

Condition 4: For each agent but the system net there is exactly one place p in B such that A is marked in p.

Condition 5: True by the definition of the standard labeling function.

Induction step: Let $\beta' \in BP((H, \mathcal{M}_0))$ be such that (β', can) satisfies the conditions of Definition 30, and let β be the $\mathcal{C}$-net generated by adding to β' the event $e = \{\Gamma, S\}$, and one place $(\langle A, \gamma(A)^\bullet\rangle, \{(\Gamma, S)\})$ for each passive agent A in the consortium Γ. We will prove that (β, can) also satisfies the conditions of the axiomatic definition of branching process showed in Definition 30.

Condition 1: The only event added to β' is $e = \{\Gamma, S\}$ and $can(e) = \Gamma \in \Sigma_2$. Moreover, by construction each place $p = (\langle A, \gamma(A)^\bullet\rangle, \{(\Gamma, S)\})$ added to β' has the property: $can(p) \in \Sigma_1$.

Condition 2a: For event e we have $can({}^\bullet e) = \{\langle A, p\rangle \mid A \in C$ and $p ={}^\bullet \gamma(A)\}$.

Condition 2b: Similar to the previous condition.

Condition 3: Since β is canonical, two events with the same preset, and the same labelling are encoded in the same event.

Condition 4: No places without an input event are added. Therefore the set of minimal elements still satisfy this condition.

Condition 5: By definition the function *can* satisfies this property.

Now, let us consider a net $\beta \in BP((H, \mathcal{M}_0))$ which is the union of $Y \subseteq BP((H, \mathcal{M}_0))$. By hypothesis if $\beta' \in Y$, then (β', can) is a branching process of H.

Condition 1: The nodes in β keep the mapping of the components β', therefore by hypothesis the first condition is preserved.

Condition 2a and condition 2b: All the events of the components satisfy these two conditions, and no new events are added. Therefore also all the events of β satisfies this condition.

Condition 3: As for the previous part of the demonstration, since the net is canonical two events with the same preset, and the same labelling are encoded in the same place.

Condition 4: All the components β' have the same $Min(\beta')$ by hypothesis, and no new minimal elements are added with the union operator.

Condition 5: Again true by definition. $\square$

From now on let $\beta = (O, h)$ with $O = (B, E, F)$ be a branching process of $(H, \mathcal{M}_0)$. To prove that each pair (O, h) which satisfy the conditions of the axiomatic definition can be built using the steps defined in the inductive definition we proceed in the following way: first, we consider the net generated by an event e of O, denoted $Gen(e)$, which is a subnet of O built considering the backward causal closure of e. We prove that, together with the restriction of h to $Gen(e)$, it also is a branching process; then, it is proved that the net generated by an event e can be inductively constructed using the steps of the inductive definition; finally, it is proved that the union of the subnets generated by all the events of O is again O.

Definition 32. *Let $\beta = (O, h)$ with $O = (B, E, F)$ be a branching process of $(H, \mathcal{M}_0)$. Given $e \in E$, the net generated by the event e is denoted with $Gen(e) = (\{p \mid p \in Min(O) \vee \exists e' \in [e]$ such that $p \in e'^{\bullet}\}, [e])$.*

For example, the process generated by the event *drink* of the net in Figure 6.1(f) is a net composed of the places and the transitions that causally precede *drink* plus the places which represents the initial marking, as it is shown in Figure 6.3.

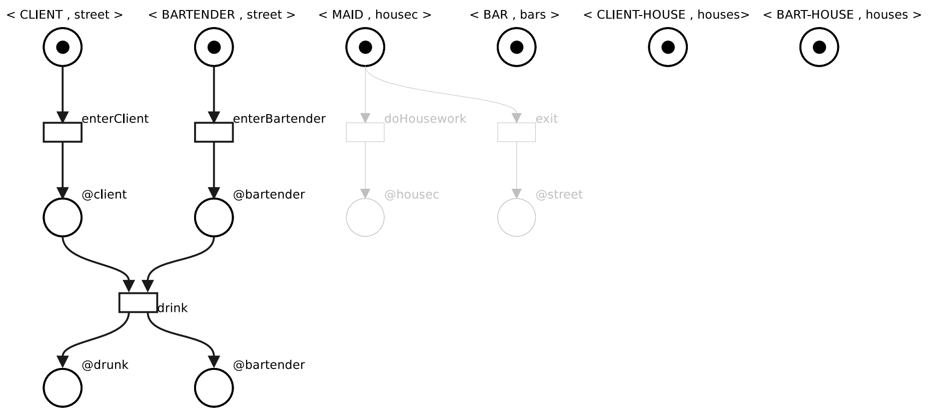


Figure 6.3: The net generated by an event

The next proposition proves that each net generated by an event of a branching process of a hypernet H is a branching process of H itself.

Proposition 3. *Let $Gen(e) = (B', E')$, and let h' be the restriction of h to $B' \cup E'$. $(Gen(e), h')$ is a branching process.*

Proof. Conditions 1, 2a, 2b, 3, 5 are trivial since the elements of the nets taken from a branching process, and all pre-post set are added. Condition 4 can be proved by induction on $[e]$ observing that by lemma 1 no initial places are added. $\square$

The next lemma shows that the net generated by an event of a branching process can also be generated by applying the steps of the inductive definition of branching process (Definition 31). The proof is by induction. It starts with the net generated by the empty set, which is equivalent to the net of step 1, and then it is shown that adding an event e to that net using step 2, or generating the net using $Gen(e)$ leads to the same conclusion. Some preliminary notions are required in ordeer to prove this lemma:

Let $\beta = (O, h)$ with $O = (B, E, F)$ be a branching process of $(H, \mathcal{M}_0)$.

Let $Y \subseteq B$ be a cut, $\Gamma \in Cons(H)$ be a consortium enabled in the hypermakring associated to Y, i.e.: $HM(Y)[\Gamma\rangle$. With $S_2(Y, \Gamma, O)$ the net obtained applying step 2 of definition 31 to the net O is denoted. With S_1 the net defined in step 1 of definition 31 is denoted.

For example, if we consider the Bar hypernet, S_1 denotes the net which only contains the places representing the initial hypermarking (Figure 6.1(a)). Let O be the net shown in Figure 6.1(d); let Y be the final cut of O, i.e.: $Y = \{@client, @bartender, @housec, \langle BAR, bars\rangle, \langle CLIENT\text{-}HOUSE, houses\rangle, \langle BART\text{-}HOUSE, houses\rangle\}$; let Γ be the consortium associated to the transition labelled *drink* which has been added to Figure 6.1(e): $S_2(Y, \Gamma, O)$ is the net represented in Figure 6.1(e).

Finally, let $E' = \{e_1, e_2, ..., e_n\} \subseteq E$ be such that $\{e_1, ..., e_i\} \oplus \{e_{i+1}\}$ for all $i = 2, ..., n-1$. With $GenBP(e_i, ..., e_1)$ we denote the net generated starting from S_1, and then repeatedly applying step 2 of Definition 31 to the net described in step 1, i.e.: $GenBP(e_1, ..., e_i) = S_2(h(e_i), \mathrm{Fin}(e_{i-1}), S_2(h(e_{i-1}), \mathrm{Fin}(e_{i-2}), S_2(..., S_1)))$.

For example, in Figure 6.1(c), $GenBP(enterClient, enterBartender)$ is the net obtained taking the net $S1$, and then applying step 2 with the consortium associated to *enterClient* and the the net representing the initial cut, and then again step 2 to the consortium associated to *enterBartender* and the net obtained by applying the previous step 2 (which, as we will show in the next theorem, is again the net shown in Figure 6.1(c)).

Lemma 2. *Let $\beta = (O, h)$ with $O = (B, E, F)$ be a branching process of $(H, \mathcal{M}_0)$. Let $e \in E$, and $[e] = \{e_1, e_2, ..., e_n\}$ such that $\{e_1, ..., e_i\} \oplus \{e_{i+1}\}$ for all $i = 2, ..., n-1$ Then, $Gen(e) = GenBP(e_n, ..., e_1)$*

Proof. **Induction basis**: $BP(\emptyset) = Gen(\emptyset)$ has to be proven. Both will contain the initial set of the process, which are the same according to condition 4 of definition 30, and step 1 of definition 31.
Induction step: Assume $Gen(\{e_i\}) = BP(e_1, ..., e_i) = (B_i, E_i)$. Moreover, let $S = \{(\langle A, p\rangle, X) \in \mathrm{Fin}(e_i) \mid A \in \mathrm{PASS} \wedge \gamma(A) = p\}$. We have to prove that $BP(e_{i+1}, ..., e_1) = Gen(e_{i+1})$. In order to obtain $BP(e_{i+1}, ..., e_1)$ we have to apply step $S2$ to the net $BP(e_i, ..., e_1)$, i.e.: $S2(h(e_{i+1}, \mathrm{Fin}\{[e_i]\}, (B_i, E_i))$. By Definition 31 the places of this net are $B_i \cup \{(\langle A, trg(A)\rangle, \{\Gamma, S\}) \mid A \in \mathrm{PASS}\}$. and its events are $E_i \cup (\Gamma, S)$. By Definition 32, and using some simple insiemistic operation the net $Gen\{e_{i+1}\}$ is equal to $(B_i \cup e^{\bullet}_{i+1}, E_i \cup e_{\{}i+1\})$. Since condition 22a of Definition 30 holds, and since $Gen\{e_{i+1}\}$ is a $\mathcal{C}$-net, then $e_{i+1} = (\Gamma, S)$.

Moreover, since condition 22b of Definition 30 holds, and since $Gen\{e_{i+1}\}$ is a $\mathcal{C}$-net, then $e_{i+1}^{\bullet} = \{(\langle A, trg(A)\rangle, \{\Gamma, S\}) \mid A \in \text{PASS}\}$. ☐

The next lemma proves that the union of the net generated by all the events of a branching process is the branching process itself. Together with Lemma 3, and the application of step three of the inductive definition, it proves that the class of net identified by the inductive and the axiomatic definitions are the same.

Lemma 3. *Let $\beta = (O, h)$ with $O = (B, E, F)$ be a branching process of $(H, \mathcal{M}_0)$. Then $\bigcup_{e \in E} Gen(e) = \beta$*

Proof. All places added by the $\bigcup$ operator belong to β. We have to prove that all places are taken. Let $\bigcup_{e\in E} Gen(e) = (B', E')$, and assume that $\exists p$ such that $p \in B \wedge p \notin B'$. If p is a minimal place then it belongs to B' by Definition 32. Otherwise p has an event such that $p \in e^{\bullet}$. But again if it happens then p must belong to B'.

For events it is simple because all the events of β are added. ☐

Finally, the following theorem is the main result of the section. It proves that branching processes can be built using the inductive definition.

Theorem 7. *Let $\beta = (O, h)$ with $O = (B, E, F)$ be a branching process of $(H, \mathcal{M}_0)$. O can be built using the three step of Definition 31.*

Proof. For each event $e \in E$ we take the causal closure $[e] = \{e_1, ..., e_n\}$ of e. Let $(e_1, ..., e_n)$ be such that $\{e_1, ..., e_i\} \oplus \{e_{i+1}\}$ for all $i = 2, ..., n-1$. By Proposition 3, $Gen(e)$ is a branching process, and by Lemma 2 each one of these processes can be built using the inductive definition. By Lemma 3, if we take the union of all these processes (step 3 of the inductive definition), we obtain the net O. ☐

6.4 Complete Prefix of the Unfolding

In this Section we first give the definition of Unfolding of an hypernet, a branching process which contains full information about reachable states. Then, the notion of complete finite prefix of the unfolding is defined. In this thesis completeness means that the prefix contains as much information as the unfolding, in the sense that every reachable hypermarking of the hypernet is represented. Finally, a Theorem we prove that every reachable hypermarking of a hypernet is represented as a cut in its finite complete prefix of the unfolding.

Being the previous Section dedicated to the study of branching processes of an hypernet, the definition of the unfolding of a hypernet shuold be clear and quite straightforward:

Definition 33. *The* unfolding *of a hypernet H, denoted $\mathcal{U} = (B_{\mathcal{U}}, E_{\mathcal{U}}, F_{\mathcal{U}})$, is the union of all the branching processes of A.*

The main idea for getting a prefix of the unfolding is to cut it at some point if the parts which follow the cut point have already been explorated. Consider two configurations C and C' of the unfolding whose associated hypermerking is the same. Then, a sufix E' of C is also a suffix of C', but if we append E'

to C, then we do not need to append it to C', and viceversa. Configurations which do not need to be explored are called *cut-offs*. To be able to stop at a certain point, and mark a configuration as a *cut-off* we need an order between configuretions. This order must satisfy these three conditions:

Definition 34. *Let $(H, \mathcal{M}_0)$ be a marked hypernet, and let $\prec$ be a partial order over $Conf(\beta)$. The order $\prec$ is* adequate *if:*

1. *it is well founded,*
2. *it refines the set inclusion relation, i.e.: $\forall C, C' \in Conf(\beta)$, $C \subset C' \Rightarrow C \prec C'$*
3. *it is preserved by finite extension, i.e.: $\forall C, C' \in Conf(\beta)$ such that* $\text{Mark}(C) = \text{Mark}(C')$, *then $\forall E$ suffix of C, $\exists E'$ suffix of C' such that* $\text{Mark}(C \cup E) = \text{Mark}(C' \cup E')$

The main idea behind algorithms which generate a prefix of an unfolding is to start adding events to the branching process which represents the initial marking, and then to stop when two configurations have the same associated marking and one of the two is lower than the other one. The following definition gives an axiomatic characterization of the finite complete prefix of an unfolding.

