Diss. ETH No. 13758

SYNTAX AND SEMANTICS OF GRAPHS

An approach to the specification of visual notations for discrete-event systems

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY

ETH ZURICH

for the degree of

DOCTOR OF TECHNICAL SCIENCES

presented by

JÖRN WILHELM JANNECK

born 11 October 1966

citizen of Federal Republic of Germany

accepted on the recommendation of
PROF. DR. LOTHAR THIELE, examiner
PROF. DR. EDWARD A. LEE, co-examiner

2000

Contents

Abstract

Visual notations are a powerful medium for expressing algorithmic structures. Their two-dimensionality is particularly suited to support the definition of concurrent systems, which are characterized by a notion of state, transitions between states, and communication between concurrently executing parts of the system. This kind of system is usually referred to as a *discrete-event system*.

There exists a large variety of visual notations for specifying such systems, visualizing different aspects of a system and thus being suited to different kinds of systems—from purely state-oriented notations to purely dataflow-based languages, with languages like Petri nets combining both aspects. Many complex systems consist of *heterogeneous* components which are best described in different visual notations.

This work defines a computational framework for describing discrete-event systems which consist of a number of communicating components. The communication structure, as well as the set of these components, is allowed to change dynamically during the execution of the model. The framework supports an abstract notion of *time*, which is used to schedule concurrent activities in a model.

This framework is then used as the basis of a generic approach to the definition of the semantics of graph-like visual languages. This approach is operational, defining the behavior of a visual description, a *picture*, by an *interpreter* for an abstract structure corresponding to the pictures of a visual language.

As a next step, a (textual) language is defined in which these interpreters may be formulated in an abstract but fully executable manner. This language is based on *Abstract State Machines*, borrowing from them the concept of state and state transition, and extending them with a notion of *component* and communication between components.

Finally, these techniques are then applied to a series of increasingly complex visual notations, exemplifying and illustrating some of the issues in defining realistic visual languages for discrete-event systems.

Kurzfassung

Visuelle Notationen sind ein mächtiges Medium zur Beschreibung algorithmischer Strukturen. Ihre Zweidimensionalität eignet sich besonders für die Definition nebenläufiger Systeme, die durch Begriffe wie Zustand, Zustandsübergang, und Kommunikation zwischen nebenläufigen Teilen des Systems charakterisiert sind. Solche Systeme werden üblicherweise als *diskrete Ereignissysteme* bezeichnet.

Es existiert eine grosse Bandbreite von visuellen Notationen für die Spezifikation solcher Systeme, von denen jede verschiedene Aspekte eines Systems visualisiert, was sie für unterschiedliche Arten von Systemen besonders geeignet macht—von rein zustandsorientierten Notationen hin zu reinen Datenflusssprachen, mit Sprachen wie etwa Petrinetzen als Kombination dieser Aspekte. Viele komplexe Systeme bestehen aus *heterogenen* Komponenten, die am Besten in verschiedenen visuellen Notationen beschrieben werden.

Diese Arbeit definiert ein Berechnungsmodell für die Beschreibung diskreter Ereignissysteme, die aus einer Reihe kommunizierender Komponenten bestehen. Hierbei kann sich die Kommunikationsstruktur, und auch die Menge dieser Komponenten selbst, während des Modellablaufs ändern. Das Berechnungsmodell unterstützt eine abstraktes Zeitkonzept, das der Ablaufplanung der Aktivitäten innerhalb eines Modells dient.

Auf der Basis dieses Berechnungsmodells wird dann ein allgemeiner Ansatz zur Semantikdefinition graphenartiger visueller Sprachen vorgestellt. Dieser Ansatz ist operational, er definiert das Verhalten einer visuellen Beschreibung, eines *Bildes*, durch einen *Interpreter* für eine abstrakte Struktur die die Bilder einer visuellen Sprache repräsentiert.

In einem nächsten Schritt wird dann eine (textuelle) Sprache definiert, in der diese Interpreter aufgeschrieben werden können, in einer Form die abstrakt und dennoch ausführbar ist. Diese Sprache basiert auf *Abstrakten Zustandmaschinen*, von denen sie die Konzepte für Zustand und Zustandsübergang übernimmt, und die sie um einen Komponentenbegriff und um ein Konzept der Kommunikation zwischen Komponenten erweitert.

Schliesslich werden diese Techniken dann auf eine Reihe komplexer werdender visueller Notationen angewendet, an denen beispielhaft einige Problemstellungen bei der Definition realistischer visueller Sprachen für diskrete Ereignissysteme illustriert werden.

Acknowledgments

First of all, I would like to thank Prof. Lothar Thiele—for providing the environment for this research, his patience to give me the freedom and time to find a topic I would like to work on, for his invaluable academic input and advice, and for his personal support. I would also like to thank Prof. Edward A. Lee, for his encouragement and his constructive criticism, his spontaneous readiness to be my co-advisor and the efforts he made to keep the deadlines resulting from a rather short-termed planning, and quite simply for providing a major part of the scientific foundations this work draws on.

I am also greatly indebted to Prof. Egon Börger who found the time to discuss much of this work, and who provided many insights into extensions of ASM, practical ASM modeling, and ASM semantics.

Special thanks go to Dr. Robert Esser. First of all, it was his work that got me involved in the field. Then, besides being just a great friend, he is one of the finest software engineers I know, and working with him in the Moses project was an experience I would not want to miss. Finally, he spent endless hours meticulously proofreading this text, resulting in literally hundreds of suggestions for improvement. Thanks, Rob.

First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems.

[...]

Underlying our approach to this subject is our conviction that ”computer science” is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is what might best be called *procedural epistemology*—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of ”what is.” Computation provides a framwork for dealing precisely with notions of ”how to.”

Harold Abelson, Gerald Jay Sussman
Structure and Interpretation of Computer Programs [2]

1

Introduction

Languages are structures used to communicate. They are usually described by a set of rules, which the sender of a message uses to render that message to the medium of communication, and which the receiver uses to reconstruct the original content of the message. Of course in general, and in especially with natural languages, the language might not be fully defined by any given set of rules, the media might in principle be inadequate to represent certain aspects of an intended message, and message encoding and decoding might therefore be incomplete and ambiguous.[1]

All branches of engineering, and computer science in particular, make extensive use of languages other than 'natural' languages. They use them to communicate, document, or represent the artifacts and objects of the engineering process, models of aspects of the world relevant to the engineering problem at hand, data and collections of facts (statistical and otherwise) pertaining to some problem, and sometimes even the engineering process itself. There exists an abundance of notations for these very different purposes—e.g. programming languages, knitting patterns, architectural blueprints, modeling notations for traffic networks, circuit diagram notation for representing electronic circuits.

In many areas of computer science, the communication, documentation, and representation of *computation* (algorithms, computational procedures, or programs) is of major importance. In this work, the term *(computational) system* is used rather broadly being applicable to anything which can be described algorithmically. In practice, these may be proper programs, but could also be descriptions (or *models*) of (aspects of) the behavior of real-world systems, such as electronic circuits, mechanical systems or traffic networks. Hence we will not distinguish between programming languages and (discrete-event) simula-

[1] cf. [36]

tion modeling notations, and in fact we will present examples of both.

Naturally, 'traditional' programming and simulation modeling are fundamentally different application areas and consequently have different engineering problems associated with either discipline.[2] In treating them as special cases of a more general concept we simply acknowledge their common semantic properties as notations for describing the behavior of discrete systems, irrespective of their concrete application contexts.

The notations typically used for these purposes roughly fit into two classes, viz. *textual* languages and *visual* languages.[3] Textual languages are those which define the structure of finite strings of characters, i.e. essentially one-dimensional entities. By far most programming languages are of this kind, and there is a vast amount of knowledge concerning all aspects of handling these languages. This includes how to analyze strings and reconstruct their *abstract syntax tree* (essentially the logical structure contained in them), what classes of syntactical rules exist, and how different types of syntactical rule affect the efficiency with which a string can be analyzed etc.

1.1 Why visual languages

Despite their popularity in computer science, textual languages often lead to inadequate representations of a computational system. Their inherent syntactical linearity often suggests a sequentiality in execution that might not be present. Furthermore, for the same reason they are sometimes ill-suited to represent complex structure, such as dependency or dataflow relations between parts of a larger system. By contrast, visual languages make use of the two-dimensionality of a flat surface (as well as a host of other visual cues) to represent just this kind of information—relations between entities are visualized, e.g., by lines or arrows between them, by geometric inclusion/overlapping of entities, or by their relative placement, size, or coloring.

It has been suggested that the structure of human cognitive processes make visual languages a better medium for expressing many types of content [17, 99]. A discussion of this is beyond the scope of the present work, but at least it seems plausible that visual languages are superior at least in cases where the system to be modeled is not of a linear structure such, such as real-world material flows, a traffic networks, etc. The examples we will encounter in this work seem to suggest that in fact their applicability extends far beyond this kind of straightforward structural resemblance.

The visual languages that we will be concerned with here and textual languages do not differ in *what* can be expressed in them *in principle*—it is usually

[2]See for instance [2] on programming and [46, 113] on simulation modeling.

[3]See [34] for a comprehensive classification scheme for visual (programming) languages and its relation to the ACM Computing Reviews system usually employed for classification of research in textual programming languages.

not particularly difficult to devise a scheme for transforming one into the other without loss of relevant information. This work, dealing with visual languages, will therefore not be concerned with the equivalence of both representations, but rather with their adequacy. Starting from the assumption that visual languages are often an adequate representation of some computational entity, we will present how the specification of their syntactical structure and also their *semantics* can be given directly in terms of the elements of the visual notation and the structures they represent.

Of course, in practice the distinction between textual languages as one-dimensional and visual languages as two-dimensional is not that strict—e.g. strings of textual languages are *laid out* on a page to represent some degree of structure (such as indentation in programs indicating forms of lexical containment), while many visual languages make use of textual languages to annotate the graphical entities. In the following we will largely gloss over the first point, especially because most often layout does not change the semantics of the textual expression, and often leads to very cumbersome language definitions where it does. However, integrating textual languages into visual ones plays an important part of our approach to visual language syntax.

1.2 Diversity of (visual) languages

Languages describing computational systems (in the sense outlined above),[4] be they textual or visual, need both their syntax and semantics to be precisely defined. The most obvious reason for this is that often the descriptions are to be processed, simulated, executed, or run by a machine, which requires an executable definition of their meaning.

Furthermore, the systems described in such languages might be subject to a formal mathematical analysis, which not only requires a formal definition of their meaning, but one that lends itself (and thus the descriptions it defines) to such an analysis. Also, a concise formal definition often helps to avoid ambiguities in the language itself, and thus also tends to improve its utility for communicating system descriptions between people. If it is executable, it may be used to directly generate implementations in some language, which serve at least as prototypes that further refinements may be validated against or even formally derived from.

There are very many different kinds of computational systems with very different characteristics. They may be sequential or highly parallel, their control flow may be fixed or dependent on the flow of information/data through the system, they may be deterministic or non-deterministic, synchronous or asynchronous and so on. Different languages provide varying support for the explicit

[4]Since this work will almost exclusively be about this kind of language, we will henceforth omit the qualification.

representation of the idiosyncrasies of a given system. Dataflow languages (e.g. Kahn process networks [71] or Synchronous Data Flow diagrams [39]) focus on the flow of information through the system, state-oriented languages (e.g. StateCharts [58]) allow for a better (more explicit) description of the control aspects of a system, while more architecturally oriented languages (e.g. the class diagrams of UML [93]) provide especially good support for the high-level compositional aspects of the system. Clearly, when describing a system, one would like to be able to choose the language that seems most appropriate to the task at hand.

Languages may also be geared towards specific application domains. The rationale behind this is that of *domain specific languages* [9, 33, 107]. Here the additional effort required to acquire sufficient fluency in a new description technique is outweighed by the additional support (as compared to more general-purpose languages) it provides for the most common description/modeling tasks in that domain.[5] This means that a certain amount of *domain expertise* went into the design of the language and is thus made more readily available for reuse than would otherwise be possible.[6]

These two points already suggest that for a wide variety of possible modeling tasks a set of different languages be available. To make matters worse, complex systems are often *heterogeneous* in the sense that they are most usefully described as consisting of interacting subsystems with very different properties. For instance, a material flow system may contain a subsystem controlling and supervising its operation. In this case, the language in which the material flow is best described is very likely to differ from the best choice for a language to model its control. Ideally, one would like to formulate the two systems each with the most adequate description technique, and then compose the partial models to obtain the complete system model. This would not only enhance the overall understandability of the system, it might also aid its engineering process—if these different parts of the system are designed by different teams of domain experts, these will most likely prefer to use the modeling notation they are most fluent in.

The consequence of this is that not only do we need a variety of languages to adequately deal with the various kinds of systems, it must also be our aim to make them interoperable, i.e. to allow for systems described in different languages to interact meaningfully. The semantic framework presented in this work allows us to do this—for visual languages, as well as for others.

One way to approach this problem would be to provide translations of these languages into one another, roughly parallel to the solution used for natural languages (Fig. 1a). This obviously implies that the number of translations grows quadratically with the number of languages, which leads to significant overhead when introducing a new language and thus inhibits extension and interoperabil-

[5] In fact, using domain-specific notations may also be motivated by other factors, such as better support for analysis, implementation etc. However, the focus of this work will be on the modeling aspects of notations.

[6] Cf. the notion of *domain analysis* in [9].

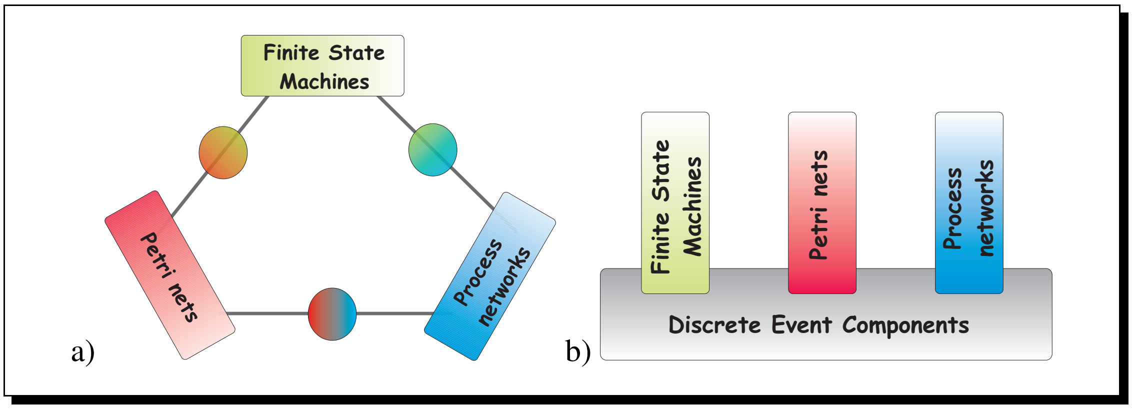

Fig. 1: Approaches to language interoperability.

ity. Alternatively, one could devise a general basic framework, such as e.g. *Discrete Event Components* in Fig. 1b that every language is mapped into, and have them interoperate via this framework. New notations are connected to existing ones simply by providing a mapping of the new language into the common framework. Such a mapping might consist of a compilation/translation scheme, or it might be some form of interpreting engine for the new language.

Both approaches assume that there is an underlying 'fundamental' similarity between all the languages in our scope so that translation into each other, or into some common base model is possible. This is the case for the class of languages we are looking at, but it is easy to think of languages that have sufficiently different semantics to make a translation of one into the other seem rather unlikely—UML class diagrams and knitting patterns, for example.

Assuming this fundamental semantic similarity, the second approach obviously needs fewer translations than the first for more than three languages, it scales significantly better. At least as importantly, it also allows a new language to be interconnected to languages it does not even "know" about—more precisely, interoperability does not depend on the language designers being aware of the existence of other languages, let alone their semantic idiosyncrasies. As this greatly facilitates implementation of new languages, and their use in a heterogeneous model (i.e. one consisting of parts formulated in different languages), it is the approach we chose in this work.

1.3 Some visual languages denoting computation

Before the approach of this work is discussed in more detail a few examples of the kind of visual language that we will be concerned with here are presented. These will serve to illustrate the common characteristics (both in appearance and in meaning) of pictures in these languages,[7] as well as the variation in syntax and semantics that we aim to describe.

In practice for the languages we will be looking at in this work, *visual languages* are really a combination of graphical elements and text. They will also differ in *what* exactly they visualize—which aspects of the modeled systems described are represented by the graphical elements. When discussing the following introductory example languages (and some of their variants), we will focus on three main aspects:

- their *form*, i.e. the graphical and textual elements that are used to form pictures in that language, as well as the rules governing their composition,
- their *meaning*, i.e. the kind of computational system they describe, and specifically the way in which the descriptions translate into the meaning, and finally
- their *object of visualization*, i.e. which aspect of the system is actually depicted graphically.

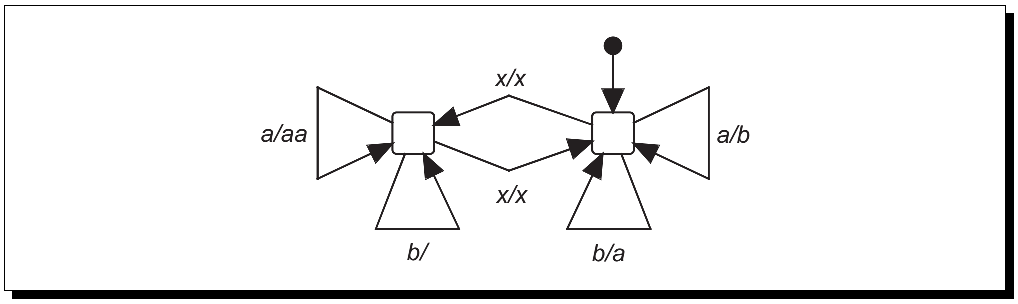

Fig. 2: A small finite state machine.

1.3.1 Finite state machines

Finite state machines (FSM) [62] are simple systems that have a notion of *state*, that accept some *input*, make *transitions* to the next state depending on this input, and possibly produce some output. Fig. 2 shows an example of a small FSM.

This machine defines an automaton that takes an input string in the alphabet a, b, x and, depending on the sequence of characters in that string, performs certain transitions and produces some output. The states are denoted as rounded rectangles, the transitions as arrows between them. The transitions are annotated with short text strings of the form f/g. The intended interpretation is that the transition causes the consumption of the character f from the input[8] and the production of the character g at the output. Either may be missing, in which

[7] We will use the term *picture* for an expression, an utterance in a visual language, just as we would use 'string' in the textual case.

[8] Of course, this makes the respective transition *conditional* on the presence of this character as the next character in the input.

case no input is consumed or output produced, respectively. The little black dot is connected to the *initial state* of the system. In our example, the FSM is initially in the right of the two states, and remains there (more precisely, it makes a transition leading back to the state it started from) as long as it finds only a and b characters in the input string. This situation is shown in Fig. 3a.[9] The output it produces will just exchange these two characters, e.g., the feeding string $abab$ as input to this automaton yields the string $baba$ as its output.

When the machine encounters an x in the input, the corresponding transition leads it to the other state, where it remains again as long as it encounters only a and b. In this state, however, it produces the sequence aa for each a it finds, and an empty sequence for each b, effectively 'swallowing' the b characters. Thus, the input string $ababxaba$ produces the output $babaxaaaa$. Yet another x leads back to the original state, in which a and b are exchanged until the next x and so on. $ababxabaxbbaa$ yields $babaxaaaaxaabb$.

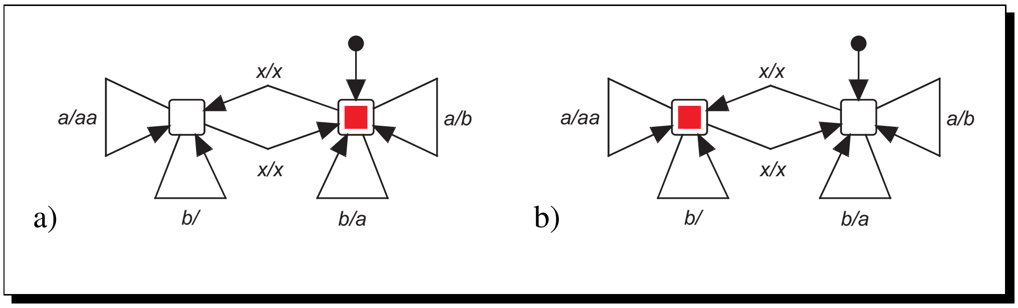

Fig. 3: States of an FSM.

Finite state machines illustrate all three basic semantic concepts that this work is based on:

- a notion of *state*,
- a concept of discrete *transitions* between these states,
- systems may receive *input* and produce *output*, which can be construed as ordered sequences of individual data items (characters in the above example).

In the case of FSMs, the states have no structure of their own—they only need to be distinguishable, atomic entities. Transitions between them need not necessarily be deterministic (although they are in the example)—there is nothing in principle that disallows two transitions from the same state "consuming" the same input character.

The above representation of finite state machines further exemplifies the syntactical elements of the languages we will be concerned with:

[9]Note how additional decoration (which is not part of the visual language proper) is used to represent the system state. We will do this as appropriate whenever we want to discuss the concrete state a system modeled by a picture is in.

- (possibly different kinds of) vertices
- (possibly different kinds of) edges,
- (textual) attributes to the graphical elements.
- rules that describe admissible compositions of these elements

For instance, our example contains two different kinds of vertices (states and the dot to denote the initial state), and one kind of edge (an arrow).[10] It attributes the edges representing transitions with text that indicates the input consumed and output produced in that transition's firing. Furthermore, certain rules apply to the composition of such a picture for it to be *well-formed*: There must be exactly one initial-state marker, there must be exactly one arc leading from it (and ending at some other node), and arcs must never end at the initial-state marker. Finally, all arcs except the one leading from the initial state marker carry an input/output inscription, which in turn has to satisfy (textual) syntactical criteria.

In the picture in Fig. 2 the *state space* of the finite state machine is fully explicit, i.e. its set of possible states and the transitions between them. Indeed, this is precisely the object of visualization in the visual language of finite state machines. Obviously, this is only possible for *finite* state spaces and only practical for *small* state spaces—too small for many real-world applications.

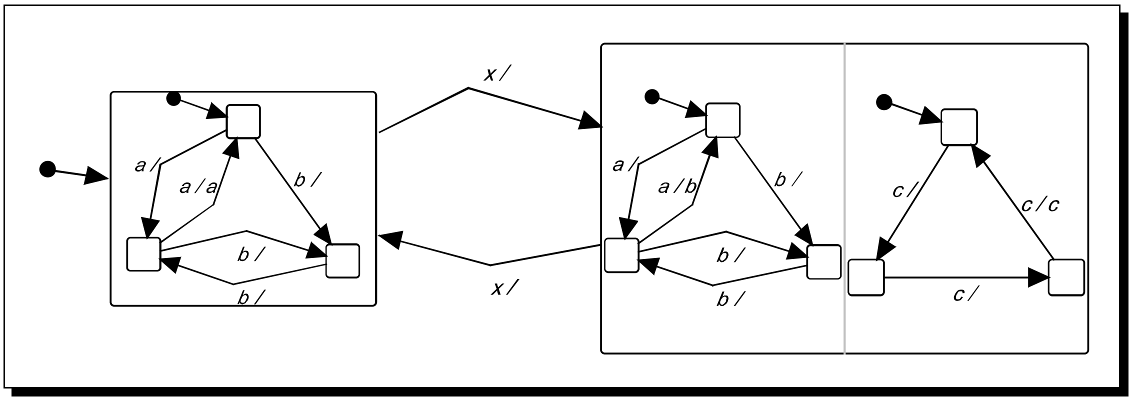

Fig. 4: A hierarchical FSM.

However, in practice FSMs are often represented as *hierarchical* graphs, making use of some structure in their state space to reduce the number of visual elements in the picture. Fig. 4 shows such a hierarchical FSM. Hierarchy is visualized by graphical containment of one or more FSMs inside a state. In this case, the contained picture becomes a decoration, an attribute of the vertex it is contained within. Fig. 5 shows an equivalent 'flat' finite state machine—without going into the semantics of hierarchical FSM, it is obvious, even for

[10]It is of course possible to consider the arc leading from the initial-state marker to the initial state as a different *kind* of arc from those between states.

this toy example, that the hierarchical version is much more concise and better structured than its flat equivalent (15 vertices and 17 edges as opposed to 14 vertices and 42 edges).

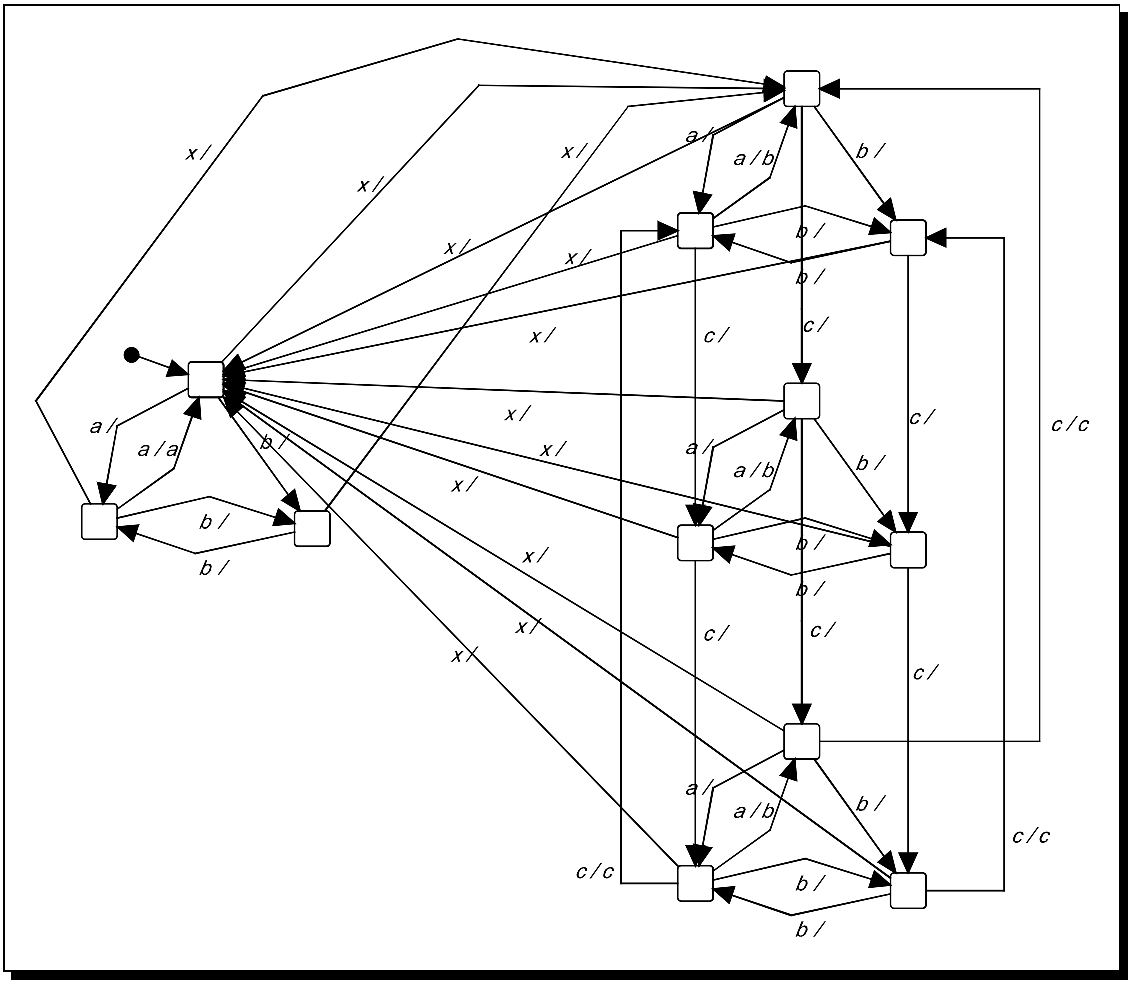

Fig. 5: The 'flat' equivalent of the FSM in Fig. 4.

Representing FSM hierarchically breaks the one-to-one correspondence between graphical elements (vertices) and states in the state space, which is a salient feature of flat finite state machine pictures. In exchange, it makes use of structures in the state space to reduce the number of visual elements used to represent it. Furthermore, it supports new ways of *thinking* about the system—if several machines are contained in the same state (as in the right state of Fig. 4), these may be thought of as running concurrently, and independent of each other. Even though, semantically, every hierarchical FSM picture always corresponds to some ordinary finite state machine, the success of hierarchical FSM notations such as StateCharts [58] suggests that from a modeling point of view this does make a difference.

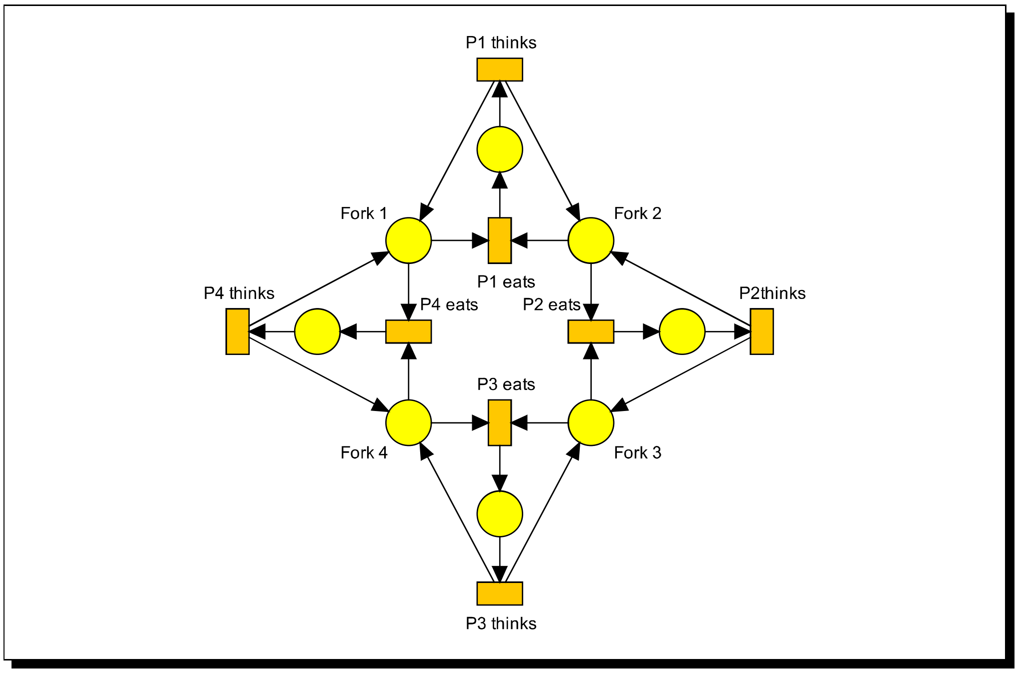

Fig. 6: A simple Petri net.

1.3.2 Petri nets

Petri nets [84, 88, 89] are a well-known formalism that represents a very different approach to modeling system behavior. They are usually depicted as graphs made up from two kinds of vertices, called *places* and *transitions*, connected by one kind of directed arc going from places to transitions or vice versa. Fig. 6 shows a visual representation of such a Petri net, specifically the dining philosopher's problem with four philosophers. Places are drawn as circles, transitions as rectangles.

