Visualizing Actor Programs using Predicate Transition Nets*

Shakuntala Miriyala† and Gul Agha
Dept. of Computer Science
University of Illinois
Urbana, IL 61801, USA
{shaku | agha}@cs.uiuc.edu

Yamina Sami‡
LRI, UA410 CNRS
Bât 490, Université Paris-Sud
91405 Orsay Cedex, France
sami@lri.fr

Abstract

The Actor model of concurrent computation unifies the functional and object-oriented programming language paradigms. The model provides a flexible basis for concurrent programming. It supports local state, dynamic creation and configuration, and inherent parallelism. Because of the fluidity of Actors, specifying and debugging actor systems is often considered difficult. We believe visual programming techniques are of fundamental importance in addressing powerful concurrent systems of this nature. Not surprisingly, a number of methods to visualize actor programs have been proposed. We give an outline of visualization techniques and their relation to actors. We then discuss one such proposal, namely the use of Predicate Transition nets, to visualize actor programs.

1 Introduction

The problem of developing suitable visual representation to support specification and debugging of concurrent systems is a fundamental one. We are particularly interested in dynamically evolving

*This research described has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195). The authors wish to thank Guy Vidal-Naquet for his work in the area, Anna Patterson, Rajendra Panwar and other members of the Open Systems Laboratory at the University of Illinois for their suggestions, help and discussions, and the anonymous referees for their comments.

†This work was primarily done when the first author was at the University of Illinois at Urbana-Champaign. The first author is currently at Vista Technologies, 1100 Woodfield Road, Suite 108, Schaumburg, IL-60713.

‡The third author is also affiliated with LIFAC, ENS de Cachan, 61, Avenue du Pŕesident Wilson, 94235 Cachan Cedex, France

scalable concurrent systems open to interaction with their environment. Our research in visual formalisms has three broad objectives: visual specification, program visualization, and visual interaction with the system. The objectives may be described as follows:

Visual Specification: A programming environment should permit the visual specification of both the structure and the dynamic process flow in a concurrent system.

System Visualization: A programming environment should allow observation of program execution in terms of the high-level visual notation which is used to specify a program. In other words, a user should be able to receive feedback about her/his program's execution behavior at the same level at which she/he provides a specification for the program.

Visual Interaction: A user should be able to dynamically modify and debug the visual representation of an executing program without stopping the entire system.

In this paper, we focus on a system for observing the behavior of actor programs in terms of a visual specification language. We are continuing to study methods for incremental system construction and debugging using the visual formalism. Visual representation of concurrent systems is far more complex than that of sequential ones. In a conventional sequential program, only transformations of the state of a single process need to be represented. By contrast, in a concurrent system, state transitions within a number of processes can occur in parallel. It is not sufficient to simply represent all processes that are executing in parallel – inter-process interaction must be represented in a visually meaningful manner.

A visual programming environment which required the explicit specification of all processes would not be very useful in a large-scale system. Explicitly representing a very large number of processes would compromise the ability to use the system as a visualization tool: despite the relatively high bandwidth of visual perception in humans, it is not possible to visually track the execution of a very large number of processes. Our goal is to develop abstractions which can represent the behavior of concurrent systems at a high-level. An example of this kind of abstraction is the ability to provide the similar visual representation to systems which have identical structure but distinct dynamics. It is important to provide a visual denotation of programs which allows composition of the visual representation of constituent expressions of a program to determine the visual representation of the entire program. In particular, the representation should use expressions in the source program, not the interpretation of the expressions at a lower level.

A difficulty in developing a general visual notation for concurrent systems is the fact that there is no single universal model of concurrency. For example, an analysis of available parallel architectures suggests at least three distinct models of parallel computation: data parallelism, shared memory, and distributed memory. The corresponding programming models have different

kinds of support for abstraction and different measures of complexity. It is not clear that a single general visual model is the best approach for supporting all models of parallel computation. Because it is the most scalable architecture, we have chosen to focus on a distributed memory model – namely that of multicomputers. A multicomputer is an ensemble of computers (processors with local memory) connected by a global (virtual) network.

The actor model combines procedural and data abstraction with concurrency. Thus, Actors provide a powerful programming model for multicomputers [AS88, Dal86]. Actors were originally proposed in the mid 70's by Carl Hewitt [Hew77]. The concept has continued to evolve and has recently generated increasing interest. The actor model has been formally characterized by means of power domain semantics [Cli81], by a transition system [Agh86], by term rewriting and category theoretic constructs [Mes90] and by Petri Nets and related models [SVN91, ELR90, JR89]. Actors have been proposed as a universal model for parallel complexity [BVN91]. The model has also been proposed as a basis for multiparadigm programming [Agh89] and variants of it have been used to support concurrent object-oriented programming [Agh90b, Yon90].

We use a net-theoretic model, namely *Predicate Transition net* (PrT-net), to provide a visual representation of actor systems. PrT-nets are a high-level extension of the Petri Net model of concurrency. Under certain finiteness restrictions, PrT-nets can be mapped to equivalent Petri nets. Net theory is a well-studied model of concurrency. In particular, a number of net theoretic techniques have been developed to formally verify properties of concurrent systems such as reachability, deadlock, livelock, etc. (see, for example, [GL81, Vau87, Rei85]).

Note that Predicate Transition nets are formally equivalent to Colored Petri Nets used by Sami and Vidal-Naquet to model actors. In fact, our model is similar to the Sami and Vidal-Naquet approach; the main difference between the two representations is that we represent actor behaviors defined in the source program as encapsulated visual objects, while in the CPN approach, actor behaviors are interpreted in terms of their primitive actions. One way to look at the difference is that we provide a layer of abstraction for ease of visualization, while the work of Sami and Vidal-Naquet provides a formal justification for the use of Predicate Nets. In particular, the latter work shows that CPN's can emulate important aspects of arbitrary actor programs defined in Agha's semantics [Agh86].

Our interest is a more practical one – we have implemented and experimented with the system we describe. We have also developed a number of important generalizations and specializations which simplify the visual descriptions of unserialized actors, enabledness conditions, multiparty interactions, hierarchical abstractions, and meta-level architectures.

The outline of this paper is as follows. Section 2 discusses related research in visual formalisms for concurrent systems. Section 3 describes the actor model. Section 4 provides a basic introduction to Predicate Transition nets. Section 5 builds on the previous two sections by means of an

example illustrating the use of PrT-nets for representing a class of actor systems. Section 6 gives transformation rules for deriving PrT-nets from actor programs. Section 7 presents a comparison between the work presented in this paper and the Sami and Vidal-Naquet approach to modeling of actor programs. Section 8 describes a prototype implementation and our experience in using it. The final section outlines directions for future research.

2 Related Work

A number of visual representations for specifying or observing the execution of concurrent programs have been proposed. The representations include the specification language G-LOTOS, Roman's shared dataspace visualization system, event diagrams, graph grammars, actor grammars, and net theoretic models. In particular, a number of visual models, including GARP, Augmented Event Diagrams, and various net models, have been used to specify the structure or behavior of actor systems. We describe some of these systems and their applicability to developing representations for actor-like dynamic systems.

G-LOTOS

Graphic-LOTOS (G-LOTOS) [BLP88] is a visual notation for the LOTOS specification language. LOTOS is closely related to Milner's Calculus of Concurrent Systems – it allows synchronous communication between agents whose interconnection pattern is fixed at compile time. In G-LOTOS, each basic element of a program is mapped to a pictorial representation. In particular, behavior expressions (in process definitions) and process instantiations are given different syntactic representations.

A basic goal of G-LOTOS is to provide unambiguous specifications which serve as a sound basis for the analysis and implementation of concurrent programs. The pictorial syntax is meant to improve the readability of behavior expressions and to highlight aspects such as sequentiality, parallelism, synchronization and choice. However, because of limitations in the underlying LOTOS language, G-LOTOS models only a static interconnection topology of processes. This makes it unsuitable for a number of applications such as fault-tolerant computing.

Roman's Visualization System

Roman proposes a declarative approach to visualization and applies the methodology to a shared dataspace model of concurrency [RC89]. In the shared dataspace model, concurrent processes communicate via addressable data structures. Roman's visualization approach is to require a formal mapping between computational states of a program and their rendering in terms of graphical objects. This is in contrast to conventional algorithm animation systems which require

that explicit directives to the visual rendering system be issued within programs. The directives are to be used to affect the visual representation of the state during the execution of the program.

Roman argues that visualizing the state of concurrent programs makes it easier to infer their correctness: deadlock is lack of activity, repetitious cyclical patterns represent livelock, and so on. While the declarative approach is in principle superior to explicit directives, the current state of the art requires specifying a rendering function on a per system basis. Further work is required to provide a basis for building reusable abstractions.

GARP

In GARP (Graphical Abstraction for concurRent Processing) [Goe90], a running program consists of a set of processes which communicate by message passing. Internal behavior of each process is defined structurally using text and communication between processes is specified graphically. GARP uses $\triangle$-grammar rewrite rules to expand complex nodes into simpler subgraphs [KGC89].

GARP provides an interleaving model of concurrency: computation may be carried out by repeatedly choosing a production from a set of rewrite rules, and applying it to transform the graph. The specifications inside the internal nodes take care of evaluating expressions, sending and analyzing messages and making decisions to control the computation [Goe90]. Although GARP effectively represents the potential primitive actions of an actor, it does not provide a mechanism for representing complex multiparty interaction patterns as a single visual abstraction.