Definition 35. *Let $\prec$ be an adequate order on $Conf(\mathcal{U})$. An event of the unfolding is a* cut-off *event if there exists an event e'', such $[e''] \prec [e]$ and* $\text{Mark}([e'']) = \text{Mark}([e])$. *An event $e \in E_{\mathcal{U}}$ of the unfolding of H is* feasible *if no event $e' < e$ is a* cut-off. *The $\prec$-complete prefix is the prefix containing all the feasible events.*

Finally, in the following theorem it is proved that there is a correspondance between the hypermarking of an hypernet, and the cuts of its finite complete prefix.

Theorem 8. *Let $(H,\mathcal{M})$ be an hypernet, and let $\mathcal{U}$ be its unfolding. Let $\prec$ be an adequate order on $Conf(\mathcal{U})$. A marking $\mathcal{M}$ of H is reachable iff there exists a configuration C of the $\prec$-complete prefix of $\mathcal{U}$, that does not cut-off events, such that* $\text{Fin}(C) = \text{SET}(\mathcal{M})$.

Proof. ($\Leftarrow$): Without loss of generality let f$C = \{e_1, ..., e_n\}$ be such that $\forall i$ suchthat $<$ $i \leq n \{e_1, ..., e_i\} \oplus \{e_{i+1}\}$, and let $\Gamma_1, ..., \Gamma_n$ be the consortia associated to $e_1, ..., e_n$ through h. The proof is by induction repeatedly applying the condition 2 of definition 30.

Induction basis: $\text{Mark}(\emptyset) = \mathcal{M}_0$ which is a reachable hypermarking.

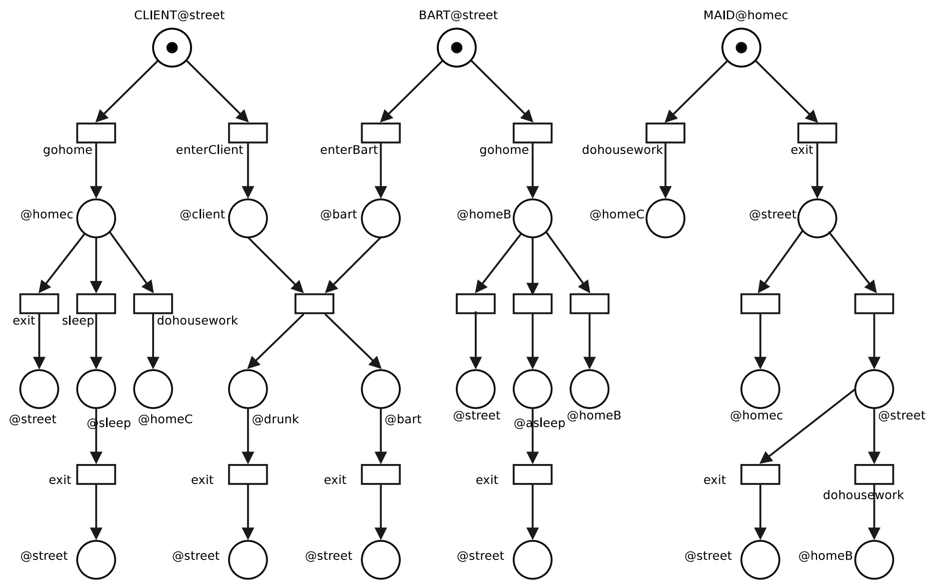
Induction step: Let $C_i = \{e_1, ..., e_i\}$ be such that $\text{Fin}(C_i) = \text{SET}(\mathcal{M}_i)$ where $\mathcal{M}_i$ is a reachable hypermarking. By condition 22a $\text{Fin}(C_i)$ enables Γ_{i+1}. Let $\mathcal{M}_{i+1}$ be such that $\mathcal{M}_i[\Gamma_i\rangle\mathcal{M}_{i+1}$. By condition 2 2b $\text{Fin}(C_i \cup \{e_{i+1}\}) = \text{Fin}(C_i) \setminus \{\langle A, p\rangle \mid A \in \text{PASS}\} \cup \{\langle A, p\rangle \mid A \in \text{PASS} \wedge p = trg(A)\}$ By the definition of firing rule (Definition 13) $\mathcal{M}_{i+1}(A) = \begin{cases} \mathcal{M}_i(A) & \text{if } A \notin \text{PASS} \\ trg(a) & \text{if } A \in \text{PASS}, \end{cases}$.

Therefore $\text{SET}(\mathcal{M}_{i+1}) = \text{SET}(\mathcal{M}_\rangle) \setminus \{\langle A, p\rangle \mid A \in \text{PASS}\} \cup \{\langle A, p\rangle \mid A \in \text{PASS} \wedge p = trg(A)\}$

($\Rightarrow$): Since $\mathcal{M}$ is reachable there exists at least one firing sequence $\tau = \mathcal{M}_0[\Gamma_1\rangle\mathcal{M}_1...\mathcal{M}_{n-1}[\Gamma_n\rangle\mathcal{M}_n$. The inductive definition of branching process ensures there exists a configuration of C the unfolding such that $C = \{e_1, ..., e_n\}$ such that $\forall i,\ 0 < 1 < n$ then $h(e_i) = \Gamma_i \wedge \{e_1, ..., e_i\} \oplus \{e_{i+1}\}$. Assume that τ is the minimal firing sequence, i.e.: there is no τ' with $C' \prec C$, where C' is the configuration associated to the firing sequence τ'.

It will be proved that C does not contain a cut-off event, and therefore it belongs to the $\prec$-complete prefix. By contradiction assume that $e_i \in C$ is a cut-off event, and let e' be an event of the unfolding such that $[e'] \prec [e_i]$ and $\text{Mark}([e_i]) = \text{Mark}([e'])$. Let $[e'] = \{e'_1, ..., e'_j\}$, and let $\Gamma'_1, ..., \Gamma'_j$ be a firing sequence such that $\forall\ k < j\ h(e'_k) = \Gamma'_k$. Since $\text{Mark}([e_i]) = \text{Mark}([e'])$ the firing sequence $\tau' = \Gamma'_1...\Gamma'_j, \Gamma_i + 1, ..., \Gamma_n$ leads to the hypermarking $\mathcal{M}$. Let C' be the configuration such that τ'. Since $\prec$ is preserved by finite extensions then $C' \prec C$ which contradicts the minimality of C. $\square$

In Figure 6.4 it is shown how the finite complete prefix of the unfolding of the Bar example appears using the total order defined by Esparsa, Romer, and Voegler for one safe net in [22].



As it can be seen, the unfolding procedure has been stopped when two configurations which brings to the same markings have been reached.

Chapter 7

Hierarchical Net Models

7.1 Survey of hierarchical Petri net models

We have already discussed how the use of the nets-within-nets paradigm can be useful to model mobility. Locations are modeled as places of nets, and agents located in a place are modeled as nets themselves. In this Chapter the main hierarchical Petri nets formalisms which use the nets-within-nets paradigm are considered and reviewed. The goal of this Chapter is to give the reader an overview of which kind of hierarchical Petri net models have been introduced in the past, with particular attention to models used for mobility. For each class of net discussed in this thesis we give an informal description of the basic model by means of an introductory example. Then, extensions, and results concerning the model are described. The discussion is kept at an informal level: only an idea of the main characteristics of each model are discussed. The formalism used to model mobility are compared with the hypernet model.

The Chapter is structured with a first part where several formalisms for mobility which use the nets-within-nets paradigm are discussed, and a second part which introduces the Renew tool, and two other approaches which have not been specifically used to model mobility, but which contain interesting ideas. The first part begins with object nets, one of the most studied formalisms which implements the nets-within-nets paradigm. These are discussed in Section 7.2.1. Then nested nets and its extensions are discussed in Section 7.2.2. Section 7.2.3 discusses object nets for mobility, a particular approach for the problem of modeling systems of mobile agents acting in distributed namespaces. Modular nets for mobility are introduced in Section 7.2.4. In the second part we discuss the Renew tool (Section 7.3.1), the high level net with rules as token paradigm (Section 7.3.2), and recursive Petri nets (Section 7.3.3).

7.2 Nets-Within-Nets Paradigm and Mobility

7.2.1 Object Nets and MULAN

The idea of using a nesting structure where tokens of a Petri net can be Petri nets themselves had been discussed by Valk in his earlier work about task flow in systems of functional unit [48]. The term nets-within-nets was not coined here, but the intuition of using nets (called task flows) as tokens of another net (the system of functional units) can be tracked back to this earlier work. This raw idea was refined by Valk itself in his work on elementary object nets [49] in 1998. A two level structure with a system net (the net at the higher level) and several object nets (the tokens which are nets themselves) was considered, and it was also discussed how to handle situations where fork and join of object nets are required, like for example when a transition t has only one input place, but several output places. Different semantics were proposed (see for example [50]). The value semantics and the reference semantics recall the two different mechanisms of passing parameters to a function in a programming language, by value and by reference. If the value semantics is used, two copies of the net bound to the input arc of t are created. They are two independent objects, each with its own marking. If the reference semantics is used then the two tokens situated in the output places of t are both references to the same net, and a change of marking done on one of the two references is reflected immediately in the other reference. Other semantics introduced are the process semantics in which the tokens are process of the object net instead of the net itself, and the distributed semantics where the tokens of the object net in the input place of t are distributed between the two copies of the object net in the output places of t when the transition is fired.

Properties of object nets have been studied in [33]. In particular, it was shown that reachability becomes undecidable while boundedness remains decidable for elementary object net systems. It was also shown that allowing an infinite nesting structure is enough to obtain a Turing equivalent model. Although this result is useful from a computational point of view, it is a negative result from the modeling side of the formalism since interesting properties like reachability and boundedness are not decidable for Turing powerful formalisms.

Other extensions of the object nets model have been proposed. For example, in [35] the formalism has been extended by adding the possibility of a *vertical* synchronization. This means that tokens are no more restricted to move in the places of the same net, but they can also move in other nets. In this paper it was also shown that the model with this extension is Turing complete. In [34] an algebraic extension of object nets was proposed. Operators which compose nets can be added in arc expression. These operations allow to modify the structure of net-tokens at run-time.

The paradigm of nets-within-nets has been implemented in the tool Renew. This topic will be discussed in detail in in Section 7.3.1.

A first attempt to use object nets for modeling mobility can be found in [32]. To be more precise, the starting point of this paper was the Renew tool. The authors discuss how the nets-within-nets paradigm can be used to describe mobility, showing that it is attractive, since it allows an intuitive representation of mobile entities as well as an operational semantics which is implemented in the RENEW-simulator. In particular, the Petri net based multi agent system architecture MULAN was presented (see figure 7.1). A system of mobile agenta is modeled with a four level structure. Each layer represent a level of abstraction and it is modeled with a Petri net. The first level of abstraction represents the agent system with the mobility and communication structure. The tokens of the net modeling the first level are agent platforms. Platforms offer services to agents and handle the creation/deletion of agents, and the receiving/sending of agents to other platforms. Therefore platforms are the environments (also locations) where the agents are located. Agents are also modelled in terms of nets. They are encapsulated, since the only way of interaction is by message passing. Agents can be intelligent, since they have access to a knowledge base. The behavior of the agent is described in terms of protocols, which are again nets. Protocols are located as templates on the place protocols. Protocol templates can be instantiated, which happens for example if a message arrives. An instantiated protocol is part of a conversation and lies in the place conversations.

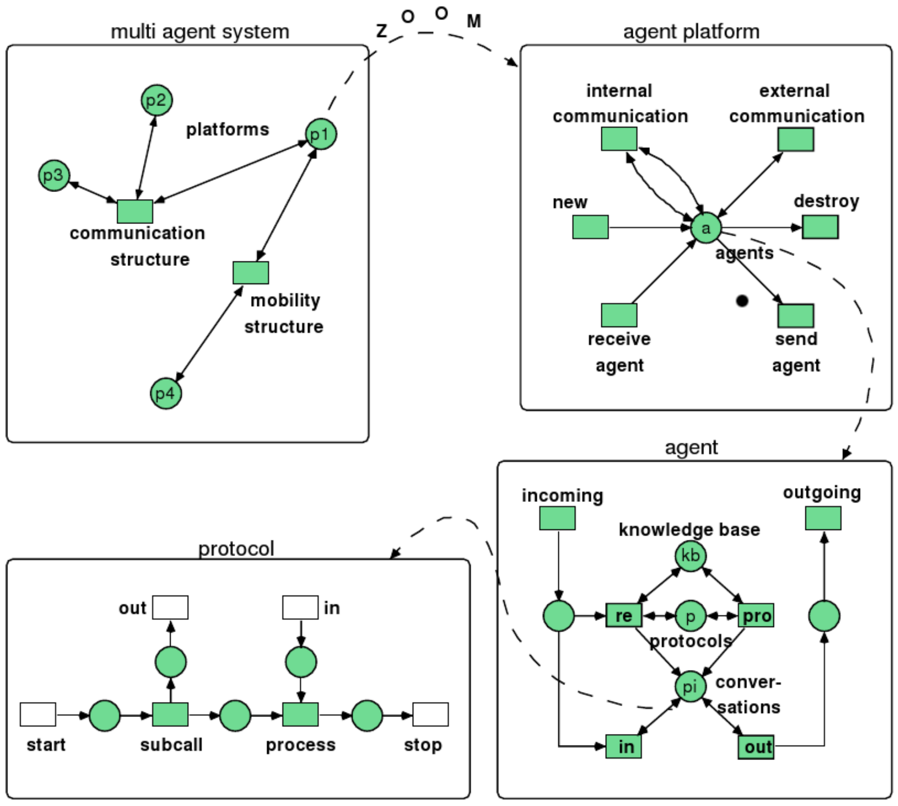


Figure 7.1: The MULAN architecture

It is hard to compare this approach with the hypernet one. MULAN is not a high level Petri net formalism like hypernets are, or like all the formalisms which will be discussed in the following Sections are. MULAN is a framework based on the reference net formalism, and loosely speaking it builds one more level of abstraction on a specific formalism: the reference net formalism. Therefore, from a modeling point of view, MULAN has more features, like the possibility to model the intelligence of agents, protocol of communication etc. On the other hand hypernets are more similar to Petri nets, and also offer the possibility to apply analysis techniques typical of Petri nets to the modeled system. As we will see, this last point is the main advantage the hypernet model provides compared to the other nets-within-nets formalism used to model mobility.