The state of a Petri net is defined by the distribution of *tokens* over its places. Each place may hold any number of tokens, including zero. In the example, the inner four places are initially assigned one token each (depicted as black dots inside the places in Fig. 7a). These tokens represent the four forks placed on the table between the philosophers.

A transition is called *activated* if all places in its preset (i.e. the set consisting of those places from which an arc goes to the transition) contain at least one token. In Fig. 7, this is denoted by an inner rectangle inside the transition symbol. An activated transition may *occur* (or *fire*), resulting in a new state, i.e. a new distribution of tokens over the places. This distribution is computed by removing a token from each place in the preset and adding one to each place in the postset (i.e. the set of those places to which an arc leads from the transition) of the transition that fired. Fig. 7 shows a possible sequence of state transitions due to transition firing.[11]

[11] Unfortunately, the term 'transition' is somewhat overloaded here, used in 'state transition' of system as well as to refer to a special kind of vertex in a Petri net.

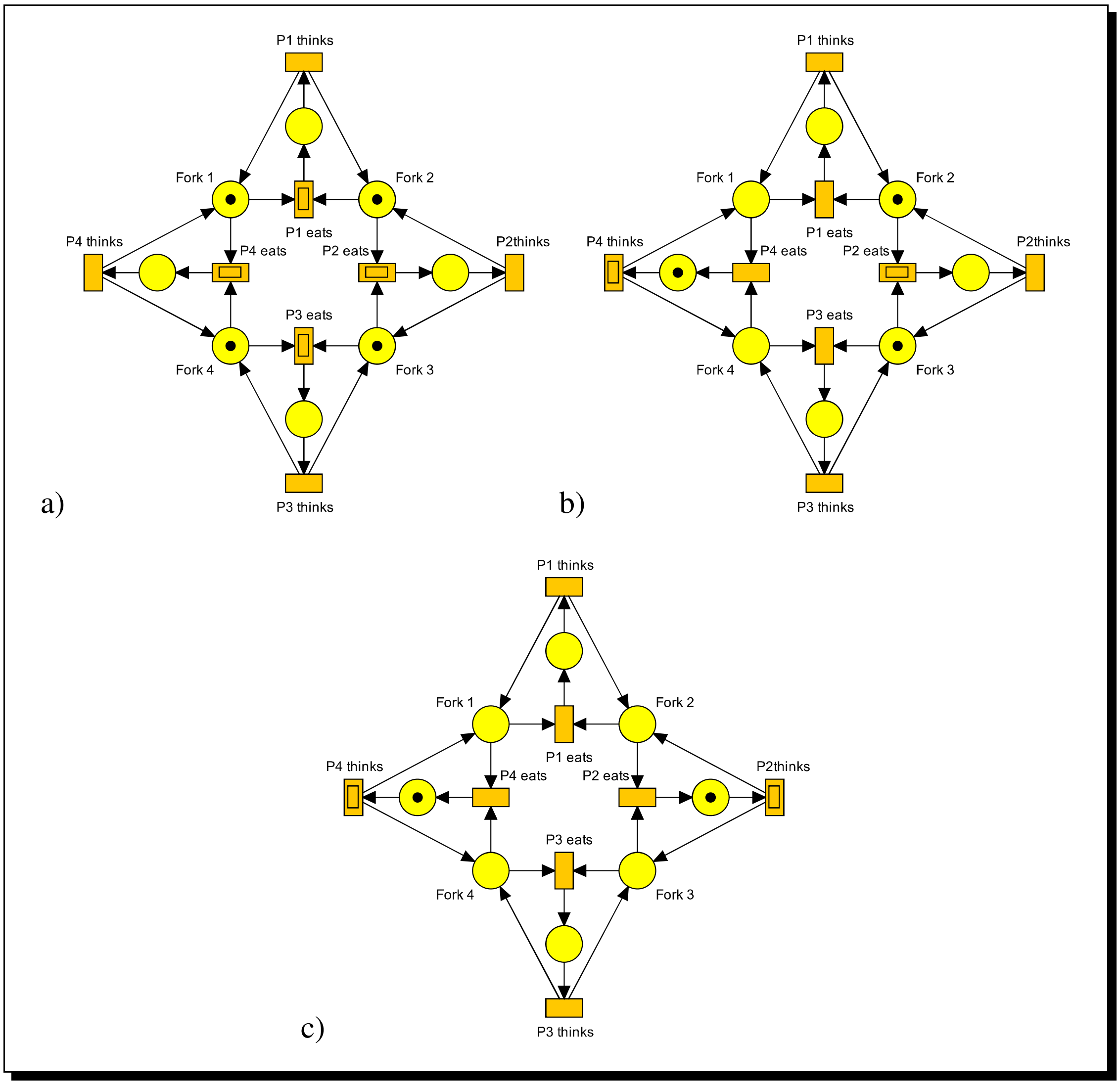

Fig. 7: A few states of the dining philosophers system.

Abstractly, the state of a Petri net may be represented as a mapping from the places of the net to the natural numbers.[12] The state space of a Petri net, although related to its structure, bears in general no resemblance to the net itself—in fact, its structure may depend heavily on other factors, such as the initial token distribution . These points are illustrated in Fig. 8, which depicts a Petri net and Fig. 9 which shows its state spaces for slightly different numbers of initial tokens on the place $P*$.[13]

Instead of the structure of their state space, the usual representation of Petri nets visualizes local causality relations. This makes it a very powerful and ap-

[12] Assuming an ordering on the places, this is equivalent to a vector of natural numbers. Petri nets can then be considered as vector addition systems. [64]

[13] The Petri net in Fig. 8 (taken from [69]) uses a small extension to the version described above, in that it allows for arc *weights*, that specify how many tokens flow across an arc when the transition fires.

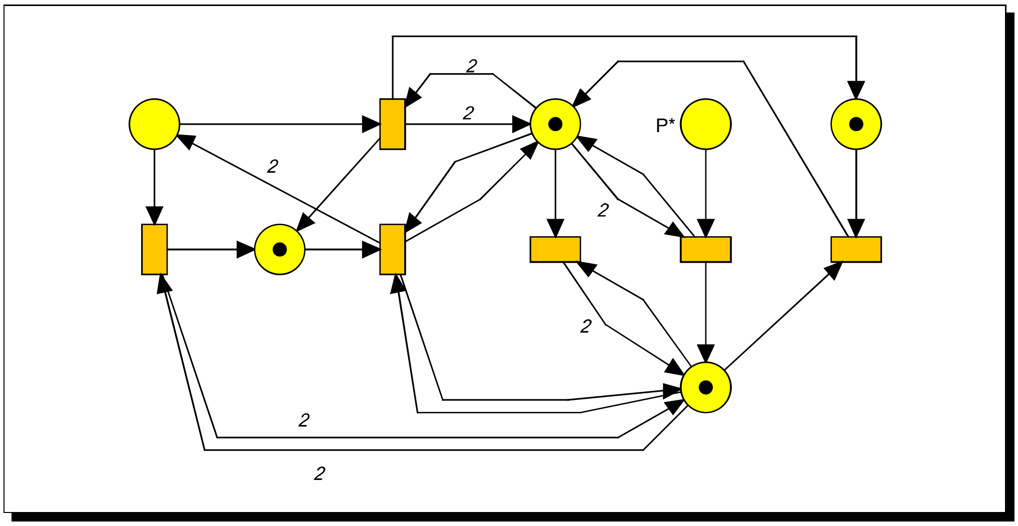

Fig. 8: A small Petri net.

propriate technique for describing distributed and parallel computation, as well as flow-oriented systems. Note that the state spaces in Fig. 9 can be construed as finite state machine models of essentially the same computational process—obviously, in this case, the Petri net representation is much more concise.

Petri nets as we presented them above exhibit certain weaknesses when applied to some real-world modeling tasks. Two of the most important shortcomings are, that

- they are closed in the sense that they have no connection to anything external to them. For instance, an FSM would consume input from a stream and produce a stream of output, and that
- they do not allow for any manipulation of data structures since they only store 'unstructured' tokens in their places.

Several extensions to Petri nets have been suggested to make them more compositional or to include tokens representing complex data and associated functions [29, 70, 74, 106, 108]. One possible approach to compositionality would be to provide input/output ports, through which tokens may enter or leave the Petri net [68]. Manipulation of complex data could be achieved by allowing tokens to represent data structures and transitions to specify some algorithm (in some appropriate language) on the tokens consumed to define the tokens that are produced (along the lines of [70]). An example of such a Petri net is shown in Fig. 10.

Tokens (which are assumed to represent numbers in this example) may enter this Petri net at the input ports *In1* and *In2*. They are put onto the place connected to the entered port and may cause the transition to be activated. If it is, it fires and produces a new token by computing the sum of the tokens consumed, which then leaves the net via the output port *Out*. In this way, Petri nets—much like the finite state machines in the previous section—become compositional

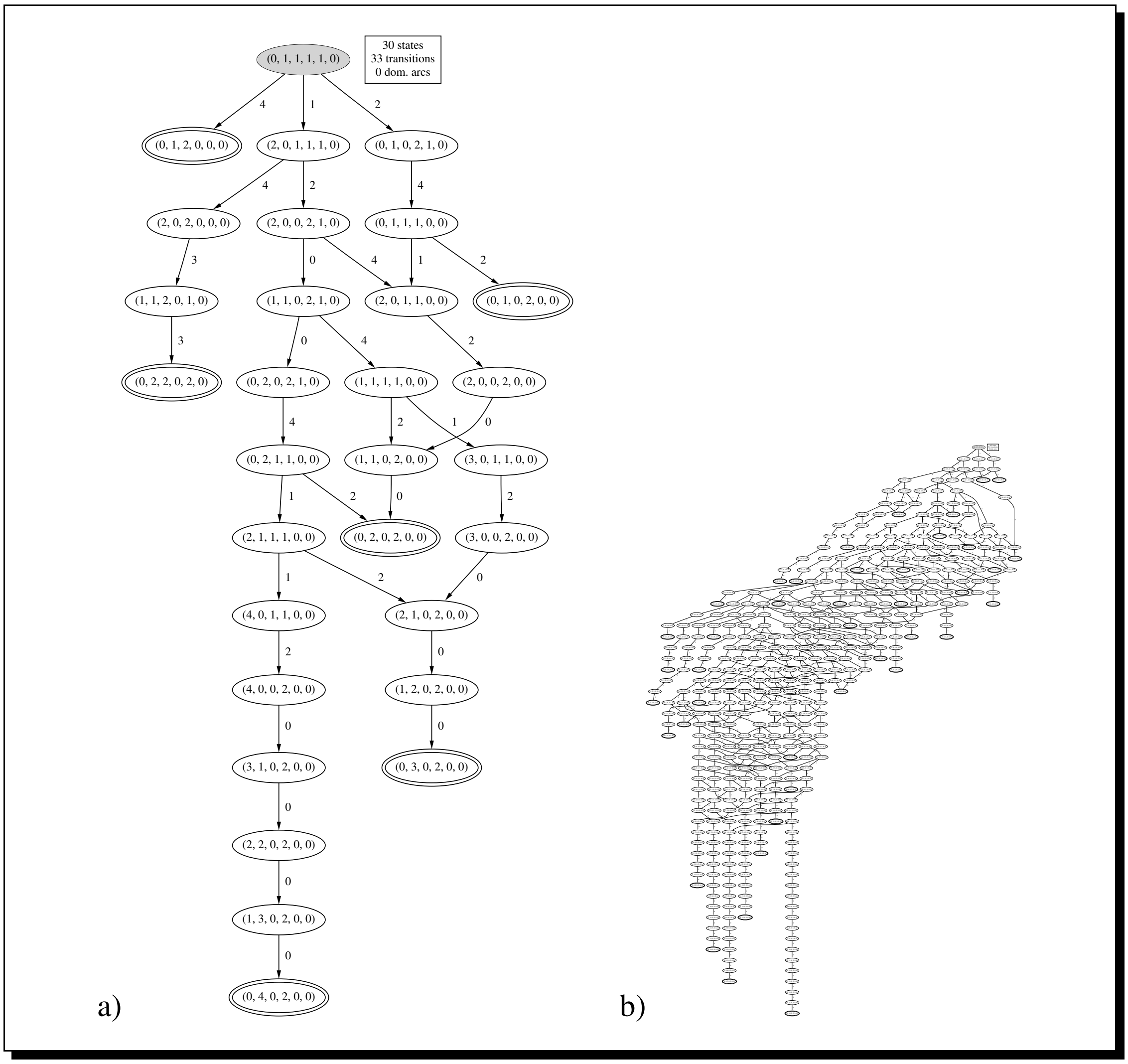

Fig. 9: State spaces of the net in Fig. 8 for (a) $M(P*) = 0$ and (b) $M(P*) = 1$.

entities, *components*, that may be composed with each other, but also with components specified in some other formalism, for instance as finite state machines.

Note that this example also illustrates a more elaborate combination of visual and textual elements used in representing the 'function' of a component. This is a pervasive feature of many visual languages used for modeling or programming, and the decision of which aspects of a system are to be represented visually and which are to be written down textually is a major one in the design of the visual language, and often in the modeling of a concrete system as well.

1.3.3 Dataflow networks

Dataflow networks represent yet another approach to specifying systems [98]. Essentially, they consist of a collection of concurrently executing *processes* which have input/output ports of the kind we have seen for FSM and Petri nets in the previous sections, and a set of connections between the output ports of

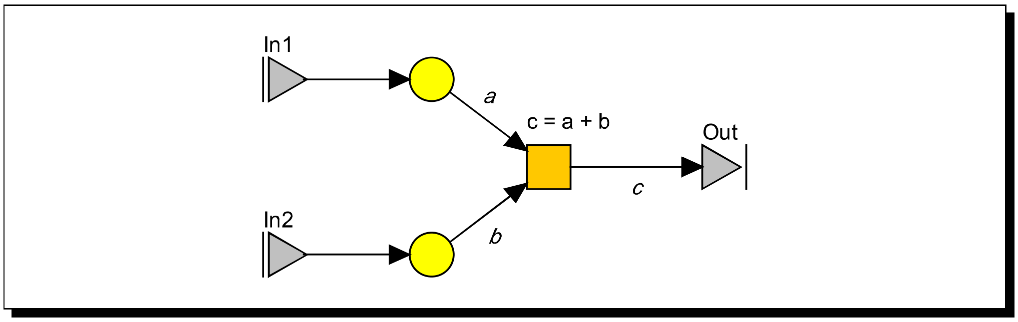

Fig. 10: A very simple Petri net with input/output ports.

these processes and their input ports.

Fig. 11 shows a small example of a process network, describing a simple prime generator. Each rectangular node denotes a process, the triangle represents an output port of this process network. Directed arcs are drawn from output ports of processes to input ports of processes, and also to the output of the process network. Graphically, ports of embedded processes are represented as decorations of the respective node in the process network graph. Arcs, instead of just pointing to a node representing a process, go from one of the decorations denoting an output port to one of the decorations denoting an input port.

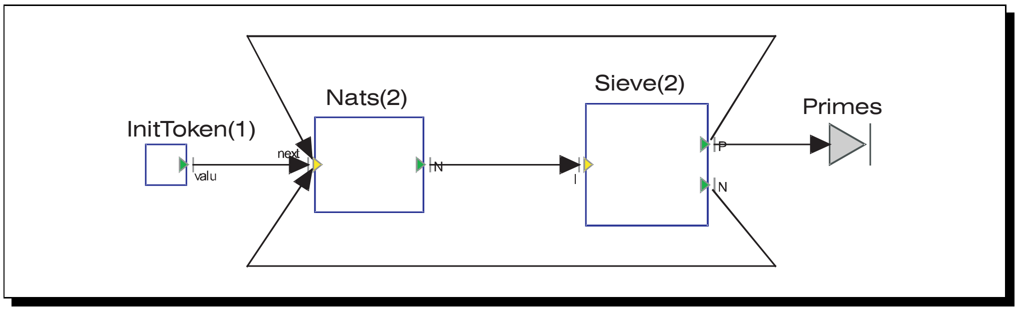

Fig. 11: A simple dataflow process network.

A process network is executed by simply executing the embedded processes and directing data flowing into the process network from the outside, or data coming from one of the embedded components to either an output port of the process network or an input port of an embedded component—according to the connections inside the network. A *step* of the process network consists of a step of any of its embedded components.

In contrast to finite state machines and Petri nets, neither the state nor state transitions are visualized in a process network. In fact, since the process network fully abstracts from the computational processes embedded into it, there is no way of talking about the 'state' of a process network without considering the individual states of the embedded components. In a process network, only the flow of data sent between processes (and to and from the environment of the process network itself) is the object of visualization. In order to understand the working of the example in Fig. 11 we need not draw state diagrams[14]—instead, we only need to know about the input/output relation of the individual processes in the picture. If we know that the left-most block generates a token, that the center block generates the natural number in sequence, starting from 2, for each incoming token, and that the right-most block applies a prime sieve to the incoming sequence of tokens, producing the primes at its *P* output and the non-primes at its *N* output, then this information is sufficient to conclude that the entire network will produce the sequence of primes at its one output.

The information about the individual processes is not contained in the graphical symbols themselves used to represent them but in their inscriptions—the formalism interprets these as expressions instantiating the processes, passing the appropriate parameters to constructor functions. Furthermore, in addition to these textual annotations there are also *graphical* decorations of the main icons (the triangular shapes at their left and right edges) symbolizing input/output ports across which the processes exchange information units. These serve as anchor points for the arcs denoting dataflow connections between those processes, more precisely between their ports, facilitating graphical editing as well as visualization of these connections.

1.4 Problem statement and overview

This work presents a method for defining the instances of a certain class of visual notation. We will characterize this class both in terms of *form* and *meaning*, i.e. we will restrict our attention to notations that have a particular 'look', and notations have a particular meaning.

Specifically, we will focus on notations that are 'graph-like' in appearance, i.e. whose graphical elements consist mainly of nodes of various kinds which are connected by possibly different kinds of arcs, directed and undirected. Both, nodes and arcs, may be decorated with textual and graphical *attributes*.

These notations will be used to depict computational systems, which are informally characterized as follows:

- They have some notion of *state*, which may be described as exactly the amount of information required to be able to predict the future behavior of a system given knowledge about its input.

- There are discrete *state transitions* between states of the system.

- Systems may have input ports and output ports, over which discrete amounts of information (*tokens*) may enter or leave the system, respectively.

[14]In fact, given only the information from that figure we are patently unable to draw the state diagram!

As these interfaces to their environment makes systems inherently compositional, and because their execution occurs in discrete state transitions, we will call these systems *discrete event components*.

The main contributions of this work are the following:

- A formal model of discrete-event components that is compositional, facilitates arbitrary and dynamic component structures, and is sufficiently general to allow the translation of a broad variety of visual notations for such components to be appropriately mapped into it.

- Two languages to define syntactical properties of visual notations as well as their operational semantics in terms of these discrete-event components, effectively providing this mapping mentioned above.

Together these results provide a technique for defining and implementing visual notations for discrete-event computations. We will illustrate its applicability by a few case studies, using a prototype implementation of these concepts inside the MOSES tool suite [1]—cf. App. E.

The remainder of this work is structured as follows. Chapter 2 develops the formal framework for the semantics of discrete event components, the basic model of computation that constitutes the common semantical platform for our specification.

Building on this, chapter 3 elaborates a concept of abstract syntax of visual languages, which will form the link to our notion of discrete event component by introducing the concept of *abstract syntax interpreters* of visual languages. Finally, chapter 4 provides a concrete notation based on *Abstract State Machines* to specify such interpreters in a way that is formal and executable.

These concepts are then applied in chapter 5 to a number of concrete examples in an attempt to cover a reasonable spectrum of visual languages denoting computational systems. Finally, we conclude in chapter 6 with a summary and discussion of our results as well as of future work in this field.

2

A discrete event model of computation

As we have seen in the previous chapter, a discrete event component is essentially characterized by

- a set of *states*, one of which is called the *initial state* of the component,
- a transition function, relating a state and some input to possible successor states and some output,
- an *I/O signature*, abstractly identifying the input/output ports of the component that other components may be connected to.

Abstractly, a component acts on a number of input *streams* fed to its input ports, and produces a number of output streams, one at each output port. This kind of component is also called an *actor* or *dataflow actor*,[1] to emphasize the fact that the streams represent flows of data (usually between more than one actor). The pictures we draw in the visual languages that are of interest to us are essentially representations of such discrete event components.

In this section we first make explicit the notion of *streams* or *sequences* of data upon which actors work. We will then proceed to define the concept of an actor, and discuss some of the properties of actors, including the relation between their denotational and operational semantics. Finally, we will identify a particular subclass of dataflow actors that serves as the basis for our semantical description of discrete event components, and discuss their composition into systems. This forms our basic model of computation into which the semantics of our visual languages are embedded.

[1] Here we follow the terminology of [77], extending the actors presented there by a notion of state.

2.1 Sequences

Our computational systems will act on items of data, or *tokens*. For the moment, the will not be concerned with the structure of the individual tokens, but simply assume that they be from some set A, the set of all tokens.

Effectively, during its operation, an actor produces and consumes sequences of these tokens, which may either be finite or infinite. We will now introduce some basic notation for manipulating sequences.

The set of all finite sequences over A is called A^*, the set of all infinite sequences over A is $A^{\mathbb{N}}$. Since in the following we will mostly work with sequences over A, rather than with A itself or elements of it, we will simple define

$$S =_{def} A^* \cup A^{\mathbb{N}} \tag{2.1}$$

The empty sequence will be written as λ. For convenience of expression, we will further introduce

$$S_{fin} =_{def} A^* \tag{2.2}$$
$$S_{inf} =_{def} A^{\mathbb{N}} \tag{2.3}$$

Concatenation, denoted by the $+$ operator, is the most fundamental operation on these sequences. Thus for any $s, r \in S$, $s + r$ denotes the concatenation of s and r. Note that

$$s \in S_{inf} \Longrightarrow \forall t \in S : s + t = s \tag{2.4}$$

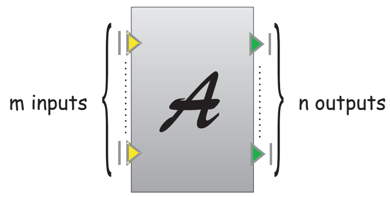

Fig. 12: An actor

2.2 Dataflow actors

Now we will define the notion of *dataflow actor* introduced above, as an entity with states and discrete transitions between them which consumes input from a number of input sequences and produces output. This actor definition builds on and slightly generalizes the one found in [77] by adding a notion of state.

Informally, we can think of actors as automata with a number of input ports, and a number of output ports, that are in one of any number of states, and which consume a 'prefix' of their input sequences in an event called *firing*—cf. Fig. 12.[2] The firing of an actor has two effects: it may change the state of the actor, and it may produce a tuple of output sequences (one sequence per port).

Hence an actor transforms input sequences, from which subsequences are consumed during firing, to output sequences where the result of each firing is concatenated to a single tuple of streams—which is of course exactly how we want to view actors. In fact, a certain class of actors may even be viewed as iterative realizations of functions on streams—this will be discussed in App. A.

So formally, we will write actors in their most general form as follows:

Def. 1: (Dataflow actor with firing)

Assuming a set of sequences S over some set of tokens A, a dataflow actor with firing $\mathcal{A}$ *(also simply* actor *for short) mapping a tuple of streams from S^m to S^n, with $m, n \in \mathbb{N}_0 \cup \{\infty\}$, (written as $\mathcal{A} : S^m \rightsquigarrow S^n$) is defined as a tuple*

$$\mathcal{A} = (\Sigma, \sigma_0, (P_\sigma)_{\sigma\in\Sigma}, (\tau_\sigma)_{\sigma\in\Sigma})$$

with

- *a set of states Σ*
- *an* initial state $\sigma_0 \in \Sigma$
- *for each $\sigma \in \Sigma$, we have*[3]
  - $P_\sigma : S^m \longrightarrow \wp(S^m_{fin})$ *such that $p \in P_\sigma s \Rightarrow p \sqsubseteq s$ a function mapping an m-tuple of input sequences to the set of (finite) prefixes (which we call its* active prefixes*) in σ. The* prefix domain *D_σ of σ is the union of all possible active finite prefixes in that state: $D_\sigma =_{def} \bigcup_{s\in S^m} P_\sigma s$ We further require that $\forall p \in D_\sigma : p \in P_\sigma p$.*
  - $\tau_\sigma : D_\sigma \longrightarrow \wp(\Sigma \times S^n_{fin}) \setminus \emptyset$ *a* transition function *mapping active prefixes from D_σ to a non-empty set of possible next states and the corresponding (finite) outputs generated in that transition.*[4]

[2] Conceptually, as far as our actor model is concerned, firing an actor will be a conceptually atomic step.

[3] $\wp(A)$ denotes the powerset of a set A.

[4] Obviously, the P_σ and τ_σ can also be represented by a transition relation $\rho \subseteq \wp(S^m_{fin} \times \Sigma \times \Sigma \times S^n_{fin})$ such that $P_\sigma s = \{p \mid p \sqsubseteq s \wedge \exists \sigma', v : (p, \sigma, \sigma', v) \in \rho\}$ and $\tau_\sigma p = \{(\sigma', v) \mid (p, \sigma, \sigma', v) \in \rho\}$. We use the functional description because it is more convenient later on.

So in any state σ, the function P_σ can be presented with a tuple of input sequences (those, that have not yet been consumed by the actor, for instance) and it returns a set of prefixes of this tuple. These prefixes essentially identify that part of the input that the actor decides it can operate on. Note that this set general depends on the current state.

Having identified these 'active' prefixes, the function τ_σ represents the actual firing. Passing it an active prefixes will produce a new state and a tuple of output sequences. More precisely, it will produce a set of new state/output tuple combinations, reflecting the fact that an actor may be non-deterministic. Note that the transition function is required to always return at least one possible result of the firing.

It is important to note that the actor definition itself does not contain any notion of 'consumption' of input or 'production' of output, other than that P_σ identifies prefixes of some sequence tuple, and τ_σ produces output tuples. The actual handling of input and output sequences, e.g. the reduction of the input by the prefix that the actor fired on, or the concatenation of its collected output, are considered to be defined outside of the actor. In the next section this will be formally defined. The following examples assume that there is an input and an output tuple of sequences, and that prefixes the actor fires on are removed from its input (thus consumed by the firing), and output tuples are appended to the current collected output.

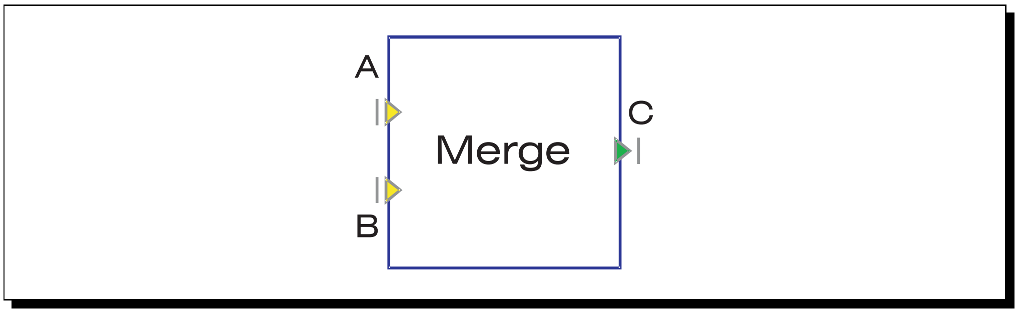

Fig. 13: An actor

As an example, we will elaborate various ways of realizing a *Merge* actor (Fig. 13). Intuitively, this actor takes two streams of input tokens and merges them into one output stream in such a way that their relative order in each stream is preserved, but not necessarily the order between the two input streams. For example, taking the two input streams (ab, cde), admissible outputs would include $abcde$, $cdeab$, $acbde$, $acdeb$ etc. Writing this as an actor according to the above definition, this could look as follows:

Ex. 1: (Non-deterministic merge)

$$\Sigma = \{Q\}$$
$$\sigma_0 = Q$$
$$P_Q : (s_1, s_2) \mapsto \begin{cases} \{(x_1, \lambda), (\lambda, x_2)\} & \textit{with} \quad s_1 = x_1 + s_1', s_2 = x_2 + s_2', \\ & \quad x_1, x_2 \in A, s_1', s_2' \in S \\ \{(x, \lambda)\} & \textit{with} \quad s_1 = x + s', s_2 = \lambda, x \in A, s' \in S \\ \{(\lambda, x)\} & \textit{with} \quad s_1 = \lambda, s_2 = x + s', x \in A, s' \in S \\ \emptyset & s_1 = s_2 = \lambda \end{cases}$$
$$\tau_Q : s \mapsto \begin{cases} \{(Q, x)\} & s = (x, \lambda) \\ \{(Q, x)\} & s = (\lambda, x) \end{cases}$$

This actor has only one state (Q), and its active prefixes include the first tokens of either input, if there are any. For example, $P_Q(ab, cde)$ is $\{(a, \lambda), (\lambda, c)\}$, whereas $P_Q(ab, \lambda)$ is $\{(a, \lambda)\}$.

Thus, in the interesting case where both input sequences are not empty, P_Q produces two possible active prefixes (one part of which is always the empty sequence) which the transition function τ_Q then maps to the respective non-empty component. Nothing in this actor definition specifies the order in which inputs will be processed. Hence its result is indeterminate, and any possible merge of the two input sequences might be its output.

A merge actor that alternatively takes tokens from the two inputs as long as both sequences are not empty, starting with a token from the first input sequence, can be described as follows:

Ex. 2: (Deterministic merge, version 1)

$$\Sigma = \{Q\}$$
$$\sigma_0 = Q$$
$$P_Q : (s_1, s_2) \mapsto \begin{cases} \{(x_1, x_2)\} & \textit{with} \quad s_1 = x_1 + s_1', s_2 = x_2 + s_2', \\ & \quad x_1, x_2 \in A, s_1', s_2' \in S \\ \{(x, \lambda)\} & \textit{with} \quad s_1 = x + s', s_2 = \lambda, x \in A, s' \in S \\ \{(\lambda, x)\} & \textit{with} \quad s_1 = \lambda, s_2 = x + s', x \in A, s' \in S \\ \emptyset & s_1 = s_2 = \lambda \end{cases}$$
$$\tau_Q : s \mapsto \begin{cases} \{(Q, x_1 x_2)\} & s = (x_1, x_2) \\ \{(Q, x)\} & s = (x, \lambda) \\ \{(Q, x)\} & s = (\lambda, x) \end{cases}$$

Here, in contrast, $P_Q(ab, cde)$ results in the singleton prefix set $\{(a, c)\}$, and applying τ_Q to it produces the output ac. The facts that this actor has at most one active prefix, and that τ_Q always produces exactly one result, makes this actor deterministic.

Note that this actor can process two input tokens in a single firing, where it first outputs the token from the first input and then the one from the second input. If the actor would be required to consume exactly one token on each firing (this will be a requirement for the subclass of actors we will be looking at in section 2.5). This behavior would have to be encoded in the state, as in the following example.