Actor Grammars

Actor grammars are a formalization of finitary actor systems in terms of a graph grammar model [JR89]. The actor grammars framework is particularly suitable for representing *configurations* (distributed snapshots) of actor systems. Corresponding to each configuration of an actor system, there is a graph and, corresponding to each transition in an actor system, there is a graph transformation. The execution of an actor system is controlled by configuration graph transformations based on actor grammars. The actor grammar representation is somewhat more abstract than the $\triangle$ representation. Productions in actor grammars handle an entire event whereas in $\triangle$ semantics a combination of productions is needed to handle a single event. On the other hand, the $\triangle$ approach uses fewer production rules than actor grammars to derive graph transformations.

Event Diagrams

In event diagrams, an actor is represented by a vertical line which marks the local order of *events* at an actor, where an event is the (atomic) processing of a communication. A communication between two actors is represented by an arrow from the lifeline of the sender to the lifeline of the recipient. The origin of an arrow is the event that caused a communication to be sent while

the destination is the event that represents the processing of the communication. Each lifeline is augmented with a box like structure to represent pending events, i.e., communications which have been sent but not processed.

Augmented Event Diagrams were used by Manning in the Traveler observatory to support the debugging of actor programs [Man87]. A significant source of complexity in event diagrams is that they contain every event, and in any realistic concurrent systems, there are simply too many events. Manning addresses this difficulty by structuring computations into sets of events which causally connect a response to a request (much like a transaction). Traveler allows the selective unfolding of event diagrams to visualize finer grains of activity.

Although the Traveler has proved to be a useful debugging tool, it suffers from a number of problems as a visualization tool. First, it does not provide a visual representation of the internal state (behavior) of an actor. In particular, it does not abstract over groups of actors which are instances of the same behavior. Second, traveler maintains the entire history of the computation distributed over large numbers of actors. While saving the history allows retrospective analysis, it is space inefficient and masks some of the higher-order static relations between actors. Finally, Traveler has no inherent ability to abstract over groups of actors and events representing some interaction pattern – its grouping mechanism is in terms of events rather than actors. Traveler is an evolving system and some of these deficiencies may be addressed by ongoing research.

Net Theoretic Approaches

Net theoretic models have been used in visual programming for a number of applications. For example, the Trellis Hypertext system [SF90] uses timed petri nets for temporal hyperprogramming. At least two distinct net theoretic approaches have been proposed to model actor-based systems: the Parallel Object-Based Transition System (POTS) and Colored Petri Nets (CPN). In fact, the model we use is closely related to CPN.

Engelfriet, Leih and Rozenberg [ELR90] developed the POTS formalism to represent parallel *object-based* systems. POTS is a Petri net model augmented by an additional annotation used to indicate the *acquaintances* of an object (acquaintances are objects an object knows about). An actor A is known as an acquaintance of actor B if B knows the mail address of A. The terminology comes from actor literature where an actor's acquaintances represent its local state and the acquaintance relations determine the interconnection topology. Traditional Petri nets are used to specify the execution behavior of a system; specifically, objects are represented by places and events by transitions. The annotations allow reasoning about the evolution of the interconnection topology of objects. Actor systems are defined as a special subset of object-based transition systems. Note that the Petri net model is static; in order to represent the behavior of actor systems, the set of actors which may be potentially created, must be pre-determined. The

number of such actors can be quite large – providing a visual representation to each of them may make it harder to visualize large systems. On the other hand, POTS was not designed to be a visualization system: it simply provides a basis for understanding the behavior of actor systems in terms of another fundamental model of concurrency.

Sami and Vidal-Naquet proposed the use of a high-level net, namely Colored Petri Net (CPN), to model actors [SVN91]. The transformation from actor programs to a CPN is based on the actor commands within a behavior definition. Additional places for address generator and communication are introduced in the transformation.

3 The Actor Model

The *actor model*, first proposed by Carl Hewitt [Hew77] and later developed by one of the authors [Agh86], captures the essence of concurrent computation in distributed systems at an abstract level. In the model, the universe contains computational agents, called *actors*, which are distributed in time and space. Each actor has a conceptual location (its *mail address*) and a *behavior*.

The only way one actor can influence the actions of another actor is to send the latter a communication. An actor can send another actor (or itself) a communication only if it knows the mail address of the recipient. Because information is localized the mail address of the target actor must be locally accessible to the sender before the latter can send a message to the target actor. Laws of locality restrict which actors are locally accessible when an actor receives a communication to the following: acquaintances of the actor processing the message, mail addresses contained in the message itself, and new actors created as a result of processing the message. Mail addresses can be used to send a message, they can be communicated in messages, and they can become acquaintances of actors which process such messages.

If more than one actor sends a message to the same actor, the messages are interleaved; in particular, this implies that the communications between actors are buffered. By contrast, in a synchronous communication model, a sender is required to wait until the recipient is free to accept a communication. The recipient is blocked from accepting any other communications until after it has finished accepting the first communication. One can think of each actor as having a mailbox which is always available to receive messages. Furthermore, mail addresses can be freely communicated. This assumption implies that the interconnection network topology of actors is dynamic. A dynamic topology provides generality, for example, it allows resource management decisions such as object to processor mapping to be programmed.

Process models assume that each process carries out one action at a time. By contrast, the behavior of actors is inherently concurrent. Actors support local state: an actor computes its *replacement* behavior in order to account for changes in its responses subsequent to the commu-

nication it has just processed. A program in an actor language consists of [Agh86]:

- *behavior definitions* which associate a behavior schema with an identifier, without actually creating an actor.
- *new expressions* which create new actors and return their mail addresses. A new expression is wrapped in a let command to bind the mail address it returns to local variables.
- *send commands* which create communications. A specific, known actor to which the communication is sent (called the *target* actor) must be specified. Communications may contain mail addresses of other actors, thus changing the interconnection network topology of actors.
- *become commands* specifies how the behavior of an actor is to be changed. A become command must be embedded in a behavior definition and it will be used to determine the new behavior of an actor whose definition is specified using the behavior schema.

The behavior definition for checking accounts below illustrates programs in Rosette, an Actor language developed at MCC [TSS89]. A behavior definition is used in a `new` command to create individual actors with specific initial parameters. In this case, a number giving the initial value of `balance` and `my-savings` would be specified for creating a `check-acc` actor. Note that the behavior of a checking account changes over time as a function of the balance in the account. The checking account can process *deposit*, *show-balance* and *withdraw* requests. A request to deposit some amount causes an update of the balance in the checking account. A *show-balance* request sends a message to a customer indicating his/her balance. A *withdraw* request results in either updating the checking balance and sending some money to the customer or sending a withdrawal request to the savings account or sending a message to the customer and teller indicating that the transaction could not be processed depending on the balance in the checking and savings account.

```
(defActor check-acc (slots& balance my-savings)
   (deposit [amount]
     (become check-acc (+ balance amount) my-savings))
   (show-balance []
     (send customer (display balance)))
   (withdraw [amount]
      (if (>= balance amount)
         ((become check-acc (- balance amount) my-savings)
          (send customer amount))
         ((let [[b = amount - balance]]
            (if (>= b my-savings)
                ((withdraw savings-acc b customer)
                 (become check-acc 0 (- my-savings b)))
                ((send customer 'cannot-process-request)
                 (send bank-teller 'cannot-process-request)
                 (become insens-acc balance b ))))))))
```

An insufficient amount in the savings account results in not being able to carry out the requested transaction. The customer and bank-teller are informed that the request cannot be processed and the checking account becomes insensitive. The checking account remains insensitive till a certain amount is deposited or there is a withdrawal request for a lesser amount. We have not shown the behavior definition of an insensitive account.

Because actors may be created and mail addresses may be communicated, the Actor model does not require the structure or shape of a computational problem to be statically determined, or that the number of actors participating in the computation be fixed. Furthermore, with the help of the behavior changing primitive, it is possible to delineate critical sections and support synchronization. Note that actors can process communications concurrently as long as synchronization requirements have not been specified by the structure of an actor system. the Actor model guarantees delivery of a communication – a form of *weak fairness*. The guarantee of delivery is useful for proving liveness properties in actor systems. In our visual specification of actor systems, we do not directly address fairness issues but assume that the underlying implementation provides fairness.