7.2.2 Nested Nets

Nested nets have been introduced in [40] by Lomazova and Schnoebelen. They are a nets-within-nets formalism which uses the value semantics. A nested net can have four kinds of steps. A transfer step is a step in a system net, which can "move", "generate", or "remove" objects, but does not change their inner states. An object-autonomous step changes only an inner state in one object. There are also two kinds of synchronization steps. Horizontal synchronization means simultaneous firing of two object nets, situated in the same place of a system net. Vertical synchronization means simultaneous firing of a system net together with some of its objects involved in this ring. It has been demonstrated that the reachability and the boundedness problems are undecidable for nested Petri nets, while the termination, the control state maintainability[1],and the inevitability[2]problems are decidable.

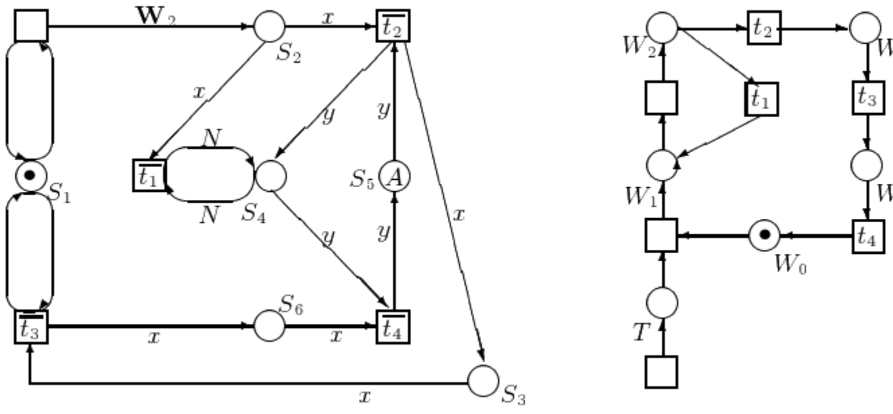


Figure 7.2: A system net SN (left), and an object net EN (right)

Figure 7.2 represents an introductory example in which a set of workers receive some tasks from time to time. In the initial marking the unlabeled transition in SN may fire, putting a net token **W2** (EN with marking $\{W_2\}$) into place S_2. This step creates an instance of EN in S_2 . After that the transition marked by t_2 in SN may fire synchronously with the transition marked by t_2 in the element net lying in $S2$. After that the net EN with the marking $\{W_3\}$ will be situated in the place $S3$, the set A in S_5 will be diminished by one token and the place S_4 gets one token. Then the transition marked by t_4 in SN may fire synchronously with the transition marked by t_4 in the element net lying in $S3$. Note that initially there are no net tokens (workers) in SN, so the element net EN for a worker plays a role of type description.

In [51] a nested net for adaptive systems has been introduced to model adaptive workflow systems. This is a net-within-nets inspired formalism in which the

[1]The Control-State Maintainability Problem is to decide, given an initial marking M and a finite set $Q = \{q_1, q_2, ..., q_m\}$ of markings, whether there exists a computation starting from M where all markings cover (are not less than w.r.t. some ordering) one of the q_is.

[2]The Inevitability Problem is the dual problem of the Control-State Maintainability Problem, and consists in deciding whether all computations starting from M eventually visit a state not covering one of the q_is, e.g. for Petri nets we can ask whether a given place will eventually be emptied.

nets are workflow nets enhanced with an exception handling mechanism (EWF). To manipulate token nets in an adaptive workow net a number of operations on EWF have been identified. These include: sequential composition, parallel composition, and choice. Adaptive workflow nets are Turing complete, therefore in order to be able to check properties of systems a restriction of this model, called adaptive workflow net, has been introduced in [52]. It has also been shown that two typical properties of workflow nets, namely soundness and circumspectness, can be verified. Soundness means that a proper final marking (state) can be reached from any marking which is reachable from the initial marking, and no garbage will be left. Circumspectness means that the upper layer is always ready to handle any exception that can happen in a lower layer.

Both nested nets and hypernets use the value semantics, a choice which seems natural for modeling an agent because each agent has its own identity. However, as we will see in Section 7.2.3, there are some cases where also the reference semantics makes sense. Nested nets allow the creation of new nets, feature that is useful for modeling purposes. However, the constraint in the hypernet model which forbids the creation/deletion of tokens is calculated. In fact, it is thanks to this constraint that it is possible to expand the hypernet model to the equivalent 1-safe net. Moreover, the result about the prefix of the unfolding of Chapter 6 is based on the fact that hypernets have a finite state space.

As it has been just discussed in nested nets several problems which are decidable for Petri nets are no more decidable in the nested net formalism, while in hypernets are a compact representation of 1-safe nets, and therefore all the problems decidable for 1-safe nets are still decidable.

Finally, another advantage of hypernets is that they have a dynamic hierarchy, while nested nets have a static hierarchy of agents which cannot change during the evolution of the system.

7.2.3 Object Nets for Mobility

Object nets for mobility are a nets-within-nets formalism introduced by khöler and Farwer in [31]. This work investigates the problem of formalizing mobile agents acting in a mobility infrastructure. Consider a situation where two buildings A and B are present, and a mobile agent can be in eitherbuildings as shown in Figure 7.3. The two buildings are connected via the mobility transfer transitions t_4 and t_6. One mobile agent is present inside building A as a net token.

When modelling this scenario we have to distinguish two kinds of movement: movement within a building and movement from one building to another. When moving within a building, the agent has full access to all services. On the other hand, when moving to a different building the environment may change dramatically: services may become unavailable, they may change their name or their kind of access protocols. This leads to the usual problem that within the same environment (e.g. the memory of a personal computer) we can use pointers to access objects (as done for Java objects), which is obviously impossible for a distributed space like a computer network: For example when a Java program transfers an object from machine A to B via remote method invocation (RMI) it does not transfer the object's pointers (which are not valid for B); instead Java rather makes a deep copy of the object (called serialization) and transfers

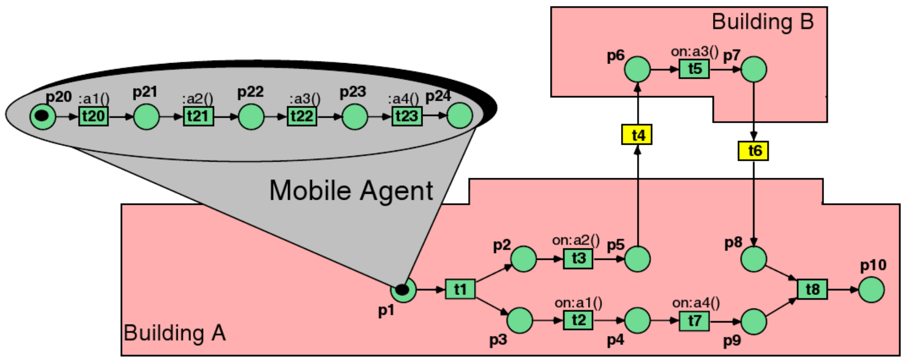


Figure 7.3: A mobile agent's environment

this value over the network. The value is used to generate a new object at B which can be accessed by a fresh pointer.

Therefore belonging to a name space and migrating between name spaces is not the same. An object belonging to a name space can be accessed directly via pointers, but when migrating between name spaces, objects have to be treated as values that can be copied into network messages. For the modelling of mobile systems it is essential that the formalism used supports both representations. To accomplish this, the firing rule of the elementary object net model has been modified in order to provide both reference and value semantics.

Hypernets do not consider this particular problem and only use the value semantics. On the one hand this choice restricts the flexibility of the formalism from a modeling point of view. Situations where agents act in different namespaces cannot be modeled as simply as in object nets for mobility. On the other hand adding mechanisms to handle this kind of systems would have complicated the model of hypernets.

As it has already been discussed for nested nets in the previous Section, in the hypernet model it is possible to use all the analysis techniques available for 1-safe nets, while in objects nets for mobility problems like boundedness and reachability are not decidable.

7.2.4 Modular Petri Nets For Mobility

In [38] C.A. Lakos, in an attempt to capture mobility in a Petri net formalism, extended the formalism of modular Petri nets in a hierarchical fashion. Modular Petri nets are nets made up of a number of subnets, which interact in the standard Petri net way by place and transition fusion. They have been extended adding the notion of *locations*, which are subnets with a specific fusion environment. Locations are nested, and thus capture the notion of locality or proximity. Moreover, modular nets have been extended with the capability of shifting locations: a subsystem resident in one location can be shifted to another location by firing a transition with arcs incident on locations. A colored version of this formalism has also been discussed.

In Figure 7.4 a simple mail agent is shown, while Figure 7.5 shows its associated system. If place *empty* is marked in Figure 7.4, then the agent has no

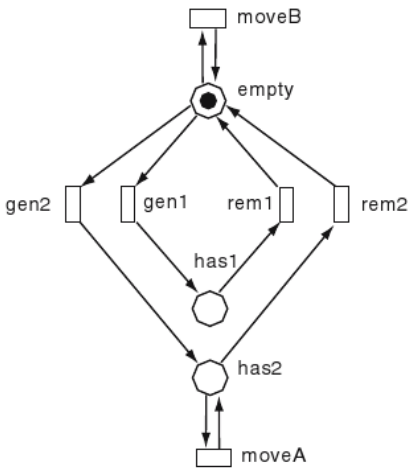


Figure 7.4: A simple mail agent

mail to deliver, while if place has_1 or has_2 is marked, then the agent has mail to deliver to $site_1$ or $site_2$, respectively. The transitions gen_1 and gen_2 generate these mail messages, while transitions rem_1 and rem_2 consume (or deliver) them. Transitions $move_A$ and $move_B$ are used to constrain or allow movement of the agent.

The composite mail system is shown in Figure 7.5. Each rounded rectangle is a location, which is a subnet together with a fusion environment. The main or root location is labelled *System*, and it contains three locations labelled $Site_0$, $Site_1$ and $Site_2$. Within each of these locations there is a nested location for the mail agent, labelled Loc_0, Loc_1 and Loc_2, respectively. Transition fusion is indicated either by name correspondence, by a double line, or by a line through the transition (these transitions are fused with disabled transitions in the system which are not shown).

As indicated by the double lines, the transition mov_{01} in the location $Site_0$ is fused with the transition mov_{01} in the location *System*, which is also fused with the transition mov_{01} in the location $Site_1$. mov_{01} in the location $Site_0$ has a *broad input arc* incident on the location Loc_0, while mov_{01} in $Site_1$ has a *broad output arc* incident on the location *Loc*1. This is shorthand for shifting the location of a subsystem: a broad input arc removes all the tokens from the source location, and the broad output arc deposits the tokens into the target location.

A location is occupied if at least one of the local places is marked. Thus, in the example, the initial marking indicates that locations *System*, $Site_0$, $Site_1$, $Site_2$ and Loc_0 are occupied, while locations Loc_1 and Loc_2 are not. The transitions like mov_{01} which shift the location of a subsystem (or more generally consume a subsystem at a location or generate a subsystem at a location) are shown with a broad arc. This is a shorthand notation for indicating that all the tokens in the local places are consumed. Where a consume is paired with a corresponding generate, it is assumed that the marking is shifted from one location to another. It is worth noting that for $Site_0$, the shifting of the agent is determined solely by the site, whereas for $Site_1$ and $Site_2$ the agent collaborates

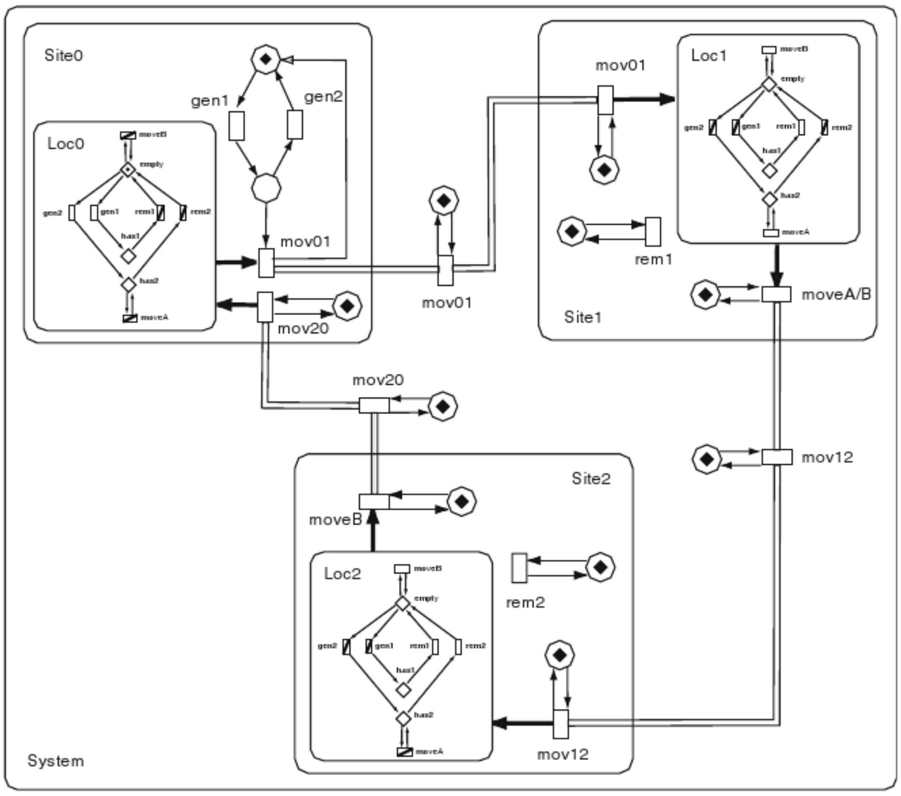


Figure 7.5: The composite mail system

with the shift transition.