Ex. 3: (Deterministic merge, version 2)

$$\Sigma = \{Q_1, Q_2\}$$
$$\sigma_0 = Q_1$$
$$P_{Q_1} : (s_1, s_2) \mapsto \begin{cases} \{(x, \lambda)\} & \textit{with} \quad s_1 = x + s', x \in A, s' \in S \\ \{(\lambda, x)\} & \textit{with} \quad s_1 = \lambda, s_2 = x + s', x \in A, s' \in S \\ \emptyset & s_1 = s_2 = \lambda \end{cases}$$
$$P_{Q_2} : (s_1, s_2) \mapsto \begin{cases} \{(\lambda, x)\} & \textit{with} \quad s_2 = x + s', x \in A, s' \in S \\ \{(x, \lambda)\} & \textit{with} \quad s_2 = \lambda, s_1 = x + s', x \in A, s' \in S \\ \emptyset & s_1 = s_2 = \lambda \end{cases}$$
$$\tau_{Q_1, Q_2} : s \mapsto \begin{cases} \{(Q_1, x)\} & s = (\lambda, x) \\ \{(Q_2, x)\} & s = (x, \lambda) \end{cases}$$

In this version of the deterministic merge, if the actor is in state Q_1 (its initial state), it will take a token from its second input only if its first input sequence is the empty sequence, otherwise, it will take a token from its first input sequence only—and vice versa. The state transition function τ (which is the same for both states) will lead into state Q_2 if a token was taken from the first input sequence, and to Q_1 otherwise. Effectively, this leads to taking tokens from the input sequences in an alternating fashion, as long as both are not empty. The output computed by the two deterministic merges is identical, but they differ in the number of firings it takes them to compute their results.

Now we formally define the semantics of actors. In this work, we will be primarily concerned with the operational notion of *running* an actor. However, it is possible to construct a denotational semantics for the important subclass of *deterministic* actors, which is shown in App. A.

2.3 Operational semantics of an actor

In this section we formalize the execution of an actor already informally described in the previous section. The execution of an actor will be defined by the notion of a *run*, i.e. a stepwise transition of the actor from one state to the next—which depends on the input sequences fed to the actor. Each step in such a run consists of determining the set of active prefixes, consuming one of them from the input, firing the actor on the consumed prefix(computing the new state and appending the resulting output to the output up to this point), and repeating the process on that part of the input tuple that was not consumed until the set of active prefixes is empty.

In order to do this, we not only have to keep track of the state of an actor, but also of the unconsumed input at any step, and the collected output up to that point. Formally, we can write this as follows:

Def. 2: (Run of an actor)

Given a dataflow actor $\mathcal{A} : S^m \rightsquigarrow S^n$ as defined in Def. 1, and a tuple of input sequences $s \in S^m$ a run *on this tuple is a sequence*

$$(s_i, q_i, r_i)_{i \in \mathbb{N}_0}$$

with s_i the unconsumed input tuple, q_i the state of the actor, and r_i the collected output up to this step, such that

$$\begin{aligned}
& s_i \in S^m, q_i \in \Sigma, r_i \in S^n \\
& s_0 = s, q_0 = \sigma_0, r_0 = \lambda_n \\
P_{q_i} s_i \neq \emptyset : \quad & \textit{take some} \quad p \in P_{q_i} s_i \quad \textit{and some} \quad (q, v) \in \tau_{q_i} p \\
& s_i = p + s_{i+1} \\
& q_{i+1} = q \\
& r_{i+1} = r_i + v \\
P_{q_i} s_i = \emptyset : & s_i = s_{i+1} \\
& q_{i+1} = q_i \\
& r_{i+1} = r_i
\end{aligned}$$

It is in this definition that the consumption of input and the production of output is defined, viz. in the definitions that $s_i = p + s_{i+1}$ for the active prefix p selected for firing, and, respectively, by $r_{i+1} = r_i + v$, where the output produced in that firing is appended to the output tuple up to that point.

This definition assumes that the actor is run on one specific input tuple from which prefixes are successively removed. We will later embed actors into more dynamic contexts, where input sequences become added to, for instance, but for the moment we will consider actors in isolation, running on fixed input.

Another aspect worth noting is that runs are always infinite, even though they 'effectively' terminate when the set of active prefixes becomes empty (cf. Def. 13). Note that this is *not* the same as the absence of input tokens—the set $\{\lambda_m\}$ is a perfectly valid prefix, so that an actor can fire without consuming input, and even in the absence of input altogether.

A valid run of the actor in Ex. 1 with input sequences $(ab, cdef)$ would be

$$\begin{aligned}
((ab, cde), Q, \lambda) \to ((b, cde), Q, a) \to ((\lambda, cde), Q, ab) \to ((\lambda, de), Q, abc) \\
\to ((\lambda, e), Q, abcd) \to ((\lambda, \lambda), Q, abcde) \to \ldots
\end{aligned}$$

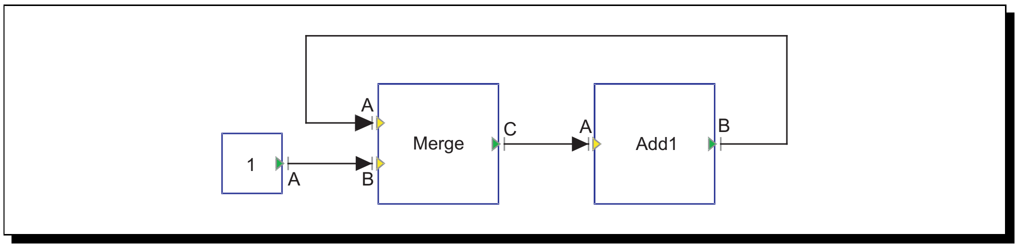

Fig. 14: A simple network of actors.

or

$$((ab, cde), Q, \lambda) \to ((ab, de), Q, c) \to ((b, de), Q, ca) \to ((b, e), Q, cad)$$
$$\to ((b, \lambda), Q, cade) \to ((\lambda, \lambda), Q, cadeb) \to \ldots$$

Note that we have omitted those steps in the run where the set of active prefixes would be empty, as this implies the termination of the actor for the given input sequence.

As expected, the deterministic actor from Ex. 2 has exactly one run:

$$((ab, cde), Q, \lambda) \to ((b, de), Q, ac) \to ((\lambda, e), Q, acbd)$$
$$\to ((\lambda, \lambda), Q, acbde) \to \ldots$$

Similarly, the deterministic variant in Ex. 3 has exactly one run, finally computing the same output, but arriving there in a somewhat different manner:

$$((ab, cde), Q_1, \lambda) \to ((b, cde), Q_2, a) \to ((b, de), Q_1, ac) \to ((\lambda, de), Q_2, acb)$$
$$\to ((\lambda, e), Q_1, acbd) \to ((\lambda, \lambda), Q_1, acbde) \to \ldots$$

2.4 Composition—networks of actors

So far, we have only looked at dataflow actors in isolation, their behavior when they change state while consuming input tokens and producing (sequences of) output tokens. Now we will compose these actors to form *systems* of communicating actors, where tokens produced as output of one actor may serve as input to another.

Consider the network depicted in Fig. 14. It contains three actors, 1, *Merge*, and *Add1*. Assume that 1 outputs the sequence of length 1 containing only the integer number 1 (in this case the basic alphabet A contains at least the set of integers), *Merge* merges its two input streams (for the moment it is irrelevant whether it does so deterministically), and *Add1* consumes its input token by token and for each integer n it outputs $n+1$. The arrows in the network indicate that output of one actor is to be considered input to another. Obviously, in this case, the output of *Merge* will be the natural numbers starting at 1. (We will of course make the notion of a *run* of a network more precise below.)

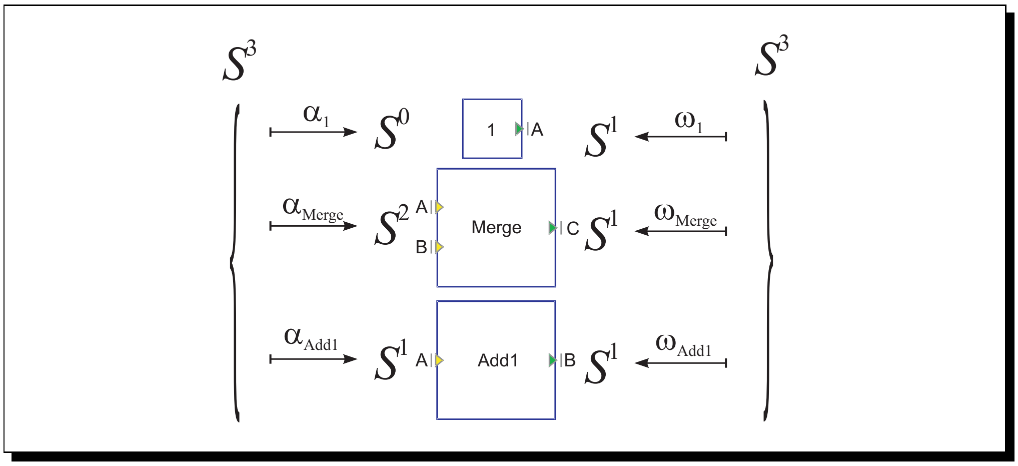

Fig. 15: Products of input/output tuples and their projections.

Note, however, that the above network is rather special in a number of ways. For one, it does not depend on whether *Merge* is deterministic, because there is always at most one token to be acted upon, anyway—so even if the *Merge* actor itself would be implemented nondeterministically, the network behavior would still be deterministic. Furthermore, the network structure is static, i.e. the connection structure between actors, as well as the number of actors in the network, does not change during its execution. Also, there are exactly as many inputs as there are outputs, and all of them are connected. In the general case, we will require none of these properties.

We define a system as a set of actors and some communication structure between them. This structure will be one-to-one, i.e. at any given time each output is connected to at most one input and vice versa. In the first step, this structure is static, i.e. the connections between actors remain the same during the execution of the system. Then, however, we will turn to the more general case of dynamic network structures, where actors (or rather a slightly extended version of them) may change the network structure as a result of a firing.

2.4.1 Static network structures

First, we need to abstractly characterize the communication structure between actors, or rather their input sequences and output sequences. We will do this in two steps using products and projections.[5] In the first step, we will simply define the product of all inputs and all outputs of a set of actors. In a second step, we define a way to connect some of the inputs to some of the outputs.

In 14 it can be seen that the total number of actor inputs and the number of actor outputs is three. In other words, putting them side by side, a 3-tuple of input sequences is fed into them, and a 3-tuple of sequences comes from their outputs. We now need to relate the individual actor inputs and outputs to the collective input/output tuple.

[5] A short introduction to this is given in App. B.

This is achieved by defining a pair of projections, as can be seen in Fig. 15. A possible system of projections for the example in Fig. 15 might look like this:

Ex. 4: (Concrete projections for Fig. 15)

$$\begin{aligned}
\alpha_1 &:(s_1, s_2, s_3) \mapsto () \\
\alpha_{Merge} &:(s_1, s_2, s_3) \mapsto (s_1, s_2) \\
\alpha_{Add1} &:(s_1, s_2, s_3) \mapsto (s_3) \\
\omega_1 &:(s_1, s_2, s_3) \mapsto (s_1) \\
\omega_{Merge} &:(s_1, s_2, s_3) \mapsto (s_2) \\
\omega_{Add1} &:(s_1, s_2, s_3) \mapsto (s_3)
\end{aligned}$$

Here, the (s_1, s_2, s_3) is the collective tuples of input (for the α_i-projections) or output (for the ω_i-projections) sequences s_i, from which the projections pick the appropriate sequences in such a way that each input sequence and each output sequence is projected exactly once.

Obviously, we would like all input/output projections, to (a) exhaust the collective input/output tuple and (b) not to overlap, i.e. we would like there to be a one-to-one correspondence between positions in the collective input/output and all the positions in all actor inputs/outputs taken together. This can be done by simply requiring all the input projections, and likewise all the output projections to form a *product* (together, of course, with corresponding sets of collective input and output tuples)—cf. App. B for a short introduction.

We will call such a collection of actors with a corresponding set of projections a *system of actors*, which we define as follows:

Def. 3: (System of actors) *Assume a set of actors $\{\mathcal{A}_i : S^{m_i} \rightsquigarrow S^{n_i} \mid i \in I\}$. Then we have $M = \sum_{i \in I} m_i$ actor inputs and $N = \sum_{i \in I} n_i$ actor outputs. If we picture all inputs of all actors as some M-tuple, there is for each $\mathcal{A}_i$ exactly one projection $\alpha_i : S^M \longrightarrow S^{m_i}$ from this S^M onto S^{m_i} that results in the input tuple of that actor. In other words, $(S^M, \{\alpha_i \mid i \in I\})$ form a product. Likewise, for each actor there is a projection $\omega_i : S^N \longrightarrow S^{n_i}$ that yields its output tuple from the tuple of all outputs in the system, so that $(S^N, \{\omega_i \mid i \in I\})$ is a product. These projections are depicted at the top and at the bottom of Fig. 16.*

We call this set of actors $\mathcal{A}_i$ and corresponding families of projections α_i and ω_i a system of actors *iff these families of projections define products.*

Since projections define a product uniquely up to isomorphism, we may assume the α_i and ω_i to be fixed for a given set of actors without loss of generality. This allows us to conveniently extract from the collective input tuple (which will play an important role in the execution of a network of actors) the inputs for any specific actor without having to keep track of the indices of the actor's input sequences inside the collective input tuple—and likewise for the output.

$$\begin{tikzcd}
S^M \arrow[rr, "\alpha_i"] \arrow[dr, "p^{in}_{i,k}"'] & & S^{m_i} \arrow[dl, "\pi_k"] \\
 & S & \\
S^N \arrow[rr, "\omega_i"] \arrow[ur, "p^{out}_{i,k}"] & & S^{n_i} \arrow[ul, "\pi_k"']
\end{tikzcd}$$

Fig. 16: The projections identifying actor ports.

The next step is to identify individual input and output ports and to connect them. First of all, we have to realize that each input and output port of an actor is uniquely identified by some projection $p : S^M \longrightarrow S$ for an input port, or $p : S^N \longrightarrow S$ for an output port, which extracts exactly the individual sequence from the collective tuple of input or output sequences which goes to or comes from the port in question. Obviously, no two input ports should have the same projection, and likewise for the output ports. All we currently have, however, are projections from the collective tuples to the full input or output tuples of actors, not to their individual ports (which are the ones we want to connect). The question is how to construct the projection for each individual port from the projections for the complete actor.

The idea here is to use the same technique once again, using the standard cartesian product projections, which are defined as follows:

$$\pi_k : (a_1, ..., a_k, ..., a_n) \mapsto a_k$$

In other words, π_k extracts the k-th position from its argument tuple. Looking at actor $\mathcal{A}_i$, its input tuple is m_i-ary, and its output tuple n_i-ary, giving rise to the corresponding number of cartesian projections, which *relative to the actor* identify a given input or output port—hence we call them *actor port projections*. They are, however, of course not unique inside the complete network—e.g., all actors with two inputs will have the same two input port projections $\pi_1 : S^2 \longrightarrow S$ and $\pi_2 : S^2 \longrightarrow S$.

Here we concatenate the actor port projections to the input projections of the actor, which yield exactly the projections from the collective input to each individual output of an actor (the *system port projections*) in the way shown in Fig. 16. Fig. 17 demonstrates this for the input projections of our example network. This way, e.g., the k-th input port of the actor $\mathcal{A}_i$ is defined to be $p^{in}_{i,k} = \alpha_i \circ \pi_k$.

Def. 4: **(Input/output ports of actors/a system of actors.)** *Given a system of actors*

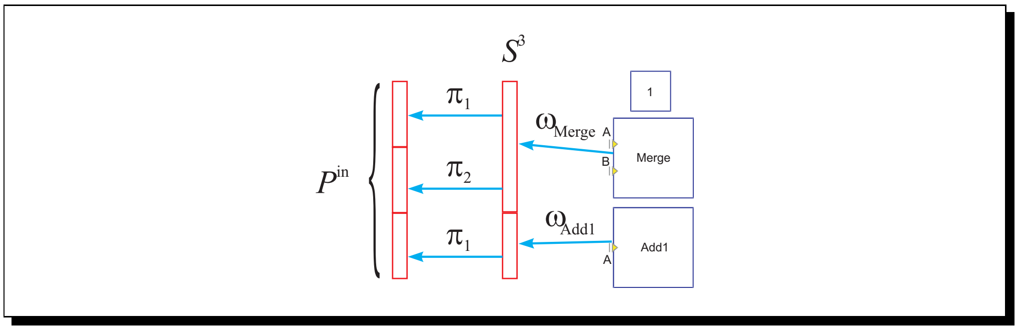

Fig. 17: Identifying individual ports by concatenating projections.

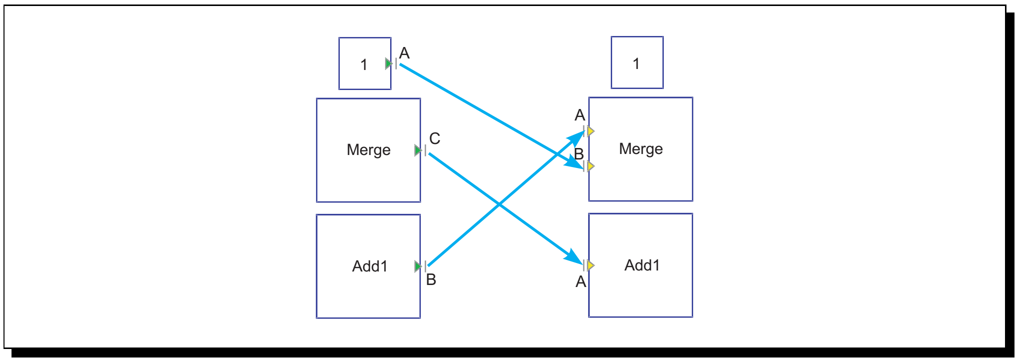

Fig. 18: Unfolding the actor network in Fig. 14—first step.

$\mathcal{A}_i : S^{m_i} \rightsquigarrow S^{n_i}$ with corresponding α_i and ω_i. We define

$$P_i^{in} = \{p_{i,k}^{in} = \alpha_i \circ \pi_k \mid 1 \leq k \leq m_i\}$$

to be the set of input ports of the actor $\mathcal{A}_i$ and

$$P_i^{out} = \{p_{i,k}^{out} = \omega_i \circ \pi_k \mid 1 \leq k \leq n_i\}$$

as the set of its output ports.

From this, we arrive at

$$P^{in} = \bigcup_{i \in I} P_i^{in}$$

as the set of input ports *of the system and*

$$P^{out} = \bigcup_{i \in I} P_i^{out}$$

as the set of its output ports.

These projections are depicted in Fig. 16.

Note that the $p_{i,k}^{in}$ go from S^M to S, and the $p_{i,k}^{out}$ from S^N to S. Obviously, these are all distinct and uniquely identify any port in the system.

Now we can connect outputs to inputs simply by means of a relation between these to sets. Fig. 18 shows the connections of the network in Fig. 14, and

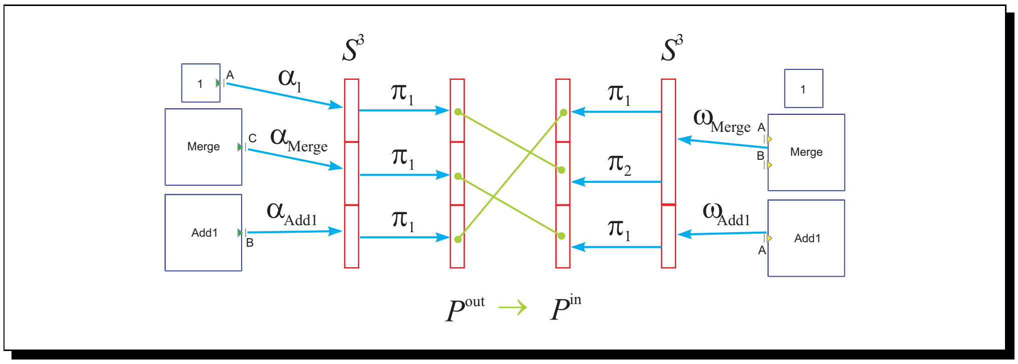

Fig. 19: Unfolding the actor network in Fig. 14—introducing the projections.

Fig. 19 illustrates how the individual ports are obtained and the relation $\longrightarrow$ connecting P^{out} and P^{in}. Note that since $\longrightarrow$ is a relation, it allows arbitrary n-m connections between ports (although the example only contains one-to-one connections).

Ex. 5: **(Network structure of Fig. 14.)**

$$\longrightarrow = \{(\alpha_1 \circ \pi_1, \omega_{Merge} \circ \pi_2), (\alpha_{Merge} \circ \pi_1, \omega_{Add1} \circ \pi_1), (\alpha_{Add1} \circ \pi_1, \omega_{Merge} \circ \pi_1)\}$$

Giving a relation between P^{out} and P^{in} is all there is to defining the network structure. In the general case, we do not assume all actor inputs and outputs to be connected, neither do we require that the total number of input and output ports be equal. Therefore, the general definition is as follows:

Def. 5: **(Network of actors)** *Assume a system of actors* $\{\mathcal{A}_i : S^{m_i} \leadsto S^{n_i} \mid i \in I\}$ *with corresponding families of projections* α_i *and* ω_i*, and resulting sets of input and output ports* P^{in} *and* P^{out}*. Then any relation*

$$\longrightarrow \subseteq P^{out} \times P^{in}$$

defines a network structure *in that system.*

A system together with a network structure is simply called a network (of actors).

Now we would like to define a *run* of a network of actors. We assume that there is a collective input tuple of sequences, and that this need not be empty, which essentially provides the 'initial' input data of the system. Informally, a *step* of the network will consist of one of its actors firing, transforming its own state, consuming input and producing output. This output is then added to the sequences of unconsumed input tokens according to the network structure—e.g., in our example in Fig. 14, a token that is produced by the *Merge* actor is added to the sequence of hitherto unconsumed input of the *Add1* actor. This means that these input sequences are part of the state of the network, in addition, of course, to the states of all the actors.

More precisely, running a network of actors on some 'initial' input data in S^M consists of discrete steps that

- pick one actor that has an active prefix,
- fire it and
- add the tokens thus produced to the respective input streams, removing the consumed tokens from the actor's input stream(s).

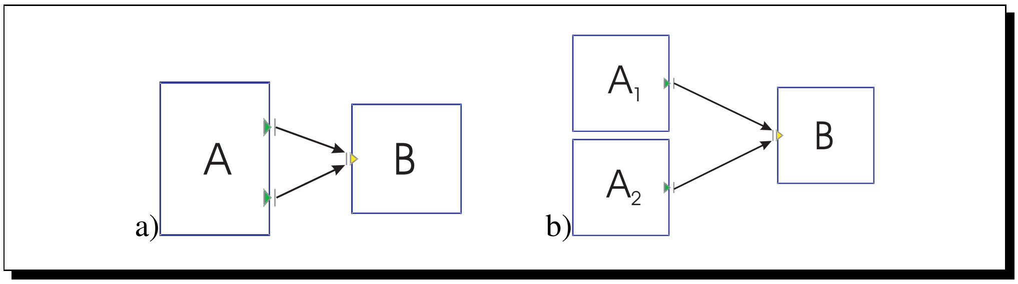

Fig. 20: Different ways of connecting two output ports to one input port.

When adding the tokens produced by the actor firing we have to consider the possibility that more than one output of the firing actor might be connected to the same input port—such as in Fig. 20a. If actor *A* produces tokens at both output ports in the same firing, then, to all intents and purposes, these tokens were produced simultaneously. However, when adding them to the input sequence corresponding to the input port of actor *B*, we have no choice but to somehow arrange them sequentially. The way we do this is by nondeterministically choosing one of the possible permutations for the order in which we add tokens from any of the outputs of the actor that has just fired.

Note that this situation only occurs if the same input port is connected to more than one output port *of the same actor*. In a structure like the one in Fig. 20b there is no such problem, as the actors A_1 and A_2 never produce tokens simultaneously, as actors are sequentially.

Def. 6: **(Run of a static network)** *Given a network of actors defined by* $\{\mathcal{A}_i : S^{m_i} \rightsquigarrow S^{n_i} \mid i \in I\}$ *with corresponding families of projections* α_i *and* ω_i *and a network structure* $\longrightarrow$*, we call a sequence of states*

$$(s_k, (\sigma_{i,k})_{i\in I}, r_k)$$

a run *of the network (starting from initial state* $(s_0, (\sigma_{i,0})_{i\in I}, \lambda_N)$*) iff*

- $s_k \in S^M$ *and* $r_k \in S^N$
- $\sigma_{i,k} \in \Sigma_i$

and each subsequent state $(s_{k+1}, (\sigma_{i,k+1})_{i\in I}, r_{k+1})$ *is is related to its predecessor as follows.*

We need to distinguish between the case that in state $(s_k, (\sigma_{i,k})_{i\in I}, r_k)$ *there is at least one actor that can fire, i.e. that has an active prefix of its part of the* *input tuple, and the case that there is no such actor. In the latter case, nothing changes, i.e.* $s_{k+1} = s_k$, $r_{k+1} = r_k$, *and for all* $i \in I$, $\sigma_{i,k+1} = \sigma_{i,k}$.

Now assume there is at least one actor that can fire. We compute the next state as follows:

- *First choose an actor* $\mathcal{A}_i$ *with a non-empty set of active prefixes.*
- *Choose one such prefix* $p \in P_{\sigma_{i,k}} \alpha_i s_k$.
- *Then choose some* $(q, v) \in \tau_{\sigma_{i,k}} p$.
- *Finally, choose some permutation of* μ *of the numbers* $\{1, ..., n_i\}$.

First we determine the residual tuple of input sequences s'_k *that is left after 'consuming'* p. *This we can define by*

$$\begin{aligned} \alpha_i \; s_k &= p + \alpha_i s'_k \\ \alpha_j \; s_k &= \alpha_j s'_k \quad \textit{for} \quad j \neq i \end{aligned}$$

Now we define the N*-ary output tuple* r*, which represents the output of the firing in* N*-ary space, by 'padding' the output produced by the actor with* λ*:*

$$\begin{aligned} \omega_i \; r &= v \\ \omega_j \; r &= \lambda_{n_i} \quad \textit{for} \quad j \neq i \end{aligned}$$

The M*-ary tuple of new inputs* $v_k \in S^M$ *for the* $\mu(k)$*-th output is defined using the* $p \in P^{in}$ *as follows:*

$$p \; v_k = \begin{cases} \pi_{\mu(k)} v & \textit{for} \quad p^{out}_{i,k} \longrightarrow p \\ \lambda & \textit{otherwise} \end{cases}$$

So that finally the new state is

$$\begin{aligned} s_{k+1} =& s'_k + v_1 + ... + v_{n_i} \\ \sigma_{j,k+1} =& \begin{cases} q & \textit{for} \quad j = i \\ \sigma_{j,k} & \textit{otherwise} \end{cases} \\ r_{k+1} =& r_k + r \end{aligned}$$

Note that the r_k contain the accumulated output for the entire run. Strictly speaking, this is a redundant part of the system state, as it does not influence its behavior (nothing in the execution depends on it). It is included here, however, for practical reasons, so we can later use it as part of the *result* of a run.

This definition allows us to 'execute' a network of discrete event components by firing one actor in each step. It assumes that the network structure itself be static, i.e. that the connection structure defined by $\longrightarrow$ is given initially, and will remain the same during the execution of the system. In general, we will allow this to change, and the following section will provide a framework for this.

Note further how the above definition isolates four potential sources of nondeterminism in the execution of a network of actors:

1. There may be more than one actor activated.
2. Each actor may be activated by more than one active prefix.
3. Firing an actor may in itself be nondeterministic in the sense that more than one pair of next state/output tuple may be returned.
4. Finally, overlap inside the connection structure of a network may make the choice of the order in which output sequences are added to the result a relevant one.

This classification can be used to distinguish implementation and scheduling strategies of this semantics. For instance, in the implementation in the Moses Tool Suite [1] (see Appendix E), when executing networks of actors, the first choice is made by the environment (a scheduler cf. section 2.6), while the other three are made by the actors themselves.

2.4.2 Dynamic network structures

The assumption of a static connection structure between the actors may often be good enough, but it turns out to be too restrictive in many modeling situations. Specifically, there are three kinds of situations where a more general approach is needed:

- New actors are generated and connected to other actors in the network.
- Actors are removed and disconnected from the network.
- Actors are moved through the network, possibly disconnecting them from some actors and connecting them to others.

We have developed visual notations providing constructs for these situations (e.g. [68]) and demonstrated their utility in specific modeling tasks (for instance to develop generic simulation frameworks [45]). This section will extend the semantical framework of static network structures to accommodate the semantical description of these languages.

The creation of actors can be handled by providing a sufficiently large base set of 'fresh' actors, from which we always take a new one when we conceptually 'create' an actor. Sufficiently large here usually means denumerably many, making I (the index set for our actors) denumerable, and also N and M (the number of output and input ports, respectively). This does not pose a problem, as none of our definitions above relied in any way on their finiteness. So what remains is to be able to modify the network structure among all these actors during a run. However, we obviously only want to execute those actors that have already been created in this sense, therefore we need to keep track of the set of actors that we are actually running, which will be indexed by a subset $J \subseteq I$ of the set of all actor indices.

As shown above, for a given system of actors (including their α_i and ω_i projections), the network structure is completely defined by the $\longrightarrow$ relation between ports. In general, the structure of a system of actors may change during its execution, therefore this relation must be part of the overall state of the system, in addition to the states of the individual actors and the tokens which have been produced but not yet consumed by an actor.

However, the actors defined in Def. 1 are not related to any notion of network structure, and therefore there is no way they could modify it. What we therefore need is a concept of an *extended actor*, that not only consumes and produces tokens, but is also aware of the structure of the network and is able to modify it. Hence an extended actor can form new connections between ports and remove old ones, by simply returning (from its transition function τ_σ) relations between P^{out} and P^{in} that are added to and subtracted from the current network structure. Furthermore, an extended actor also can create new actors by returning their indices.

Def. 7: **(Extended actor, system of extended actors)** *We call a structure*

$$(\Sigma, \sigma_0, (P_\sigma)_{\sigma \in \Sigma}, (\tau_\sigma)_{\sigma \in \Sigma})$$

with Σ*,* σ_0 *and the* P_σ *as in Def. 1 and a transition function*

$$\tau_\sigma : D_\sigma \longrightarrow \wp(\Sigma \times S^n_{fin} \times (P^{out} \times P^{in}) \times (P^{out} \times P^{in}) \times \wp(I)) \setminus \emptyset$$

an extended actor*, assuming a* system *of such actors with corresponding projections and sets of input/output ports defined analogous to Def. 3 and Def. 4.*[6]

Note that an extended actor depends on the system it runs in, which of course makes sense since it can modify its structure. Intuitively, the two additional return values of the τ_σ contain the new connections between ports and those that are to be removed, respectively.

Before we come to the definition of a run of a system of extended actors, we need to define the structure of the state of such a system.

Def. 8: **(State of a system of extended actors)** *Assume a system of extended actors* $\{\mathcal{A}_i : S^{m_i} \rightsquigarrow S^{n_i} \mid i \in I\}$ *with corresponding families of projections* α_i *and* ω_i*, and sets of input and output ports* P^{in} *and* P^{out}*.*

We call a tuple $(J, \longrightarrow, s, (\sigma_i)_{i \in I}, r)$ *a* state *of this system iff*

[6] This definition might seem somewhat circular due to its reliance on a given system of extended actors which is needed to identify the set of ports. However, the definition of a system of (extended) actors really only needs their input/output signature, which could easily be factored out.