4 Predicate Transition Nets

Petri Nets were proposed thirty years ago to model distributed systems. To model increasingly large and complex systems, models of high-level nets, such as Predicate Transition Nets and Colored Petri Nets, have been developed. We first present a brief introduction to Petri nets and then describe Predicate Transition Nets. A Petri net structure consists of three components: a set

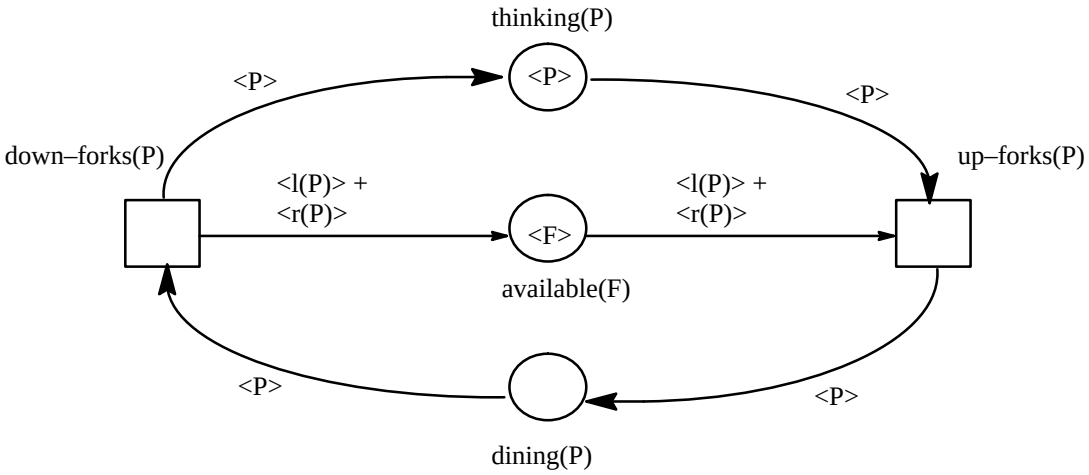

Figure 1: A predicate transition net for dining philosophers: *thinking(P)* represents the state of the thinking philosophers. *dining(P)* represents the dining philosophers. The transition *up-forks(P)* indicates that the philosopher moves from thinking to dining state after receiving the required forks. The transition *down-forks(P)* indicates that after dining, a philosopher puts down his forks and enters the thinking state (figure adapted from [Rei85])

---

of *places*, a set of *transitions*, and a set of *arcs*. The arcs are represented by arrows from places to transitions and from transitions to places known as *input* and *output* arcs respectively. The place from which an input arc originates is known as an *input place*. The destination of an output arc is called an *output place*. A Petri net is executed by *firing* one or more of its transitions, i.e., by removing tokens from its input places and creating new tokens in its output places. A transition is fireable if each input arc has a token at its input place. On receiving a token, a place may fire any one of the transitions to which it has an outgoing arc.[1] Synchronization is automatically provided for in the firing condition for transitions. The places and transitions that participate in a Petri net come from predefined sets.

A Predicate Transition net (PrT-net) is a high-level Petri net. Tokens in a PrT-net are structured objects carrying values. A set of places with identical functions is replaced by a single

[1] In actor systems, different actors with the same behavior, may accept communications concurrently. Thus, it is necessary to modify the classical one-transition-at-a-time semantics of Petri nets. Transitions in the net may be fired concurrently for all available sets of tokens.

place and a predicate. Similarly, a set of transitions is represented by a single transition and a predicate wherever possible. Incoming and outgoing arcs are annotated by *symbolic sums* – i.e., by linear combinations – of tokens with non-negative integer coefficients. Each token is represented as an n-tuples $< v_1, \ldots, v_n >$ of terms where a term is a constant or a variable, or a function of terms. The integer coefficient of an n-tuple indicates the number of identical tokens. The symbolic sum annotating an arc then indicates the number of tokens of each kind that are needed to cause a firing.

Transition firing in a PrT-net is controlled by using predicates to impose conditions on the token values of the transition's input arcs. Note that transition firing is demand driven: a transition fires when the predicates annotating the incoming places evaluate to true. A transition fires by moving the tokens from its input places to its output places. The annotation on the input and output arcs of a transition determine this transfer of tokens. Note that deadlock conditions may be avoided by ensuring that token access is atomic. Thus nets provide high-level specifications of systems which may need to be executed using more primitive constructs. By contrast, observe that actors are a lower-level model which represents point to point asynchronous communication inherent in distributed architectures.

The working of a PrT-net is best understood through an example. Consider the dining philosophers problem modeled in Fig. 1 (from [Rei85]). A collection of philosophers is sitting around a table. Each philosopher has a plate in front of him. Between any two neighboring plates lies a fork. Whenever a philosopher eats he uses both forks, the one to the right and the other to the left of his plate. When a philosopher has finished eating he replaces both his forks and starts thinking. The predicate *thinking(P)* annotates the philosophers in the thinking state. The predicate *available(F)* represents the forks that are available. The predicates *up-forks(P)* and *down-forks(P)* represent the state of a philosopher when he is waiting for his left and right fork and the state when he is ready to release his forks respectively. The annotations on the incoming arcs and outgoing arcs of the net represent the values or functions of values flowing through them. The annotation $< l(P) + r(P) >$ on the arc to the *available(F)* state, indicates that philosopher P has released his left and right forks. The annotation $< P >$ on the arc to the *up-forks(P)* indicates that a philosopher P moves from *thinking(P)* to *dining(P)* state when all the conditions are met. Let us assume there are three philosophers p_1,p_2 and p_3 who enter the system. Their forks are f_1,f_2 and f_3. Since P is a variable for a philosopher, the predicates annotating the places are *thinking*(p_1), *thinking*(p_2) and *thinking*(p_3). We also have $f_1 = l(p_1) = r(p_2)$, $f_2 = l(p_2) = r(p_3)$, and $f_3 = l(p_3) = r(p_1)$.

It should be noted that the value for the tokens in a PrT-net come from a finite set of values. This limits the instantiation of the predicates in the PrT-net. Since the predicates are an annotation for places and transitions, the number of places and transitions that participate in

a net are finite. This assumption is made to support reducibility to Petri nets which represent decidable functions. Since we are more interested in PrT-net's as a visual notation, we can weaken this assumption when we want to.

A PrT-net can be defined as a directed graph with five components: a set of places, a set of transitions, input arcs, output arcs and annotations on places, transitions and arcs. Places and transitions are annotated using predicates, while arcs are annotated by symbolic sums. In a PrT-net, a place is represented by ◯ and a transition is represented by □. Annotation of a place/transition is a mapping from a set of places to a set of predicates. In Fig. 1 *thinking(P)*, *dining(P)*, and *available(F)* are the annotations for places. The annotations for transitions are *up-forks(P)* and *down-forks(P)*. The annotations for incoming and outgoing arcs is a mapping from a set of arcs to a set of symbolic sums of tuples of terms.

An initial marking of a net assigns to each place a symbolic sum of constants. The marking is such that it can allow firing of some transition and thereby cause the flow of tokens within the system. Consider the following initial marking for the dining philosophers problem. Philosophers p_2 and p_1 are in *thinking* state, and philosopher p_3 is in the *dining* state. This will be indicated by the symbolic sums $< p_1 > + < p_2 >$ and $< p_3 >$ marking places *thinking* and *dining* respectively. After philosopher p_3 has dined, he releases his forks and any of the other philosophers can enter the dining state.

Following Genrich [Gen87], the basic form of a PrT-net, PN, is defined as (N, A, M^0), where

1. N is a directed net, $N = (S, T, F)$ where S is a set of places, T is a set of transition nodes and F is a set of arcs.

2. A is the annotation of N, $A = (A_N, A_S, A_T, A_F)$ where

   - A_N is the set of all predicates that are true.
   - A_S is a bijection between places, S, and the set of predicates.
   - A_T is a mapping of the set of transitions, T, into the set of predicates.
   - A_F is a mapping of the set of arcs F, into the set of symbolic sums of tuples of terms.

3. M^0 is a (consistent) marking of the places: it is a mapping that assigns to each place s in S a symbolic sum of tuples of constants such that if n is the index of the predicate annotating s, then M^0 is in $S^{(n)}$.

In the following section, we discuss how PrT-nets provide an abstract representation for actor programs as modeled in [Agh86]. The behavior definitions of an actor program are captured by the static places and transitions, while the dynamic parts of an actor program – namely, actors and messages are mapped to the tokens of a PrT-net. Such a mapping works provided that the two assumptions noted below hold.

- *Fixed behavioral interconnections.* Consider the interconnection topology induced on behavior definitions corresponding to the actors, i.e., two behaviors are linked if any actor using the first definition may know an actor using the second. We require that this induced topology is static. Although this assumption is somewhat restrictive, it is less so than requiring a fixed topology on actors themselves. Because a fixed topology on behavior templates only requires the predetermination of which behaviors may relate to which others – it holds whenever the conceptualization of the problem is clear.

- *No behavior creation.* The actor system modeled by a PrT-net does not cause new behavior definitions to be added to the system during its evolution. In particular, this means that *reflective* actor architectures may not be modelled. Such architectures allow the system level actors implementing an application to be dynamically accessible to the application

5 Modeling Actor programs with Predicate Transition nets

Colored Petri nets (**CPN**'s) have been used by Sami and Vidal-Naquet [SVN91] to formalize the behavior of actors. We will call their algorithm for transforming actors programs into **CPN**'s the SV-N algorithm. Internal concurrency is depicted in the SV-N algorithm by interpreting actor program in terms of its primitive actions. Our algorithm for deriving a PrT-net from an actor program is a variant of the SV-N method. In particular, we simplify the visual aspects of the pictorial representation by folding collections of atomic actions. Instead of associating separate places for address generation in each behavior, we assume a tagging scheme on message tokens which allows extension of the tags to locally generate globally unique mail addresses and new tags (as in the semantic operational model in [Agh86]). As a simplification, we simply ignore the details of an address generation scheme. We assume that the underlying actor execution system generates unique actor mail addresses which appear in the appropriate tokens. Recall that our purpose in transforming an actor program to a PrT-net is primarily to visualize its execution.