Among the formalisms discussed in this Chapter, modular nets for mobility are probably the class of nets more similar to hypernets. In fact, they both use the value semantics and they both provides a hierarchy of net which can dynamically change. However some differences exist. Modular nets for mobility allows the creation of black tokens, while hypernet don't. This choice has again been made to allow the expansion toward 1-safe nets. The other difference is more technical. When transferring an agent from one location to another one, what it is transfered is not the entire agent (structure and marking), but only the marking. This means that places and transition modeling an agents must be duplicated in every location the agent can go. For example, in Figure 7.5 the net modeling the mail agent (the net in Figure 7.4) is placed in all the four locations the agents an go.

As far as I know, modular nets for mobility have not been studied from the decidability of property point of view.

7.3 Renew and Other Approaches

7.3.1 Reference Nets and Renew

The reference net formalism was described in [36]. It is a high level Petri net formalism based on the nets-within-nets paradigm. The reference net formalism uses reference semantics. This means that tokens within a net do not exclusively correspond to their object/net (value semantics), but only reference their object/net. As a result of this, multiple tokens can refer to the same object.

Communication between different net instances within the reference net formalism is handled via synchronous channels, based on the concepts proposed in [11]. Synchronous channels connect two transitions during firing. Transitions inscribed with a synchronous channel can only fire synchronously, meaning that both transitions involved have to be activated before firing can happen. During firing arbitrary objects can be transmitted bidirectionally over the channel. While the exchange of data is bidirectional there is a difference in the handling of the two transitions. The transition, or more accurately the inscription of the transition, initiating the firing is called the *downlink*. The downlink must know the name of the net in which the other transition, the so-called *uplink*, is located. The inscription of the downlink has the form *netname:channelname(parameters)*, in which the parameters are the objects being send and received during firing. If the downlink calls an uplink located in the same net the net name is simply replaced by the keyword *this*. The uplink's inscription is similar, but loses the net name, so that it has the form *:channelname(parameters)*. Uplinks are not exclusive to one downlink and can be called from multiple downlinks, so that this construct can be used in a flexible way. It is also possible to synchronize transitions over different abstraction levels. While during firing one downlink is always linked to just one uplink, it is possible to inscribe one transition with multiple downlinks, so that more than two transitions can fire simultaneously.

Figure 7.6 shows a simple example of a reference net system. The example was modelled using the RENEW tool, which will be described later. It models a producer/consumer system, which holds an arbitrary number of producers and consumers. The system consists of three kinds of nets: the system net, the producer nets and the consumer nets. The producer and consumer nets both possess the same basic structure, but use different channels. The system net serves as a kind of container for the other nets. The transitions labeled **manual** initiate the creation of new producers and consumers by creating new tokens when a user manually fires them during simulation[3]. The transitions labeled C and P actually create new producer or consumer nets when firing. These new nets are put onto the places below the transitions. The transition labeled I synchronizes the firing of a transition in one consumer and one producer each (labeled I1 and I2 in the other nets). In this way it is possible to simulate the behavior in such a way, that whenever a producer produces a product, an arbitrary consumer consumes it. It is of course possible to enhance this model by, for example, adding an intermediary storage, which can store items from arbitrary producers until consumers need them. Another way of making the model more realistic is to explicitly model the products as nets as well. That way they would not just be simple tokens but actual objects being exchanged via

[3]This is a special function of the RENEW tool, which was used for this example.

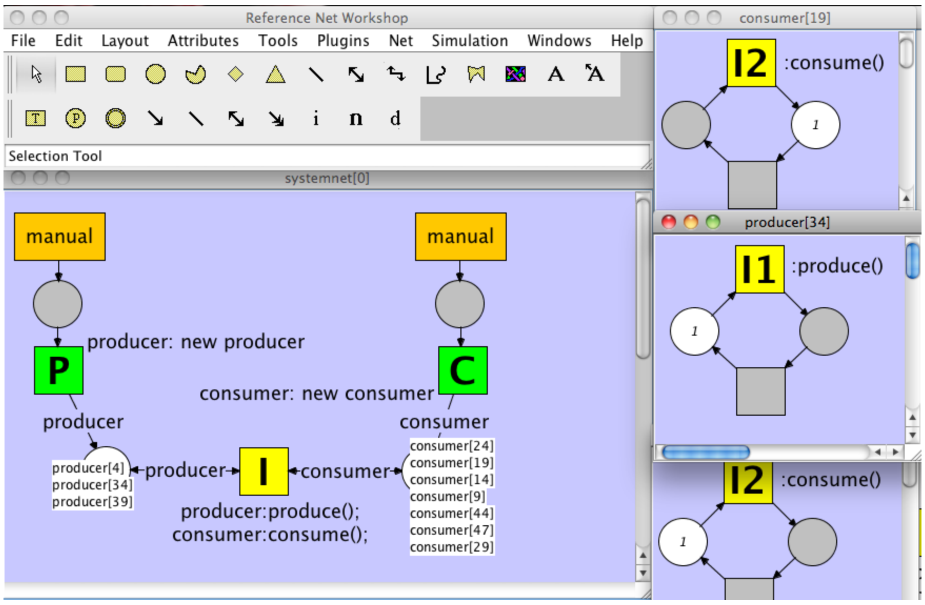


Figure 7.6: Reference net example

the synchronous channels between the producers and consumers. In this case the parameters of the channels would be the nets, which would be transmitted from within the producer nets into the consumer nets.

The Petri net editor and simulator Renew (The **RE**ference **NE**t **W**orkshop) was developed alongside the reference net formalism, and is also described in [36] as well as in [37]. It features all the necessary tools needed to create, modify, simulate and examine Petri nets of different formalisms. It is predominantly used for reference nets, but can be enhanced and extended to support other formalisms. It is fully plugin based, meaning that all functionality is provided by a number of plugins that can be chosen, depending on the specific needs. Plugins can encapsulate tools, like a file navigator or certain predefined net components, or extensions to the standard reference net formalism, like hypernets or workflow nets. Renew is freely available online and is being further developed and maintained by the Theoretical Foundations Group of the Department for Informatics of the University of Hamburg. Since the tool supports the idea of nets within nets and is flexible enough to support multiple formalisms, it was chosen as the basic environment for the experiments of this thesis. In particular, in Chapter 9 a Renew plugin which allows to draw and to analyze a hypernet will be discussed.

7.3.2 High Level Nets ith Nets and Rules as Tokens

In [28] Hoffmann, Ehrig, and Mossakowski introduced a high level model which uses a modification of the nets-within-nets paradigm, called nets with nets and rules as tokens. In this new paradigm the tokens of a Petri net can be Petri nets themselves (like in the ordinary nets-within-nets paradigm), or they can be rules which can be used to change the structure of net tokens. This new concept has been used to model the main requirements of a modified version of the dining philosopher example in which a philosopher sitting at the table has the capability to introduce a new guest. The higher level net of this example

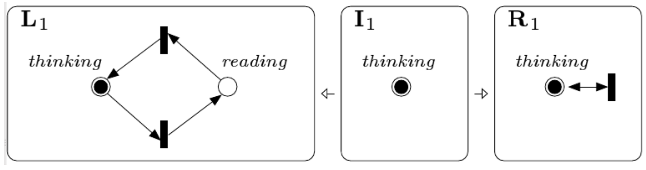


Figure 7.7: Rule1

is shown in Figure 7.8. There are three different locations in the house where the philosophers can stay: the library, the entrance-hall, and the restaurant. In the restaurant there are different tables where one or more philosophers can be placed to have dinner. Each philosopher can eat at a table only when he has both forks. The philosophers in the entrance-hall have the following additional capabilities: they are able to invite another philosopher in the entrance-hall to enter the restaurant and to take place at one of the tables; they are able to ask a philosopher at one of the tables with at least two philosophers to leave the table and to enter the entrancehall.

In order to explain the basic concept of this formalism it will be detailed the execution of transition *leave library*, which includes the application of the rule $rule_1$ shown in Figure 7.7. *leave library* needs an assignment for the variables n, r and m in the net inscriptions of the transition. They are assigned to the net phi 1 (see Figure 7.9), the rule $rule_1$, and a match morphism $m1 : L \to G$ between P/T-systems. The first condition $codm = n$ requires $G = phi_1$ and the second condition $applicable(r, m) = tt$ makes sure that rule $rule_1$ is applicable to phi_1 , especially $L = L1$, s.t. the evaluation of the term $transform(r, m)$ leads to the new net phi_1 isomorphic to R_1 of $rule_1$ in Figure 7.7. As result of this ring step phi_1 is removed from place Library and phi'_1 is added on place Entrance-Hall. In general, a rule $r = (L \leftarrow I \to R)$ is given by three P/T-systems called left-hand side, interface, and right-hand side respectively.

7.3.3 Recursive Petri Nets

Recursive Petri nets are a family of nets introduced by Haddad and Poitrenaud in which the firing of a transition of a net can generate a copy of the net itself with a new marking. There are several models belonging to the family of the recursive Petri net. The first model that is considered here is the sequential recursive Petri net (SRPN) [26], a restriction of the general model introduced because, unlike the general model, the model checking problem is decidable for them. In order to informally explain the basic concepts of this formalism, a comparison with ordinary Petri nets is described. As an ordinary Petri net, a sequential recursive Petri net has places and transitions. Moreover, there is a set of final markings. Transitions are split in two categories: elementary and abstract transitions. Informally speaking, if the semantics of the basic Petri net model can be explained as a thread which plays the token game by firing a transition and updating the current marking, then, by comparison, in the SRPN model there is a stack of threads (each one with its current marking), where

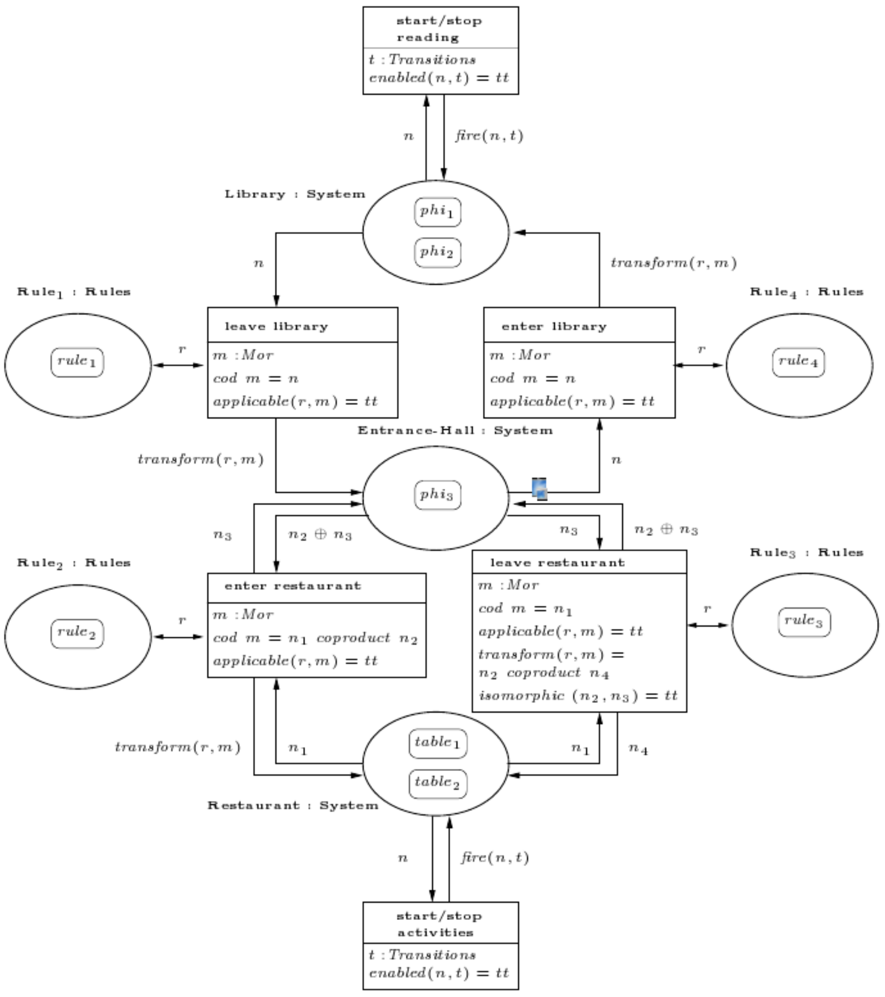


Figure 7.8: The hurried philosophers case of study

the only active thread is the one on the top of the stack. When a thread fires an elementary transition, it consumes the tokens in the way they are consumed in ordinary Petri nets. But if an abstract transition is fired, then a new thread is created, and put on the top of the stack becoming the active one. When a final marking is reached, then the thread on top of the stack is removed, and the abstract transition which created the thread is fired.