- $J \subseteq I$
- $\longrightarrow \subseteq P^{out} \times P^{in}$
- $\sigma_i \in \Sigma_i$, *and*
- $s \in S^M$ *and* $r \in S^N$.

The relation $\longrightarrow$ defines the current network structure, while J contains the indices of the *active* actors—as we will see, only actors that are active in a given state may be fired in the corresponding step of the run.

This allows us to define a run of such a system. Intuitively, we start with some initial network structure $\longrightarrow_0$ and some initial input, and then each step consists of picking an actor that has an active prefix, firing it on this prefix, and updating the connection structure as well as the state of the actor and the currently unconsumed tokens.

Def. 9: **(Run of a system of extended actors.)** *Assume a system of extended actors* $\{\mathcal{A}_i : S^{m_i} \rightsquigarrow S^{n_i} \mid i \in I\}$ *with the usual projections. We call a sequence of states*

$$(J_k, \longrightarrow_k, s_k, (\sigma_{i,k})_{i \in I}, r_k)$$

a run *of the system (starting from initial state* $(J_0, \longrightarrow_0, s_0, (\sigma_{i,0})_{i \in I}, \lambda_N)$*) iff*

- $J_k \subseteq I$
- $\longrightarrow_k \in \wp(P^{out} \times P^{in})$
- $s_k \in S^M$ *with* $M = \sum m_i$
- $\sigma_{i,k} \in \Sigma_i$

and for each k *its successor state* $(J_{k+1}, \longrightarrow_{k+1}, s_{k+1}, (\sigma_{i,k+1})_{i \in I})$ *is defined as follows:*

We need to distinguish between the case that in state $(J_k, \longrightarrow_k, s_k, (\sigma_{i,k})_{i \in I})$ *there is at least one actor that can fire, i.e. that has an active prefix of its part of the input tuple, and the case that there is no such actor. In the latter case, nothing changes, i.e.* $J_{k+1} = J_k$, $\longrightarrow_{k+1} = \longrightarrow_k$, $s_{k+1} = s_k$, *and for all* $i \in I$, $\sigma_{i,k+1} = \sigma_{i,k}$. *Now assume there is an actor that can fire. We compute the next state as follows:*

- *First choose an actor* $\mathcal{A}_i$ *with* $i \in J_k$ *with a non-empty set of active prefixes.*
- *Now choose one such prefix* $p \in P_{\sigma_{i,k}} \alpha_i s_k$.
- *Then choose some* $(q, v, c^+, c^-, N) \in \tau_{\sigma_{i,k}} p$.
- *Finally, choose some permutation of* μ *of the numbers* $\{1, ..., n_i\}$.

First we determine the residual tuple of input sequences s'_k that is left after 'consuming' p. This we can define by

$$\alpha_i s_k = p + \alpha_i s'_k$$
$$\alpha_j s_k = \alpha_j s'_k \quad \textit{for} \quad j \neq i$$

Now we define the N-ary output tuple r such that

$$\omega_j r = \begin{cases} v & \textit{for} i = j \\ \lambda_{n_j} & \textit{otherwise} \end{cases}$$

The M-ary tuple of new inputs $v_k \in S^M$for the $\mu(k)$-th output is defined using the $p \in P^{in}$ as follows:

$$pv_k = \begin{cases} \pi_{\mu(k)} v & \textit{for} \quad p^{out}_{i,k} \longrightarrow_k p \\ \lambda & \textit{otherwise} \end{cases}$$

So that finally the new state is

$$\begin{aligned} J_{k+1} &= J_k \cup N \\ \longrightarrow_{k+1} &= (\longrightarrow_k \cup c^+) \setminus c^- \\ s_{k+1} &= s'_k + v_1 + ... + v_{n_i} \\ \sigma_{i,k_1} &= q \\ \sigma_{j,k+1} &= \sigma_{j,k} \qquad \textit{for} \quad i \neq j \\ r_{k+1} &= r_k + r \end{aligned}$$

Of course this definition parallels the one for static network structures, but there are two things worth noting. First, every output of the firing actor is distributed using the existing network structure. Second, when changing the network structure, the new connections are first added and then the connections to be removed are removed, so that removal takes precedence over addition of connections in cases of conflict.

2.5 Single-token actors

So far, we have developed our notions for the very general class of actors presented in section 2.2. In the following, we will specialize this general notion and will thus arrive at a subclass of actors that will form the basis for our definition of visual language semantics.

The dataflow actor as defined above is a very powerful and general entity. In particular, the family of functions P_σ determining the active prefixes of its input can perform arbitrarily complex operations on the input sequences, in other words the property of being an active prefix may be a far from trivial one, and may depend on the values of input prefixes from any number of input ports. In many cases, however, one would like actors to act on minimal amounts of information.

Another aspect of the way active prefixes are determined is that this effectively means that an actor may look at data, and then choose not to process it—without 'consuming' it. It is the actor that grabs a piece of data and then does something with it—or refuses to act on it, leaving the input as it is. In many applications, however, one would like to model a notion of asynchronous *sending* of data, i.e. the receiver (the actor in our model) has nothing to say about *whether* it accepts the data, its job only consists of handling any input that is sent to it. This is the basic communication mechanism that our model will be based on, and we will now embed this into our actor framework.

In this section we show how the actor definition can be specialized to characterize a subclass of actors that never refuse data. The easiest way to do this is to have them accept every possible single piece of data, i.e. every possible (available) token. We will call this kind of actor a *single-token (dataflow) actor* and we define it by requiring that the set of active prefixes be exactly those tuples of sequences, where one element is a single-token prefix of one of the input sequences, and all other elements are λ, which we write for the i-th input sequence as $(\lambda_{i-1}, a, \lambda_{m-i})$, assuming of course that the i-th input sequence is not empty, and that a is its first token.

Def. 10: (Single-token dataflow actors) *A dataflow actor*

$$\mathcal{A} = (\Sigma, \sigma_0, (P_\sigma)_{\sigma\in\Sigma}, (\tau_\sigma)_{\sigma\in\Sigma},)$$

is called a single-token (dataflow) actor *iff*

$$\forall \sigma \in \Sigma, s \in S^m : P_\sigma s = \{p \mid p = (\lambda_{i-1}, a, \lambda_{m-i}), a \in A, i = 1..m, p \sqsubseteq s\}$$

Note that this definition in particular implies that the set of active prefixes is the same for each state, i.e. the property of whether some input activates an actors does not depend on the actor's state. Also, an actor always consumes exactly one token per firing (we will say that it 'fires on' that token), which in particular means that it cannot fire in the absence of tokens.

Fig. 21 shows the relation between actors, extended actors and single-token actors. Note that single-token actors are a proper subclass of actors. This is in contrast to extended actors, where the actor definition was modified by adding a notion of network structure to it and the ability to change it. One important property of a single-token actor is that it is not selective on its input—as long as tokens are available, it is completely indifferent as to which of its input sequences it takes the token for the next firing from. This means that, e.g., the non-deterministic merge actor in Ex. 1 is a single-token actor, while neither of its deterministic versions is.

A consequence of this lack of selectivity on input tokens is the fact that single-token actors cannot 'recognize' the end of an input sequence—as would

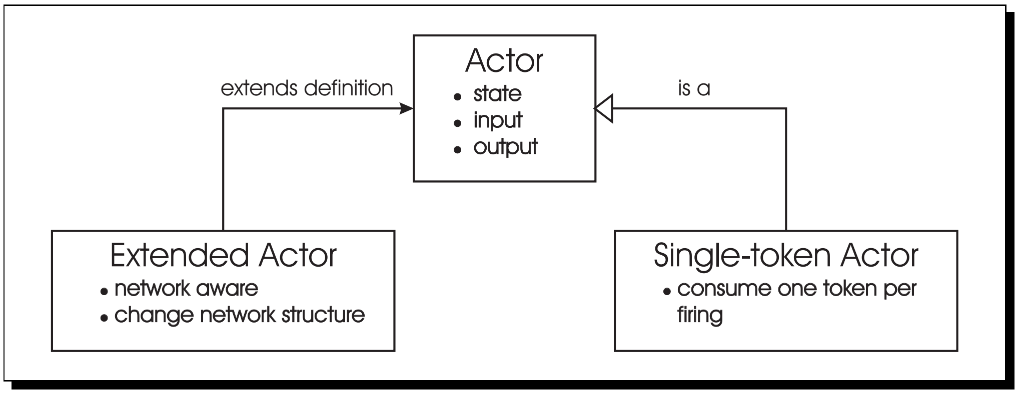

Fig. 21: Single-token actors, actors, and extended actors.

be necessary to realize the deterministic merge actors above. As can be seen below (we repeat the prefix function from Ex. 3), the active prefix function, in e.g. state Q_1, only selects a non-empty prefix of its second input sequence (i.e. some (λ, x)) if the first was empty. This in turn means that if the transition function τ_{Q_1} is fired on this prefix it can 'assume' that the first input sequence is empty and act accordingly.

$$P_{Q_1} : (s_1, s_2) \mapsto \begin{cases} \{(x, \lambda)\} & \text{with} \quad s_1 = x + s', x \in A, s' \in S \\ \{(\lambda, x)\} & \text{with} \quad s_1 = \lambda, s_2 = x + s', x \in A, s' \in S \\ \emptyset & s_1 = s_2 = \lambda \end{cases}$$

$$P_{Q_2} : (s_1, s_2) \mapsto \begin{cases} \{(\lambda, x)\} & \text{with} \quad s_2 = x + s', x \in A, s' \in S \\ \{(x, \lambda)\} & \text{with} \quad s_2 = \lambda, s_1 = x + s', x \in A, s' \in S \\ \emptyset & s_1 = s_2 = \lambda \end{cases}$$

So clearly, not every actor can be expressed as a single-token actor. What do we gain from restricting the more general actor class in this way?

Obviously, single-token dataflow actors effectively relinquish this kind of control over the consumption of their input, thus introducing a degree of nondeterminism and making their functioning dependent on the (undetermined) order in which they consume their input tokens. However, a different way of looking at this is that single-token actors accept any input at any time, and can thus be viewed as units that can be driven *from the outside* by sending them tokens in an asynchronous fashion, inducing them to make state transitions or to produce output tokens. This is a salient feature of our model of computation, viz. the fact that in a network of actors, the order of token consumption is beyond the control of an actor and is only constrained by the availability of tokens. We will use this property by controlling the availability of tokens for actors and thus driving the computation according to some notion of *schedule* (cf. section 2.6).

Another property of single-token actors is that their prefix function P_σ exhibits a kind of monotonicity in the sense that if for any two tuples of input sequences s_1, s_2 we have $s_1 \sqsubseteq s_2$, then in every state σ we have

$$P_\sigma \; s_1 \subseteq P_\sigma \; s_2$$

This implies that adding new tokens to the input never removes active prefixes. This is not so for general actors. Consider for example the deterministic merge actor in Ex. 2: The input sequences (ab, λ) result in the active prefix set

$$P_Q \; (ab, \lambda) = \{(a, \lambda)\}$$

Adding a token, say c, to the second input sequence, however, gives the input tuple $(ab, c) \sqsupseteq (ab, \lambda)$, and here we have

$$P_Q \; (ab, c) = \{(a, c)\}$$

which does not contain the formerly active prefix (a, λ).

This monotonicity of the prefix function applies to the actor behavior (in terms of state transitions taken and output produced) as well: adding more tokens to the input streams allows the actor to make additional steps, thereby possible producing additional output. Note, however, that this does not imply the simple monotonicity of deterministic actors, because adding tokens may not only add to output produced for some shorter input, it may start off completely new branches in the tree of possible firing sequences, possibly leading to output sequences for which the shorter input produced no prefix. The non-deterministic merge actor from Ex. 1, for instance, produces the unique output (ab) when run on the input tuple (ab, λ). Running it on (ab, c) produces, e.g., (abc) (which the former output certainly is a prefix of), but also (cab) and (acb), none of which has a prefix in the output of the actor for the shorter input tuple.

In other words, we have two dimensions of monotonicity in single-token actors when adding tokens to their input. First, all existing output may be extended, and second, the set of all possible outputs might increase in size and be augmented by output tuples that have no prefix in the output generated from shorter inputs.

Actors in general need not exhibit this kind of monotonicity. Consider the outputs of the deterministic merge actors in Ex. 2 and Ex. 3. For the input tuple (ab, λ), their output, too, is the unique (ab), while for the longer input (ab, c) both produce the unique (acb). The former output sequence is obviously not a prefix of this.

2.6 Scheduling and a notion of time

So far, execution within the network was driven exclusively by the availability of data tokens. The basic feature of our model of computation so far was that the 'consumption' of a token from some input stream triggered the state transition inside an actor and also possibly led to the production of new tokens. In

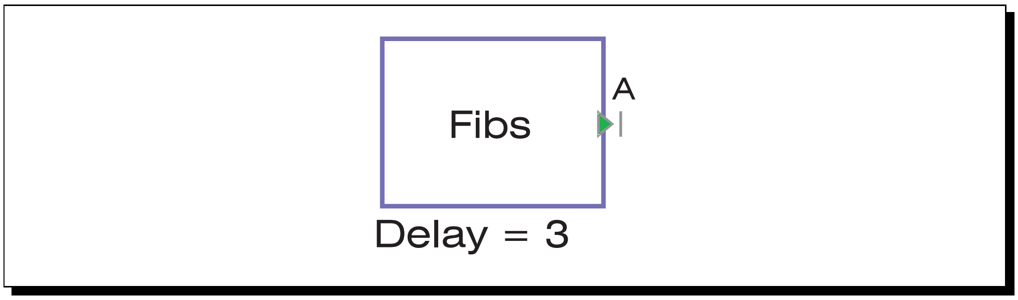

Fig. 22: A timed actor.

the definition of *run* (Def. 9) we identified several sources of potential nondeterminism, leading, in general, to a set of different runs from a given initial state.

Often one would like to constrain these sources of nondeterminism by specifying a *schedule* in which potentially concurrent activities (concurrent at least as far as the availability of data is concerned) are executed in order to model scheduling policies of some system.

Another very common area were the initiation of activities (read: the firing of actors) may depend not only on the availability of data are discrete event simulation models [113]. Here, activities often conceptually consume (virtual) *time* and thus need to be executed in some order that depends on their relative 'speeds' in addition to the availability of data to be processed. A salient feature of these kinds of models is that coordination between actor activities in principle needs to be global in the sense that no part of the system may be ignored when deciding which actor is to fire next. Naturally, the network structure may be used to impose a hierarchy onto the set of communicating actors and thus make the scheduling hierarchical. This section will introduce a technique that allows us to construct these kinds of models based on the framework outlined above.

2.6.1 The scheduling problem

First consider the actor in Fig. 22. We will think of it as producing the Fibonacci numbers in proper sequence, one per firing. However, in addition to this it has a *temporal* aspect in that before each firing it has a conceptual delay of 3 (virtual) time units.[7] We will, n this example for the sake of simplicity, assume time to be 'measured' by real numbers, as well as that the system starts to run at time 0.

Thus, if we write $a@t$ to denote that value a was produced at time t, the sequence of outputs produced by that actor would be

$$1@3, 1@6, 2@9, 3@12, 5@15, 8@18, 13@21, ...$$

More accurately, this means that there were firings at times 3, 6, 9, etc. and that these produced the indicated outputs (so it would really be more accurate

[7] We will henceforth omit the qualification 'virtual' for time, since in the context of a model of computation this will be the only concept of time we will be concerned with.

to list the actor firings and the time when they happened). Of course, with just a single actor, time has little influence on the order in which things happen (although it might still be an interesting measure from an evaluation point of view—e.g. to answer question about the speed/throughput/latency etc. of an actor). With more than one actor, the temporal behavior of the actors may influence the functional aspects of the system.

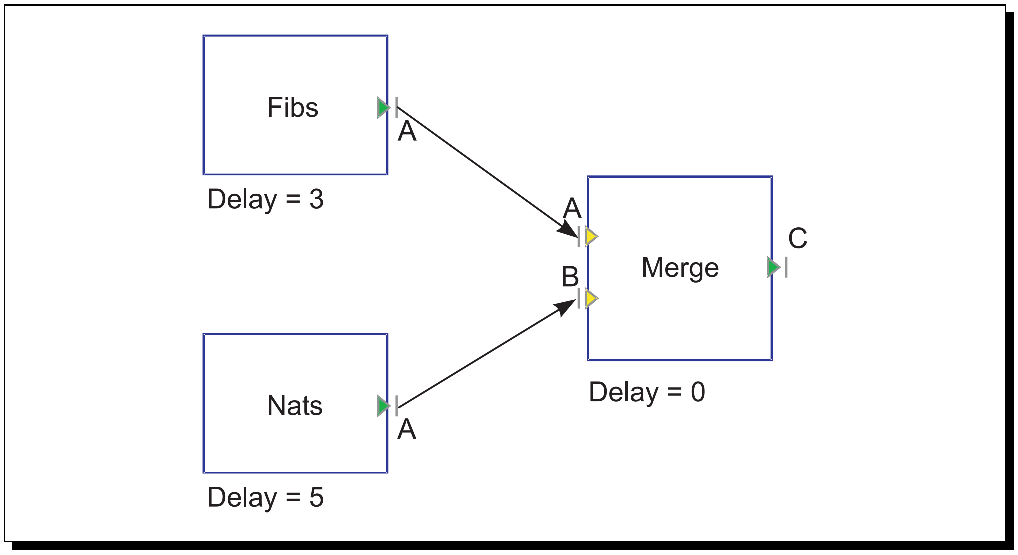

Fig. 23: A small network of timed processes.

Consider for example the system in Fig. 23. It contains the actor from Fig. 22 and another, similar one, which produces the natural numbers in sequence, each with a delay of 5 time units. Both outputs are merged by a *Merge* actor, which is instantaneous. Obviously, if the delays attributed to the $Fibs$ and $Nats$ actors are taken to model their behavior in physical time, we would expect the output of the *Merge* actor to be something like

$$1@3, \underline{1@5}, 1@6, 2@9, \underline{2@10}, 3@12, 5@15, \underline{3@15}, 8@18, \ldots$$

This sequence results by merging the underlined outputs of the $Nats$ actor into the appropriate places (as indicated by the timestamps) of the previous output sequence of the $Fibs$ actor. Note, incidentally, that at time 15 both actors fire, and thus *Merge* can choose between two tokens at that point.

However, our current concept of executing networks of actors (as defined in section 2.4) does not provide us with any way of controlling the sequence of firing of the actors in the way outlined above. Consider, for example, the $Fibs$ actor. A possible definition of the $Fibs$ actor might be the following:

Ex. 6: (The *Fibs* actor, without a concept of time)

$$\Sigma = \mathbb{N} \times \mathbb{N}$$
$$\sigma_0 = (1, 1)$$
$$P_{(a,b)} : () \mapsto \{()\}$$
$$\tau_{(a,b)} : s \mapsto \{((b, a+b), a)\}$$

This actor has no input ports, and it never reaches a state in which it could not fire. This means that in all its states (a, b), its set of active prefixes is $P_{(a,b)}\ () = \{()\} \neq \emptyset$. This in turn means that one possible run of the above network according to Def. 6 or Def. 9 would be to simply fire it an infinite number or times, without any other actor firing in between. Of course, the temporally 'well-behaved' run giving rise to the sequence above is also a possible run under these definitions, but the problem is that our concepts so far provide no mechanism to select runs that respect the temporal properties of the actors, and in fact we have no way to even formulate temporal aspects of actors.

So what is required is a way of expressing temporal properties of actors and a mechanism that allows us to coordinate the firings of the actors in a network in such a way that the temporal order is respected. In the following we will present an approach that allows us to do this. Specifically, it consists of the following parts:

- the concrete notion of *time* that we are dealing with,
- some requirements on the behavior of the actors as well as on the system structure,
- how to enforce a scheduling among the actors that respects the firing times they are scheduled for.

The pivotal concept of this mechanism will be the *schedule* which contains actors that are ready to fire at some future point in time. The execution of the system will be done in two alternating phases:

1. Choose one actor which to fire *next*, i.e. an actor which is scheduled at a point in time such that no other actor is 'earlier', and fire it.
2. The actor may have produced tokens, which activate other actors. Fire them, in the usual manner, until the system is 'dead', i.e. no actor can fire (we require this procedure terminates after a finite number of steps). See, if any actors have scheduled themselves for future firing (again, this will be made explicit below), and update the schedule accordingly. Then go to step 1.

Conceptually, the firings in step 2 all happen at the same time, i.e. whatever happens in step 2 is instantaneous. Time is only (potentially) advanced in step 1, when the next actor to be scheduled is chosen.[8]

[8]Out model of time has been inspired by the time concept of Colored Petri Nets [70].

2.6.2 Scheduled actors and systems

Before we come to the scheduling technique proper we need to be somewhat more precise about our concept of time. Although the term 'time' suggest 'physical time', which (at least in its Newtonian conception) is totally ordered and continuous, the following definitions will not commit us to any specific notion of time, except that 'points in time' (which we will call 'tags', see below) be partially ordered. There are several reasons for this decision:

- The central function of the notion of time is to coordinate actions (the firing of actors) in the system. Using a totally ordered concept of time might lead to overspecification of a system in the sense that it unnecessarily forces order among actions that we would like to model as unordered.
- Many existing system models use partial orders to describe the coordination of actions, e.g. in distributed systems [75], or in Petri nets [89], others are based on a totally ordered but discrete time concept [16]. If possible, we would like to be able to incorporate these models into our actor framework.
- Totally ordered notions of time remain available to us if we need them, since they constitute a special case of our model.[9]

So, following [78], in this framework, time will be represented abstractly by a partially ordered set of *tags*. As a consequence each token will have a tag component as follows:

Def. 11: **(Tag, tagged value)** *A* tag *is an element of a partially ordered* tag set $(T, \prec)$. *If* $t_1 \prec t_2$ *we say that* t_1 *is* (strictly) before t_2.

A set of tagged values *is some set* A *and a function* $\theta : A \longrightarrow T$*, assuming* $(T, \prec)$ *is a tag set. Furthermore, there must be a function* $\upsilon : A \longrightarrow V$ *to some non-empty set of values* V *such that* (A, θ, υ) *is a product.*

The scheduling mechanism outlined above depends on two interactions with the actors: When its time has come, it must be able to fire an actor, and actors must be able to communicate their scheduling requests to it. The usual way to fire an actor is to provide it with input tokens that activate it (and then, of course, fire it on those tokens), while the normal way to get information from an actor is by one of its output ports. Thus for an actor to able to be scheduled it must fulfill the following requirements:

- The actor must have two designated ports used to initiate firing it and to send scheduling requests. Fig. 24 shows two actors from the system in Fig. 23, exhibiting their additional ports needed to schedule them (the input used for firing them is labeled $fire$, the output where they produce scheduling requests is labeled $sched$).

[9] In fact, our implementation of these concepts is based on a totally ordered time concept.

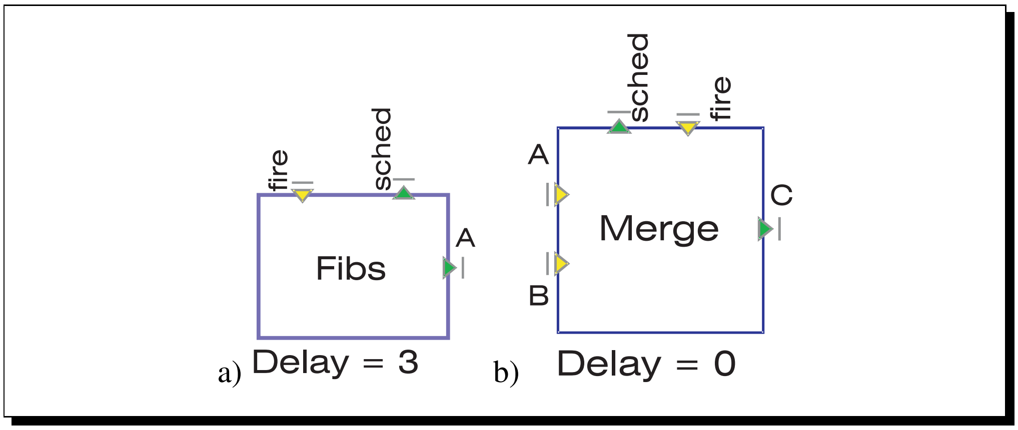

Fig. 24: Actors exhibiting their scheduling ports.

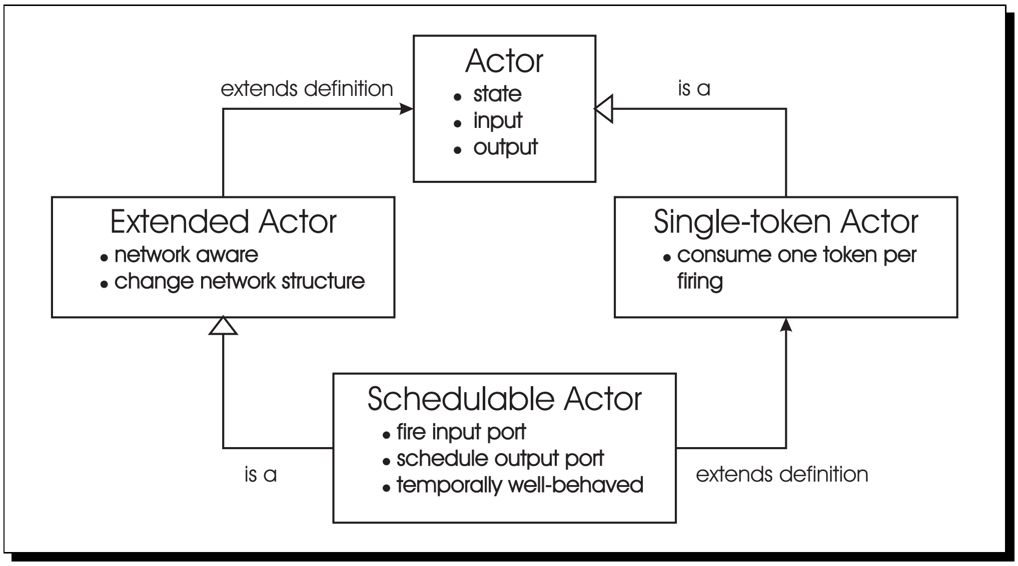

Fig. 25: Schedulable actors and their relation to other kinds of actors.

- In addition, the input/output behavior of the actor must be consistent with the order of the tags that denote temporal progress. In particular, all tags except those of tokens produced at the scheduling output must be equal to the one that was used to fire the actor. Also, the actor must be firable by any single token along the $fire$ input. The tag of tokens produced at the $sched$ output must be greater than or equal to the tags of the tokens it fired on. In order to reduce possible inconsistencies with different time stamps of simultaneously consumed tokens, we will require schedulable actors to be single-token actors in general.

Fig. 25 shows how this new kind of actor, which we call *schedulable actor*, fits into the concepts introduced so far. It is really a special kind of extended actor, that behaves in a single-token fashion, i.e. always consumes exactly one token per firing.

Def. 12: (Schedulable actor, schedulable system) *Given a system of extended single-token actors and a set of tag values $(T, \sqsubseteq)$. We call a structure*

$$(\Sigma, \sigma_0, (P_\sigma)_{\sigma\in\Sigma}, (\tau_\sigma)_{\sigma\in\Sigma}, p_{fire}, p_{sched}, t_0)$$

such that $(\Sigma, \sigma_0, (P_\sigma)_{\sigma\in\Sigma}, (\tau_\sigma)_{\sigma\in\Sigma})$ is an extended single-token actor $\mathcal{A}_k$, p_{fire} and p_{sched} are two functions and $t \in T \cup \{\infty\}$ a schedulable actor *with* starting time t_0 *iff*

- $p_{fire} \in P_k^{in}$
- $p_{sched} \in P_k^{out}$
- *If a is the token on which the actor fires (i.e. $(\lambda_{i-1}, a, \lambda_{n_k-i})$ is an active prefix which the actor fires on), then any token v produced at any output port $p \neq p_{sched}$ must be such that $\theta a = \theta v$, while for any token r produced at p_{sched} we require $\theta a \preceq \theta r$.*
- *Furthermore, if N is the set of actor indices of actors created in that firing, then the following must hold:*
$$\forall i \in N : \theta a \preceq t_{0,i}$$
- *If m, m' are two tokens such that $\theta m = \theta m'$ and s, s' two input tuples such that $p_{fire}\ s = m$ and $p_{fire} s' = m'$ and for all input ports $p \neq p_{fire}$ we have $p\ s = p\ s' = \lambda$, then for any state σ the following must hold:*

$$\tau_\sigma \alpha_i s = \tau_\sigma \alpha_i s'$$

A system consisting exclusively of schedulable actors is a schedulable system.

The first two conditions require p_{fire} and p_{sched} to be an input and an output port of the actor, respectively. The third condition states that the actor be *temporally well-behaved* in the following sense. Since it is a single-token actor, it consumes one token, say a, in every firing. All its output tokens are required to carry the same tag as the token is consumed, that is θa, except for tokens coming from it scheduling output: their tags are required to greater than or equal to the tag of a. The fourth condition requires something similar for the actors created in the firing: their starting time tag must be greater than or equal to the current firing time tag, i.e. the tag of the token the actor fired on.

The last condition simply states that the actor must not depend on the value component of the tokens arriving at its firing input: any two tokens at the p_{fire} input port with the same time stamp should produce the same value when fed to the transition function.

The starting time t_0 is a tag that identifies when the actor schedules itself for first firing—this does not necessarily imply that it will not fire before t_0. In particular, if it receives any data from other actors it is connected to before this time, it will fire on it, and even may change its scheduled firing time. Note that t_0 can assume the special value $\infty \notin T$, which denotes an unscheduled actor that remains inactive unless it receives input from other actors in the system. We will discuss this in more detail below.

Coming back to the $Fibs$ actor, a version of which we modeled without a notion of time in Ex. 6, we are now able to modify it so that it is schedulable and correctly behaves in time.

Ex. 7: (The *Fibs* schedulable actor, including its delay)

$$
\begin{aligned}
&t_0 = 0\\
&p_{fire} : (s) \mapsto s\\
&p_{sched} : (s_1, s_2) \mapsto s_2\\
&\Sigma = \mathbb{N} \times \mathbb{N}\\
&\sigma_0 = (1, 1)\\
&P_{(a,b)} : (s) \mapsto \begin{cases} \{(x)\} & \textit{for} \quad s = xw, x \in A \\ \emptyset & \textit{otherwise} \end{cases}\\
&\tau_{(a,b)} : (x) \mapsto \{((b, a+b), (a', y), \emptyset, \emptyset, \emptyset)\}\\
&\qquad \textit{such that} \quad \nu a' = a, \theta a' = \theta x\\
&\qquad \textit{and} \quad \theta y = \theta x + 3
\end{aligned}
$$

By defining p_{sched} we have identified the second output port of the actor as the one scheduling messages are sent from.[10] It has only one input port, which therefore must be the p_{fire} port. The firing function produces two output tokens, (a, y): the first is the data token, the next Fibonacci number in the sequence (here we have to construct a token a' such that a is its value projection, and θx (the tag of the fire message) its tag, but the second is the schedule message, scheduling the actor at $\theta\ x + 3$, i.e. 3 time units in the future.