As discussed in Section 3, an actor program consists of *behavior definitions* and initializations. Within each behavior definition, the response of an actor to a communication is defined by a *method*. A behavior definition is thus composed of different methods. In turn, a method may be a combination of any of the primitive actor commands – provided that there is exactly one executable *become* command in the body of a method. The topology of behavior definitions – defined by the possibility of replacing one behavior by another – is static, although the interconnection topology of individual actors is dynamic. Note that this model of actors precludes passing behaviors as first-class objects.

A place in our PrT-net representation corresponds to an actor behavior, and a transition to a method. Corresponding to each method of a behavior definition, there is an incoming arc

from the place representing the behavior definition to the transition representing the method. Intuitively, a transition is a function which receives a message on its incoming arc, acts on it, and sends out a message on its outgoing arc. However, the formula annotating a transition may contain free variables. These variables can be bound to existential quantifiers. Thus free variables provide a mechanism for explicitly representing data encapsulation in PrT-net's. Our current implementation does use this feature of PrT-net's.

The tokens in a PrT-net represent actors and messages (we will refer to them as actor tokens and a message tokens, respectively). The value of an actor token is the address of an actor and its acquaintances. The value of a message token consists of the address of the target to which a message is to be sent and the communication itself. An actor token is created when a *new* expression is evaluated. Upon creation, an actor token is located at the place which represents its initial behavior.

A message token is created when a *send* command is executed. The value of the message token is the method name[2] which represents the name of the function to be invoked in the target actor, its parameter list and the address of the target actor. A message token is placed at the behavior definition corresponding to the target's current behavior. In the initial program, this can be statically deduced from the new expression creating the program. In an executing program, the token is sent down an arc as described below.

A transition is fired when an actor token and a (corresponding) message token arrive at a place. Note that the target mail address field in the message token must correspond to the mail address field in the actor token. Thus behavior execution in an actor program corresponds to the firing of a transition in a PrT-net. A `send` or `become` command in a method is represented by outgoing arcs from the corresponding transition to places. In case of a message send, the place corresponds to the current behavior of the target actor. A message token is simply moved from the transition to the place. In case of a become command, there is an arc from the transition which points back to the place corresponding to the behavior the actor is adopting. An actor token will be passed along this arc together with the values of its current acquaintances. This provides the necessary synchronization for executing any communications pending to the actor.

Upon executing one of its methods, if an actor changes the behavior definition it is using, the situation is somewhat complicated. It is possible to define synchronization tokens and new transitions to keep the places corresponding to different behavior definitions distinct. We take a more direct approach: the places representing all the behavior definitions an actor may use are connected to a *master place* which sends messages to the place corresponding to the current behavior. Thus, if the behavior representing a given actor is not known at compile time, the transition executing the become command sends the actor token to the master place together

[2]In actor literature, a method name is often called a communication handler.

with its acquaintances. The master place will fire a transition to forward the actor token to its current behavior. However, if the behavior definition to which the actor changes is known at compile time, a direct approach is adopted. The actor and its acquaintances are moved directly to the place that represents the current behvaior of the actor.

5.1 A Simple Example

We present an example to demonstrate the modeling of actor programs using Predicate Transition nets. We have chosen the canonical bank account example to illustrate the intuition behind the formal modeling of actors with PrT-nets. Consider a bank which employs two managers and three tellers (M_1, M_2 and T_1, T_2, T_3 respectively). A bank manager services customer requests for opening checking and savings accounts. On processing a request, the manager informs the tellers the account numbers of the newly created accounts.

A teller receives the account numbers and maintains the checking and savings accounts separately. In Figure 2, places are used to represent the behavior templates for the bank managers, tellers, checking and savings accounts. Transitions are used to represent the requests that can be processed by the manager and the teller. The two managers M_1 and M_2 are represented as tokens located at the place *bank-manager*. Similarly tokens T_1, T_2 and T_3 mark the place *bank-teller(T)*. A customer approaches manager M_1 with a request to open a bank account (*create-acc(A,*M_1*)*). Manager M_1 creates the checking and savings accounts $a1$ and $b1$ respectively. Upon receiving the tokens, the tellers fire a transition that moves these tokens to the part of the net representing the checking and savings account. Once a checking or the savings account is created, a teller may receive requests for transactions from the customers for whichever accounts have been created. In Figure 2 the teller T_1 receives a request to deposit some amount into the checking account $a1$ from a customer.

In Section 3, we provided the Rosette code for a checking account. The behavior definition for the checking account corresponds to the place `check-acc` in Figure 3, and each method in the behavior definition corresponds to one of the transitions – namely, `deposit`, `show-balance`, or `withdraw`. The behavior given below is expressed Figure 3 is a portion of a net representing this behavior. In Figure 3, depending on the amount in the savings account the transition *suff*( ), or *insuff*( ) is fired.

The PrT-nets which describe the bank account, process two types of tokens: tokens which represent actors ($a1, b1, M_1, M_2, T_1, T_2, T_3$) and tokens which represent messages (*create-acc* and *checking* requests). Message tokens represent the communications received by actors. The predicate *bank-manager* captures the behavior of all bank managers in the bank and is represented by a single place and a set of predicates. Places in a net are OR-nodes and transitions are AND-nodes. Firing a transition in the net is driven by the values of the message tokens and actor tokens. A

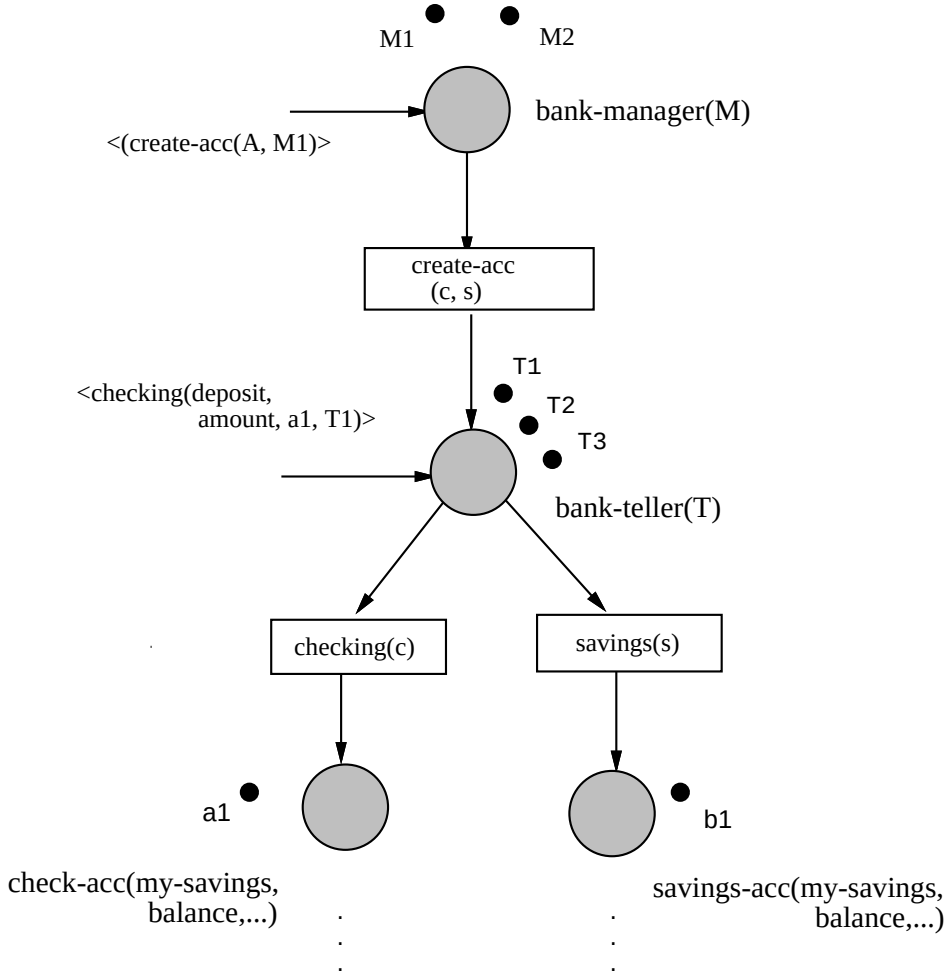

Figure 2: A PrT-net illustrating the creation of a checking and savings accounts of a customer A.