The net of Figure 7.10 illustrates the characteristic features of SRPNs. Abstract transition are represented by a double border rectangle. The starting marking of the net created by an abstract transition is specied in a frame. Note that contrary to ordinary nets, SRPNs are often disconnected since each connected component may be activated by the ring of different abstract transitions.

The left upper part of the gure models the application level of a processor in an abstract way. The cycle p_run , $t_correct$, p_run represents the correct execution of the current instruction and the abstract transition t_ex models a faulty execution of the instruction yielding a second level with pex marked. In

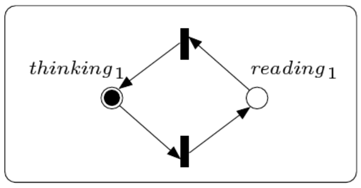
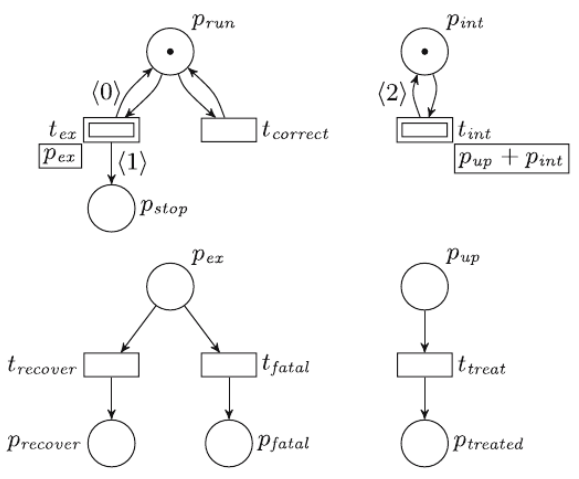


Figure 7.9: A philosopher



$\Upsilon_0 = \{m \mid m(p_{recover}) = 1\}$ $\Upsilon_2 = \{m \mid m(p_{treated}) = 1\}$
$\Upsilon_1 = \{m \mid m(p_{fatal}) = 1\}$

Figure 7.10: A simple sequential recursive Petri nets

this level, the system either recovers from the fault ($t_recover$) or detects a fatal error (t_fatal). The sets γ_0 and γ_1 model these two cases. Depending on the fault type, when returning to the rst level, the process resumes its activity (place p_run marked) or stops it (place p_stop marked). The interrupt modelling outlines the capabilities of the SRPN. When the abstract transition t_int is red, the current execution is interrupted and a second execution level, modelled by a token in p_up and p_int , is activated. The same construction applies again on this component net making possible a recursive interrupt process. Figure 7.11 represents an extract of the reachability graph of the SRPN of Figure 7.10. An extended marking is graphically represented as a path whose nodes correspond to levels and are labelled by the associated ordinary markings, and whose edges connect level i to $i + 1$. The dashed arcs denote steps between extended markings.

In the general recursive petri net model (RPN) threads playing the token game of a Petri net can be dynamically created, and concurrently executed. From the expressivity point of view it has been proved that RPNs are strictly more expressive than the union of Petri nets and context free grammars w.r.t. the language point of view. Moreover, it has been proved that the reachability problem and some related one remain decidable for RPNs, while event-based

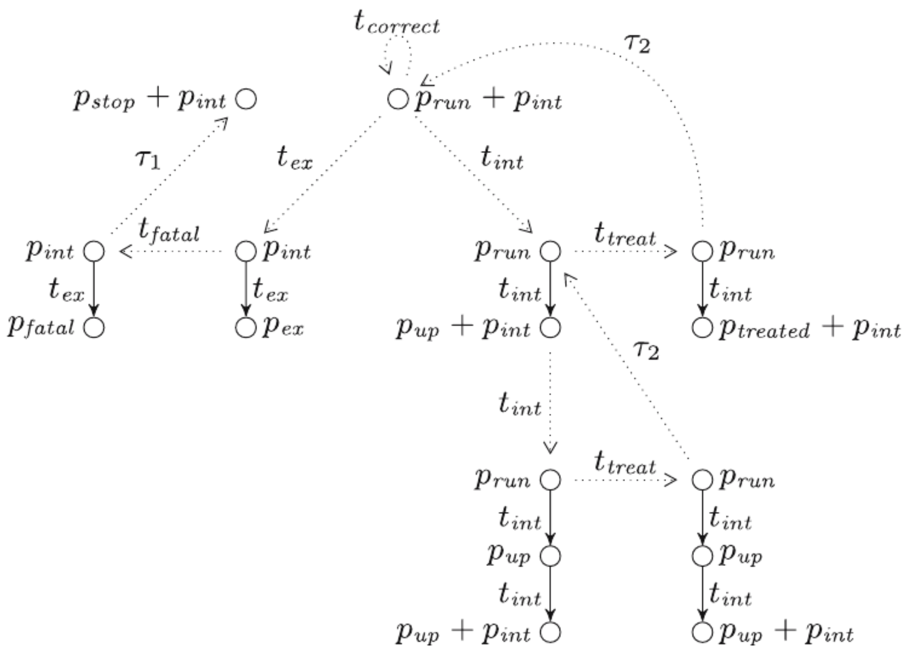


Figure 7.11: The reachability grapf of the system in Figure 7.10

linear time model checking is decidable for SRPNs, but not for RPN (see [24, 25] if you want to deepen these topics).

In [27] some other results concerning recursive Petri nets has been shown. First, an extended version of RPN which includes new mechanisms like place capacities, test arcs, parametrised initiation and termination of threads, and interrupt capabilities has been introduced. It has been also shown that the reachability and boundedness are still decidable if you consider these extensions. Then, it is demonstrated that the bisimulation problem between a restricted class of SRPN and a finite automaton is decidable, like it is decidable between Petri nets and finite automaton.

Chapter 8

Modeling a Grid Tool for High Energy Phisics

In this Section an example in which the nets-within-nets paradigm has been successfully used to model a real Grid application is deeply discussed. Section 8.0.4 introduces the application context, a Grid distributed data analysis tool developed to serve the community of the Compact Muon Solenoid (CMS) experiment at the CERN Large Hadron Collider (LHC). In Section 8.0.5 it is shown how the considered use case has been modeled. Finally, in Section 8.0.6 it is possible to find some details about how the model has been built starting from the source code of the implemented system, and from the documentation.

8.0.4 The Application Context: Grid distributed analysis

The CMS experiment at CERN produces about 2 Petabytes of data to be stored every year, and a comparable amount of simulated data is generated. Data needs to be accessed for the whole lifetime of the experiment, for reprocessing and analysis, from a worldwide community: about 3000 collaborators from 183 institutes spread over 38 countries all around the world.

The CMS computing model uses the infrastructure provided by the Worldwide LHC Computing Grid (WLCG) Project [10] through the supporting projects EGEE, OSG and Nordugrid. Grid analysis in CMS is data driven. A prerequisite is that data is already distributed to some remote computing centers, and correspondingly published in the CMS data catalogue, so that users can discover available datasets. Parallelization is provided by splitting the analysis of large data samples into several jobs. The output data produced by the analyses are typically copied to the storage of a site and registered in the experiment specific catalogue. Small output data files are returned to the user. In the CMS experiment the CRAB tool set has been developed in order to enable physicists to perform distributed analysis over the Grid. The role of CRAB is to allow the user to run over distributed datasets the very same analysis she/he ran locally, and collect the results at the end. CRAB interacts with the distributed environment and the CMS services, hiding as much of the complexity of the system as possible. CMS community members use CRAB as a front-end which provides a thin client, and an Analysis Server which does most of the work in terms of automation, recovery, etc. with respect to the direct interactions with the Grid. The Analysis Server enables full workflow automation among different Grid middlewares and the CMS data and workload management systems. Indeed, the main reasons behind the development for the Analysis Server are:

- automating as much as possible the whole analysis workflow;
- reducing the unnecessary human load, moving all possible actions to server side, keeping a thin and light client as the user interface;
- automating as much as possible the interactions with the Grid, performing submission, resubmission, error handling, output retrieval, post-mortem operations;
- allowing better job distribution and management;
- implementing advanced use cases for important analysis workflows

The server architecture adopts a completely modular software approach. In particular, the Analysis Server is comprised of a set of independent components (purely reactive agents) implemented as daemons and communicating asynchronously through a shared messaging service supporting the "publish & subscribe" paradigm. Most of the components are themselves implemented as multi-threaded systems, to allow a multi-user scalable system, and to avoid bottlenecks. The task analyses are completely handled during their lifetime by the server through different families of components: there are components devoted to monitoring the Grid status of the single jobs in a task, other groups of agents coordinate to manage the output retrieval and the recovery of the failed jobs by scheduling their resubmission automatically. A relevant part of the agents is designed in order to handle the submission chain of user tasks to the Grid. As the Analysis Server internal architecture is a natural candidate for being analyzed with the nets-within-nets paradigm, as aforementioned, we decided to model and study the Grid submission chain. The aim of this study is to check that the involved agents behave correctly and efficiently with respect to the foreseen submission workflow. We decided to consider the system at the component-task-job level, as it represents a good compromise between the effects perceived by the tool final users and the large number of technical details that a complete representation of the Grid would require.

8.0.5 Modeling the submission use-case

In this Section we describe in detail the process of submitting jobs to the Grid through the CRAB Analysis Server. For each relevant component of the system its net representation is discussed. In addition, the bugs that have been discovered thanks to the net models are presented with the solutions that the actual code has adopted in order to solve the issues. The CRAB analysis suite was modeled using nets in a hierarchical fashion, as shown in Figure 8.1. A vertical line with multiplicity n, indicates the presence of n nets in the higher one (e.g.: the CRABClient net contains from 1 to N Task nets); a horizontal dashed line indicates that the linked nets are references to the same net. In our modeling we consider one client just for the purpose of simplicity. Of course, the discussed functionalities and use cases still hold when a larger number of

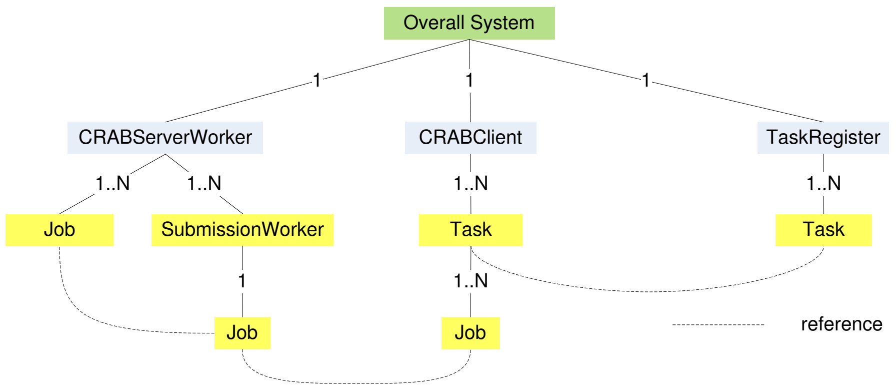


Figure 8.1: The Nets hierarchy for the CRAB suite.

clients is considered, as the client server model assumes no direct interactions among the clients. In addition, for the use case that will be discussed, the server code separates properly the session of work for every task.

The OverallSystem net, which is the system net, contains three nets which respectively model the behavior of the client who is using the CRAB server (*CRABClient net*), the TaskRegister component which is a thread running on the CRAB server (*TaskRegister net*), and the CRABServerWorker which is also a thread running on the server (*CRABServerWorker net*). *Tasks* are the objects a client creates, and deals with. They are composed of *jobs*, the single units of work that need to be performed. The TaskRegister component is responsible for registering tasks, i.e. creating some data structures on server disks, checking if each task has all the inputs it needs to be executed, and checking if the Grid can access the proper security credentials to execute it. The CRABServerWorker component continuously receives jobs, schedules them for execution on the Grid infrastructure, and creates a SubmissionWorker thread which monitors the lifecycle of each job on the Grid. The clients interact with the server, and can initiate some operations like: submitting jobs, killing them if needed, and asking for the results.

CRABClient, Tasks, and Jobs

The first component we are going to discuss is the CRAB client, which is modeled with the net in Figure 8.2. This component is what enables all the action sequences that the users can do on their Grid analyses.