2.6.3 Executing schedulable systems

At this point we come to the execution of a schedulable system. As mentioned above, this happens in two phases:

1. Inject a token into the $fire$ port of an actor. Of course, this actor is chosen as the next to be fired according to the current schedule.

2. This activates this actor. Now run the system until no actor is activated. Then repeat the cycle.

This obviously assumes that the run in the second step terminates (when no actor is activated any more). So far, however, our definition of *run* (Def. 9) is an infinite one. Therefore we must define a finite run.

[10]To simplify the notation we have taken the liberty of defining these ports relative to the actor, rather than relative to some imaginary system.

Def. 13: (Finite run, result of run) *Given a system of extended actors $\mathcal{A}_i$, and an initial state $(J_0, \longrightarrow_0, s_0, (\sigma_{i,0})_{i\in I}, \lambda_N)$. We call a run* finite *if there is a k such that in state $(J_k, \longrightarrow_k, s_k, (\sigma_{i,k})_{i\in I}, r_k)$ the set of activated prefixes of all actors is empty, i.e. no actor is can fire.*[11]

In that case, we call $(J_k, \longrightarrow_k, s_k, (\sigma_{i,k})_{i\in I}, r_k)$ the result *of the run.*

A run that is not finite has the result $\perp$, which is distinct from any state of the system.

It is important to realize that *running* a system from a given state may yield any number of results, including 'none', denoted by $\perp$. Therefore when we run a system from a particular state we are in effect selecting one of possibly many runs and looking at the result. If the result is $\perp$, then the run never finished, and hence the system does not terminate. It is of course perfectly possible for a system (in a given state) to have both, runs that terminate and those that do not.

Now it is necessary to define the state of a schedulable system. This will be the state of the system as defined in Def. 8, plus a *schedule*. This schedule will be defined as a function from I (the index set of the actors) to $T \cup \{\infty\}$, the tags plus an 'unscheduled' special tag not in T.

Def. 14: (Schedule, firable actors) *Assume a schedulable system of actors $\mathcal{A}_i$, with $i \in I$. We will call a function $\phi : I \longrightarrow T \cup \{\infty\}$ a* schedule. *Extend the partial order on T to $T \cup \{\infty\}$ such that for all $t \in T$, $t \prec \infty$. The set*

$$I_{firable} = \{i \in I \mid \phi i \neq \infty \wedge \neg\exists j \in I : \phi j < \phi i\}$$

is called the set of firable *actors.*

The state of a schedulable system then becomes essentially the combination of a schedule with a state of system of extended actors. Additionally, we will allow the state to be undefined, denoted by the special value $\perp$. The scheduled system will go into this state if any of its runs do not terminate.

Def. 15: (State of a schedulable system)

For any schedulable system of actors, if $(J, \longrightarrow, \lambda^M, (\sigma_i)_{i\in I}, r)$ is a state of the system of actors according to Def. 8, $J \subseteq I$, and ϕ is a schedule, then

$$(\phi, J, \longrightarrow, (\sigma_i)_{i\in I}, r)$$

is a state of the schedulable system *of actors, iff*

$$\forall i \in I \setminus J : \phi\, i = \infty$$

Additionally, that system can be in a state written as $\perp$, different from any of the previously defined states.

[11]This obviously implies that each following state is identical to the k-th state.

This definition demands that the scheduled time for each non-active actor is ∞, which implies $J \subseteq I_{firable}$, i.e. every actor that is not active is not firable and thus will not be fired by the scheduling mechanism that we will describe below.

As a consequence of the assumption that all actors are single-token and that no actor has an active prefix, we can assume that our input sequences are all empty, which is why we need not keep the input sequences as part of the system state.

The execution of a schedulable system starts with a schedule that is constructed from the starting times of all actors. The execution then proceeds as follows:

Def. 16: (Execution of a schedulable system)

Assume a schedulable system $(\mathcal{A}_i, p_{fire,i}, p_{sched,i}, t_{0,i})$ *with* $i \in I$ *and an initial state*

$$S_0 = (\phi_0, J_0, \longrightarrow_0, (\sigma_{i,0})_{i\in I}, \lambda_N)$$

such that for all $i \in I$*,*

$$\phi_0 : i \mapsto \begin{cases} t_{0,i} & \textit{if} \quad i \in J_0 \\ \infty & \textit{otherwise} \end{cases}$$

An execution *of the schedulable system is a sequence of states* S_k *such that for any* S_k *its successor state* S_{k+1} *is computed as follows:*

- *If* $S_k = \bot$*, then* $S_{k+1} = \bot$*.*
- *Otherwise,* $S_k = (\phi_k, J_k, \longrightarrow_k, (\sigma_{i,k})_{i\in I}, r_k)$*. If* $I_{firable} = \emptyset$*,* $S_{k+1} = S_k$*.*
- *Otherwise, there exists a firable actor. Now we perform the following steps:*
  1. *Choose some* $i \in I_{firable}$*. Let* $s \in S^M$ *be defined by* $p_{fire,i}s = a$*, for some* a *such that* $\theta a = \phi i$*,* $ps = \lambda$ *for any port* $p \in P^{in} \setminus \{p_{fire,i}\}$*.*
  2. *Run the system of actors on state* $(J_k, \longrightarrow_k, s, (\sigma_{i,k})_{i\in I}, \lambda_N)$*. If the result of that run is* $\bot$*,* $S_{k+1} = \bot$*.*
  3. *Otherwise, the result is a state* $(J', \longrightarrow', s', (\sigma'_{i,k})_{i\in I}, r)$*. Since all actors are single-token actors,* $s' = \lambda_M$*. The set* N *of newly created actor indices is* $N = J' \setminus J_k$*. Now define*

$$\phi_{k+1}j = \begin{cases} \phi_k j & \textit{if} \quad p_{sched,j}r = \lambda \wedge j \notin N \\ t_{0,j} & \textit{if} \quad p_{sched,j}r = \lambda \wedge j \in N \\ \theta a & \textit{with} p_{sched,j}r = wa, w \in A^*, a \in A\} \end{cases}$$

$$J_{k+1} = J'$$

$$\longrightarrow_{k+1} = \longrightarrow'$$

$$\sigma_{i,k+1} = \sigma'_{i,k}$$

$$r_{k+1} = r_k + r$$

The key to this definition is the way that the new schedule, ϕ_{k+1} is defined. If no scheduling token is received by an actor, its next firing time remains unchanged if the actor was active before. If it was created during this iteration, its starting time is taken instead (note that temporal well-behavedness of schedulable actors ensures that this is not smaller than the current firing time). Otherwise, i.e. if a scheduling token came from the actor, we take the *last* scheduling token, and its time stamp will become the actor's new next firing time. Here the monotonicity condition of the scheduling output of schedulable actors ensures that this time will not be in the past.

2.7 Summary—scheduled systems of discrete event components

This completes our model of computation. The systems we will be looking at will be schedulable systems of actors. We will also refer to a single actor in such a system as a *discrete-event component*, for its discrete-event behavior (due to discrete firings) and its component-like interface (e.g. [31]) to its environment. In the remainder of this work, this will be the only kind of actors we will be concerned with, hence we will use the terms 'discrete-event component', 'actor', and 'component' interchangeably, unless stated otherwise.

As mentioned at the beginning of this chapter, these discrete-event components will be the central concept in our semantics description. The following chapters will describe a technique for constructing an actor from a visual program, or *picture*. One of the consequences of this approach is that once that visual program is mapped into an actor it becomes abstract in the sense that the the program itself, i.e. its syntax, becomes invisible, thus allowing us to combine any kind of visual program with any other kind, as long as there exists such a mapping into the domain of discrete-event components.

As we will see by studying a few applications, there is a large variety of visual languages for which this is the case, including those mentioned in the introduction. However, first we need to describe pictures in a form that allows us to manipulate them by an automaton, i.e. we must define them as a formal structure.

2.8 Discussion and related work

The main purpose of our model of computation is the representation of structured, concurrent systems—in a way that facilitates operational modeling of individual actors. In this section we will discuss features of our model and important design decisions, and contrast it with the choices made in other approaches. We will in particular focus on dataflow actors as defined in [77], the class of approaches we will sum up here as 'process algebraic' (most notably CCS [81], CSP [61], and the π-calculus [82]), DEVS [113] and Agha's actors (sometimes referred to as 'agents') [4, 5].

As mentioned at the beginning of this chapter, the concept of actor presented here has its origin in the dataflow actors in [77], which it extends by a notion of state. Key concepts have been borrowed from dataflow actors, most notably the firing concept and the notion of ports. However, we have generalized the notion of network structure, which in our model is an n-m relation between output ports and input ports, as opposed to dataflow networks, where the network structure allowed at most one input port to be connected to any one output port and vice versa. For static networks, this is not particularly limiting, as one can always conceptually insert fork or merge actors to distribute output from or collect input to a port. In the case of dynamically changing network structures, however, this would lead to artifacts which seem intractable. In particular, when establishing a new connection, one might need to implicitly create a new actor and wire it up to achieve the desired effect. The price for allowing n-m relations between ports is that this produces a new source of non-determinism in the execution of a system (cf. Def. 9), viz. the simultaneous output of tokens at different ports in the same firing of the same actor. When these outputs are merged, their order is not determinate. It is our view, however, that this is, in a sense, a real indeterminacy of the model itself, that a general model of computation should not resolve. If at all, it would be the responsibility of the model to make this behavior determinate, e.g. by explicitly inserting a merge actor.

One of the novel features of our approach is the way we model system communication structures. We adopt the common notion of ports (occurring in similar forms in such diverse approaches such as dataflow actors, DEVS, but also in denotational frameworks [3, 30, 32, 78]) as the interface between actor and environment. Unlike other approaches, ports as well as actors themselves are first-class entities, connected by an arbitrary relation between output and input ports. Actors need to be able to identify ports and connect and disconnect them. However, actors do not have read-access of the communication structure itself, i.e. they cannot inquire about the presence or absence of a connection. The rationale behind this design was to decouple the behavior of an actor from who it was communicating with, allowing an actor to be written in full ignorance of the 'outside' world but still interchanging information with it. For an actor, its ports form its view of the rest of the system. Likewise, from the outside an actor is exclusively viewed in terms of the communication that happens via its input and output ports. Disallowing an actor to obtain knowledge about the communication structure is necessary to maintain strict encapsulation and separation of the states of the actors from one another. Otherwise, the communication structure would become a 'global' state, and actors could influence each other by other means than just communication (i.e. they could exchange information by establishing or removing connections).

In static networks, the communication structure, or ports, need not be represented to the actors at all, neither need the actors themselves be first-class—which is why the work on dataflow actors [77], focusing on static network structures, does not address these issues. Process algebraic approaches have an abstract concept of their communication medium in the form of 'channels'. This is not equivalent to ports or connections, because a channel does not necessarily belong to an actor/a process, nor does it constrain the set of processes communicating via the channel, allowing in general any number of processes to read and write to a channel. Also, they need not be first-class objects—the π-calculus extends CCS by first-class channels, which means they can be dynamically created and passed as values inside of messages. Although process algebras support the dynamic creation of processes, they are not data values that may be passed as messages to other processes.

Agha's actor model, on the other hand, makes actors ordinary data objects. However, it does not formalize a notion of network structure, so that actors need to explicitly address (and therefore know) any actor they are sending a message to. DEVS builds models hierarchically from components that are connected at ports to form a 'coupled model', which in turn represents a component that can be integrated into other models. The connection structure is explicitly represented as part of the coupled model, but it is not first class, and neither are ports. This implies that models may not change the connection structure—it is static.

Like dataflow actors, our communication model is an asynchronous one. Actors can always produce output, and the inputs of actors can be thought of as being equipped with unbounded buffers. On the other hand, actors need not, indeed cannot, 'wait' for some specific input to arrive. An actor has no control over the order in which input is delivered to it. Other models of computation (e.g. most that could be called 'process algebras' such as CSP and CCS) choose synchronous communication as the underlying communication paradigm. Synchronous communication can be simulated by asynchronous communication primitives [6, 31], so in principle the latter should suffice in a semantical model. Many practical modeling tasks, however, are best dealt with using a synchronous communication model, so a practical modeling language might usefully include both, in spite of the conceptual redundancy [5].

The notion of time used in our model of computation was inspired by the tagged signal model in [78]. However, while there a signal (roughly corresponding to our sequences of tokens) is a function from (time) tags to values, we allow any number of values to carry the same time stamp in any one sequence. Furthermore, the order of tokens in a sequence and the time stamps attached to these tokens are only loosely related, especially in scheduling messages. In fact, the relation between the order of tokens in a token sequence and their time stamps is a *derived* property, following from the temporal behavior of a schedulable actor (Def. 12 on page 44) and the mechanism of executing a schedulable system (Def. 16). The notions of causality and temporal order are realized in these definitions.

Simulation modeling languages and formalisms are a usually operational description techniques that involve some notion of time, most often a model of continuous physical time. The most important example is DEVS. DEVS components have a built-in notion of continuous time, which is realized by a function computing the time of the next internal state transition. Internal transitions are those happening without external input, just due to the passage of time. External input, on the other hand, is processed immediately. Both kinds of transitions may lead to a new state and output.

By contrast, the temporal behavior of our actors is simply a specialization of atemporal behavior (Def. 12). Every schedulable actor is a valid 'atemporal' actor. The scheduling and firing of an actor is realized in our model of computation by using the communication mechanisms—port, passing of tokens, firing. In a sense, our actor model reflects the dichotomy between DEVS' time-consuming internal and instantaneous external events in the distinction between input on the *fire* input put and input on other input ports. The big difference is simply that once again, the concept of time-consumption and the scheduling of state transitions is a derived notion, a notion being associated with a way of executing a system. Actors themselves have no notion of time.

3

Visual language syntax and semantics

This chapter will present our general approach to specifying the syntax and semantics of a visual programming language by an automaton that is parameterized by a representation of a particular visual program, or picture. This representation will be referred to as its *abstract syntax*, and the automaton is consequently an *abstract syntax interpreter*. We will subsequently embed this notion into the actor framework outlined in the previous chapter. As a first step, we will address the representation of pictures as abstract mathematical structures, which is then used as the basis for the interpretation process.

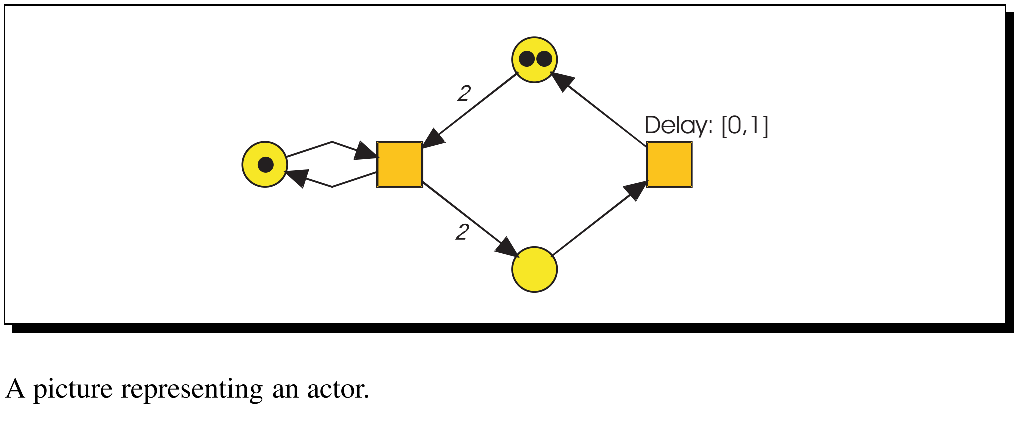

Fig. 26: A picture representing an actor.

3.1 Abstract syntax of pictures

3.1.1 An example

To illustrate the relation between concrete and abstract syntax that we will base out interpretation on consider the picture in Fig. 26, incidentally depicting a time Petri net [80]. The basic structure of this picture is that of a *graph*, i.e. a set of vertices and (directed) edges connecting them. Although this is not the case in this simple example, we will allow for more than one edge connecting the same two vertices (in the same direction), thus more generally the basic abstract structure of our pictures is that of a *multigraph*, which can be defined by a tuple

$$(V, E, s, d)$$

such that V and E are the disjoint sets of vertices and edges, respectively, and $s, d : E \longrightarrow V$ map an edge to its respective start or destination vertex.

However, this structure is clearly not sufficient to represent all relevant information[1] contained in the picture. For instance, we have vertices of different *kind* (representing the concepts of places and transitions in Petri nets [84, 88]), and vertices as well as edges can be inscribed with (different kinds of) textual annotations. [2]

These pieces of information we will consider as *attributes* of the multigraph structure, and it is 'connected' to it by means of an *attribution function* μ : $(V \cup E) \times A \longrightarrow U$, where A is the set of attribute *names* and U the set of all possible attribute values. A useful candidate for this set will probably contain the character strings, numbers, but maybe also more structured data such as lists etc.—and often also V and E themselves. To make the attribution function total, we assume that U contains an element $\perp$ denoting an 'undefined' attribute value. We will refer to edges and vertices collectively as *graph objects*, and we will say that some graph object x has an attribute value v for attribute a iff $\mu(x, a) = v$. Often it is convenient to talk about the complete attribution of some graph object x as a function $\mu_x : A \longrightarrow U$.

Fig. 27 shows the structure of an abstract syntax for the picture in Fig. 26 (leaving out all 'undefined' attribute assignments). The items on the left are the edges, those on the right the vertices. Note that each graph object has at least one attribute called *type*, which in the case of the vertices distinguishes between *Place*s and *Transition*s. Since we have only one edge type, *Arc*, this is the same for all edges. Some edges have an additional *Weight* attribute, some vertices

[1]We will pretend, for a moment, that this notion is actually an objective one, i.e. that we would know for any given picture what exactly this relevant information *is*, and which information is incidental, implying that the latter need not be represented in the abstract syntax. Of course, in general this depends on the semantics of the picture, and since our abstract syntax representation is a generic one, we will usually have a lot of structure represented in the abstract syntax that is not interpreted and thus irrelevant semantically.

[2]Additionally, there are of course graphical and geometrical informations (position, size, color, orientation, ...) that we will gloss over for a moment. It will become clear how these fit into the structure we propose here.

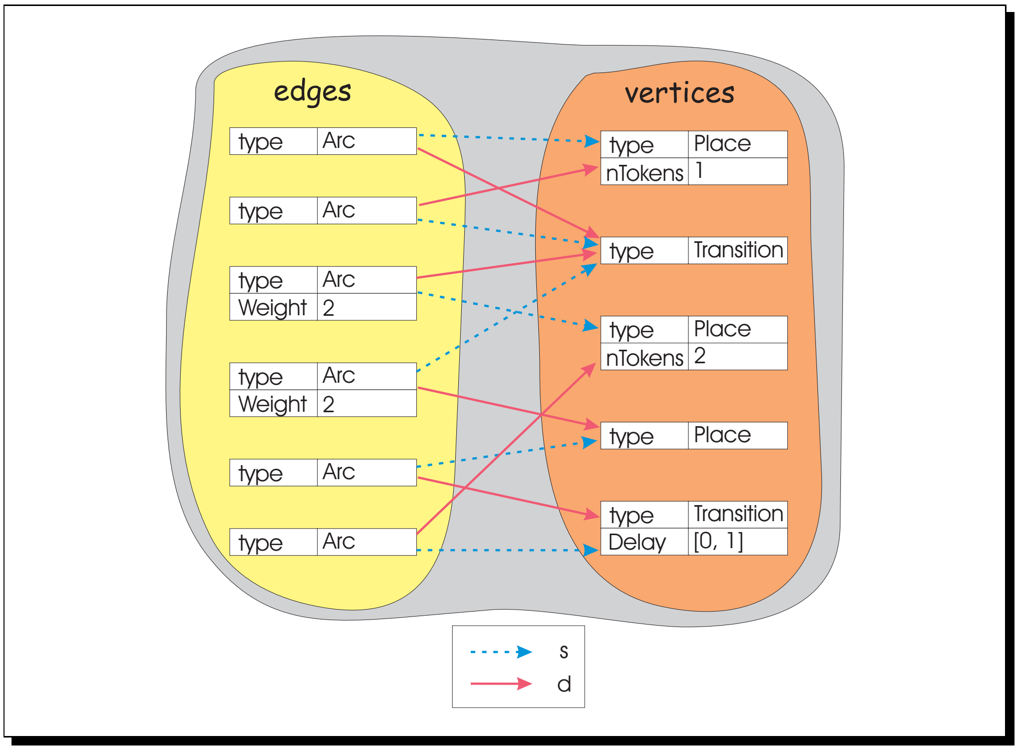

Fig. 27: An abstract syntax of the picture in Fig. 26.

have an *nTokens* attribute, and there is one that bears a *Delay* attribute. Finally, the arrows indicate the way the edges connect the vertices, distinguishing for each edge between the vertex it 'starts' from and the vertex it 'ends' at.

Incidentally, for representational purposes we broke the μ attribution function into its μ_x components for each graph object x, which is displayed as the little table representing the graph object. It is often conceptually convenient to view the attribution in this way, because the way some μ_x looks often depends on the specific graph object, in particular on its type. For instance, in the above example, only vertices of type *Place* (i.e. for which their $\mu_x(type) = "Place")$ could have an *nToken* attribute, while a *Delay* attribute only makes sense for *Transitions*. This kind of property, which is somewhat analogous to the static semantics of textual programming language, will be discussed in greater detail below.

3.1.2 Abstract visual syntax

As we have seen above, our abstract syntax is an attributed multigraph, where an attribution function attaches additional information to each graph object. The subsequent definition will also allow the graph itself to carry attributes. We will see an application of this in chapter 5. We adopted the concept of abstract visual syntax outlined in [41].

Def. 17: **(Abstract syntax of a picture/attributed graph.)** *Assume a set A of* attribute names *and a set U of* attribute values, *containing an 'undefined' value $\perp$. We* *define the* abstract syntax of a picture *as an* attributed graph*, i.e. as a structure*

$$(V, E, s, d, \mu, \overline{\mu})$$

such that V and E are disjoint sets of vertices *and* edges*, respectively (collectively called* graph objects*), and*

$$\begin{aligned} &s, d : E \longrightarrow V \\ &\mu : (E \cup V) \times A \longrightarrow U \\ &\overline{\mu} : A \longrightarrow U \end{aligned}$$

The set of all attributed graphs for a given A and U is written as $\Gamma(A, U)$.

As mentioned above, we will also write μ_x for the function defined by $\mu_x\ a = \mu\ (x, a)$, for any graph object x.

Obviously, by choosing A and U as needed we can make the structures attached to the edges and vertices of an attributed graph arbitrarily simple or complex.

However, not all such graphs are 'meaningful' as representatives of a given visual notation. For instance, in the Petri net language we used in the example above, it does not mean anything for a *Transition* type vertex to have a *Weight* attribute. Furthermore, the actual attribute values attached to graph objects may have to conform to certain syntactical rules—they are usually described by some textual grammar themselves. For example, we would like to exclude a *Weight* attribute value of "-4", since arc weights need to be positive integers.

Thus, for an attributed graph to be considered a representative of some visual language it needs to satisfy a set of predicates, say $\mathbf{P}$. An element $P \in \mathbf{P}$ then is a predicate, and some graph G is said to *satisfy* this predicate if $P(G)$ holds. If a graph satisfies all predicates in the $\mathbf{P}$, we will call it *well-formed*. A visual language is simply the set of all well-formed attributed graphs.

Def. 18: (Visual language) *Given a set of attributed graphs $\Gamma(A, U)$ and a set of predicates over these graphs $\mathbf{P}$, these define a visual language $\mathcal{L}(\mathbf{P})$ as follows:*[3]

$$\mathcal{L}(\mathbf{P}) = \{G \in \Gamma(A, U) \mid \forall P \in \mathbf{P} : P(G)\}$$

Here are some predicates describing well-formedness requirements for attributed graphs representing Petri nets in the way we sketched above. Of course, the collection is incomplete.

Ex. 8: (Some Petri net syntax predicates)

$$\begin{aligned} &P_1(V, E, s, d, \mu, \overline{\mu}) : \forall e \in E : \mu(se, type) = \text{"}Place\text{"} \Rightarrow \mu(de, type) = \text{"}Transition\text{"} \\ &P_2(V, E, s, d, \mu, \overline{\mu}) : \forall e \in E : \mu(se, type) = \text{"}Transition\text{"} \Rightarrow \mu(de, type) = \text{"}Place\text{"} \\ &P_3(V, E, s, d, \mu, \overline{\mu}) : \forall e \in E : \mu(e, Weight) \in \mathbb{N} \cup \{\bot\} \\ &P_4(V, E, s, d, \mu, \overline{\mu}) : \forall v \in V : \mu(v, type) = \text{"}Place\text{"} \Rightarrow \mu(v, nTokens) \in \mathbb{N} \cup \{\bot\} \end{aligned}$$

[3]Obviously, this is formally equivalent to just one predicate that is simply the conjunction of the $P \in \mathbf{P}$. We chose to present it in this way because it mirrors more closely the actual implementation and also allows us to talk about a graph violating a specific syntax predicate.

The first two predicates ensure that places are always connected to transitions and vice versa. The third requires arc weights to be positive (or undefined), and the fourth makes a similar requirement for the *nTokens* attribute of places.

From a practical perspective, these syntax predicates can be considered *preconditions* for the interpreter—graphs that are not well-formed need not be interpreted in any meaningful manner, thus an interpreter may assume that the syntax predicates hold for the structures it has to deal with.

We will now turn to the problem of, given such a graph $G \in \mathcal{L}(\mathbf{P})$, how to define what it *means* by what it (or rather the computational structure denoted by it) *does*.

3.2 Abstract syntax interpreters

We will define the semantics of a visual language such as Petri nets by giving an interpreter for it, its abstract syntax interpreter. However, defining such an interpreter is *not* equivalent to specifying a concrete actor—such an actor has a fixed number of input and output ports, it has a specific initial state, a specific state space and so on. All these are things that may, in general, depend on the specific visual program (represented by an attributed graph) that an interpreter will execute. The situation is roughly equivalent to the relation between an object and its class in object-oriented programming languages. There, the class represents a general description of the structure and the behavior of objects, and the programmer is concerned with describing the class, rather than the individual objects themselves. These are then created from the class description by a process called *instantiation*, which typically involves parameters describing specific details of the object's initial setup.

Similarly, when describing an interpreter for a visual language, it is obviously generic in that we would like it to be a description of the behavior of all actors that are defined by a graph belonging to a visual language. Therefore, strictly speaking, we do not define each of these actors, or any of them, we define something that, given a graph in a visual language, *creates* an appropriate actor.

We will call an entity that generates actors in this manner an *(actor) schema*, and in this work we will only be concerned with one particular kind of schema, viz. with those that generate actors that interpret visual languages, or *interpreter schemata*.

Such a schema will be parameterized by the particular abstract syntax structure, as well as possibly other parameters of the discrete event component, and the current point in time (a *tag* in the sense of Def. 11, i.e. an element of T), and it constructs a 'fresh' interpreter for the given syntax structure.[4]

[4]Of course, we will formally assume all possible components to be already 'present', so that the schema only needs to identify one that is unused so far.

A schema must, in general, produce more than just an actor identifier. Since components may contain embedded subcomponents, constructing an actor may involve constructing other actors and connecting them to each other and to the main actor. Also, the creation procedure may itself be non-deterministic, i.e. it may return a set of more than one result.

Note that we allow our components to be 'parametric', i.e we can pass a number of values to the schema that may influence the construction of the component. Parameters could influence the initial state of the component to be created, or delays inside it, or we could want to pass functions into the actor which affect its computations. For simplicity, we will assume that these values, like the values of graph attributes, are elements of U, and since we want to leave open exactly how many parameter values we pass to the schema, we just assume the *component parameter* list to be in U^*.

Def. 19: **(Interpreter schema)** *Assume a schedulable system of actors. We will call a function*

$$\mathcal{I} : \mathcal{T} \times \mathcal{L}(\mathbf{P}) \times U^* \longrightarrow \wp\left(I \times \wp(I) \times \wp(P^{out} \times P^{in}) \times \wp(P^{out} \times P^{in})\right)$$

that identifies for a given abstract visual program $P \in L(\mathbf{P})$ and parameter structure of an appropriate set of possible component parameters U^* *an* interpreter schema *for the language $L(\mathbf{P})$ a set of structures*

$$(i, N, c^+, c^-)$$

where i is the index of the new actor, N is a set of indices of other actors created in the process of creating $\mathcal{A}_i$, and c^+ and c^- are connection sets.

Each resulting A_i of the application of the *abstract (visual) syntax interpreter schema* is then expected to realize the behavior of the entity denoted by the corresponding abstract syntax structure. Of course, in general this is necessarily an informal notion, as we will not assume pictures to come with a given semantics, but rather that their semantics is defined by this interpreter schema.

In this sense, languages such as Petri nets are an exception, because these do have a 'predefined' formal semantics that the interpreter schema can be formally compared to. In such a case, it is of course necessary to show, or make reasonably plausible, that the operational semantics defined by the abstract syntax interpreter is in fact compatible with the original semantics of the language in question. We will come back to this issue in chapter 5.

3.3 A simple example

Let us now illustrate the construction of an abstract syntax interpreter using the small finite-state machine notation from the introduction as an example.

First, we need to make explicit the abstract syntax of this notation. We will assume two types of vertices, one with *type* tag *InitMarker* and another one with tag *State*, corresponding to the marker of the initial state and the actual states, respectively. We will only allow for one kind of arc, whose type thus becomes irrelevant. The arcs are inscribed by objects denoting the input consumed in the state transition and the output produced, where the latter may be a string of characters of the alphabet.

If we fix the alphabet to be $\{a, b, x\}$ as in the example of Fig. 2, we can formulate the following predicates to describe admissible abstract syntax structures:

Ex. 9: (Syntax predicates for a simple FSM-notation)

$$P_1(V, E, s, d, \mu, \overline{\mu}) : \forall v \in V : \mu(v, \mathit{type}) \in \{"InitMarker", "State"\}$$
$$P_2(V, E, s, d, \mu, \overline{\mu}) : \mid \{v \in V \mid \mu(v, \mathit{type}) = "InitMarker"\} \mid = 1$$
$$P_3(V, E, s, d, \mu, \overline{\mu}) : \mid \{e \in E \mid \mu(se, \mathit{type}) = "InitMarker"\} \mid = 1$$
$$P_4(V, E, s, d, \mu, \overline{\mu}) : \forall e \in E : \mu(de, \mathit{type}) = "State"$$
$$P_5(V, E, s, d, \mu, \overline{\mu}) : \forall e \in E : \mu(e, Input) \in \{"a", "b", "x"\}$$
$$P_6(V, E, s, d, \mu, \overline{\mu}) : \forall e \in E : \mu(e, Output) \in \{"a", "b", "x"\}^*$$

Now we must specify an actor that interprets any given graph structure that conforms to these predicates, or rather a schema producing such an actor for a given abstract syntax structure. The actors generated by it will all have the same number of input and output ports: They have two input ports, one for the actual data coming in and the *fire* input port, and likewise they have two output ports, one for the output data and one the *schedule* requests. For simplicity, we will assume our actors to be instantaneous, so we do not have to consider firing inputs or produce scheduling requests. Furthermore, they do not change the network structure, so the corresponding results of the transition function will always be empty.