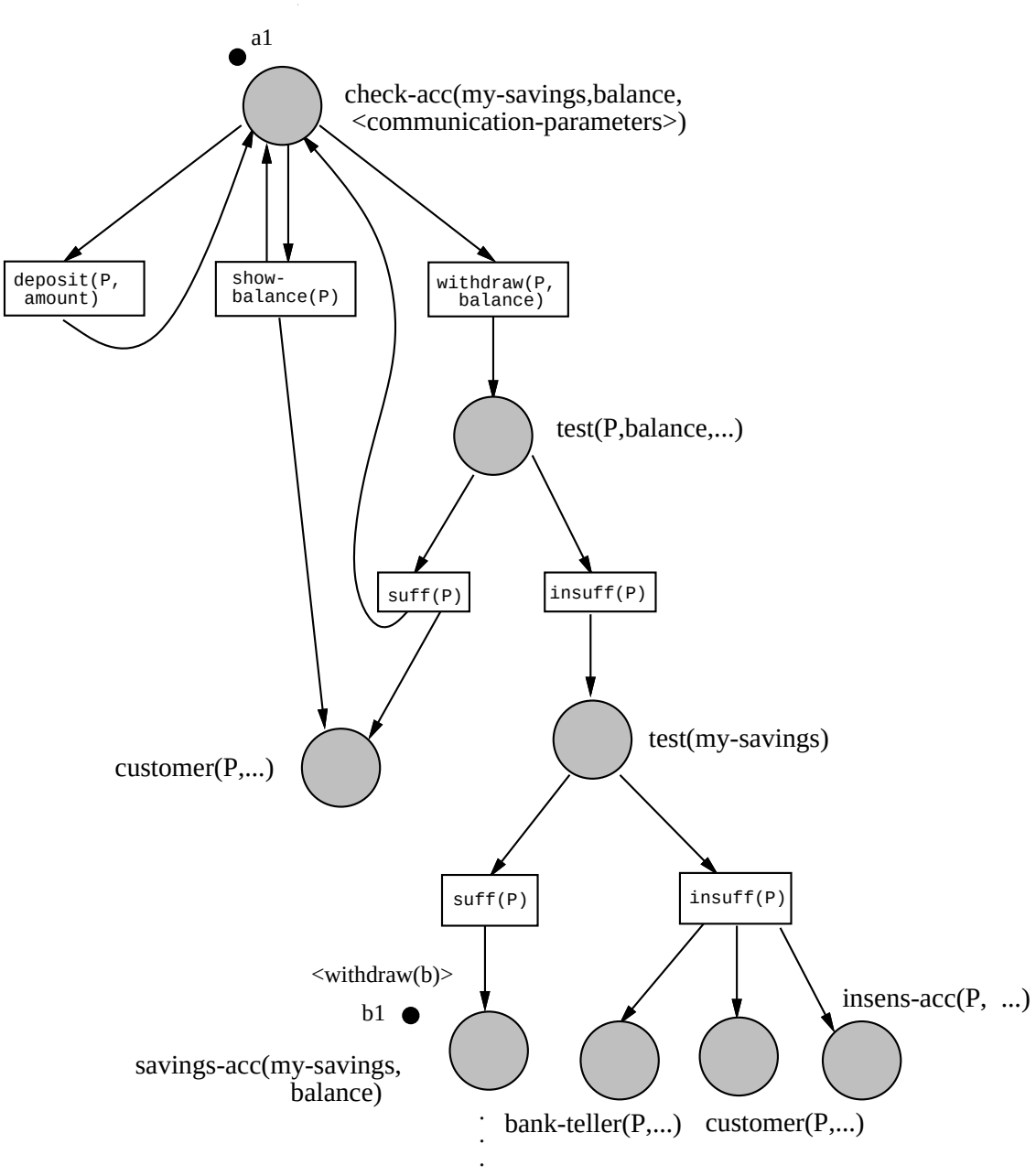

Figure 3: A PrT-net representing overdraft from a checking account. Some of the behaviors, such as that of customers and savings accounts, are not shown.

message token with a mail address equal to the mail address of an actor token causes a transition to fire. Synchronization constraints in actor programs are imposed by replacement behaviors. In a PrT-net this is depicted by moving a token from one part of the net to another, where each part of the net represents a behavior.

6 Deriving a PrT-net from an Actor Program

This section discusses the formal modeling of actors using PrT-nets. We illustrate the modeling process by a simple algorithm and then provide an algorithm for deriving a PrT-net from an actor program is given.

6.1 A Simple Recursive Program

Consider the code for a factorial program based on the actor language *Rosette* [TSS89]. Note that we have modified the syntax slightly for readability.

```
(defActor Toplevel()
    [val [k]
      (return k)])

(defActor Factorial
   (fact [n,cust]
      (if (= n 0)
         ((send cust val 1))
         ((let [[ c = new Customer(n,cust)]] )
             (send self [fact (- n 1) c])))))

(defActor Customer(n,cust1)
   (val [k]
      (send cust1 [val (* n k)])))

(define ft (new Factorial()))
(define tl (new Toplevel()))
(send ft [fact 3 tl])
```

In the above code, the `Toplevel` actor receives the result and returns it to the user. A `Factorial` actor, such as `ft` delegates the task of calculating the product of `n` and factorial of `n-1` to a new actor created using the `Customer` behavior definition and sends a message to itself to calculate the factorial of `n-1`. The new actor waits for a value to arrive which it will multiply with n and send the result to the original customer.

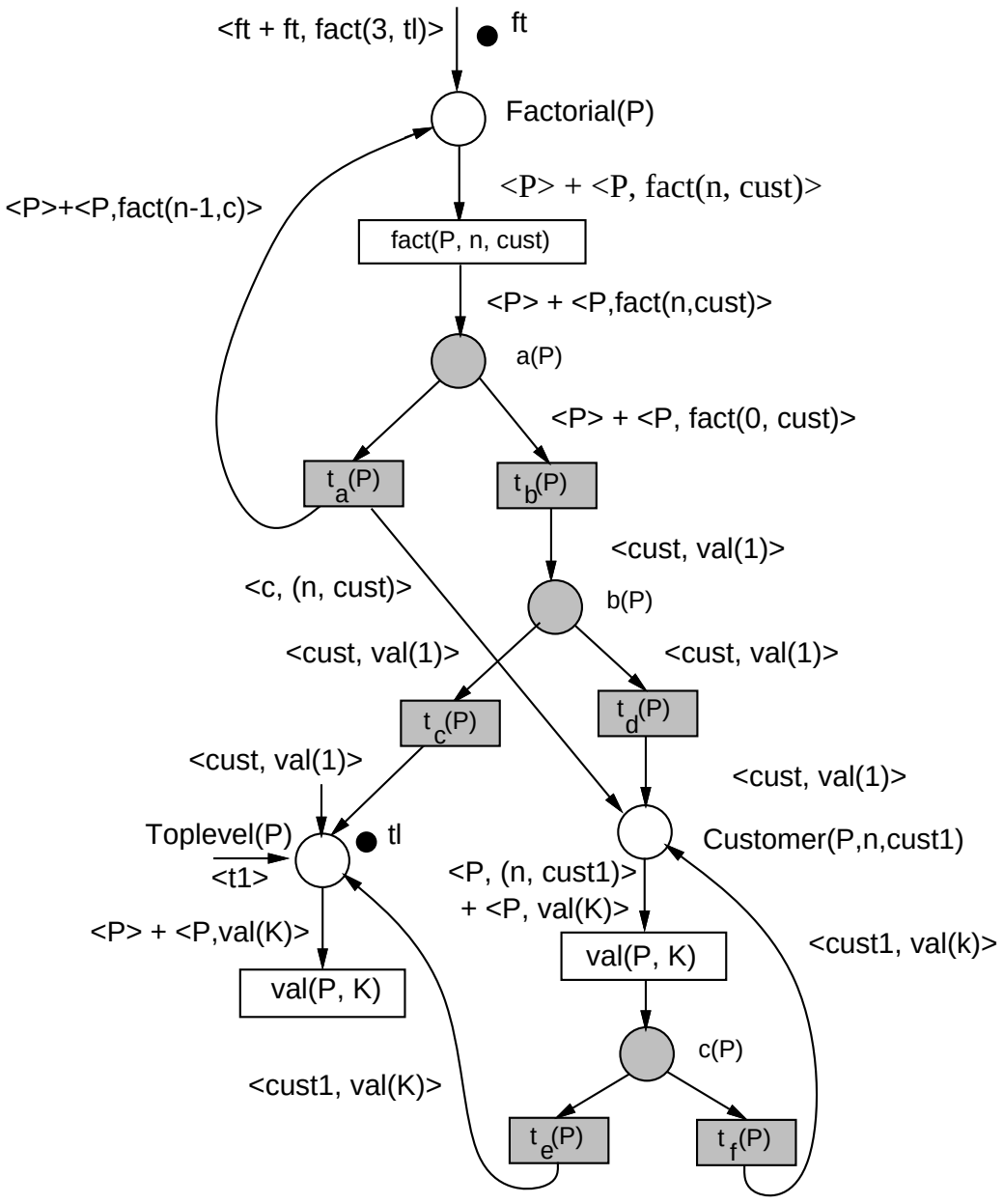

Figure 4: A PrT-net representing the factorial program. The variables used in the labelling of the arcs, places and transitions are obtained from the Rosette code for the factorial program. For example, `c` is an actor token created by the `fact` method. The conditional places and transitions are shaded in the above figure for clarity. Note that after execution of the `val` method, a *Customer* actor should be moved back to the place `Customer`. We have omitted this in the figure for simplicity.

Algorithm to transform actor programs

We first define some terms which represent the components of an Actor program. We then present an algorithm to transform an Actor program into its syntactic constituents.

- $\mathcal{B}$ is the set of behaviors defined in an actor program. Note that in Rosette behaviors are defined using `defActor`. For the factorial program given above, $\mathcal{B} = \{$ `Toplevel` , `Customer` , `Factorial` $\}$. We will represent the elements of $\mathcal{B}$ by $b, b_1, b_2, \ldots$ below.

- $\mathcal{B}^*$ is the set of behaviors augmented with the corresponding formal parameters for acquaintances of actors. Thus, $\mathcal{B}^* = \{\ (b, b_s) \mid b \in \mathcal{B}$, and b_s are the formals associated with $b\}$. For the factorial program $\mathcal{B}^* = \{$(`Toplevel,()`),(`Factorial,()`),(`Customer,(n,cust1)`)$\}$.