The first thing a client does is to create a new task on the client machine. The typical usage pairs a unique task with a CRAB analysis session. For this reason we assume that the *tasksPool* can contain a finite number of tokens. After the task has been locally created on the client machine, the client can perform a submit operation, which is of course the most important one as it starts the submission chain. The first time a task is submitted to the server, it is also registered by the TaskRegister component. Subsequent submits are handled directly by the CRABServerWorker component. In our model the difference between the two types of submits is modeled as two different transitions. In particular **c**_rab_ -**s**_ubmit(_**f**_irst)_ transition has an uplink (*:csf(task)*), which means

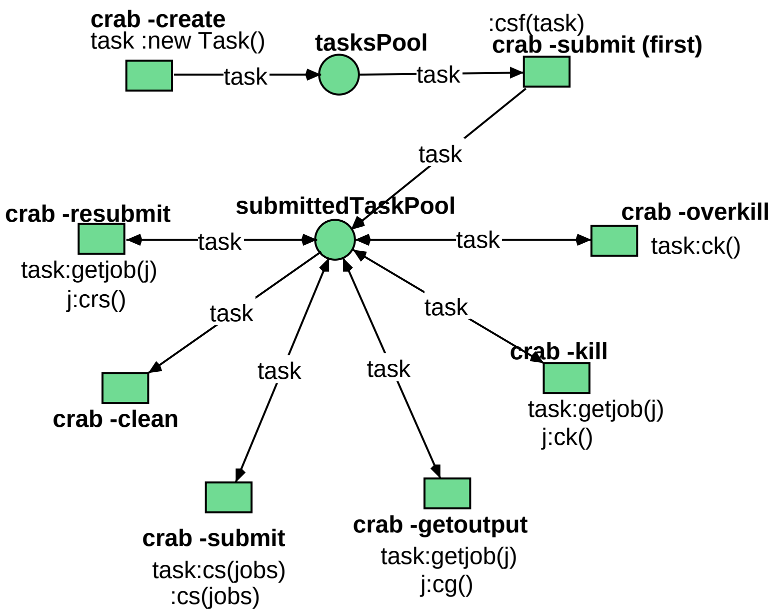


Figure 8.2: The CRABClient net.

that it must be synchronized with the upper level. As a result the task reference is copied to the TaskRegister component by the Overall System net. After creation, the main operations a user can do are submit, resubmit, kill, getoutput, and clean. All these operations require an interaction with the server, but since we have focused on the submission use case, these interactions have not been explicitly modeled. For example the *getOutput* command is modeled as an interaction between the client and the job by means of two inscriptions. Handling all the possible interactions between the actors involved in the system would have resulted in a very big model, making it impossible to describe in this paper.

A task, see Figure 8.3, is a bag of jobs (the system allows to collect up to 4000 jobs into a singe task) and it is a representation that CRAB uses to perform collective actions on the Grid processes. Places *notRegistered, registering, registered* of the Task net contain information about the state of a task itself. These places control the enabledness of transitions *crab -submitFirst*, and *taskRegistered*, which are respectively called by the CRABClient when a job is submitted, and by the TaskRegister component when the task has been successfully registered after a *submit first* operation. The submit transition is called when a CRABClient performs a *submit subsequent* action. In our model both *taskRegistered*, and *submit* transitions send upward two jobs through a synchronous channel, and make the job move to the submission request state.

The net representing the state of Grid jobs and their allowed actions is reported in Figure 8.4. This net has been modeled combining the finite state machine reported in the CRAB official documentation with the information extracted directly from the portion of code devoted to the Grid job state handling. Several transitions of this net contain uplinks, and therefore have to be synchro-

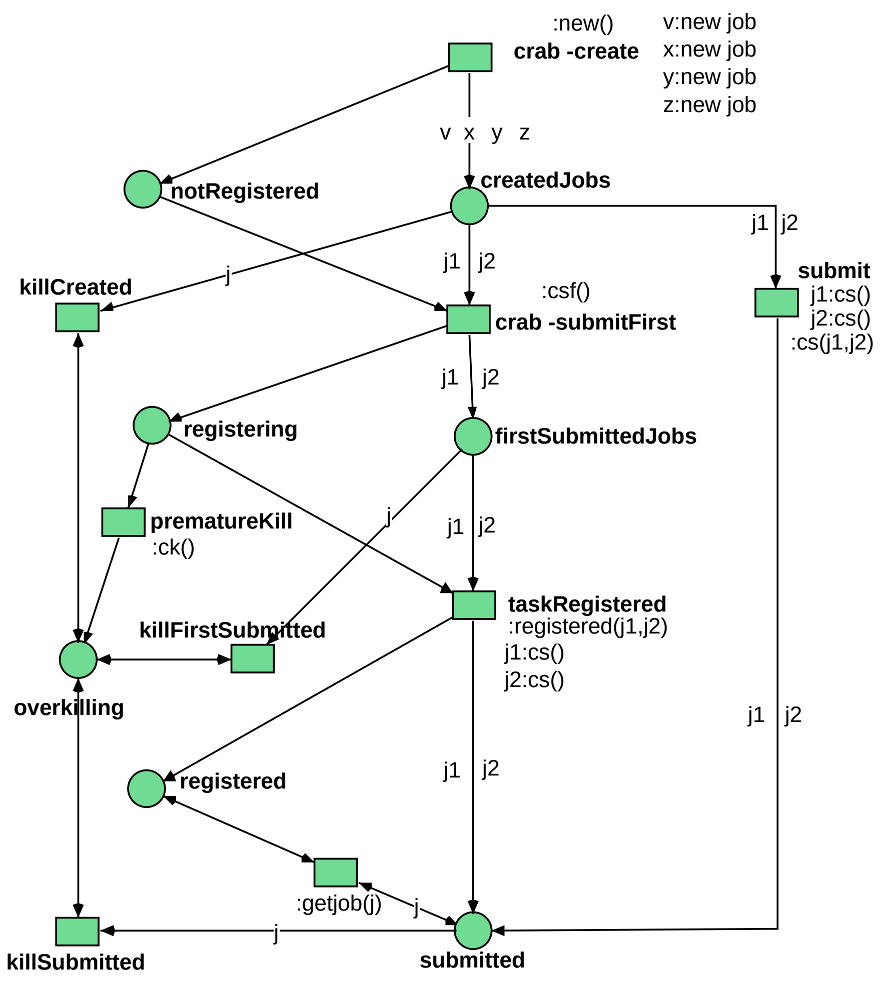


Figure 8.3: The Task net. Only four jobs are considered in order to exemplify the relation with the job net.

nized with some other net. Transitions with a *:crs()* uplink (CRAB Resubmit) are transition enabled only if the job is in a state where a resubmit is possible, and are synchronized with the *crab -resubmit* transition of the CRABClient net, or the *resubmit* transition of the SubmissionWorker net. In the same way killings (channel *:ck()*), failures (channel *:f()*), submission (channel *:s()*), and output retrieving (channel *:cg()*), have to be synchronized with a correspondent transition in another net.

The integration of the documentation and the code with the formalism of the nets has allowed us to identify a bug in the way job states are modified. In particular, the net allows some transitions that are not actually activated by any event observed by the system (bug 1, b1). For example let us consider the unlabeled transition between the *sub.success* and the *cleaned* places in Figure 8.4: the latter denotes that a job has been abandoned because the user security

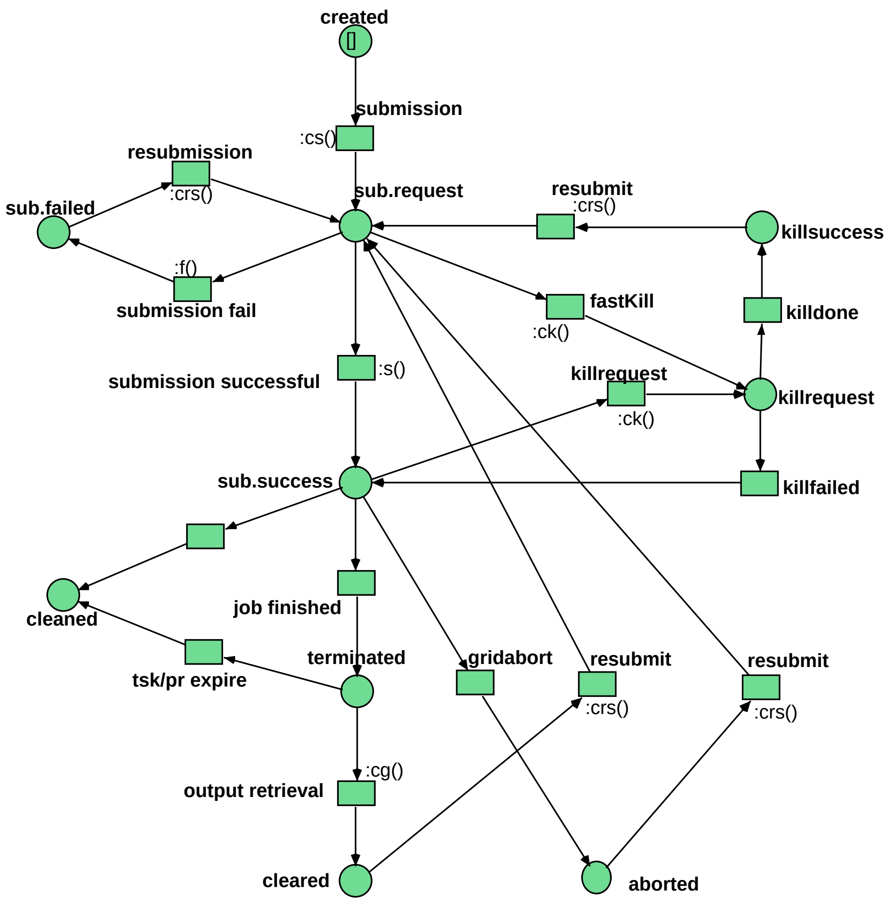


Figure 8.4: The Job net.

credentials are expired and the Grid will not manage processes whose owner cannot be recognized. A malicious code interacting with the clients in place of the proper server could move jobs arbitrarily to this terminal state. The fix for this bug consisted in a review of the code managing the job state automata in accordance with what is stated by the presented Job net. Also, the preconditions that allow a client to perform a *kill* request over the jobs are not granted properly (b2): jobs can be killed when they are in states where the killing is dangerous. For example, a user could run into a condition where a failed job cannot be resubmitted as the system requires to kill it. That means the job is in a deadlock, as a failed job cannot be killed on the Grid.

TaskRegister

The TaskRegister component, shown on the left of Figure 8.5, duplicates the task and jobs structures that have been created at the client side and alters all the object attributes in order to localize them with respect to the running environment of the server, taking care also of security issues (like user credentials

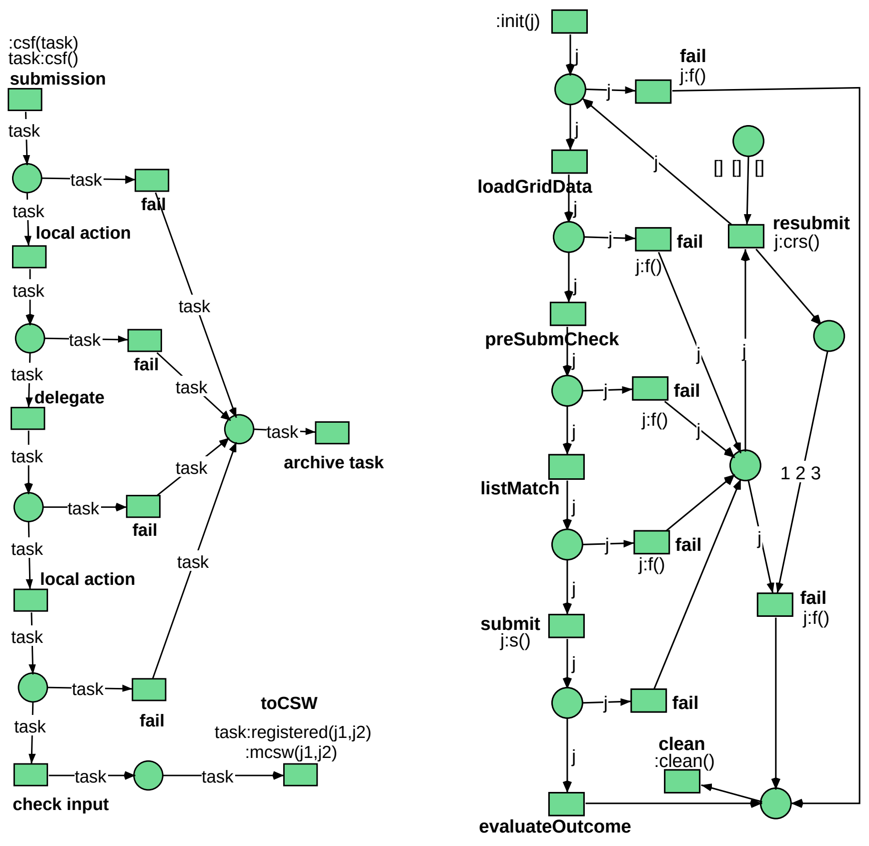


Figure 8.5: TaskRegister and SubmissionWorker nets respectively

delegation) and files movement (check the existence of input). We modeled this cloning by means of the *reference semantics*: the TaskRegister component receives from the client a copy of the reference which points to the Task.