For simplicity, we will in the following assume some comprehensive set $\mathcal{U}$ containing all possible data objects, all actors indices, the set of all tokens A, all graphs, graph objects, attributes and attribute values—in short, everything, that we can manipulate. We also assume that some special value $\perp$ be contained in $\mathcal{U}$.[5]

The states of our actors must contain information about the graph structure, as well as the particular state (which for simplicity we represent by the corresponding vertex of the graph) it is "in". Because in general the state spaces of our actors will be very large and their structures very complex, we will also assume them to be in $\mathcal{U}$.

The following actor definition is further simplified by the requirement that the actor be single-token which implies that the set of active prefixes is fixed for all s by Def. 10. This allows us to focus on the transition function in the

[5]We will talk more about this set in chapter 4.

definition of the actor. To make it somewhat shorter, we will first define the following functions:

$$
\begin{aligned}
& I(E, s, d, \mu) = v \\
& \qquad \text{with} \quad \exists e \in E : \mu(se, type) = \text{"}InitMarker\text{"} \wedge de = v \\
& T(v, a, E, s, \mu) = \{e \in E \mid se = v \wedge \mu(e, Input) = a\}
\end{aligned}
$$

Obviously, the first function produces the initial state of the finite state machine, while the second computes the transitions possible from a given state with a specific input (more precisely, it computes the set of edges that represent these transitions, of course). Note that I is well-defined for graphs that comply with the syntax predicates above—the syntax predicates effectively acting as preconditions in this case.

The actors generated by this schema all have two input ports and two output ports. The input ports are the actual data input port and the firing input p_{fire}, similarly the output ports are the data output port and the scheduling output port p_{sched}. We will define p_{sched} and p_{fire} to correspond to the second position in the respective (output/input) sequence tuple. They will have no effect in this simple interpreter.[6]

As discussed above, the states must contain the relevant structural information of the graph, as well as the vertex that represents the current FSM state. Thus, the states of our FSM interpreter are structures of the form

$$(V, E, s, d, \mu, v)$$

where V, E, s, d, and μ are the relevant parts of the graph structure, and v is the vertex that represents the current FSM state.

Our actor schema does not take any component parameters, therefore we do not use the $pars$ parameter.

[6] Here, in contrast to the *Fibs* actor from Ex. 7 we specified the ports relative to the network, since we conveniently have the actor index (and thus the input/output projections α_i and ω_i) available.

Ex. 10: (FSM interpreter schema)

$$
\begin{aligned}
&\mathcal{I}_{FSM}(t, (V, E, s, d, \mu, \overline{\mu}), pars) = \{(i, \emptyset, \emptyset, \emptyset) \mid \\
&\textit{such that} \quad \mathcal{A}_i = (\Sigma, \sigma_0, (P_\sigma)_{\sigma\in\Sigma}, (\tau_\sigma)_{\sigma\in\Sigma}, p_{fire}, p_{sched}, t_0) \\
&\textit{with} \\
&\qquad t_0 = \infty \\
&\qquad p_{fire} = \alpha_i \circ p'_{fire} \\
&\qquad\quad \textit{where} \quad p'_{fire} : (s_1, s_2) \mapsto s_2 \\
&\qquad p_{sched} = \omega_i \circ p'_{sched} \\
&\qquad\quad \textit{where} \quad p'_{sched} : (s_1, s_2) \mapsto s_2 \\
&\qquad \sigma_0 = (V, E, s, d, \mu, I(E, s, d, \mu)) \\
&\qquad \tau_{(V,E,s,d,\mu,v)} : \\
&\qquad\qquad (a, \lambda) \mapsto \begin{cases} \{((V, E, s, d, \mu, v'), (w, \lambda), \emptyset, \emptyset, \emptyset)\} \\ \qquad \textit{forall} \quad e \in T(v, \nu a, E, s, \mu) \\ \qquad \textit{with} \quad \nu w = \mu(e, Output), \theta w = \theta a, v' = de \\ \qquad \textit{if} \quad T(v, \nu a, E, s, \mu) \neq \emptyset \\ \{((V, E, s, d, \mu, v), \lambda_2, \emptyset, \emptyset, \emptyset)\} \qquad \textit{otherwise} \end{cases} \\
&\qquad\qquad (\lambda, a) \mapsto \{((V, E, s, d, \mu, v), \lambda_2, \emptyset, \emptyset, \emptyset)\}\}
\end{aligned}
$$

Note that we have omitted the definition of the state space (because it contains little information and could be inferred from the transition function and the initial state) and the prefix function (because this is fixed since our actor is required to be single-token).

While this kind of definition seems just about manageable for the finite state machine example, it seems clear that for more complicated semantics descriptions this soon becomes somewhat cumbersome. This problem will be addressed in the next section, where we will develop a more concise and better structured description language, without compromising the formality and executability of the resulting descriptions.

3.4 Related work

There is a substantial body of work on the syntactical aspects of visual languages—cf. [79] for a comprehensive survey. While most of these approaches are concerned with aspects of the concrete syntax of specific visual languages, there are some that suggest to separate the notion of *abstract visual syntax* from the concrete representation [7, 90, 91], though there the former is still very closely modeled on the latter.

Dropping these restrictions, [41, 42, 43] propose a much more abstract and general structure for the abstract syntax of a visual language, which our work is based on. Though there still may be a strong structural correspondence between abstract and concrete syntax (as is the case for the kind of languages that we are looking at in this work), this is in general not necessary.

Much less work has been done on the semantical aspects of visual languages. Often, especially for visual languages that are not visual *programming* languages, the term *semantics* has a different meaning, which we will not be concerned with here, viz. that of a static structure it describes or a set of pictures satisfying a set of constraints [28, 60]. The concept of semantics that is the topic of this paper is one of a picture representing a computational structure and the computations performed by this structure.

A general approach to this latter concept of semantics of visual programming languages can be found in [42, 43]. It builds on the abstract visual syntax mentioned above, and proposes a method to define the denotational semantics on this basis. Our work can be seen as supplementing this approach with an operational specification technique.

Most other work on visual language semantics is based on graph grammars [12, 11, 44, 101], sometimes augmented with algebraic specification techniques to model data manipulation. While these technique are certainly appropriate to describe a certain class of visual languages, it becomes somewhat artificial or even cumbersome and impractical in those cases where the execution of a component is not best described by a syntactical transformation of its abstract syntax. This is especially true for languages that manipulate other components (cf. the examples in Sections 5.3 and 5.4) and in which directing the flow of data, rather than performing modifications of their own local state, is a major part of their activity.

4

Specifying actors using Abstract State Machines

In principle, we are now able to specify visual notations in terms of the syntactical properties of their abstract syntax (assuming we have some environment providing a useful mapping between this and some concrete representation) as well as their semantics as defined by some discrete event component acting as its abstract syntax interpreter—both using simple mathematical notation, as has been done at the end of the previous chapter. However, this approach—while apparently appropriate for the definition of syntactical constraints—seems somewhat awkward when used to define the execution of a visual program.

In this chapter we will show that this kind of semantics definition is unwieldy, and propose a different way of structuring definitions, based on a well-known operational description technique known as *Abstract State Machines* [51, 53]. We will embed the resulting specifications into our actor-based model, effectively using it to define discrete-event components that interpret abstract syntax structures of visual programs.

4.1 Motivation

Looking at the specification in Ex. 10, we see that it consists of two parts: a comparatively short definition of the initial state of the actor, and one defining the transition behavior. The latter is structured according to which input port a token may arrive at (denoted by the active prefixes (a, λ) and (λ, a)). Each rule computes a 4-tuple consisting of a full description of the new state, a full description of the tuple of output sequences, as well as the set of connections to be added to or removed from the relation between ports defining the network structure of the system.

This style of specification is barely adequate for an interpreter for our FSM notation, despite the state being very simple and unstructured. For systems with more structured states it becomes rather redundant, because the complete state has to be described when defining the transition function, where it might be more adequate to simply specify which parts of the state *change* with respect to the previous state.

As an example of a language with simple structured state consider Petri nets. Here each firing corresponds to one Petri net transition occurrence (until there are no activated transitions). Let us further assume that each arc carries an attribute *Weight*, which is a positive integer, and each *Place* vertex has an attribute *initialTokens*.

The state of such a Petri net is simply a map, called a *marking*, from the places (represented by the *Place* vertices) to the non-negative integers—a very simple structured state. A transition (represented by *Transition* vertices) may fire if all incoming arcs come from places whose marking is greater than or equal to the weight of the respective arc. When a transition fires, the weight of incoming arcs is subtracted from the marking of the corresponding places, while the weights of all outgoing arcs are added to the markings of the places they point to.[1]

Note that the style of the above formulation is different from the one used for defining the state transition of an FSM in Ex. 10 in that it assumes a certain state structure (the marking in the case of Petri nets) and describes local *modifications* of it (the change of the marking at certain points in its domain, the places). For many discrete-event languages, especially when the state of a component has some structure, this is often the more natural way of defining the transition.

Using our current technique for defining interpreter schemata, a schema for simple Petri nets would look like this:

[1]We assume the Petri nets to be loop free, i.e. no transition has an arc coming from and another one going to the same place. Without this assumption, we would have to handle this as a special case.

Ex. 11: (Petri net interpreter schema)

$$\mathcal{I}_{PN}(tm,(V,E,s,d,\mu,\overline{\mu})) =$$
$$\left\{ (i,\emptyset,\emptyset,\emptyset) \;\middle|\; \begin{array}{l} \mathcal{A}_i = (\Sigma,\sigma_0,(P_\sigma)_{\sigma\in\Sigma},(\tau_\sigma)_{\sigma\in\Sigma},p_{fire},p_{sched},t_0), \\ \sigma_0 = M_V, \\ t_0 = tm, \\ p_{fire} = \alpha_i \circ p'_{fire}, \\ \quad \textit{where} \quad p'_{fire} : (s) \mapsto s \\ p_{sched} = \omega_i \circ p'_{sched}, \\ \quad \textit{where} \quad p'_{sched} : (s) \mapsto s \\ \tau_M : \\ \qquad (a) \mapsto \begin{cases} \{(M'_{(V,E,s,d,\mu,M)}(t),(a),\emptyset,\emptyset,\emptyset)\} & \\ \quad \textit{forall} \quad t \in T(V,E,s,d,\mu,M) & \\ \quad \textit{if} \quad T(V,E,s,d,\mu,M) \neq \emptyset & \\ \{(M,\lambda,\emptyset,\emptyset,\emptyset)\} & \textit{otherwise} \end{cases} \end{array} \right\}$$

We assume that the initial state is defined by a function

$$M_V : \mathcal{U} \longrightarrow \mathcal{U}$$
$$a \mapsto \begin{cases} \mu(a,initialTokens) & \textit{if} \quad a \in V \wedge \mu(a,type) = \text{"}Place\text{"} \\ \bot & \textit{otherwise} \end{cases}$$

and that the set of activated transitions $T_{(V,E,s,d,\mu,M)}$ is defined as follows:

$$T(V,E,s,d,\mu,M) = \{t \in V \mid \mu(t,type) = \text{"}Transition\text{"}$$
$$\wedge \forall e \in E : d(e) = t \Rightarrow Mse \geq \mu(e,Weight)\}$$

Finally, the marking $M'_{(V,E,s,d,\mu,M)}(t)$ resulting from firing t in state (V,E,s,d,μ,M) is defined as

$$M'_{(V,E,s,d,\mu,M)}(t) : p \mapsto \begin{cases} Mp + \mu(e,Weight) & \textit{if} \quad \exists e \in E : se = t \wedge de = p \\ Mp - \mu(e,Weight) & \textit{if} \quad \exists e \in E : se = p \wedge de = t \\ Mp & \textit{otherwise} \end{cases}$$

As can be seen from this description, our specification technique does not scale very well. As mentioned above, the need to always constructively describe the full state structure M (as opposed to only those things that actually change) leads to some redundancy. Also, the need to explicitly construct a set of possible successor states in the definition of τ_M requires some notational overhead.

Furthermore, the specification is not very well-structured. The structure of the state is implicit: We know that M is the state because of its use as an index of τ, and we know its structure because of the way $M'_{(V,E,s,d,\mu,M)}$ is constructed. The sequentiality of the computation of the set of activated transitions (in the 'forall' clause in the definition of τ_M), the selection of the next to fire (implied by the choice of one t in that 'forall' clause), and the computation of the next state (the call to $M'_{(V,E,s,d,\mu,M)}$ in the first case of t_M)is also not represented explicitly—in fact, the call to $M'_{(V,E,s,d,\mu,M)}$ appears textually before the other two. Looking at this definition, it is not immediately obvious at which point the actor behaves nondeterministically and why.

One can easily imagine cases where this kind of specification scheme is even less adequate. For instance, if the actual computation of the next state is in itself somewhat more 'algorithmic', more iterative (instead of, as for Petri nets, essentially one quantification over one variable—the places in the definition of $M'_{(V,E,s,d,\mu,M)}(t)$) the above way of writing this down soon becomes impractical.

Therefore, although the semantics framework as such seems to be sufficiently general and powerful to cover many different kinds of visual notations, we are still lacking a proper specification language that can be used to specify actors in. From our discussion above we can see that this language should have the following characteristics:

- It should allow the representation and manipulation of structures representing an actor state.

- It should be possible to define a new structure by locally changing an old one in a number of places, i.e. the next state should be defined 'differentially' by specifying its differences to the previous state.

- The language should support iterative computations on that structure.

- The language should have a natural concept of non-determinism with as little notational overhead as possible.

Note that *compositionality* is not a requirement for our specification language. Indeed from the current state of our discussion we are not even able to say precisely what it would mean for the specification of an interpreter schema to be compositional. Of course, by virtue of being embedded into our actor framework, it automatically inherits its compositionality as far as individual actors/abstract syntax interpreters are concerned, but that does not imply that the specification of these interpreters themselves is compositional. We will come to this point later, after we have applied our language to some example applications.

Fortunately, a specification language already exists that addresses these issues: *Abstract State Machines (ASM)* [51, 53, 52]. ASM have a proven track record as a specification technique for many programming languages, focusing primarily on languages that lend themselves very well to an operational semantics description (e.g. Java [27, 26], Oberon [73], Modula-2 [83], C++ [109])—even though one of the first programming language semantics (after that of Modula-2) given as an ASM was, in fact, that of Prolog [19, 20, 24]. This makes ASM an interesting platform for our actor semantics, since actors are, by definition, state-based, 'operational' entities.

In this chapter we will present our approach of how to use concepts developed in the context of ASM to describe the behavior of an actor. Also examples of ASM descriptions for the Petri net example and the FSM example will be given. The next chapter will then show applications of this technique to somewhat larger examples.

Our main focus will now turn to developing a language to describe a state transition of the form

$$\tau_\sigma : (\lambda, ..., a, ..., \lambda) \mapsto (\sigma', w, c^+, c^-, N)$$

which is an entity that, in a state σ, maps a single-token input a to a new state σ', some output $w \in S^n$, possibly some changes to the connection structure of the system (by connection sets c^+ and c^-), and new active actors N. But before we can discuss the definition of a state transition function, we need to elaborate a few preliminary concepts.

4.2 Preliminaries

This section discusses some of the basic concepts needed to define a language that allows the differential specification of states by 'changing' an existing state. First we elaborate the concept of state of an actor, presenting a simple formal model we claim will be rich enough to capture all cases of interest to us. We then introduce a few notions needed in discussing the definition of differential state changes. We also define a simple concept for modeling side-effect-free computation on the data items we want to manipulate, which allows us to abstract from any manipulation of the objects of our universe themselves, as long as our actors do not change state.

Then we describe a simple concept of 'structuring' objects and defining relationships between them, which together with the interpretation of symbols (in Section 4.5) provides a simple foundation for incorporating functional (i.e. side-effect free) computation into our model.

4.2.1 The universe

We assume a universe $\mathcal{U}$ that contains all possible objects of our computation (we will refer to the elements of $\mathcal{U}$ technically as *objects* from now on). In particular, it includes not only those that are 'used' at any point in time, but also all 'past' and 'future' ones. This allows us to talk about a static collection of things, and 'creating' new objects (such as new data structures, or new actors) essentially amounts to gaining access to objects that were previously not accessible (we will refer to them as *fresh* objects). This will be discussed below.

Obviously, for most practical applications, the universe will be a rather large one—it could contain (at least) one object for each of the usual 'atomic' data items (possibly different kinds of numbers, strings, boolean values, ...), as well as structures among them (lists, sets, tuples, ...). Specifically, it is required to contain a 'special' object $\bot$, which is understood as the 'null' object, used by us to denote the absence of a 'proper' object, but otherwise a perfectly normal member of $\mathcal{U}$.

These requirements are the same as those introduced in section 3.3. As we will see, this is convenient because it facilitates a direct representation of the abstract syntax of a picture inside the state of the system.

4.2.2 The structure of the state of an actor

In this section a structure will be given to the states of an actor, i.e. the σ that the transition function τ_σ depends on, and thus the state space of an actor, Σ. We adopt the concept developed for Abstract State Machines [51, 53], which is based on Tarski's notion of structure [102].

The essential elements of a state can be illustrated by the state descriptions we used in the examples so far. In the finite state machine semantics of the previous chapter, states looked like this:

$$(V, E, s, d, \mu, v)$$

where V and E were sets of objects, s and d were unary functions, μ was a binary function, and v was just one particular object (a state in V). Similarly, a Petri net state in the example above had the following layout:

$$(V, E, s, d, \mu, M)$$

Most components were very similar, except the last, which was a unary function.

So it seems as though in specifying states we should provide a facility to include sets, functions of various arities, and simple variables. However we can simplify our definitions if we regard sets as unary functions and employ the convention that

$$a \in s \Leftrightarrow (sa) \neq \bot$$

If furthermore we see variables (such as the v component in the FSM state) as nullary functions, we are left with states being simply composed of a collection of functions of various arities, including zero.

However, there are two aspects of a state of an actor—there are the symbols, and their associated arities. In a sense, these are the 'syntactical' aspect of a state, which we will use to specify the actor behavior, and which at any given moment 'stands for' a concrete realization of the state, actual functions of the corresponding arity. Following standard ASM terminology, we will call the symbols and their arities a *vocabulary*, the functions they are associated with an *algebra*.

Def. 20: **(Vocabulary)** *Assuming a set of (function) symbols* $\mathcal{S}$, *a* vocabulary $\mathcal{V}$ *is a function*

$$\mathcal{V} : \mathcal{S} \longrightarrow \mathbb{N}_0 \cup \{\bot\}$$

We will call the set

$$\mathit{dom}\,\mathcal{V} = \{s \mid \mathcal{V}s \neq \bot\}$$

the set of defined (function) symbols *and require it to be finite for the vocabulary to be valid. For any defined function symbol s, the natural number $\mathcal{V}s$ will be its* arity.

Ex. 12: (Vocabulary of FSM interpreter schema) *The fragment of our specification language defining the vocabulary of the FSM interpreter schema looks like this:*[2]

function V, E **arity** 1 **;**
function s, d **arity** 1 **;**
function μ **arity** 2 **;**
function $state$ **arity** 0 **;**

To enhance readability, we feature the keywords **set** *and* **attribute** *for unary functions that are intended to be used as sets, and functions of arity 0, respectively. This makes the above vocabulary somewhat more readily understandable:*

set V, E **;**
function s, d **arity** 1 **;**
function μ **arity** 2 **;**
attribute $state$ **;**

The carrier set of all our algebras (they will only have one) will always be $\mathcal{U}$, so that an algebra will be defined by its functions only. A state of an ASM consists of binding such an algebra to a vocabulary by interpreting the symbols of the vocabulary as functions of the corresponding arity. Such a binding will be called a *valuation*.[3]

Def. 21: (Valuation, compatibility with vocabulary) *A valuation* $\mathbf{V}$ *is a function*

$$\mathbf{V} : \mathcal{S} \longrightarrow \bigcup_{n \in \mathbb{N}_0} \langle \mathcal{U}^n \longrightarrow \mathcal{U} \rangle \cup \{\bot\}$$

assigning to each symbol either a function over $\mathcal{U}$ of some arity, or $\bot$.

A valuation $\mathbf{V}$ *is* compatible *with a vocabulary $\mathcal{V}$ iff*

$$\begin{aligned} &\forall s \in \mathit{dom}\,\mathcal{V} : \mathbf{V}s \in \langle \mathcal{U}^{\mathcal{V}s} \longrightarrow \mathcal{U} \rangle \\ \wedge &\forall s \in \mathcal{S} \setminus \mathit{dom}\,\mathcal{V} : \mathbf{V}s = \bot \end{aligned}$$

The set of a valuation compatible to a vocabulary $\mathcal{V}$ will be written as $\Sigma_{\mathcal{V}}$.

[2] For better readability, the examples in this section may include characters, such as μ, which might not be accepted in an implementation.

[3] The more common term 'interpretation' will be used for something different below.

In other words, a valuation is compatible with a vocabulary if it assigns only the defined symbols a function, and if that function is of the proper arity defined by the vocabulary.

For each interpreter schema we will define a vocabulary, and then a state of an interpreter of that schema is simply a valuation that is compatible with this vocabulary.

4.2.3 Basic concepts of state transitions

So far, we have simply formalized the concept of actor state we had already used in the examples. Remember that one of the issues we wanted to improve on was the way we described the new state—instead of providing a complete new valuation (to use the new terminology), we wanted to specify only those locations in the state that actually changed, and of course the values they changed to. This then, together with the current state, would define exactly the next state.

So first we need to define this concept of location. For instance, if in the Petri net semantics, the marking is denoted by the symbol M, and we would like to add w to the marking of the place p, then we would like to write something such as

$$M(p) := M(p) + w$$

with the intention that in the next state, the M symbol will be bound to a function that is like the current M function, except that at point p its value has been updated to the sum of the current value and w (of course, there may also be other changes to M).

The location we are changing in the above line would be uniquely identified by the function symbol (M) and the point of the function given by the object p evaluates to. In other words, a location is a symbol and the respective parameter tuple identifying the point of the associated function:

Def. 22: (Location, compatibility with vocabulary) *A set of* locations *is defined as*

$$\mathcal{L} = \mathcal{S} \times \bigcup_{n \in \mathbb{N}_0} \mathcal{U}^n$$

A location $(s, \mathbf{a})$*, where* $\mathbf{a}$ *is some tuple in some* $\mathcal{U}^n$ *is* compatible *with a vocabulary* $\mathcal{V}$ *iff*

$$s \in dom \;\; \mathcal{V} \wedge a \in \mathcal{U}^{\mathcal{V}s}$$

In other words, the symbols s *must be defined in* $\mathcal{V}$ *and its arity must be* n*. The set of all locations compatible with* $\mathcal{V}$ *is written as* $\mathcal{L}_{\mathcal{V}}$*.*

In the above example, the term $M(p)$ on the left-hand side of the assignment would denote a location—with $\hat{p}$ being the object that results from p, the location would be $(M, (\hat{p}))$.

The basic construction that we will write to change the state of an actor will be a *rule*. The result of such a rule will be a number (any number) of locations and their new values. Such a pair of location/new value we will call *update*, and a collection of these an *update set*. When describing the semantics of an ASM rule, we will do these in terms of the update sets generated by them (as well as a few other things, cf. below).

Def. 23: (Update, update set, compatibility with vocabulary) *An* update *is an element in $\mathcal{L} \times \mathcal{U}$, an* update set *is a set of updates.*

A specific update (l, v), where l is a location and v its new value, is compatible *with a vocabulary $\mathcal{V}$ iff its location l is. An update set is compatible with $\mathcal{V}$ iff all its updates are.*

In the example above, if x is the value of the right-hand side of the assignment, i.e. x is $M(p) + w$, the update denoted by this rule would consequently be $((M, (\hat{p})), x)$.

Obviously, in order for the next state to be well-defined, an update set should at most contain one new value for any given location. This gives rise to the notion of consistency:

Def. 24: (Consistency of update set) *An update set U is called* consistent *iff*

$$\forall (l_1, a_1), (l_2, a_2) \in U : l_1 = l_2 \Rightarrow a_1 = a_2$$

Obviously, if we had a rule

$M(p) := M(p) + w,$
$M(p) := M(p) - w$

and w would be non-zero, then having the updates generated from these two rules in the same update set would essentially mean that in the next state we try to assign the location $(M, (\hat{p}))$ two different values. Consistency means that an update set does not do this.

Now we are at a point where we can define what it means to have a state (valuation) $\mathbf{V}$ changed by a set of pointwise updates, say U.

Def. 25: (Updating a valuation) *Assume we have a vocabulary $\mathcal{V}$ and a valuation $\mathbf{V}$ that is compatible with it. Updating $\mathbf{V}$ with an update set U (also compatible with $\mathcal{V}$) yields a new valuation which we will write as $\mathbf{V}[U]$. If U is consistent, then $\mathbf{V}[U]\ s$ for any defined symbol s is defined such that*

$$\mathbf{V}[U]\ s\ \mathbf{a} = \begin{cases} v & \textit{if} \quad ((s, \mathbf{a}), v) \in U \\ \mathbf{V}\ s\ \mathbf{a} & \textit{otherwise} \end{cases}$$

If s is not defined in $\mathcal{V}$, $\mathbf{V}[U]\ s = \bot$, ensuring that $\mathbf{V}[U]$ is also compatible with $\mathcal{V}$

If U is not consistent, $\mathbf{V}[U] = \mathbf{V}$.

So for a consistent U, $\mathbf{V}[U]s$ is the function that maps any tuple $\mathbf{a}$ (of correct arity) either to the value v , if the update $((s, \mathbf{a}), v)$ is contained in U (with $(s, \mathbf{a})$ being the location and v the value), and simply the original value $\mathbf{V}\ s\ \mathbf{a}$ if there is no such update.

Note that applying an inconsistent update set to a state produces no change, and that in any case $\mathbf{V}[U]$ will always be compatible to $\mathcal{V}$ if $\mathbf{V}$ and U are. Now we will turn to defining a language (and its evaluation) that will be based on these concepts of differential computation of a state.

4.2.4 Structuring the universe

So far, we have focused on the way the state of an actor defines relationships between objects. However, since we require actor states to be separate from each other[4], this might lead to the following problem when they want to exchange data. Say an actor sends another actor an object, i.e. an element of $\mathcal{U}$. As we defined it above, this object is essentially unstructured, it is 'featureless'. It has no relation to other objects, except being distinguishable from them. It doesn't help the receiver that the valuation of the sender puts the object into some context, because it has no access to that valuation.

Actors therefore need access to a structure that is not part of their own valuation, nor that of any other actor, but accessible to all actors in the same way. It is important that this structure never *changes*, because then actors could influence each other by changing that 'global' structure, which would thus become part of the state of the system.

For this structure, we will choose one binary function, Φ:

Def. 26: (Structure of the universe) *Given a universe* $\mathcal{U}$, *a function*

$$\Phi : \mathcal{U} \times \mathcal{U} \longrightarrow \mathcal{U}$$

creates a structure *in it.*

We assume this Φ to be fixed for a given system. We may consider it to be bound to an 'invisible' binary function symbol in the vocabulary. It is invisible in the sense that may not be assigned to, since Φ must be constant. We will discuss this in Section 4.6 on terms.

One way of looking at this function is as a table indexed on both axes by $\mathcal{U}$, with objects contained in the cells. Another interesting interpretation, however, results from realizing that for each Φ, we can define a function $\overline{\Phi}$ as follows:

$$\overline{\Phi} : \mathcal{U} \longrightarrow (\mathcal{U} \longrightarrow \mathcal{U})$$
$$\overline{\Phi}(a)(b) = \Phi(a, b)$$

[4]This is a necessary condition, since otherwise the transition of one actor could change the state of another, which is something we do not allow in our model of computation—the execution of a schedulable system (Def. 16 on page 47) changes at most one actor state per step.

This means that $\overline{\Phi}$ associates with to each object x a unary function $\overline{\Phi}(x)$, which we call the *structure* of x. This very simple notion captures all important data structures: an array/a tuple is a map from indices to objects, a record is a map from record tags to objects, a linked list might be construed as a map of two special tags (”data” and ”next”, say) to objects and so on. Maps of more than one argument (multi-dimensional arrays, for example) may be constructed as either cascaded unary maps, or as one unary map of lists.

Data structures may be ’built’ up from this simple concept in the following manner. For instance, assume we have an object representing (by convention) the empty list, let us call this object ϵ. Then, the object, say l_1 representing the list $[a]$ (with a being another object) may be defined as an object that has the following properties: 1. $\overline{\Phi}(l_1)(''first'') = \Phi(l_1, ''first'') = a$, and 2. $\overline{\Phi}(l_1)(''rest'') = \Phi(l_1, ''rest'') = \perp$. Here, $''first''$ and $''rest''$ are names for other objects. Likewise, an object l_2 representing $[b, a]$ has the property that $\overline{\Phi}(l_2)(''first'') = b$ and $\overline{\Phi}(l_2)(''rest'') = l$, for any object l that represents (a).[5]

We need to pay special attention to functions operating on more then one argument. Performing an addition such as $3 + 4$ is usually understood as applying an addition function to two arguments, 3 and 4. In our context, this will be interpreted as the expression $(+3)4$, where $+$, 3, and 4 are names of objects such that the object named by $+$ is associated to a function that maps any number (and whatever else it may be defined for) to some object that is associated to a function that adds just that number to any other number. So the result of $(+3)$ is an object associated to an add-three-map, which is then applied to 4, resulting in 7.[6] For notational convenience, we will take the liberty to use infix-notation, and also to apply a function to more than one argument, as in $f(a, b)$, when this is understood to mean $(fa)b$. We will make this more precise below.

Of course, the precise structure of the universe is something that will be defined by a specific implementation, though we will assume the ’usual’ environment of numbers, strings, and lists as well as the usual operations on them. So from now on, if we talk about the universe $\mathcal{U}$, we will assume the presence of a ’useful’ Φ.[7]

In order to make this work we need to be able to actually *name* objects in the universe and then somehow ’apply’ Φ to them. We will discuss how we name

[5]Note that we do not assume that there is exactly one object representing a given data structure, thereby distinguishing between the notions of (value) equality and identity. Of course, an implementation might in fact choose to identify these notions, so that identical structures will always be represented by identical objects. From our perspective here, this is merely a choice of choosing Φ and the global interpretation we will introduce in Section 4.5.

[6]This interpretation of functions of more than one argument is called *currying* in λ-calculus [13] and functional programming. This view of functions of more than one argument has been introduced by Schönfinkel [97] and used by Curry [38].

[7]Note that although the table-view of the structure suggests this, the structure of $\mathcal{U}$ need of course not be represented by actual data, but can be implemented as code computing the required functions—as long as it has no side effects as far as the view of the actors on the universe is concerned. More specifically, this definition creates a place to plug a functional language into, such that these computations might in fact be user-specified—cf. Section 4.6.4.

objects in Section 4.5, the application of Φ to (pairs of) objects is discussed in the context of how we write terms in Section 4.6.1. This structure of the universe is what we will use for computing with individual objects. Our model of actors and their behavior is largely orthogonal to the structure of the universe in that it makes very few assumptions about the presence of certain mappings (e.g in Section 4.7.1).

4.3 Defining an actor schema

The language we are about to present will allow the definition of general actor schemata. An actor schema, which we will also call a *(actor) class*, has the following structure:

class $name[par_1, ..., par_n]$ **is**
  input/output ports
  vocabulary
  initialization rule
  rules
end

In section 4.2.2 we have already presented the syntax for specifying the vocabulary, which is a straightforward enumeration of symbols and their arities. In the following sections we will discuss the syntax and semantics for defining state transition *rules*, and then the initialization of an actor from a class specification, as well as how to specify its input and output ports.