- $\mathcal{M}$ represents the structure of the methods. $\mathcal{M} = \{(b, m, ml) \mid m$ is a method in b and ml is the parameter list for the method$\}$. For the factorial program, $\mathcal{M} = \{$(`Toplevel,val,(k)`), (`Customer, val, (k)`), (`Factorial,fact, (n,cust)`)$\}$.

- $\mathcal{C}$ represents the structure of the set of commands an actor program. $\mathcal{C} = \{(b, m, St) \mid$ St is the set of commands associated with the method m in behavior $b\}$. For the factorial program, $\mathcal{C} = \{$(`Toplevel, val, return k`), (`Customer, val, send cust1 (val (* n k))`), (`Factorial, fact, if (= n 0) then (send cust (val 1)) ((let [[c = new Customer(n,cust)]]) (send self fact(- n 1) c))`)$\}$.

- $\mathcal{I}$ represents the initial environment. $\mathcal{I} = \langle \mathcal{I}_b, \mathcal{I}_m \rangle$, where $\mathcal{I}_b = \{(a, b, a_s) \mid a$ is an actor created with behavior b command and a_s as acquaintances$\}$, and $\mathcal{I}_m = \{(a, m, ml) \mid a$ is an actor in $\mathcal{I}_b$ to which a message with method m and ml are sent$\}$. For the factorial program, $\mathcal{I} = \langle \{$(`ft,Factorial,()`), (`tl,Toplevel,()`)$\}$,$\{$ `(ft,fact,(3,tl))` $\}\rangle$.

The algorithm for deriving a PrT-net PN, from an actor program AP uses the structural relations between actors with the same behavior definition. A PN such that, $PN \cong AP$ is defined as $PN = (N, A, M^0)$, where $N = (S, T, F)$ and $A = (A_N, A_S, A_T, A_F)$. From a given actor program AP, PN is derived in the following way:

1. For each behavior in the actor program there is a place in the PrT-net. For the factorial program, $S = \mathcal{B} \cup S_b$, where S_b is a set of conditional places which is created during the transformation process. Step 6 discusses conditional places and transitions.

2. $\forall\ b \in \mathcal{B}$, $A_S(b) = b(P, args)$ where $(b, args) \in \mathcal{B}^*$ and P is a variable which gets instantiated with the value of an actor token (i.e., its mail address). For the factorial program:

$$A_S(s) = \{Toplevel(P), Factorial(P), Customer(P, n, cust1)\}$$

3. $T = T_m \cup T_b$, where $T_m = \{\ m \mid \forall (b, m, ml) \in \mathcal{M}\}$ and T_b is the set of conditional transitions created during the transformation process. For the factorial program $T_m = \{val, val, fact\}$.

4. $A_T(t) = \{\ t(P, ml) \mid \forall t \in T_m, (b, t, ml) \in \mathcal{M}$. For the factorial program,

$$A_T(t) = \{fact(P, n, cust), val(P, k), val(P, k)\}$$

5. We differentiate between the incoming and outgoing arcs in F. An incoming arc f_i is an arc from a place to a transition, and an outgoing arc f_o is an arc from a transition to a place. For every behavior b, $\forall m \in \mathcal{M}$, there is an incoming arc from b to the transition m represents, s.t $(b, m, ml) \in \mathcal{M}$. The annotation of this arc f_i is $A_{Fi} =< P, al > + < P, m(ml) >$, where P is the variable representing the address of an actor token, and al is its acquaintance list. For the factorial program $A_{Fi} = \{(< P > + < P, fact(n, cust) >), (< P, (n, cust1) > + < P, val(K) >), (< P > + < P, val(K) >)\}$.

6. The outgoing arcs in a PrT-net are determined by the *send* and *become* commands in the method represented by a transition. A syntactic analysis of the actor program will determine the places in S to which there are outgoing arc from the transitions. Let A_{Fo} be the annotation of this outgoing arc. We determine the set A_{Fo} by a case analysis of the primitive actor commands that constitute the actor program.

   - If the script St of a transition $t \in T_m$ has a *become* command, then there is a outgoing arc (t, s), from t to s, s.t. s is an identifier that syntactically follows the *become* command, and $s \in S$. $< P, al >$ is added to the symbolic sum in A_{Fo}. If s is an expression, the behavior definition to which an actor is to be moved is determined at run time. Then there is an outgoing arc from t to the master place. Recall that all the behavior definitions an actor may use are connected to this master place via transitions.
   - If there is no *become* command in the script St of a transition $t \in T$, then there is an outgoing arc (t, s), where s is a place in all the incoming arcs (s, t). The symbolic sum $< P, al >$ is then added to the annotation of the outgoing arc.
   - If the script St of a transition $t \in T_m$ has a *send* command, then there is a outgoing arc from t to s, s.t $s \in S$ and s is the behavior for the actor identifier s_a that syntactically follows the *send* command. Thus if `send` s_a m ml is the command, then the symbolic sum added to the annotation of the outgoing arc will be $< s_a, m(ml) >$.
   - A *cond* and an *if* command with *send* commands in the body is represented by a single conditional place and as many conditional transitions as there are tests in the conditional command. When a condition is satisfied the corresponding conditional

transition is fired. For example in Figure 4 place a and transitions t_a and t_b have been added after the transition *fact*. The execution of the script of the method `fact` is delayed. The transitions t_a and t_b represent the evaluation of the *if* condition. One of the transition is then enabled and fired.

- If the script St of a transition $t \in T_m$ has a *new* command, then there is a outgoing arc from t to s , s.t, $s \in S \wedge$ s is the behavior for the actor identifier s_a that is created by the *new* command. Thus the token corresponding to actor s_a is created at transition t and sent to the place s. Thus the mail address of s_a is available at s.
- In an actor program the address of the target actor of a communication is often determined at run time. To represent this communication in the PrT-net a semantic analysis of the program is required. All the potential behaviors that may execute as a result of this communication are determined. Conditional Transitions are introduced to show the potential communications.

  For example, in the code of the factorial program, the command `send cust val 1` in the script of the method `fact`, does not reveal the identity of the actor `cust`. But we know that since it is a *Toplevel* actor which first sent the message to calculate the factorial, it will at some point receive a message either from *Customer* or from *Factorial* depending on the value of `n`. Therefore there are arcs from *fact* to both *Customer* and *Toplevel* through the conditional places and transitions in Figure 4.

  Each conditional place is an element of the set S_b and each conditional transition is an element of the set T_b. For each $s \in S_b$, $A(s) = s(P)$, and for each $t \in T_b$, $A(t) = t(P)$. The annotation of the arc (t, s_b) where $t \in T$ and $s_b \in S_b$ is equivalent to the annotation of the incoming arc to t. The annotation of the arcs (s_b, t_b) where $t_b \in T_b$, is equivalent to the annotation of the arc to the place s_b. The annotation of the arcs (t_b, s) where $s \in S$, is determined by the `send` commands in the script of the method as indicated above.

7. The initial marking of the PrT-net is determined from the initial environment $\mathcal{I}$ of the actor program. For each $(a, s, a_s) \in \mathcal{I}_b$, there is an actor token $\langle a(a_s) \rangle$ in place s. For each $(a_s, m, ml) \in \mathcal{I}_m$ there is a message token $< a, m(ml) >$ in place b, where $(a, b, a_s) \in \mathcal{I}_b$.

A better understanding of how this algorithm helps in visualizing the execution of an actor program using PrT-nets can be obtained by playing the token game for the factorial example discussed above. Figure 5 depicts the important execution steps in the token game for the factorial example which calculates the factorial of 3. Step 1 shows the execution of the method `fact`. This is depicted by the firing of the transition *fact*, t_a and creation of a new actor token *c3* with acquaintances 3 and *tl* and sending the message token to calculate factorial of 2 to ft.