The component is able to handle more tasks simultaneously thanks to a pool of threads implementing the net of Figure 8.5. The first transition that is fired is *submission*, which is synchronized with the transition in the system net that receives the task reference from the CRABClient. Then four operations which can fail are executed on the task. These include local modification of the task with respect to the server environment, the user's credential retrieval (also known as delegation), the setting of the server behavior according to what the credentials allow to do and, finally, the checking that the needed input files are accessible from the Grid. If the registration fails the only possible operation available is *archiveTask* which deletes the reference to the task from the task register component. If the user has the privileges to execute the jobs in the task, and if the inputs needed by the task are available, then a range of jobs is selected from the task and passed to the CRABServerWorker by firing the *toCSW* transition (again under the supervision of the system net). The modeling and the simulation of the TaskRegister net has highlighted some relevant defects and bugs. In case of failure the TaskRegister component was not able to set properly the status of the jobs in a task to fail. This macroscopic lack in the system design implied different side effects. The server was not able to discriminate whether to retry automatically the registration process or to give up and notify the user about the impossibility to proceed (b3). In addition, the system could not tell if the registration has been attempted previously. This implies that the client transfers the input data every time a registration failure appears, with a waste of network resources (b4). Both the defects have been solved by introducing the proper synchronization between the *fail* transition in the component with *submission failed* in the job net. Mapping the synchronization into the server code has granted that the status of the jobs is set to the correct failure state and that the submission counters are properly incremented (being implementative details the counter is not reported in the Job net). With this modification the server becomes aware that a first try has been executed and also network transfers are exploited more efficiently. A second bug has been identified thanks to the study of the synchronization among the transitions for the client, the jobs and the TaskRegister nets. In detail, the handling of the *kill* commands presents some issues. If a user requires to kill some jobs while the task is being registered, the system cannot distinguish properly which jobs have to be killed and therefore it applies an over-killing strategy by halting the whole task (b5). This happens because the code performs some sort of synchronization with the Task net instead of having rendezvous with the related transitions into the lists of killing jobs.

The killing of Grid jobs is a demanding action, both in terms of network communications and in terms of coordination among the different services involved in a Grid. Furthermore the killing of an analysis job is a permitted but infrequent action. For these reasons the CRAB developers have decided to suppress this early job termination feature in order to avoid the bug. Now users are allowed to kill jobs only once they have been actually submitted to the Grid.

CRABServerWorker, and SubmissionWorkers

In our model the result of a submit operation is that the CRABServerWorker component, shown in Figure 8.6, receives a structured token in the place *accepted*. If the submit was the first, transition *newTaskRegistered* is fired after the task has been registered by the TaskRegister component by means of transition *toCSW*, which is synchronized with transition *newTaskRegistered* through the overall system. If the submit is not the first, the task has been already registered, therefore transition *subsequentSubmission* is fired. After receiving the range of jobs, the CRABServerWorker component schedules these jobs for the execution on the Grid infrastructure. The practical effect of this component is to break the task into lists of jobs in order to improve the performance thanks to bulk interactions with the Grid middleware. The Submission Worker thread spawned by the component monitors the actual submission process of the jobs. We have modeled this fact by creating a Submission Worker net for each one of the jobs in the list. Indeed, transition *triggerSubmissionWorker* creates a new Submission Worker assigned to the variable *sw* and synchronizes it with a transition labeled *init*.

The thread is responsible both for tracking the submission to the Grid infrastructure, and for resubmitting jobs when a failure occurs. Failures can occur for different reasons: network communication glitches, unavailable compatible

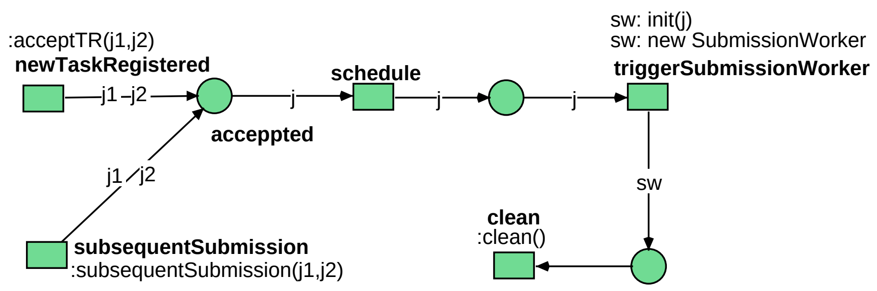


Figure 8.6: The CRABServerWorker Net

resources, etc. Some types of failures are recoverable and in those cases the Submission Worker automatically tries to resubmit the job a three times. This value can be configured in the code, but in the model we only used the actually employed value of three. If the failure persists the job is permanently marked as failed. The net shown on the right in Figure 8.5 is our model of the submission worker component.

The study of the synchronization between the job and the Submission Worker nets allowed us to identify another bug in the code. The *submission success* transition in the job net (Figure 8.4) synchronizes with the *submit* Submission Worker's transition (right of Figure 8.5). This means that the CRAB Server marks the submission as successful just after the interactions with the Grid. Actually the network latencies could delay the propagation of the job failure message (b6) and, therefore, the correct rendezvous should be enacted between *submission success* and *evaluateOutcome*.

It is relevant to observe that the approach followed for the modeling of the CRAB Server submission chain is a particular case for a quite general class of Grid systems. All the Grid middlewares rely on jobs that are represented by finite state automata and that are concurrently managed by the different services involved in the Grid. In addition, the intermediate action of a broker like the CRAB Server is becoming a common pattern with the diffusion of scientific gateways: programmatic portals that abstract the user applications from the complexities of the distributed infrastructures acting as back end.

The adoption of the nets-within-nets paradigm has provided a natural and effective way to model subtle interactions among the different net levels. It would have required a significantly greater effort to discover the same problems with a flat net approach. In the following subsection details about the process of deriving the models from the documentation and the code are given.

8.0.6 Details on the model derivation process

The model was derived from the code by analyzing both the official documentation and the source code of the system. The Job net is directly built from the documentation. A finite state automata which describes the Job is reported explicitly. After that, simply by using pattern matching we analyzed the source code relevant for the submission use case by searching for interaction with jobs. Each source module is modeled as a net (e.g.: CRABClient, TaskRegister, CRABServerWorker etc), and the interactions with the Job nets are modeled using the Renew uplink/downlink mechanism. A modification of the status of a job in the code is modeled as a pair of synchronized transitions in the model itself: one in the job net and one in the net that models the component changing the job status.

To ensure that the model is an accurate representation of the software, we made several task submissions with the CRAB tool and monitored the status of the jobs during the evolution. The request parameters were set up so that different behaviours of the system are tested. For example, jobs lacking of input files, job submitted by users with expired credentials, and jobs killed before the completion of task registration process are test cases that have been considered. After that, we simulated each submission on the model, taking care that the simulation of the status of the job net was consistent with the actual job status in the system.

Chapter 9

Tools For Hierarchical Nets

The verification of properties of a concurrent system is very important. Specifications critical to the correct execution of a system need to be verified in order to guarantee them after deployment. Unfortunately, as it has been discussed in Chapter 7, for many high level models important Petri net properties are undecidable. Therefore it is impossible to verify them using the known Petri net techniques. This is a common problem for high level Petri net models: properties which are computable with low-level formalisms become undecidable, and thus cannot be verified anymore, in some high-level models.

However, it is always possible to first restrict these formalisms in some way, so that they can later be translated into low-level formalisms, which in turn can be verified again. In the rest of this Chapter it will be described which features of reference net should be used, and which features should not be used if you want to use the RENEW plugin for modeling hypernets. The advantage of doing so is that hypernets can easily be translated into 1-safe nets, and then they can be analysed.

The main result regarding this work is the implementation of a RENEW hypernet plugin which incorporates features for computing S-invariants, and features for model checking a hypernet. As far as I know, this is the first time that analysis techniques typical of Petri nets has been implemented in a tool which support the nets-within-nets paradigm, and it is mature enough to be used in a real application context.

In the rest of the Chapter when we will talk about invariants we are always referring to S-invariants.

9.1 A Renew Plugin For Drawing And Analyzing Hypernets

9.1.1 Restricting Reference Nets to Hypernets

Restricting reference net is probably the most intuitive way to use verification techniques in RENEW. In particular, the use of a nets-within-nets formalism like hypernets as a restriction permits the use of the nets-within-nets paradigm, which is probably the most intresting feature in RENEW. The original contribute of the paper is to show how this plugin allows the use of verification techniques, like invariants and CTL model checking, to check properties of systems which are suitable to be modeled with the the nets-within-nets paradigm.

9.1.2 The Hypernet Plugin

From a technical point of view the implementation of a new formalism in Renew is done using a plugin mechanism. The most important method contained in the classes implementing the plugin is a *compile* method which takes as input a *shadow* net, a set of Java objects containing all the information about the net the user has drawn in the graphical editor of Renew, and transform it in a set of Java objects used by the simulator engine to simulate the net. This compile method is responsible for checking that the net drawn by the user is an actual hypernet in our case. In particular, in order to be able to use Renew as a hypernet simulator, the arc and transition inscriptions used in the modeling process must be restricted in such a way that the drawn net is a hypernet. Therefore the restrictions applied in the plugin are the following:

- Inscriptions (tokens) inside places can only be in the following forms: *identifier* or *identifier:netType*. In the first case the identifier represent the name of an empty net, and will be treated by the simulator engine as an black token; in the second case a new instance of the net *netType* will be created and placed inside the place.
- Inscriptions on arcs are restricted to single variables only. Each arc must contain exactly one variable inscription.
- The inscriptions of input (output) arcs must not be duplicated. In this way it is possible to preserve the identity of nets: duplication of tokens is forbidden.
- Balancing of transition has to be checked, i.e.: the set of variable names used to inscribe input arcs must coincide with the set of variable names used to inscribe output arcs.
- Communication places are deleted, and are simulated by means of synchronous channels. These channels are counted when checking transition balance.

For example, the airport agent shown in Figure 3.2 can be drawn as a hypernet in Renew using the net shown in Figure 9.1. The traveler empty tokens are place inscriptions $T1$ and $T2$, and the plane net instance is created by the $P1 : place$ inscription. Each transition is balanced. For example transition *deplane* in the airport has a bidirectional arc labelled pl, and an output arc labelled pa for which there is a correspondant downling, namely $pl : deplane(pa)$. Each communication place is deleted, and it is replaced with a synchronous channel. *Land* and *takeoff* transitions are equipped with two uplink because they were connected to two up-communication places. *Deplane* and *board* transitions contain two downlinks because they were connected to down-communicating places. The module name used to label communicating places is used to retrieve the variable name used in the downlink.

The $P1$ agent of Figure 3.1 is drawn in the hypernet plugin of Renew with the net in Figure 9.2. Again, up-communication places are replaced by channels,

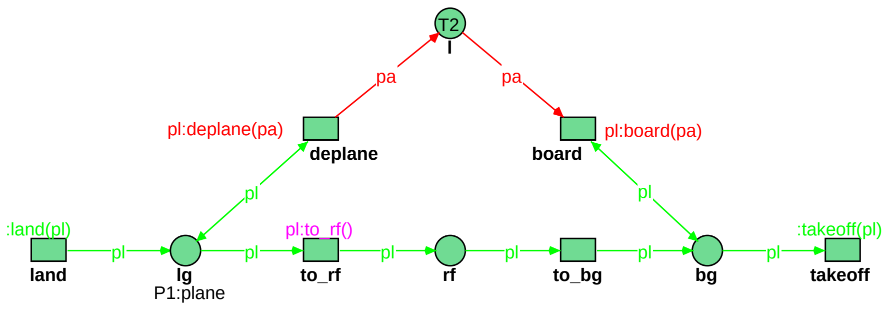


Figure 9.1: The airport agent drawn with the hypernet plugin of Renew

and transition *to_rf* must synchronise with the corresponding transition in the airport agent.

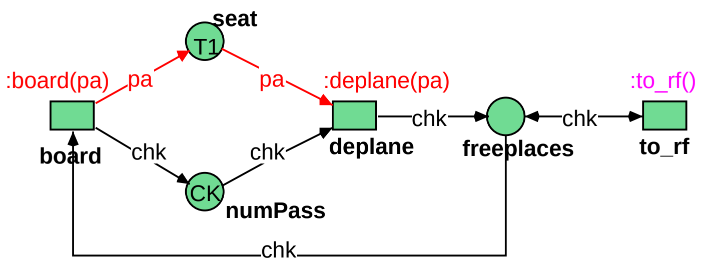


Figure 9.2: The plane agent drawn with the hypernet plugin of Renew

As we already mentioned, thanks to the expansion to 1-safe nets it is possible to use verification techniques defined for this class of net to analyse system modelled with a hypernet. Two of the most useful techniques are invariants analysis, and model checking. We explored two possibilities of using them in the plugin we implemented: internal implementation in Renew, or exporting the 1-safe net in a format understandable by other tools. Since implementing these analysis techniques in an efficient way is a difficult task (some tools are very elaborated, and have been implemented over several years), and since very efficient open source tools are available for free, we decided to use external tools to implement invariant analysis, and model checking of a hypernet.

In the following sections we will show how the extensions and incorporation can be used in a practical example.

9.2 Example

The invariant analysis, and the model checking extensions we implemented in Renew can be used to prove properties of a system. We have chosen the external tools LoLA (see `http://www2.informatik.hu-berlin.de/top/lola/lola.html`) and INA (see `http://www2.informatik.hu-berlin.de/~starke/ina.html`) for analysing purposes. Starting from the airport example of Chapter

3 shown in Figure 3.2, we will prove using invariants that there is never more than one passenger on the plane, and we will prove using the model checker that a plane never refuels if there are a passenger on board.