Note that the actor schema has parameters, as we would expect from the discussion of interpreter schemata in the previous chapter. When a new actor is constructed from this schema, its parameters are visible throughout its definition. We will come to actor setup and related issues in Section 4.9.

4.4 Overview of the rule language and its interpretation

The dynamic behavior of an ASM-defined actor is specified in a number of *rules* that eventually define the transition function. Recall that the transition function of a single-token actor has the following form:

$$\tau_\sigma : (\lambda, ..., a, ..., \lambda) \mapsto (\sigma', w, c^+, c^-, N)$$

Obviously, the value of this function (new state, output, port connections to be added, port connections to be removed, new actors created) depends on the state of the actor, but also on the input token consumed from one of the actor's input streams. We will specify the input token using three pieces of information:

- the port from which it came
- its time tag θa
- its value projection va (cf. Def. 11 on page 42)

A rule that calculates the next state is therefore written in the following format:

<u>rule</u> $ruleName[p, t, v]$:
  rule body
<u>end</u>

Each input port of the component is associated with exactly one such rule, which is executed whenever the component fires on a token coming in through that port. (We will discuss how this association is done in sections 4.7.1 and 4.9.)

In the above rule skeleton, the parameters p, t, and v are just the port, the (time) tag and the value of the input token. Of course, a class definition can contain any number of rules. Obviously, the constructions in a rule must be able to not only refer to the symbols defined by the vocabulary of a class, it also needs to refer to the parameters of a rule, and (as we will see) also to local variables defined inside the rule texts. We will deal with this aspect of the language definition in the next section.

An actor's transition from one state to a successor state has been described as an atomic step in our examples so far. However, in many cases one would like a more iterative style of specification, where the single transition is in fact a sequence of 'microsteps', each leading to a new state. The last state in the sequence is then taken to be the 'result' of the state transition, i.e. the new state of the actor. This is illustrated in Fig. 29.

In its most general form, a full rule consists of two sequences of *basic rules*, i.e. rules that result in an atomic state transition. It has the following syntax:

<u>rule</u> $ruleName$[parameters] :
<u>once</u>
  r_1 **<u>;</u>**
  ...
  r_j
<u>then</u>
  $\overline{r}_1$ **<u>;</u>**
  ...
  $\overline{r}_k$
<u>end</u>

Where the r_i and the $\overline{r}_i$ are basic rules. We will allow any rule sequence to be empty. If the first is empty ($j = 0$) the above syntax may be simplified to the following form:

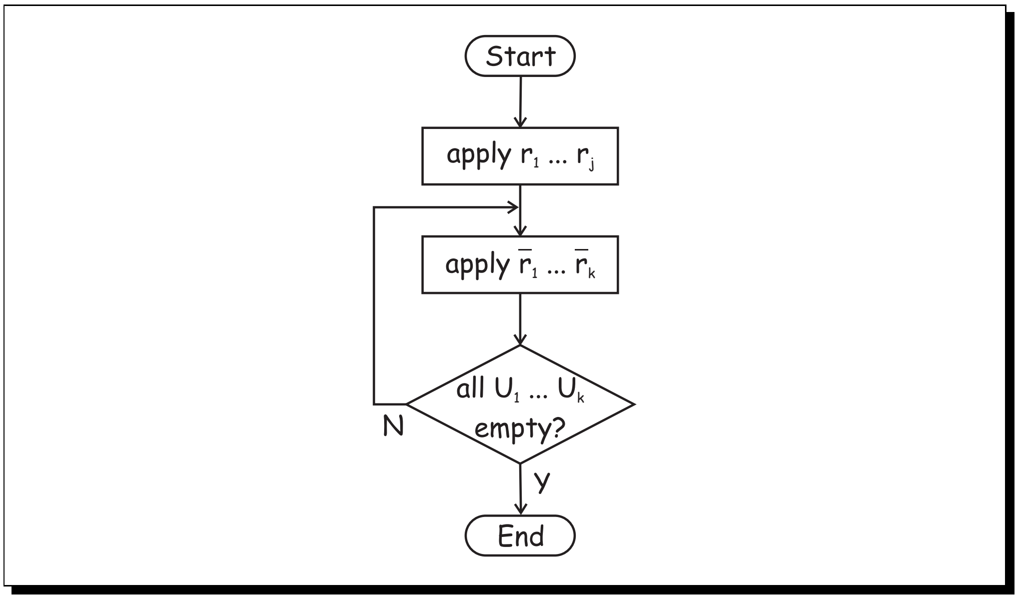

Fig. 28: The execution of a full rule **once** r_1;...;r_j **then** $\overline{r}_1$;...;$\overline{r}_k$ **end**.

rule $ruleName$[parameters] :
$\overline{r}_1$ **;**
...
$\overline{r}_k$
end

On the other hand, if the second rule sequence is empty, we can simply write it like this:

rule $ruleName$[parameters] :
once
r_1 **;**
...
r_j
end

We call such a rule an *full rule*, or *iterated rule* because of the way it is executed.

Intuitively, executing a full rule has the structure shown in Fig. 28. The first step is to execute the basic rules $r_1, ..., r_j$ in the order in which they are given, updating the state after each basic rule, so that its successor is executed in the valuation that results from this update. This is depicted in Fig. 29, which shows the transitions of state a rule makes, where each σ_i corresponds to a new valuation of the vocabulary.

Then, the second sequence of basic rules $\overline{r}_1, ..., \overline{r}_k$ is executed similar to the first, each basic rule resulting in an update set U_i, which is applied to the valuation as with the first sequence of basic rules. As before, every rule is therefore executed in the valuation resulting from the update of its predecessor. Now there are two cases: Either, all the update sets U_i of this sequence were empty, or at least one contained an update. In the first case, the execution of the full rule is finished, and the resulting state, the connection sets, the output, and the newly created actors become the result of this actor transition (Fig. 29).

Otherwise, at least one update was generated from the second sequence of basic rules. In this case the rule sequence is executed again, in the same fashion, until all update sets are empty. The result is then again the final state and the accumulated connection sets, output sequences, and newly created actors.

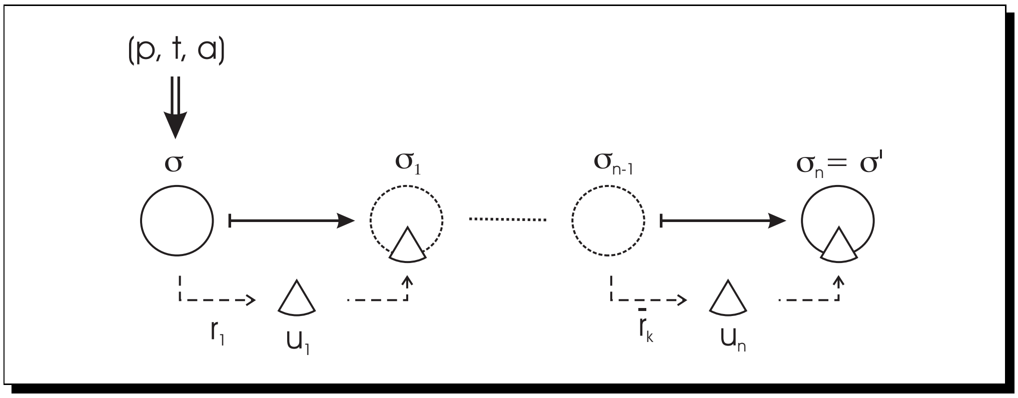

Fig. 29: The sequence of states during the execution of a full rule.

Note that once the execution of the rule has finished, we have reached a fixed point of the state transition described by the 'iterated' second sequence of basic rules $\overline{r}_i$. However, the converse is not true, i.e. having reached a fixed point does not guarantee termination of the algorithm in Fig. 28. Consider the following rule body:

once
then
$a := 42$
end

The second part of this rule will always create the same update set, which assigns 42 to the location $(a, ())$. This means that it immediately reaches a fixed point, but since the update set is not empty, the algorithm in Fig. 28 will not terminate. We do not define the behavior of the actor under these circumstances.

The reason for this choice is a practical one—it is expensive to find out whether a fixed point has been reached, i.e. whether the application of an update set changes the state. In particular, if we are only interested in fixed points of the complete sequence of basic rules, rather than of each single one of them. For instance, consider this rule body:

once
then
$a := 11$ **;**
$a := 42$
end

Each single basic rule creates a non-empty update set, and also actually changes the state. Considered as a unit, however, the sequence always returns back to the same state. If we were to detect this, we would need to keep a copy of the state at the beginning of an iteration and compare it at each point that might have changed with the state resulting at the end on an iteration. It seems that these cases are rather unusual in practice, and to require an implementation to provide this complex machinery in order to handle this case therefore seemed inappropriate.

We will come to the full formal semantics of a iterated rule in section 4.8. First we will describe the basic rules, which we need to effect atomic state transitions. In order to do this, we need to discuss how we handle parameters and local variables in the interpretation of rules.

4.5 Rule parameters and variables

Looking at the general rule format, it is clear that the rule body needs to refer to the parameters of a rule, which (like the functions in the signature) are represented by symbols.

rule $ruleName[p, t, v]$:
  rule body
end

When writing a rule fragment such as

$$M(v) := M(v) + weight(t)$$

we use several kinds of symbols—some (M, and possibly $weight$) might be part of the vocabulary of the class, while others (t and v) could be parameters or even local variables defined inside a rule (we will discuss below how this can be done), still others, such as the $+$ symbol, we might consider as 'predefined' globally. In general, we may want to use other symbols than those defined in the vocabulary.

What we need is some assignment that gives some symbols a value (in $\mathcal{U}$) that is independent of the current valuation of the vocabulary. This assignment of values to a set of symbols of our language is done by an *interpretation*.

Def. 27: **(Symbols, interpretation)** *Let* $\mathcal{S}$ *be an enumerable set of* symbols*, then an* interpretation $\mathcal{I}$ *is a partial map*

$$\mathcal{I} : \mathcal{S} \rightharpoonup \mathcal{U}$$

We will write its domain as dom $\mathcal{I}$.

Note that $\mathcal{I}$ needs to be defined as a partial map rather than denote 'undefined' symbols by assigning them the value $\perp$ because $\perp$ is of course a perfectly legal value for a symbol (in fact we might even have 'predefined' symbols explicitly representing $\perp$, e.g. the symbol *false*).

When later interpreting concrete rules in our language such as the ones above, we will assume a certain set of symbols to have 'predefined' interpretations—such as 1 and $+$, but possibly also things such as λ as the basic list constructor and many others. This *global interpretation* will be written $\overline{\mathcal{I}}$. We will leave the details of this interpretation up to the concrete implementation, but for convenience we will assume it to define the usual arithmetic operators, numbers, strings etc. as well as a list constructor λ in the way suggested above. It is via these 'predefined' symbols that we gain access to these structures of the universe.

The interpretations used for different parts of a rule may be different—new variables may be defined, adding to the interpretation and possibly shadowing a previous definition of the same variable symbol. Rules may be parameterized with different values, which also results in different interpretations being applied to the rule. This means we need to define a way to add new symbol/value pairs to an existing interpretation.

Def. 28: **(Substitution operation)** *Given an interpretation* $\mathcal{I}$*, a symbol* s *and an object* v*, we define the* substitution *of* v *for* s *in* $\mathcal{I}$ *(written as* $\mathcal{I}[s \mapsto v]$*) as follows:*

$$\mathcal{I}[s \mapsto v] : a \mapsto \begin{cases} v & a = s \\ \mathcal{I}a & a \neq s \wedge a \in \mathit{dom}\,\mathcal{I} \end{cases}$$

For convenience, when substituting a number of symbols $(s_1, ..., s_k)$ with corresponding values $(v_1, ..., v_k)$ we will write

$$\mathfrak{I}[(s_1, ..., s_k) \mapsto (v_1, ..., v_k)] = \mathfrak{I}[s_1 \mapsto v_1]...[s_k \mapsto v_k]$$

It is important to note that at any lexical point in a rule we need to be able to statically (i.e. from the rule text) compute the set of symbols that have an interpretation, i.e. to statically determine dom $\mathfrak{I}$. This is essential if the symbol sets for variables and function symbols overlap (as will usually be the case), as this allows us to determine for each occurrence of every symbol in the text whether it represents a variable or parameter (in which case we call it a *variable symbol*), or a function of the vocabulary of the actor (in which case it is called a *function symbol*).

Using the concept of interpretation, we can formulate precisely whether we interpret a symbol s as a function symbol or a variable symbol in the context of an interpretation $\mathfrak{I}$, by looking it up in the following order of precedence:

1. $s \in$ dom $\mathfrak{I}$: The symbol is a (local) variable symbol.
2. $s \in$ dom $\mathcal{V}$: The symbol is a function symbol.
3. $s \in$ dom $\overline{\mathfrak{I}}$: The symbol is a (global) variable symbol.
4. Otherwise the symbol is undefined.

Def. 29: (Function symbol, variable symbol) *Given an interpretation $\mathfrak{I}$ (and assuming a vocabulary $\mathcal{V}$ and a global interpretation $\overline{\mathfrak{I}}$), we say that the symbols in*

$$Func_{\mathfrak{I}} = \textit{dom } \mathcal{V} \setminus \textit{dom } \mathfrak{I}$$

are its function symbols *and those in*

$$Var_{\mathfrak{I}} = \textit{dom } \mathfrak{I} \cup (\textit{dom } \overline{\mathfrak{I}} \setminus \textit{dom } \mathcal{V})$$

are its variable symbols.

Now we will use these concepts to define a simple yet powerful language to write down rules.

4.6 Terms and their evaluation

One fundamental element in our rule language are *terms*. There are three kinds of terms:

- *object terms*, which are used to compute an element of the universe, and which are so common that we often omit the 'object' qualification when there is no risk of confusion,

- *location terms* (*lterms*, for short), which identify a location in an actor's state, and

- *set terms* whose result is a set of objects (these are used when we quantify variables over sets of objects).

Consider the following fragment of a rule (which we already encountered above), and assume that M is part of the vocabulary, p and w are local variables or parameters, and $+$ is a predefined variable:

$$M(p) := M(p) + w$$

Recall that we expect this rule fragment to lead to a change of the location identified by the term to the left of the $:=$-sign, to the value resulting from evaluating the term on the right of it. This means that the left-hand occurrence of $M(p)$ is a location term, while the right hand occurrence, as part of the addition, is an object term. Of course, the variable symbols themselves are object terms, and so is the addition itself.

When discussing terms, we need to talk about three aspects:

- their environment, i.e. the context in which they are written with respect to the variables/parameters/function symbols defined in that context,

- their structure, i.e. the way terms are constructed,

- their value.

The environment for terms is always defined by three components: the interpretation of local variables and parameters, the current valuation of the vocabulary, and the global interpretation $\overline{\mathcal{I}}$ of predefined symbols. Assuming the latter as fixed, the variable part of the environment of a term is a pair

$$(\mathcal{I}, \mathbf{V})$$

of an interpretation and a valuation. Then the object, location, or set that a term t evaluates to in this environment is written

$$[t]_{(\mathcal{I}, \mathbf{V})}$$

We will use the same notation for all kinds of terms, since the context usually makes clear which interpretation is intended.

We will now define the structure and the value of the different kinds of object terms, location terms, and set terms. Then we will shortly discuss the issue of how to embed 'functional', i.e. side-effect free computation into this framework.

4.6.1 Object terms

Looking at the rule fragment above we can rewrite it eliminating the syntactic sugar previously used as follows (resolving infix notation of the $+$-symbol and assuming objects to be associated with unary maps as discussed above):

$$M(p) := (+(M(p)))\ (w)$$

There are three kinds of object terms in the above example:

- *variable symbols*:[8] p (twice; as part of the left-hand location term, and inside the right-hand object term), w, $+$
- *function applications*, consisting of a function symbol applied to an appropriate number (as defined by the vocabulary) of argument terms: $M(p)$ (only the right-hand occurrence)
- *object applications*, where the result of some object (more precisely, of course, the function associated with its result by $\overline{\Phi}$, see below) is applied to (the result of) one argument term: $+(M(p))$, $(+(M(p)))\ (w)$

Naturally, we will allow infix notation and having more than one argument for object map applications, syntactically transforming them to the basic case as discussed above. Note that if we encounter a term such as

$$f(a, b)$$

we need to know whether f is a function symbol or a variable symbol in the context where it occurs in order to decide whether this denotes the function application

$$f(a, b)$$

or the object map application

$$(f(a))\ (b)$$

respectively. It is here that we need to distinguish statically between variable symbols and function symbols. We will construct our rules so that this is possible.

These are exactly the three kinds of object terms we will be defining.

Def. 30: **(Object terms and their values)** *Given a vocabulary $\mathcal{V}$, an environment $(\mathfrak{I}, \mathbf{V})$, and assuming a global interpretation $\overline{\mathfrak{I}}$, we define the following kinds of object terms:*

All $s \in Var_{\mathfrak{I}}$ are object terms. Their value is simple looked up in the 'current' interpretation, or in the global one, if they are not defined in the former:

$$[s]_{(\mathfrak{I},\mathbf{V})} = \begin{cases} \mathfrak{I}s & \textit{if} \quad s \in \textit{dom}\, \mathfrak{I} \\ \overline{\mathfrak{I}}s & \textit{otherwise} \end{cases}$$

[8] Recall that we can for each occurrence of a symbol determine whether it is a variable symbol or a function symbol.

All $f(t_1, ..., t_n)$ are object terms, if $f \in Func_{\mathfrak{I}}$, $n = \mathcal{V}f$, and the t_i are object terms. Their value is the result of applying the function bound to the symbol f (in the current valuation) to the values of the argument terms:

$$[f(t_1, ..., t_n)]_{(\mathfrak{I},\mathbf{V})} = (\mathbf{V}f)([t_1]_{(\mathfrak{I},\mathbf{V})}, ..., [t_n]_{(\mathfrak{I},\mathbf{V})})$$

If m and t are object terms then so is $m(t)$. Its value is defined as:

$$[m(t)]_{(\mathfrak{I},\mathbf{V})} = \Phi([m]_{(\mathfrak{I},\mathbf{V})}, [t]_{(\mathfrak{I},\mathbf{V})})$$

Note that the last kind of object term, $m(t)$, essentially 'applying' and object (the result of the term m) to another one (resulting from t) makes use of the common, 'global' binary function Φ (cf. Section 4.2.4). The syntax supports the intuition behind $\overline{\Phi}$, viz. that each object is associated to a unary function. We mentioned that Φ may be considered as being bound (by each vocabulary) to an implicitly defined 'invisible' symbol, say S_Φ. Then we can simply consider the object term $m(t)$ as a shorthand for $S_\Phi(m, t)$.

Object terms allow us to compute values using the predefined maps and the functions of our vocabulary. Now we need to identify locations we can assign them to.

4.6.2 Location terms

Location terms are even simpler than object terms, since there is only one structure of them: they consist of a function symbol and an appropriate (i.e. corresponding to the function's arity in the current vocabulary) list of object terms.

Def. 31: **(Location terms and their values)** *Given a vocabulary $\mathcal{V}$, an environment $(\mathfrak{I}, \mathbf{V})$, and assuming a global interpretation $\overline{\mathfrak{I}}$, every $f(t_1, ..., t_n)$ is a* location term *if $f \in Func_{\mathfrak{I}}$, $n = \mathcal{V}f$, and the t_i are object terms. Its value is*

$$[f(t_1, ..., t_n)]_{(\mathfrak{I},\mathbf{V})} = (f, ([t_1]_{(\mathfrak{I},\mathbf{V})}, ..., [t_n]_{(\mathfrak{I},\mathbf{V})}))$$

4.6.3 Set terms

Set terms are used to compute a set of objects in order to quantify a variable over them, as in

$$\forall v \in V : ...$$

Sets are represented by (unary) functions and maps; such a set comprises all those elements for which the corresponding function or map is not $\perp$.

Def. 32: **(Set terms and their values)** *Given a vocabulary $\mathcal{V}$, an environment $(\mathfrak{I}, \mathbf{V})$, and assuming a global interpretation $\overline{\mathfrak{I}}$, we define the following kinds of set terms:*

All $s \in Func_{\mathfrak{I}}$ are set terms if $\mathcal{V}s = 1$. Their value is

$$[s]_{(\mathfrak{I},\mathbf{V})} = \{a \mid (\mathbf{V}f)a \neq \perp\}$$

All object terms t are set terms. Their value is

$$[t]_{(\mathfrak{I},\mathbf{V})} = \{a \mid (\Phi[t]_{(\mathfrak{I},\mathbf{V})}, a) \neq \perp$$

4.6.4 Embedding side-effect free computation

As we have seen in Section 4.2.4, the universe has a structure, and so do the objects in it, by virtue of being related to other objects by the Φ function. This means we can consider objects as representing data structures of various kinds. For instance, as we saw above, we can interpret any object x as representing a *set* of those objects y, for which $\Phi(x, y) \neq \bot$.

In many cases, we would like to compute with these structures, e.g. given an object s representing a set of integers and some object k representing an integer, we would like to compute some object s' that represents the set of all those integers in s that are smaller than k (we assume we have an object $<$ that represents the corresponding relation between integers).

Obviously, we could do something like this by writing rules, having first added a suitable unary function, say $s2$ to our vocabulary (we will use the following rule here in its 'intuitive' interpretation, which is good enough for the purpose of illustrating the idea):

```
...
function s2 arity 1 ;
...
rule ...
once
    ...
    do forall x ∈ s :
        if x < k then
            s2(x) := true
        end
    end
    ...
end
```

This would iterate over all elements of s and set $s2$ for an element to true if it is smaller than k.

This, however, suffers from a number of drawbacks:

- We need to extend our vocabulary with a new function, even though the values stored in there might not even be part of the 'conceptual' state, i.e. $s2$ is just some intermediate result on the way to computing the next state or some output.
- The changes we made to $s2$ are only visible in the next step, but not in the current basic rule.
- The functions of our vocabularies are not first-class objects, i.e. we may not pass them as parameters to other actor schemata or output them, because this can be done only with objects. So, for example, we could not construct that subset and communicate it to another actor. The structure we have thus created is, by definition, a 'private' one.

These limitations can be overcome by applying the concept of 'interpretation' to a somewhat broader class of 'symbols'—up to now, we have only used 'symbol' in the conventional sense of an identifier (string) naming some object. However, there is nothing in the definition of the symbol set or the global interpretation that prevents us from considering the expression

$$\{x \mid x \in s, x < k\}$$

as a symbol. The interpretation is free to map this symbol to any object. This could be the object that represents just the set that we want to describe. In other words, instead of the code above, we could write something like this:

rule ...
once
    ...
    let $s2 = \{x \mid x \in s, x < k\}$:
        $...//$ *do something using* $s2$
    end
end

Here, the let-construct declares a local variable symbol, which is visible inside its body only. So this solution does not suffer from any of the problems above: The declaration $s2 = \{x \mid x \in s, x < k\}$ does not require any entries in the vocabulary (just a local symbol is defined), and therefore no state change takes place, we can nest this kind of declaration arbitrarily, and the long symbol $\{x \mid x \in s, x < k\}$ even denotes an object, which can be output and passed to the creation functions of schemata.

However, a problem remains. Strictly speaking, $\{x \mid x \in s, x < k\}$ cannot be such a symbol, because its value depends on the values of the symbols s, k, and $<$, i.e. those that occur *free* in it—as opposed to x, which is *bound* inside this expression. We can solve this problem by using λ-abstractions [13], which have the following form:

$$\lambda v.t$$

where v is a variable symbol (a simple one, not an expression itself) and t is some term that does not contain any unbound variables but v. Then our global interpretation interprets the expression $\lambda v.t$ as an object a such that for any term t',

$$\Phi(a, [t']_{(\mathcal{I},\mathbf{V})}) = [t[v \mapsto t']]_{(\mathcal{I},\mathbf{V})}$$

This means that the function associated with the object a by $\overline{\Phi}$, when applied to the value of t' in some interpretation/valuation $(\mathcal{I}, \mathbf{V})$, yields the same value as replacing every free occurrence of v in t with the argument term t' (this is written as $t[v \mapsto t']$) and evaluating that term in $(\mathcal{I}, \mathbf{V})$.

In other words, the interpretation $\overline{\mathcal{I}}$ returns for each λ-abstraction an object that is associated with the *denotation* of that λ-abstraction by the structure $\overline{\Phi}$.

However, writing those λ-terms becomes clumsy and unnecessarily complicated. We will therefore consider terms like the above $\{x \mid x \in s, x < k\}$ as syntactic sugar for the following term:

$$(\lambda s.\lambda k.\lambda < .\{x \mid x \in s, x < k\})\ (s)(k)(<)$$

The first part of which will then be considered as a symbol and interpreted by $\overline{\mathcal{I}}$ in the way described above. More generally, for any expression e in our expression language, containing free variable symbols $v_1, ..., v_n$, we can write it as the 'standard' (object or set) term

$$(\lambda\ v_1....(\lambda\ v_n.e)...)(v_1)...(v_n)$$

by simply assuming $(\lambda\ v_1....(\lambda\ v_n.e)...$ to be an expression symbol appropriately interpreted by $\overline{\mathcal{I}}$.

The set of constructions we can use to compute with objects, if e.g. we have set-comprehension-style constructs such as $\{x \mid x \in s, x < k\}$, depends now only on the interpretation. From our perspective, whatever language we introduce to write expressions is merely a choice of $\overline{\mathcal{I}}$, allowing us to completely abstract away from these details for the semantic description of the rule language. The only requirement is, of course, that the expression language be free from side-effects: it must not change the state of an actor. Otherwise, it is fully orthogonal to our rule language. Appendix D presents a short overview of the expression language we are using inside our ASM component schemata.

4.7 Basic rules and their denotation

In this section we will define a number of *basic rules*. Recall from section 4.4 that basic rules are those that produce an update set and thus lead to an atomic state transition. Like for terms, we will describe the various kinds of basic rules and their denotation. We assume that there is a current valuation of the vocabulary $\mathbf{V}$ and also a context in which the rule is evaluated defined by an interpretation $\mathcal{I}$ of variable symbols.

The denotation of a rule is its combined effect. This includes, of course, the various updates it might produce, but since we are talking about a language for specifying actors this must also include the output produced as well as port connections to be added or removed, and the actors created. Furthermore, we will allow a rule to be non-deterministic (and provide appropriate construction to express such non-deterministic rules), so in general a rule will produce a set of possible results.

Thus we will think of a rule r as representing, denoting, a map [r] such that

$$[r] : (\mathcal{I}, \mathbf{V}) \mapsto \{(U_k, P_k, c_k^+, c_k^-, N_k) \mid k \in K\}$$

where U_k is an update set, the output function P_k is a map from output ports P^{out} to $\mathcal{U}^*$ (holding the output computed in that rule), c_k^+ and c_k^- are subsets of $P^{out} \times P^{in}$, denoting the connections to be added and removed, respectively, and $N_k \subseteq I$ as a set of new actor identifiers (of actors created by that rule). K is any index set. Note that $P^{in}, P^{out} \subset \mathcal{U}$. For convenience we will use the name P_λ for the output function that maps all output ports to λ. We will call one such tuple of update set, output, connection changes, and new actors a *(rule) outcome*, and a set of these resulting from a rule a *result set*.[9]

In the following we describe each rule first syntactically, including an informal explanation of its action, and then we provide a denotation for it. We distinguish between 'atomic rules', which do not contain any other rules but only terms, rules creating objects, and structured rules, which allows us to construct various types of compound rules, which contain one or more rules.

4.7.1 Atomic rules

Atomic rules are the primitive building blocks in constructing state transition behavior. The simplest atomic rule is the do-nothing rule. It looks as follows:

$$\underline{\textbf{skip}}$$

Its effect is that it has none. Therefore, its denotation is this:

$$[\underline{\textbf{skip}}] : (\mathcal{I}, \mathbf{V}) \mapsto \{(\emptyset, P_\lambda, \emptyset, \emptyset, \emptyset)\}$$

The most fundamental ASM rule that actually does something is the atomic update of a location to a new value. It has the following form:

$$t_L := t$$

where t_L is a location term and t an object term. This rule simply creates a singleton update set $\{(l, v)\}$ (Def. 23 on page 71), such that l is the result of the location term t_L and v the value of t:

$$\begin{aligned}[t_L := t] : (\mathcal{I}, \mathbf{V}) \mapsto & \{(\{(l, v)\}, P_\lambda, \emptyset, \emptyset, \emptyset)\} \\ & \text{with} \quad l = [t_L]_{(\mathcal{I},\mathbf{V})}, v = [t]_{(\mathcal{I},\mathbf{V})}\end{aligned}$$

While these first two kinds of rules were common in traditional ASM [51, 53], the following rules deal with connecting ASM components and communicating between them, and are therefore specific to our rule language. However, before we can discuss them, we need to describe how we can formulate terms that denote input/output ports.

As we have seen above, ports are represented by a special kind of object. Identifying a port (so we can use it in the rules we are about to discuss) implies

[9] Even though we use the same brackets for term values and rule denotation, they can be distinguished by the position of the $(\mathcal{I}, \mathbf{V})$—terms values are indexed with them, so that $[t]_{(\mathcal{I},\mathbf{V})}$ is the value of the term t in the context given by $\mathcal{I}$ and $\mathbf{V}$, while $[r]$ is the rule denotation and $[r](\mathcal{I}, \mathbf{V})$ the result set produced by the rule for the interpretation $\mathcal{I}$ and valuation $\mathbf{V}$.

being able to write a term whose value is the port. To this end, we make the following assumptions about the vocabulary of components, the universe $\mathcal{U}$, its structure Φ, and the global interpretation $\overline{\mathfrak{I}}$:

The interpretation valid inside each component (schema) contains a symbol *this*, which is bound to its actor identifier (cf. Section 4.9).

There are symbols *InputPorts* and *OutputPorts* defined in $\overline{\mathfrak{I}}$, which are bound to objects *in* and *out*, respectively, such that the associated maps Φin and Φout map each actor identifier $i \in I$ to objects in_i and out_i, which in turn are associated to maps from port names (usually strings, which are also assumed to be part of the universe, but this is not necessarily so—cf. Section 5.3) to port objects. If the actor does not have a port of the name, these maps are required to associate $\perp$ to this name.

For reasons that will become clear in the examples in Chapter 5, we need to be able to distinguish between 'external' and 'internal' input and output ports, with the intention that other actors only hook up to an external port of an actor, and that the internal ports are only connected by an actor itself.[10] This means we will assume a symbol *Extern*, which represents the subset of all external input and output ports.

For example, the value of the term *InputPorts (this)("A")* is the input port of name "A" of the current component (i.e. the one we evaluate the rule for that this term occurs in). Symbolically, if a is an actor index, and n a port name, we write $\Pi^{in}(a, n)$ and $\Pi^{out}(a, n)$ for the correspondingly named input and output port of that actor, respectively. The set $\Pi^{ext} \subset P^{in} \cup P^{out}$ is the set of all external ports.