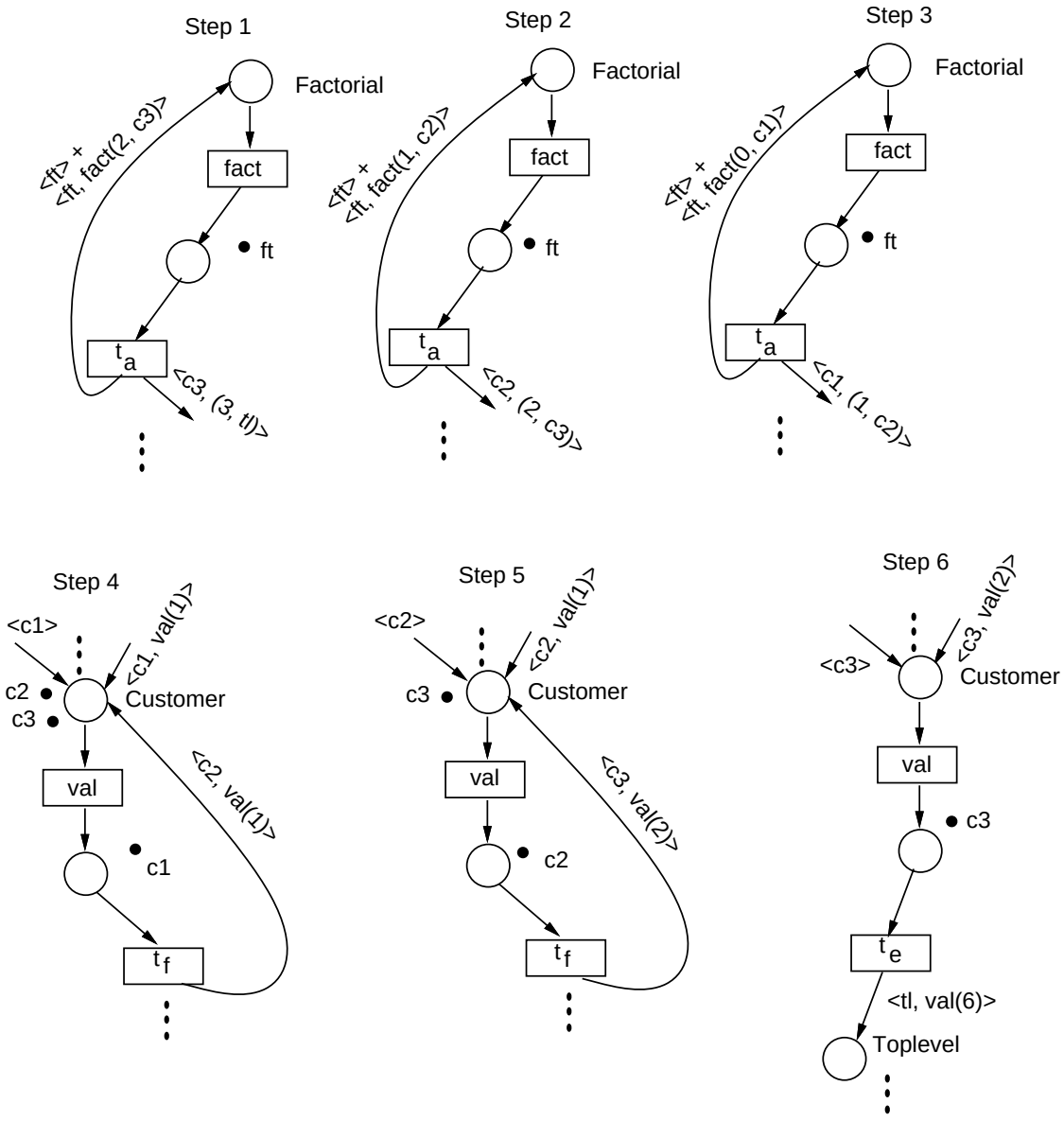

Figure 5: The steps involved in a token game for the factorial example. Only necessary places and transitions are shown in the above steps.

The actor token ft is moved back to the *Factorial* place as its behavior definition is not changed. Step 2 shows the execution of the `fact` method by the actor token ft to calculate factorial of 2. This time the actor token $c2$ is created after firing transition t_a. The actor token ft and a message token to calculate factorial of 1 are moved to *Factorial*. Eventually when the ft actor token receives a message to evaluate factorial of 0, it fires the transition t_b and causes the message token $< c1, val(1) >$ to be sent to the place *Customer*. Step 4 shows the execution of the `val` method by the actor token $c1$. $c1$ has a value of 1, and receives a value 1, in the argument of the message. It evaluates the product and sends the result in a message token to $c2$ ($< c2, val(1) >$). Step 5 shows the execution of the *val* method by the actor token $c2$. Finally in Step 6, the actor token $c3$ evaluates the product of the value 2 received in the message to it, and the value 3 in its acquaintance, and sends the result to *tl* by firing transition t_e.

An important feature of the actor system is its reconfigurability. During the execution of an actor program it is possible to create a new behavior definition as a result of processing a communication. In a PrT-net this would correspond to creating new places and transitions. It is not possible to represent an actor program with this capability through a single PrT-net. However, if properly implemented a sequence of PrT-nets can be used to depict or visualize the execution of such actor programs.

Note that variations on the basic algorithm described are possible. In particular, actor tokens are superfluous for behaviors which do not change over time and do not admit formal parameters for acquaintances; in this case, a place in a PrT-net can represent an actor. In particular, the arrival of a message token would then be sufficient to fire a transition. For example, in the above factorial program, *Factorial* place has no possible acquaintances; a factorial actor could replace the predicate *Factorial*. This results in a simplified PrT-net for the factorial program. On the other hand, requiring explicit creation of actor tokens for synchronization provides a mechanism for controlling the exact number of actors – in this case factorials – which may be active at a time. In either case, observe that actor tokens are still required to depict the customers (continuations).

An important issue in actor based systems is the problem of synchronization constraints, i.e., conditions limiting the communications an actor in a given state is able to process[Agh90a]. For example, a bounded buffer which is empty cannot service requests to dequeue. Different approaches to selectively process external communications may be taken (see [Agh90b]). One solution used by actor systems is to let an actor explicitly buffer incoming communications which it is not ready to process (insensitive actors). In the Rosette language, Tomlinson and Singh [TSS89] proposed a mechanism which associates to each potential state of actor an *enabled set* specifying the particular methods the actor is willing to invoke. The actor then processes the first message in its queue which invokes a method in its current enabled set. Synchronization constraints specified using enabled sets may be incorporated in our PrT-net representation by

simply adding another type of token, called *enabledness token* which controls the processing of incoming communications as follows. We require that a transition fire only when the following conditions are satisfied:

- an actor token, a message token, and an enabled token are all available at a place from which there is an incoming arcs to the transition;
- the address of an actor in the message token is equal to the address specified in the actor token; and,
- the value of the enabled token is equal to the method name specified in the message token.

7 Comparison with the SV-N algorithm

We will refer to the SV-N algorithm as the CPN approach and our transformation method as PrT-net approach. The current trend in the field of net-theory is to translate an analysis method developed for one kind of net to another. When PrT-nets were first introduced by Genrich and Lautenbach [GL81], generalizing place invariants and transition invariants in the net seemed difficult. The invariants contain free variables and to interpret these invariants, it is necessary to bind these variables by a complex set of substitution rules. This made the interpretation of invariants difficult. To solve this problem a first version of **CPN** was introduced by Jensen [Jen81]. The main ideas are similar to PrT-nets, but the relation between an occurrence element and token colors involved in the occurrence is defined by functions instead of expressions. However, the functions attached to arcs in a **CPN** are more difficult to read and understand than the expressions attached to arcs in PrT-nets. This led to an improved model of CPN called a high-level net [Jen83].

The modeling of actors by **CPN** and PrT-net can be divided into two stages. The first consists of deriving some information from the given actor program to be used in modeling. This stage is identical in both the approaches. The second stage consists of transforming the actor program to a net. The similarities and differences in this transformation process will be illustrated here. In both the approaches, actors and messages are represented by tokens, and behaviors are represented by the graph of the net. The use of tokens in a PrT-net and colors in **CPN** do not correspond to the definitions in [Gen87] and [Jen90] respectively. Because the address and state of the actors to be generated are not known at compile time, the set of tokens in a PrT-net and, correspondingly, colors in a **CPN**, are not fixed at the beginning of execution. Furthermore, the creation of new behaviors that are not in the given actor program is not allowed. The differences between the two approaches are:

- In contrast to the CPN approach, commands associated with a message are not represented by different transitions in the PrT-net approach.
- The generation of addresses is not handled directly in the derived PrT-net. This eliminates the need for an additional place like the address-generator.
- A variation of the SV-N algorithm models a version of Actors known as SAL, that is described in Chapter 3 of [Agh86]. Transition of actor tokens when a `become` command is executed, is handled by an additional place called 'aficionados.' The execution of `become B2` by an actor is interpreted by putting the associated actor token in the aficionado place. The name of `B2` is included as the first component of the structure of the actor token. The presence of this token in the aficionado place enables the firing of a transition. The aficionado place then puts the actor token in the place associated with `B2`. The PrT-net approach, models a more complete version of actors based on the Rosette actor-language. The execution of the *become* command is achieved in two ways. Corresponding to the aficionado place, is the master place. When the value of `B2` is known at compile time (not an expression), the routing through the master place is bypassed. The actor token is directly moved to the place that represents the behavior of `B2`. However, if the value of `B2` is not known at compile time, the actor token is moved to the master place and then to the corresponding place representing the behavior of `B2`.
- In the CPN approach the role of a communication place is much like that of a mailbox for each actor in an actor system. A communication place forwards the message tokens to the corresponding behavior causing a transition to fire. In the PrT-net approach, the task of choosing a transition to fire is performed by a place and the conditional place associated with it.
- In the PrT-net approach, a change in the behavior of a given actor, causes both its associated actor token and message tokens to be moved to the place that represents the new behavior. However, in the CPN approach all message tokens are contained in the communication place.

Although PrT-nets and CPN's are considered two different dialects of the same language [Jen90], we think that PrT-nets provides a than CPN's to model actors. Because our primary goal is to visualize actor programs rather than to apply analysis methods of Petri nets, we chose to use PrT-nets.

8 A Prototype Implementation

We discuss a prototype implementation of a tool which uses PrT-nets to specify, execute and visualize actor programs. The visual tool has two components: a visual editor and an actor system.

The visual editor is used to specify and visualize actor execution. The visual editor is based on InterViews; a toolkit built using the C++ language and X-window system protocols. The visual editor provides icons that help in building the visual representation of the actor program. The actor system used by the tool is the Rosette system. To visualize the actor execution the Rosette system needs to communicate to the visual editor the state of actor configurations. Similarly, the visual editor communicates with Rosette to specify the actor behaviors and actor commands that need to be executed. We use the UNIX system calls to carry out this communication. Currently the tool runs on SUN3 workstations.