By running the invariant analysis we get the following invariants:

T2@l	T2@seat	CHK@pass	P1@lg	P1@rf	P1@bg	CHK@freepl	T1@seat	T1@l
0	0	0	1	1	1	0	0	0
1	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1
1	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	1	0

The first four invariants are those which guarantee the truth of "law of conservation of agents", achieved thanks to the state machine decomposition in the formalism. For each agent there is a corresponding invariant indicating the places in which that agent can be located. Since the places of each invariant contains only one token in the initial marking, it is mathematically proved that each agent can be only in certain places: the places which are of the same sort of the agent itself. Moreover, these four invariants can also be used to prove that the net is 1-safe: they cover all places of the net, and contain only one token in the initial marking.

The fifth invariant is $\{\langle T2, l\rangle, \langle CHK, numPass\rangle, \langle T1, l\rangle\}$ and contains two tokens in the initial marking. Together with the second and the fourth invariants it can be used to prove that if the place $\langle CHK, numPass\rangle$ is marked then one of the two passenger is seated on the plane. The place is not marked only if both passenger are in the airport.

The sixth invariant is the counterpart of the fifth, and states that only one of the following places can be marked: $\{\langle T2, seat\rangle$, $\langle CHK, freeplaces\rangle$, $\langle T1, seat\rangle\}$. The information is clear: only one passenger can be in the *seat* place of the plane. If none of them is in the plane $\langle CHK, freeplaces\rangle$ is marked.

In Figure 9.3 a screenshot of Renew after the computation of invariants is shown.

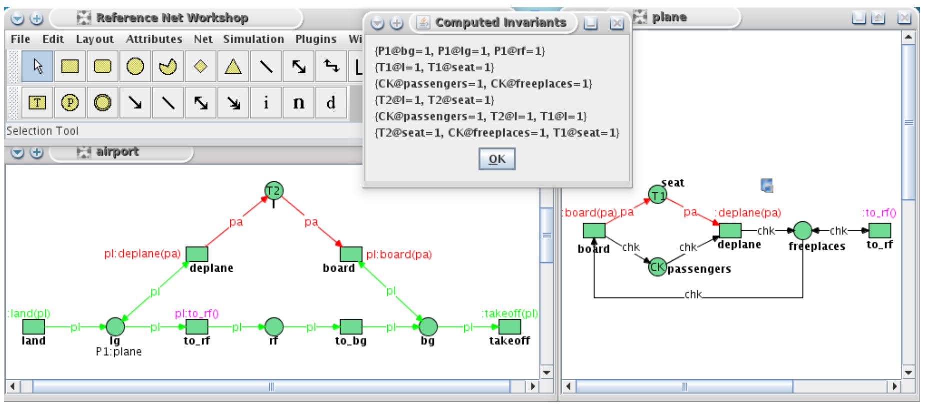

```
{P1@bg=1, P1@lg=1, P1@rf=1}
{T1@l=1, T1@seat=1}
{CK@passengers=1, CK@freeplaces=1}
{T2@l=1, T2@seat=1}
{CK@passengers=1, T2@l=1, T1@l=1}
{T2@seat=1, CK@freeplaces=1, T1@seat=1}
```

Figure 9.3: A screenshot of the invariants computed inside Renew

While invariants analysis can be launched, and the computed invariants can be analysed to extract information about the system, in order to analyze the system using model checking a formula specified in a temporal logic is needed. Since we choose LoLA, which is a CTL model checker, we need to specify the property we want to verify using this logic. For example, checking the property "if the plane is located in the place representing the refueling station then no passenger is on board" can be done by entering as input of the Renew plugin we implemented the following CTL formula:

$$\begin{aligned}&ALLPATH\ ALWAYS\\ &NOT\ ((T1.seat = 1\ AND\ P1.rf = 1)\ OR\ (T2.seat = 1\ AND\ P1.rf = 1))\end{aligned}$$

The formula checks that in every reachable state ($ALLPATH\ ALWAYS$) the situation in which both placed $\langle T1, seat\rangle$ and $\langle P1, rf\rangle$ are marked never occurs (and the same for places $\langle T2, seat\rangle$ and $\langle P1, rf\rangle$). The analysis performed confirms that the truth value of the formula is *true*, which is enough to guarantee that the property is true for the system.

As it can be seen in this simple example, the advantage of using model checking is that it is possible to express, and consequently to verify, more properties compared to invariant analysis. In our example, the information that a plane never refuels if a passenger is on board is not present in the computed invariants, but can be verified using the model checking. However, the drawback is that it is necessary to explore the whole state space of the system in order to verify a property. Invariants are computed on the static structure of the net, which is usually exponentially smaller compared to the state space of the system. In general, in real huge application both the techniques are useful: invariants give a quick overview of some properties of the system, model checking take more time and it can be used to verify specific properties of the system.

Chapter 10

Conclusions and Future Developments

In this dissertation the problem of modeling systems of mobile agents with the hypernet formalism has been addressed. The main contributions and results of the thesis can be categorized in four fields: a study of the model focused on improving its flexibility; the study of properties of the model; the modeling of a real application; the development of a prototype which allows to draw and to analyze a hypernet.

The need of a generalization of the basic model arose in [5], when it was not possible to model with basic hypernets a class of membrane systems, computational models based upon the architecture of a biological cell. The model of generalized hypernets has been studied in this thesis, and it has been shown that the main characteristics of the basic model, like the preservation of the tree-like structure of the marking, are preserved. It has also been proven that, starting from a generalized hypernet, it is possible to build a one safe net with an equivalent behavior.

One of the main results of the thesis concerns the definition of the notion of unfolding for a generalized hypernet. Unfoldings are mathematical structures which explicitly represent concurrency and casual dependence between actions, but hide information about all the possible interleavings of concurrent actions. The result is a compact representation of the state space of a hypernet, which can be exploited by behavioral techniques which explore all the possible states, like model checking.

The thesis also covers a real and concrete application context. The nets-within-nets paradigm has been used to model a component of the Grid tool for High Energy Physics data analysis used by scientists working at the Compact Muon Solenoid experiment at the CERN of Geneva. The interactions between jobs which need to be executed on the Grid infrastructure, and the software components of the tool were modeled explicitly and in a natural way using nets-within-nets and Renew.

Finally, the implementation of a Renew hypernet plugin which incorporates features for computing S-invariants, and features for model checking a hypernet was one of the subject of this thesis. This plugin checks if the net drawn by the user is a hypernet, and warns the user if hypernet constraints have not been obeyed. The modeled system can be simulated using the internal simulator of Renew. The most important feature of the plugin is that it provides functionalities for analyzing a hypernet. The tool can generate the 1-safe net equivalent to a given hypernet, and then it can invoke external tool to perform analysis on the 1-safe net.

Future developments will concern the study of how to further extend the generalized hypernet model by adding mechanism to create new agents. This is a major change in the formalism. In fact, the expansion toward 1-safe net will not be more possible if creation of agents is possible, and the state space will not be limited anymore. As a consequence, decidability issues may arise because and properties which are decidable with the current model may become undecidable

Another future line of research regards the definition of a logic for expressing properties of generalized hypernets considering both the temporal evolution of agents and their structural correlation. The starting point for this thread of research are the work on agent aware transition systems [4, 1], where two classes of logic capable of expressing the dynamic evolution of the structural correlation have been defined, and glued together in a powerful language called CTL^2.

Having a software which support the formalism and which implements analysis techniques is very important for the success of a model, and allows people to use the formalism in real application contexts. Because of that, the last line of research I want to explore concerns the optimization of the unfolding algorithm presented in Chapter 6, and its implementation in the plugin presented in Chapter 9.

Bibliography

[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, June 2006.

[4] Marek A. Bednarczyk, Wojciech Jamroga, and Wieslaw Pawłowski. Expressing and verifying temporal and structural properties of mobile agents. *Fundam. Inform.*, 72(1-3):51–63, 2006.

[5] 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.

[6] E. Best and C. Fernandez. *Nonsequential Processes–A Petri Net View*, volume 13 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.

[7] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677 –691, aug. 1986.

[8] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 253–292. Springer Berlin / Heidelberg, 1998.

[9] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Information and Computation*, 201(1):1 – 54, 2005.

[10] CERN. Worldwide LHC Computing Grid. http://lcg.web.cern.ch/lcg/public/. Accessed May, 2010.

[11] 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.

[12] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[13] P.M. Cohn. *Universal Algebra*. Springer, Berlin, 1981.

[14] 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.

[15] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *Software Engineering, IEEE Transactions on*, 24(5):315 –330, may. 1998.

[16] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2005.

[17] Joost Engelfriet. Branching processes of petri nets. *Acta Inf.*, 28(6):575–591, 1991.

[18] Javier Esparza and Keijo Heljanko. A new unfolding approach to LTL model checking. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 475–486. Springer, 2000.

[19] Javier Esparza and Keijo Heljanko. Implementing LTL model checking with net unfoldings. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2001.

[20] Javier Esparza and Keijo Heljanko. *Unfoldings: A Partial-Order Approach to Model Checking*. EATCS Monographs on Theoretical Computer Science. Springer Publishing Company, Incorporated, 2008.

[21] Javier Esparza and Keijo Heljanko. *Unfoldings: A Partial-Order Approach to Model Checking (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2008.

[22] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002. 10.1023/A:1014746130920.

[23] P. Frisco. About p systems with symport/antiport. *Soft Computing*, 9(9):664–672, September 2005.

[24] Serge Haddad and Denis Poitrenaud. Theoretical aspects of recursive Petri nets. In S. Donatelli and J. Kleijn, editors, *Proceedings of the 20th International Conference on Application and Theory of Petri Nets (ICATPN'99)*, volume 1639 of *Lecture Notes in Computer Science*, pages 228–247, Williamsburg, Virginia, USA, June 1999. Springer Verlag.

[25] Serge Haddad and Denis Poitrenaud. Modelling and analyzing systems with recursive Petri nets. In *Proceedings of the 5th Workshop on Discrete Event Systems (WODES'2000)*, pages 449–458, Ghent, Belgium, August 2000. Kluwer Academic Publishers.

[26] Serge Haddad and Denis Poitrenaud. Checking linear temporal formulas on sequential recursive Petri nets. In *Proceedings of the 8th International Symposium on Temporal Representation and Reasonning (TIME'01)*, pages 198–205, Cividale del Friuli, Italie, 2001. IEEE Computer Society.

[27] Serge Haddad and Denis Poitrenaud. Recursive Petri nets – Theory and application to discrete event systems. *Acta Informatica*, 44(7–8):463–508, December 2007.

[28] Kathrin Hoffmann, Hartmut Ehrig, and Till Mossakowski. High-level nets with nets and rules as tokens. In Gianfranco Ciardo and Philippe Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2005.

[29] Victor Khomenko, Maciej Koutny, and Walter Vogler. Canonical prefixes of Petri net unfoldings. *Acta Informatica*, 40:95–118, 2003. 10.1007/s00236-003-0122-y.

[30] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, 1994.

[31] 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.

[32] 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.

[33] Michael Köhler and Heiko Rölke. Properties of object Petri nets. In Cortadella and Reisig [14], pages 278–297.

[34] Michael Köhler-Bußmeier. Hornets: Nets within nets combined with net algebra. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 243–262. Springer Berlin / Heidelberg, 2009.

[35] Michael Köhler-Bußmeier and Frank Heitmann. On the expressiveness of communication channels for object nets. *Fundamenta Informaticae*, 93(1-3):205–219, 2009.

[36] Olaf Kummer. *Referenznetze.* Logos-Verlag, 2002.

[37] 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 [14], pages 484–493.

[38] Charles Lakos. A Petri net view of mobility. In Farn Wang, editor, *FORTE*, volume 3731 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2005.

[39] 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.

[40] Irina A. Lomazova and Ph. Schnoebelen. Some decidability results for nested Petri nets. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 1755 of *Lecture Notes in Computer Science*, pages 208–220. Springer, 1999.

[41] Carlos Martín-Vide, Andrei Păun, and Gheorghe Păun. On the power of P systems with symport rules. *Journal of Universal Computer Science*, 8(2):317–331, 2002.

[42] Carlos Martín-Vide, Andrei Păun, Gheorghe Păun, and Grzegorz Rozenberg. Membrane systems with coupled transport: Universality and normal forms. *Fundamenta Informaticae*, 49(1-3):1–15, January 2002. Special Issue: Membrane Computing (WMC-CdeA2001) Guest Editor(s): Carlos-Martín-Vide, Gheorghe Păun.

[43] K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In Gregor von Bochmann and David Probst, editors, *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer Berlin / Heidelberg, 1993.

[44] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

[45] Andrei Păun and Gheorghe Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–305, May 2002.

[46] Andrei Păun, Gheorghe Păun, and Grzegorz Rozenberg. Computing by communication in networks of membranes. *International Journal of Foundations of Computer Science*, 13(6):779–798, December 2002.

[47] Gheorghe Păun. Introduction to membrane computing. In *First brainstorming Workshop on Uncertainty in Membrane Computing, Palmade Mallorca, Spain*, 2004.

[48] Rüdiger Valk. Nets in computer organisation. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 1986.

[49] 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.

[50] 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.

[51] Kees M. van Hee, Irina A. Lomazova, Olivia Oanea, Alexander Serebrenik, Natalia Sidorova, and Marc Voorhoeve. Nested nets for adaptive systems. In Susanna Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2006.

[52] Kees M. van Hee, Olivia Oanea, Alexander Serebrenik, Natalia Sidorova, Marc Voorhoeve, and Irina A. Lomazova. Checking properties of adaptive workflow nets. *Fundam. Inform.*, 79(3-4):347–362, 2007.

[53] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246. Springer Berlin / Heidelberg, 1993.