8.1 The Visual Notation

Our visual language is a Predicate Transition net which consists of a place, a transition, and an arc connecting a place to a transition. The definitions of place, transition and arc are given in the earlier section.

- a place is represented by a $\bigcirc$ and a predicate annotating it. In Figure 4, *Factorial(P)* is an annotation of a place representing the *Factorial* behavior.
- a transition is represented by a $\square$ and an annotation. In Figure 4, *fact(P,n,cust)* is the annotation for a transition representing a method of the *Factorial* behavior.
- an arc from the place annotated by *Factorial(P)* to the transition annotated by *fact(P,n,cust)* is represented by the symbolic sum $< P > + < P, fact(n, cust) >$ in Figure 4.
- an actor token is depicted by a circle, located at the place that represents its behavior. Figure 4 shows an actor token ft, whose behavior is represented by the place *Factorial(P)*.
- a message token is not explicitly shown. Messages flowing through the system are visible only when the *ShowMessages* menu bar is selected. A texteditor displays a list of messages starting from the most recent communications in the system. The information displayed gives the source behavior predicate and the destination behavior predicate of the communication. This indicates that a communication to an actor belonging to the destination behavior class, has been created as a result of executing the source behavior. Currently we do not display the message parameter list and the actual mail addresses of the actors participating in the communication.

8.2 Features of the Visual tool

A given actor program is transformed to a PrT-net using the transformation rules given in the previous section. The user constructs the derived PrT-net with the help of the icons provided

in the visual editor. Commands are sent to the actor system by clicking on a menu bar. The execution of the program specified is visualized with the help of the constructed PrT-net, actor tokens and message tokens. Some of the important features of the visual tool are listed below.

- *communication from Rosette system to the visual editor:* The Rosette system notifies to the visual editor the new actors and communications it creates. If an actor is created, the behavior to which the actor token belongs, is indicated to the visual editor. When a message is created, it informs the editor the behavior identifiers of the source and destination of the message. The visual editor communicates to the Rosette system the behavior definitions and actor commands that the system has to execute.

- *create behavior:* is an icon in the visual editor which enables the user to draw a place, mark its annotation and also specify a behavior for it. A dialogbox is popped up to provide the annotation for the place. Following that a text editor is provided to type in the script for the behavior of an actor represented by the new place. The script provided should be in Rosette.

- *create place:* is an icon identical to the *create behavior* icon, except that it does not pop up a text editor to specify the script of the behavior. The script of all the behaviors are written using an external text editor into one single file.

- *create transition:* is an icon that allows the user to create a transition in the PrT-net and provide its annotations.

- *arcs:* An arc of a PrT-net is drawn by clicking on a line icon and choosing the brush menu to be an arc.

- *annotations of arcs:* Arc annotations can be written using a text icon that allows the user to annotate an arc in the form of text.

- *send:* is a menu which is used to send Rosette system commands from the visual editor.

- *show messages:* is a menu bar which when chosen provides the user with a list of all current messages. With the most recent messages as the first element of the list.

A sample session of the visual tool is given in Figure 6. We are working on an extension of the implementation to provide the state information of actors by clicking on the actor tokens.

An important point to note here is that in the visualization of an actor program, the order in which the transitions are fired should be identical to the order of execution of the methods in the underlying actor system. The implementation should take care to preserve this required order with the help of a manager or enabled sets which will ensure the execution of a next method only after visualization of the previous method is realized.

Figure 6: A snapshot of the visual editor used in the prototype implementation.

9 Conclusion and Research Directions

The approach we have outlined has several weaknesses. First, the behavior of an actor is encoded in a place of a PrT-net. However, in unrestricted actor systems, an actor may compute a replacement behavior as a result of processing a communication. In our current transformation, to derive a PrT-net for such an actor, we would have to identify the new behavior and move the actor token to the place that encodes this new behavior. This implies that we may not dynamically create behavior definitions. One approach to overcoming this difficulty is dynamically recomputing the predicate net – essentially, this is equivalent to recompilation. We do not have an algorithm for incremental compilation; such an algorithm would appear to require some restrictions on the otherwise arbitrary modifications which may be made on the arcs.

Visualization of actor programs with the help of PrT-nets can be used in debugging. Here a provision for real-time intervention is necessary. One way of doing this is to slow down the execution of the system with the help of a synchronous message passing protocol. A better alternative for a distributed implementation is to use a time-warp mechanism with roll-back[Jef85], where the official global clock corresponds to the display of the tokens.

Specifying and visualizing large concurrent systems with Predicate Transitions nets is cumbersome and thus error prone. A mechanism to add further layers of abstraction is to break the net into smaller subnets. Jensen [Jen90] describes hierarchical nets to model large systems. A hierarchical net is composed of smaller divisions called *pages*. Each page is a net in itself and can be used to represent a subsystem. The concept of hierarchical nets could be extended to visualize execution of large actor programs. Pages of interest can be expanded when necessary.

A high-level pictorial specification language needs to be developed to provide the data specification for systems modeled using PrT-nets. Currently, the data specifications for concurrent systems is only textual (i.e., Rosette code). Such textual notations may be replaced by pictures. However, Predicate Transition nets are not suitable for this purpose.

The actor model for concurrent computation allows us to program systems without any specific assumptions about the underlying concurrent hardware. With the use of tools that allow actor programs to be executed on any parallel or distributed computers, the visual model mentioned here can be used for visualization of programs executed on different parallel and distributed systems. Our studies in this are still at a preliminary stage. By relating Actors, which provide both a rich set of primitives and abstraction building tools, and Predicate Nets, which provide a visual syntax with a relatively well-explored formal semantics, we hope to understand the structure and dynamics of concurrent systems.

References

[Agh86] G. A. Agha. *A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass, 1986.

[Agh89] G. A. Agha. Supporting multiparadigm programming on actor architectures. In *Proceedings of Parallel Architectures and Languages Europe, Vol. II: Parallel Languages (PARLE '89)*, pages 1–19. Espirit, Springer-Verlag, 1989. LNCS 366.

[Agh90a] G. Agha. The structure and semantics of actor languages. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 1–59. Springer-Verlag, 1990. LNCS 489.

[Agh90b] G. A. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[AS88] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–23, August 1988.

[BLP88] Tommaso Bolognesi, Diego Latella, and Annamaria Pisano. Towards a graphic syntax for lotos. *Research into Networks and Distributed Applications*, pages 973–987, 1988.

[BVN91] F. Baude and G. Vidal-Naquet. Actors as a parallel programming model. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science*, 1991. LNCS 480.

[Cli81] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[Dal86] W. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Press, 1986.

[ELR90] J. Engelfrient, G. Leih, and G. Rozenberg. Parallel object-based systems and petri nets. Technical report, Leiden university, Netherlands, February 1990.

[Gen87] H.J. Genrich. Predicate/transition nets. *Lecture Notes in Computer Science*, (254):207–247, 1987.

[GL81] H.J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[Goe90] S. K. Goering. A graph grammar approach to concurrent programming. Technical Report UIUCDCS-R-90-1576, University of Illinois at Urbana-Champaign, May 1990.

[Hew77] C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[Jef85] D.R. Jefferson. Virtual time. *ACM TOPLAS*, 7(3):404–425, 1985.

[Jen81] K. Jensen. Colored petri nets and the invariant method. *TCS*, 14:317–336, 1981.

[Jen83] K. Jensen. High-level petri nets. *LNCS*, (66):166–180, 1983.

[Jen90] K. Jensen. Colored petri nets: A high level language for system design and analysis. *Lecture Notes in Computer Science: Advances in Petri Nets*, 1990.

[JR89] D. Janssens and G. Rozenberg. Actor grammars. *Mathematical Systems Theory*, (22):75–107, 1989.

[KGC89] S. M. Kaplan, S. K. Goering, and R. H. Campbell. Specifying concurrent systems with $\triangle$ grammars. pages 20–27. The fifth International Workshop on Software Specification and Design, April 1989.

[Man87] C. Manning. Traveler: the actor observatory. *Proceedings of European Conference on Object-Oriented Programming. Also appeared in LNCS*, (276), January 1987.

[Mes90] J. Meseguer. A logical theory of concurrent objects. In *OOPSLA/ECOOP Proceedings*, New York, 1990. ACM Press.

[RC89] G. Roman and K. C. Cox. A declarative approach to visualizing concurrent computations. *IEEE COMPUTER*, pages 25–36, October 1989.

[Rei85] W. Reisig. *Petri Nets*. Springer-Verlag, 1985.

[SF90] David Stotts and Richard Furuta. Temporal hyperprogramming. *Journal of Visual languages and Computing*, (1):237–253, 1990.

[SVN91] Y. Sami and G. Vidal-Naquet. Formalization of the behavior of actors by colored petri nets and some applications. *PARLE '91*, 1991.

[TSS89] C. Tomlinson, M. Scheevel, and V. Singh. Report on Rosette 1.0. Technical Report ACT-OODS-449-89(Q), Object-Based Concurrent Systems Project, MCC, December 1989.

[Vau87] J. Vautherin. Parallel systems specifications with coloured petri nets and algebraic abstract data types. *Advances in Petri Nets, Springer Verlag*, pages 291–308, 1987.

[Yon90] A. Yonezawa. *ABCL An Object-Oriented Concurrent System*. The MIT Press, Cambridge, Massachusetts, 1990.