The most straightforward use of an output port is to write a token (or a sequence of tokens) to it. The following rule does just that:

$$[t_p] \;\leftarrow\; t_{val}@t_{tag}$$

Here, t_p is a term denoting a port, t_{val} is also some term, and t_{tag} a term resulting in a tag (which will be real numbers in our examples). This rule sends a token to the port whose value projection is the value t_{val}, and whose tag projection is the value of t_{tag}. More precisely,

$$[[t_p] \;\leftarrow\; t_{val}@t_{tag}] : (\mathfrak{I}, \mathbf{V}) \mapsto \{(\emptyset, P, \emptyset, \emptyset, \emptyset)\}$$
$$\text{with} \quad P\,p = \begin{cases} w & \text{for} \quad p = [t_p]_{(\mathfrak{I},\mathbf{V})} \\ & \qquad \wedge w \in A^* \wedge p \in P^{out} \\ \lambda & \text{otherwise} \end{cases}$$
$$\text{such that} \quad \theta w = [t_{tag}]_{(\mathfrak{I},\mathbf{V})},\ \nu w = [t_{val}]_{(\mathfrak{I},\mathbf{V})}$$

Note that this denotation allows the value term to produce a sequence of tokens (assuming that the universe contains sequences). We extend the projection ν

[10]This is merely a convention and has no effect on our model of computation. Using this notion we might want to define a concept of well-behavedness of an actor that does not touch 'internal' ports of another actor, but this is a derived notion and is orthogonal to the model presented in Chapter 2.

over A^* elementwise, and $\theta w = t$ iff $\theta a = t$ for every token a in the sequence w. This rule has no effect if the port term does not yield an output port or the token sequence is not one in the alphabet A. Obviously, making A as large as possible, including in it, e.g., the set I of actor identifiers, or the ports, enhances the expressiveness of the framework, facilitating the exchange of components inside messages.

The next two constructions facilitate the creation and removal of connections between ports. Their syntax is as follows:

$$[p_{out}] \longrightarrow [p_{in}]$$

$$[p_{out}] \not\longrightarrow [p_{in}]$$

The p_{out} term must result in an output port object, and the p_{in} term in an input port object. Then the first rule will establish a connection between the two ports, while the second will remove a connection between them—it has no effect if the two ports are unconnected. Their denotation correspondingly is this:

$$[[p_{out}] \longrightarrow [p_{in}]] :$$
$$(\mathcal{I}, \mathbf{V}) \mapsto \begin{cases} \{(\emptyset, P_\lambda, \{(a,b)\}, \emptyset, \emptyset)\} & \text{if } (a,b) \in P^{out} \times P^{in} \\ & \text{with} a = [p_{out}]_{(\mathcal{I},\mathbf{V})}, b = [p_{in}]_{(\mathcal{I},\mathbf{V})} \\ \{(\emptyset, P_\lambda, \emptyset, \emptyset, \emptyset)\} & \text{otherwise} \end{cases}$$

$$[[p_{out}] \not\longrightarrow [p_{in}]] :$$
$$(\mathcal{I}, \mathbf{V}) \mapsto \begin{cases} \{(\emptyset, P_\lambda, \emptyset, \{(a,b)\}, \emptyset)\} & \text{if } (a,b) \in P^{out} \times P^{in} \\ & \text{with} a = [p_{out}]_{(\mathcal{I},\mathbf{V})}, b = [p_{in}]_{(\mathcal{I},\mathbf{V})} \\ \{(\emptyset, P_\lambda, \emptyset, \emptyset, \emptyset)\} & \text{otherwise} \end{cases}$$

4.7.2 Object creation

As mentioned before, when talking about *creating* objects we are formally only making objects accessible, which have so far not been accessible to the computation. The basic idea is that our universe of objects is sufficiently big so that we have an inexhaustible supply of 'fresh' objects if we need them, the so-called *reserve*, which contains all objects which we cannot access from any actor in the computation.

The criterion for accessibility is this:[11] If we can write a term in the current state of an actor, such that some object is the value of that term, this object is accessible inside the actor.

[11]There are some subtle issues here involving for instance quantification over infinite sets, which could be resolved in a number of different ways. We will not go into these here, and instead assume that we only quantify variables over finite sets, which makes sense anyway from a practical perspective.

The simplest rule to make an object (from the reserve) accessible, i.e. to create it in the sense above, has the following form:

<u>**import**</u> $s : r$ <u>**end**</u>

As usual, s is a symbol, and r a rule. The idea is that some object is picked from the reserve, bound to the symbol s, and then the rule r is executed which can use that object through the symbol. Describing the denotation in the terms above is straightforward:

$$[\underline{\textbf{import}}\ s : r\ \underline{\textbf{end}}] : (\mathfrak{I}, \mathbf{V}) \mapsto [r](\mathfrak{I}[s \mapsto a], \mathbf{V})$$

But the formal definition above is somewhat incomplete, since we use the object a without making explicit where it comes from. For our purposes it will be sufficient to say that it comes from the reserve, and has no properties other than being different from any other object we have used so far. It is this property of picking a 'fresh' object from the reserve that allows us to think about *import* as 'creating' an object.

We will use a variant of this rule to create a new component.[12] As mentioned before, in general components may depend on parameters, which control their internal state. These parameters are of course values of terms, which are passed to the creation mechanism constructing the initial state of the component. Syntactically, this looks as follows:

<u>**import**</u> $s =$ <u>**component**</u> $schema[par_1, ..., par_n]$:
    r
<u>**end**</u>

This creates a discrete event component of the schema corresponding to the value of the term *schema*[13] passing the parameter objects resulting from the evaluation of the terms par_i to the initialization rule of that schema, which we can think of as invoking the 'constructor' of a class (in an object-oriented programming language) to instantiate and initialize a new instance of that class[14] The important additional effect of this is that initialization creates a new actor, thereby affecting the set of new actor indices.

However, constructing a new component involves somewhat more than just initializing an actor. In particular, the new component may itself create further subcomponents, connect them to each other or to already existing components and so on. If we write the creation function for a given schema $s =$

[12] It would be more precise to talk about an actor identifier, an element $i \in I$, rather than an actor $\mathcal{A}_i$. However, this distinction will not be visible to us in the following, so we will allow ourselves to confound those two concepts and speak as if $i = \mathcal{A}_i$.

[13] We will leave open the question how this correspondence between the values of such a term and an actual schema is established. We may assume that these terms are required to evaluate to a string, which is then the name of the schema.

[14] We will discuss initialization of a component in section 4.9.

$[schema]_{(\mathcal{I},\mathbf{V})}$ as $Create_s[t, v_1, ..., v_n]$ (where t is the 'current' time tag, v_i is the value of par_i), then this creation function returns a non-empty set with elements that have the following structure:

$$Create_s(t, v_1, ..., v_n) = \{(a_k, N_k, c_k^+, c_k^-) \mid k \in K\}$$

Here, the a_k identifies the newly created actor itself (or rather the top-level actor of the newly created set of actors). N_k is the corresponding set of additional actors that were created , and c_k^+ and c_k^- are, as usual, the sets of connections to be added and removed. K is any index set (the creation of a new actor itself is a non-deterministic process and can produce any number of possible results). With this, the denotation of the rule may be given as follows:

$$\left[\begin{array}{l} \underline{\textbf{import}}\ s = \\ \ \underline{\textbf{component}}\ schemaname[par_1, ..., par_n] : \\ r\ \underline{\textbf{end}} \end{array}\right] : (\mathcal{I}, \mathbf{V}) \mapsto$$

$$\begin{array}{l} \{(U, P, c^+ \cup c_*^+, c^- \cup c_*^-, N \cup N_* \cup \{a\} \mid \\ \qquad (U, P, c^+, c^-, N) \in [r](\mathcal{I}[s \mapsto a], \mathbf{V})\} \\ \text{with} \\ \qquad (a, N_*, c_*^+, c_*^-) \in Create_{[schema]_{(\mathcal{I},\mathbf{V})}}(t, v_1, ..., v_n), \\ \qquad v_i = [par_i]_{(\mathcal{I},\mathbf{V})} \end{array}$$

We will discuss how to specify the constructor function in section 4.9.

4.7.3 Structured rules

Structured rules allow the construction of more complex rules from simpler ones. A very basic structured rule is the conditional, which allows us to execute a rule depending on the value of some term. It has the following form:

$$\underline{\textbf{if}}\ t\ \underline{\textbf{then}}\ r_1 \underline{\textbf{else}}\ r_2\ \underline{\textbf{end}}$$

where t is an object term and r_1 and r_2 are rules. As usual, we represent the 'false' value by $\perp$, so this rule is equivalent to r_1 if the term is not equal to $\perp$, and equivalent to r_2 if it is.

$$[\underline{\textbf{if}}\ t\ \underline{\textbf{then}}\ r_1 \underline{\textbf{else}}\ r_2\ \underline{\textbf{end}}] : (\mathcal{I}, \mathbf{V}) \mapsto \begin{cases} [r_1](\mathcal{I}, \mathbf{V}) & \text{if} \quad [t]_{(\mathcal{I},\mathbf{V})} \neq \perp \\ [r_2](\mathcal{I}, \mathbf{V}) & \text{otherwise} \end{cases}$$

We will allow more than one rule to be evaluated 'in parallel', in the sense that they are evaluated in the same environment, and neither is influenced by the effects of the other. Writing this for any number n of rules looks like this:

$$r_1, ..., r_n$$

Since each rule returns a set of possible outcomes, we have to 'merge' any combination of outcomes from all sets. This is straightforward for the update sets and the sets of connections to be created or removed (we simply union them). For example, assume we have two functions a and b of arity 0 in our vocabulary, which currently carry the values 7 and 25, respectively. The basic rule

$a := b, b := a$

creates the update set $\{(a, (), 25), (b, (), 7)\}$ (effectively exchanging the values of these functions in the next state, without the need for a temporary variable).

However, rules producing output potentially pose a problem, because we have to concatenate them in order to form a sequence of output tokens for each output port. Consider the rule:

$[p] \leftarrow 1@t, [p] \leftarrow 2@t$

Assuming p is (i.e. evaluates to) an output port, this rule produces either the sequence $(1, 2)$ or $(2, 1)$ at this output port. The denotation of a rule must contain all possible outcomes, therefore we must take all permutations of output into account. In other words, parallel combination of rules creating output is a potential source of non-determinism in our language.

The technicalities of the parallel combination of result sets (i.e. the set of all possible outcomes of a rule) can be found in App. C, where we define the *parallel combination operator* $\diamondsuit$, which is given a set of result sets $\mathbf{R}$ and returns the set $\diamondsuit\mathbf{R}$ of all possible parallel combinations of them.

Using the parallel combination operator, we can write the denotation of the parallel rule construct as follows:

$$[r_1, ..., r_n] : (\mathfrak{I}, \mathbf{V}) \mapsto \diamondsuit \{[r_i](\mathfrak{I}, \mathbf{V}) \mid i \in \{1, ..., n\}\}$$

This kind of parallel combination of rules contributes significantly to the power of Abstract State Machines, because it is possible to construct very large (arbitrarily large, as we will see in the next construct) update sets that change any part of the state in one atomic action, while retaining referential transparency during the construction of that update set.

However, using the above construction, a given specification can only combine a fixed number of result sets, because the rules have to be written down explicitly (so not only is the number fixed, but also finite).

This is somewhat generalized in the next rule, where we universally quantify a (new local) variable variable over some set, and combine the result sets of some rule for all values of this variable. We write this down as follows:

<u>**do**</u> <u>**forall**</u> $s \in t$:
    r
<u>**end**</u>

Here, s is any symbol, t is a set term, and r is a rule that forms the body of the construct. The result of evaluating this rule is defined to be this:

$$[\underline{\textbf{do}}\ \underline{\textbf{forall}}\ s \in t : r\ \underline{\textbf{end}}] : (\mathfrak{I}, \mathbf{V}) \mapsto \Diamond \left\{ [r](\mathfrak{I}[s \mapsto a], \mathbf{V}) \mid a \in [t]_{(\mathfrak{I},\mathbf{V})} \right\}$$

Note the way we make the symbol and its interpretation visible inside the body of the do-forall-construction, by making use of the substitution operation for interpretations.

Using this rule we can write rules that perform an unbounded, and indeed infinite, number of updates in one step—for instance, assume a symbol $\mathbb{N}$ be predefined that represents the set of natural numbers, and two binary function symbols f and g be defined. Then the following rule would construct rather large addition and multiplication tables, in one atomic step:

do forall $n_1 \in \mathbb{N}$:
  do forall $n_2 \in \mathbb{N}$:
    $f(n_1, n_2) := n_1 + n_2$,
    $g(n_1, n_2) := n_1 * n_2$
  end
end

So obviously, for purposes of interpretation and execution we require the sets over which we quantify variables to be finite.

So far, all rules were completely deterministic, apart from the output. Often, however, we would like to express non-determinism in the sense that we would like to pick an element from a set, and do something with this element only. For instance, the state transition of a Petri net should roughly have the following form:

choose $aTransition \in activatedTransitions$:
    $updatePlacesAroundThatTransition$
end

The choose-construct picks one out of any number of alternatives, again by quantifying a variable over a set, but this time existentially, instead of universally. In other words, instead of performing all rules and combining their result sets with the parallel combination operator, we choose one of the alternatives. The syntax looks like this:

choose $s \in t$:
    r_1
else
  r_2
end

Again, s is any symbol, t is a set term, and r_1 and r_2 are basic rules. r_1 is the body of the rule, and is executed for one of the elements in whatever t evaluates to. However, if this turns out to be the empty set, r_1 cannot be executed, and instead r_2 will be. (We will employ the syntactic convention that the else-branch may be left out if r_2 is the skip-rule.) Semantically, we must of course consider *all* possible rule outcomes, which effectively boils down to the union of all outcomes of all possible variations of the rule body. Formally, we thus have

$$\left[\begin{array}{c}\underline{\textbf{choose}}\ s\ \in\ t:\\ r_1\underline{\textbf{else}}\ r_2\ \underline{\textbf{end}}\end{array}\right] : (\mathcal{I}, \mathbf{V}) \mapsto \begin{cases} \bigcup_{a\in A}[r_1](\mathcal{I}[s\mapsto a], \mathbf{V}) & \text{if} \quad A \neq \emptyset \\ & \text{with} \quad A = [t]_{(\mathcal{I},\mathbf{V})} \\ [r_2](\mathcal{I}, \mathbf{V}) & \text{otherwise} \end{cases}$$

Often, we want to use the value of a term more than once inside a rule. In this case, we would like to define a local variable of that value. The let-rule construction allows us to do just that. It looks like this:

let
$s_1 = t_1, ..., s_n = t_n$:
r
end

As before, the s_i are symbols, the t_i terms, and r is a rule. The t_i are evaluated in the context valid outside of the let-construct, and r is then of course evaluated in the context where the s_i are bound to the values of the respective terms. The formal denotation is therefore this:

$$[\underline{\textbf{let}}\ s_1 = t_1, ..., s_n = t_n : r\ \underline{\textbf{end}}] : (\mathcal{I}, \mathbf{V}) \mapsto [r](\mathcal{I}[(s_1, ..., s_n) \mapsto (a_1, ..., a_n)], \mathbf{V})$$
$$\text{where} \quad a_i = [t_i]_{(\mathcal{I},\mathbf{V})}$$

4.8 Full rules and their denotation

In the last section we discussed a language for basic rules, giving each such rule a denotation of the following form (once again, K is just some index set):

$$[r] : (\mathcal{I}, \mathbf{V}) \mapsto \{(U_k, P_k, c_k^+, c_k^-, N_k) \mid k \in K\}$$

In other words, a basic rule computed a number of possible rule results (recall that in general a basic rule was non-deterministic), each such result consisting of an update set, some output sequence for each output port of the actor, sets of connections between ports to be created or removed, and a set of new actors created by that rule.

Eventually, our goal will be to use these basic rules to specify the response of a schedulable single-token actor, which looks like this:

$$\tau_\sigma : (\lambda, ..., a, ..., \lambda) \mapsto \{(\sigma'_k, w_k, c_k^+, c_k^-, N_k)\}_k$$

We will now discuss the concept of a *full rule*, which consists of any number of basic rules, and which we will use to define the firing function of an actor by binding such a rule to an input port such that, when the actor fires on a token from that port, the transition function uses this rule to determine its result.

In section 4.4 we introduced the general format of a full rule, which consisted of a number of those basic rules as follows:

rule $ruleName[p, t, v]$:
once
$\quad r_1$ **;**
$\quad \ldots$
$\quad r_j$
then
$\quad \overline{r}_1$ **;**
$\quad \ldots$
$\quad \overline{r}_k$
end

Here, each r_i and $\overline{r}_l$ is a basic rule as defined above, resulting in a set of rule outcomes, the *result set*. The three parameters of the rule stand for the port (-name) from which the token came, its tag and value, respectively.

Now if we assume that such a rule is bound to a port (section 4.9 will discuss how this is done), then obviously the transition behavior is given by the denotation of all full rules and their associations to the ports of an actor. In Section 4.4 we described the execution of a full rule as first executing the sequence of basic rules r_i once, and then iterating the sequence of basic rules $\overline{r}_i$ until all update sets resulting from these rules are empty. In this section, we will make this more precise.

The first step towards defining the semantics of a full rule will be the description of the result of executing a sequence of basic rules. Obviously, this will be a sequence of state transitions (of the actor) and outputs, connection sets, and new actors, resulting from the execution of the individual rules. We will write the denotation of a sequence of basic rules as $\langle r_1;...;r_n\rangle$ rather than using square brackets, because it results in a new valuation (instead of an update set), i.e. its denotation computes a new state (among other things). With this in mind, we can define the *simple denotation* of a sequence of basic rules as follows:

Def. 33: (Denotation of a sequence of basic rules.)

$$\begin{aligned}
&\langle r_1 ... r_n\rangle : (\mathfrak{I}, \mathbf{V}) \mapsto S_n \\
\textit{with} \quad & S_0 = \{(\mathbf{V}, P_\lambda, \emptyset, \emptyset, \emptyset, \mathbf{f})\} \\
& S_k = \{(\mathbf{V}_2, P_1 + P_2, c_1^+ \cup c_2^+, c_1^- \cup c_2^-, N_1 \cup N_2, b') \mid \\
& \qquad (\mathbf{V}_1, P_1, c_1^+, c_1^-, N_1, b) \in S_{k-1}, \mathbf{V}_2 = \mathbf{V}_1[U_k] \\
& \qquad (U_k, P_2, c_2^+, c_2^-, N_2) \in [r_k](\mathfrak{I}, \mathbf{V}_1) \\
& \qquad b' = (b \vee (U_k \neq \emptyset))\}
\end{aligned}$$

This means that in evaluating a sequence of rules, we keep the last state, concatenate the output sequences (we symbolize this with $P_1 + P_2$ which is a function defined as $(P_1 + P_2)p = P_1p + P_2p$), and simply build the union of the connection sets and the new actors. Obviously, we have to do this for all possible outcomes of a rule.

The last component of each outcome $(\mathbf{V}, P, c^+, c^-, N, b)$ is a Boolean value from the set $\{\mathbf{t}, \mathbf{f}\}$. It serves as a flag indicating whether the executions of the rules so far have produced a non-empty update set. The first flag (in the set S_0) is initialized to $\mathbf{f}$, denoting the value 'false'. At each step, we build the disjunction of the last flag and the test of non-emptiness of the current update set.[15] This value will of course be used for the termination criterion during the iteration of the second sequence of basic rules.

The above construction only allows us to execute a sequence of rules exactly once. Often, however, we would like to iterate a previously unspecified number of times, until some condition becomes true. This is exactly what the second sequence of basic rules, $\overline{r}_1$;...;$\overline{r}_k$ is for. In order to define the effect of the iterated execution of a sequence of rules, we will now proceed to define the iteration, finite iteration, and results of an iteration very similar to the Def. 9 and Def. 13 for the runs, finite runs, and results of runs of systems of actors. For simplicity, we will write $\mathbf{r}$ for a sequence of rules.

Def. 34: (Iteration, finite iteration, result of an iteration of a sequence of basic rules) *Given a sequence of basic rules* $\mathbf{r}$*, an* iteration *of* $\mathbf{r}$ *in an interpretation* $\mathfrak{I}$ *and starting from a valuation* $\mathbf{V}$ *is a sequence of structures*

$$(\mathbf{V}_k, P_k, c_k^+, c_k^-, N_k, b_k)$$

such that

$$(\mathbf{V}_0, P_0, c_0^+, c_0^-, N_0, b_0) \in \langle \mathbf{r} \rangle (\mathbf{V}, \mathfrak{I})$$

and for all k, a step *leading to the next structure in the sequence is defined as follows.*

If $b_k = \mathbf{f}$ (i.e. all update sets of this step were empty), we simply set

$$(\mathbf{V}_{k+1}, P_{k+1}, c_{k+1}^+, c_{k+1}^-, N_{k+1}, b_{k+1}) = (\mathbf{V}_k, P_k, c_k^+, c_k^-, N_k, b_k)$$

Otherwise, we choose some

$$(\mathbf{V}_*, P_*, c_*^+, c_*^-, N_*, b_*) \in \langle \mathbf{r} \rangle (\mathbf{V}_k, \mathfrak{I})$$

[15] It is not sufficient to test for equality of the current state with the original valuation, because (a) an update might not produce a change (if it updates to the same value that the point it updates has) and (b) a step may reverse the effect of its predecessor.

and define

$$\begin{aligned}
\mathbf{V}_{k+1} &= \mathbf{V}_* \\
P_{k+1} &= P_k + P_* \\
c^+_{k+1} &= c^+_k \cup c^+_* \\
c^-_{k+1} &= c^-_k \cup c^-_* \\
N_{k+1} &= N_k \cup N_* \\
b_{k+1} &= b_*
\end{aligned}$$

An iteration is finite *iff there exists a k such that $b_k = \mathbf{f}$, otherwise it is* infinite.

The result *of a finite iteration is* $(\mathbf{V}_{k+1}, P_{k+1}, c^+_{k+1}, c^-_{k+1}, N_{k+1})$.

Def. 35: (Iterated denotation of a sequence of basic rules.) *Given a sequence of basic rules* $\mathbf{r}$*, its* iterated denotation $\langle \mathbf{r} \rangle^*$ *is defined as follows:*

$$\langle \mathbf{r} \rangle^* : (\mathcal{I}, \mathbf{V}) \mapsto \{x \mid \textit{ ex. finite iteration of } \mathbf{r} \textit{ with result } x\}$$

This finally allows us to define the denotation of a full rule of the form given above. This denotation will map a valuation $\mathbf{V}$ and a number of parameters (which are objects) $v_1, ..., v_n$ to a set of possible outcomes, each of which has the form $(\mathbf{V}, P, c^+, c^-, N)$, it looks like the kind of structure we expect as a result of an iteration, which is almost (except for the representation of the output) identical to the structures returned by the transition function of an extended actor.

A rule declaration of the form

<u>rule</u> $name[par_1, ..., par_n]$:
<u>once</u>
    r_1 **;** ... **;** r_j
<u>then</u>
    $\overline{r}_1$ **;** ... **;** $\overline{r}_k$
<u>end</u>

consists of two parts, that do different things. It declares a name to stand for something (a rule), which is then defined. The rule itself is fully described by

$[par_1, ..., par_n]$: **<u>once</u>** r_1 **;** ... **;** r_j **<u>then</u>** $\overline{r}_1$ **;** ... **;** $\overline{r}_k$ **<u>end</u>**

Therefore, when defining the semantics of a full rule, it is this part that we will be looking at. The binding of this to the name of the rule (and subsequently to input ports) will be discussed below.

$$\left\langle \begin{array}{l} [par_1, ..., par_n] : \\ \underline{\textbf{once}}\ r_1 \ \underline{\textbf{;}} \ ... \ \underline{\textbf{;}}\ r_j \\ \underline{\textbf{then}}\ \overline{r}_1 \ \underline{\textbf{;}} \ ... \ \underline{\textbf{;}}\ \overline{r}_k \ \underline{\textbf{end}} \end{array} \right\rangle :$$

$$\begin{array}{l} (\mathcal{I}, \mathbf{V}, v_1, ..., v_n) \mapsto \{(\mathbf{V}_2, P_1 + P_2, c_1^+ \cup c_2^+, c_1^- \cup c_2^-, N_1 \cup N_2) \mid \\ \quad \mathcal{I}' = \mathcal{I}[(par_1, ..., par_n) \mapsto (v_1, ..., v_m)], \\ \quad (\mathbf{V}_1, P_1, c_1^+, c_1^-, N_1, b) \in \langle r_1 ... r_j \rangle (\mathcal{I}', \mathbf{V}), \\ \quad (\mathbf{V}_2, P_2, c_2^+, c_2^-, N_2) \in \langle \overline{r}_1 ... \overline{r}_k \rangle^* (\mathcal{I}', \mathbf{V}_1)\} \end{array}$$

This tells us what it means to 'run' a rule on a list of parameter values in a given state—the result is a new state, some output, connection sets, and a set of new actors. Two things need to be clarified before we can use this to describe an actor:

- We need to describe how an actor is set up, i.e. how its initial state is defined, and how rules are connected to its input ports, and also how input and output ports of an actor are defined.
- From this, we have to put the pieces together and describe the relation between the definition of an actor schema in our language as outlined in Section 4.3 and our formal concept of an interpreter schema from Def. 19.

This will be addressed in the following sections.

4.9 The semantics of a component schema

At this point, we have fully described the rule language, the structure of the state of an actor, how rules manipulate that structure, and how terms are evaluated. It is now time to bring these elements together and define the semantics of a component (or actor) schema. This is essentially given by the creation function, which we encountered in Section 4.7.2, and which had the following form:

$$Create_{schemaname}(t, v_1, ..., v_n) = \{(a_k, N_k, c_k^+, c_k^-) \mid k \in K\}$$

The t is the time tag corresponding to the 'current' time, i.e. the tag of the token which is currently fired on, and the v_i are constructor parameters. Recall that each a_k is the index of the newly created actor, N_k is a set of indices of actors created along with the one denoted by a_k (as 'subcomponents'), and the c_k^+ and c_k^- are the usual connection sets. K is again some index set, representing the non-determinism of the creation function.

Note that each $a_k \in I$ represents a schedulable actor, which according to Def. 12 has the following structure:

$$A_{a_j} = (\Sigma, \sigma_0, (P_\sigma)_{\sigma \in \Sigma}, (\tau_\sigma)_{\sigma \in \Sigma}, p_{fire}, p_{sched}, t_0)$$

In order to define the creation function of a schema, we must describe how all these items are constructed from a schema definition. To this end, we will now take a closer look at the structure of such a schema than we had done in Section 4.3. The general structure of an actor schema definition can be seen in Fig. 30.[16] Here, the dt_i are object terms, the $iset_i$ and $oset$ are set terms, $symb_{a,b}$, $rulename_i$ and $name$ as well as tm, var_i, par_i, p_i, t_i and v_i are symbols, and the n_1 are natural numbers (including 0). The $rulebody_i$ are the bodies of full rules.

<u>**class**</u> $name[tm, par_1, ..., par_n]$ <u>**is**</u>
    <u>**define**</u> $var_1 = dt_1, ..., var_h = dt_h$ <u>**;**</u>

    <u>**input**</u> $iset_1 : rulename_{r_1}, ..., iset_i : rulename_{r_i}$ <u>**;**</u>
    <u>**private**</u> <u>**input**</u> $ipset_1 : rulename_{rp_1}, ..., ipset_{ip} : rulename_{rp_{ip}}$ <u>**;**</u>
    <u>**output**</u> $oset$ <u>**;**</u>
    <u>**private**</u> <u>**output**</u> $opset$ <u>**;**</u>

    <u>**function**</u> $symb_{1,1}, ..., symb_{1,k_1}$ <u>**arity**</u> n_1 <u>**;**</u>
    ...
    <u>**function**</u> $symb_{l,1}, ..., symb_{l,k_l}$ <u>**arity**</u> n_l <u>**;**</u>

    <u>**initialize**</u> : $rulebody$ <u>**end**</u>

    <u>**rule**</u> $rulename_1[p_1, t_1, v_1] : rulebody_1$ <u>**end**</u>
    ...
    <u>**rule**</u> $rulename_m[p_m, t_m, v_m] : rulebody_m$ <u>**end**</u>
<u>**end**</u>

Fig. 30: The structure of an actor schema definition.

A text like this basically describes (denotes) an actor creation function of the kind mentioned above. We will now proceed to construct the details of this function from the denotation of the various parts of the definition. Note that the actor identifiers a_j are conceptually 'picked' from the reserve—we will build up the actor structures, and then allow any identifier from the reserve whose corresponding actor has these structures.

Before we start we need to deal with the formal schema parameters $par_1, ..., par_n$. These are symbols bound to the parameters of the creation function $t, v_1, ..., v_n$, and are to be bound to the par_i, resulting in an interpretation $\mathfrak{I}_0$.

[16] The form presented here is the canonical form. There are a few syntactic abbreviations to this form, such as using the <u>**attribute**</u> keyword for functions with arity 0. As these can be trivially translated into the canonical form, we will not clutter the semantics definition with them.

Additionally, we bind the declared symbols var_i to the values of the respective terms dt_i and we obtain the class-wide interpretation $\hat{\mathfrak{I}}$ defined as

$$\begin{array}{ll} & \hat{\mathfrak{I}} = \mathfrak{I}_h \\ \text{with} & \mathfrak{I}_{i+1} = \mathfrak{I}_i[var_{i+1} \mapsto [dt_{i+1}]_{(\mathfrak{I}_i, \mathbf{V}_\perp)}] \\ \text{and} & \mathfrak{I}_0 = \mathfrak{I}_\perp[(tm, par_1, ..., par_n) \mapsto (t, v_1, ..., v_n)] \end{array}$$

Here, $\mathfrak{I}_\perp$ is the empty interpretation, i.e. $dom\mathfrak{I}_\perp = \emptyset$ and $\mathbf{V}_\perp$ is the valuation of the actor prior to initialization, such that all functions are $\perp$ at all points.

The define-block allows the definition of symbols to stand for the values of arbitrarily complex terms—it does not add anything semantically, but it provides a convenient mechanism for making specifications shorter and more readable (we will see a first application of this in the Petri net example below).

This interpretation will be the one that defines the visible symbols inside the actor, with the exception of the *this* symbol—this must obviously be defined differently for each actor returned by the creation function. Given an actor identifier $a \in I$, we define the *local interpretaton of actor* a as

$$\hat{\mathfrak{I}}_a = \hat{\mathfrak{I}}[\text{"}this\text{"} \mapsto a]$$

In order to define the creation function for a schema, it is helpful to first extract the information contained in its definition into a more useful format. We will interpret the actor schema of Fig. 30 to define

- a vocabulary
- input port names and output port names
- a map from port names to denotations of rules
- an initialization function

Def. 36: (Vocabulary, port map, port names and initialization function of a parameterized schema) *Given a schema definition of the form in Fig. 30 this defines a vocabulary* $\mathcal{V}_{name}$ *as follows:*

$$\mathcal{V}_{name} s = \begin{cases} n_1 & \textit{if} \quad s \in \{symb_{1,1}, ..., symb_{1,k_1}\} \\ \dots \\ n_l & \textit{if} \quad s \in \{symb_{l,1}, ..., symb_{l,k_l}\} \\ \perp & \textit{otherwise} \end{cases}$$

It also defines sets of names for input and output ports, $Inputs_{name}$ *and* $Outputs_{name}$*, respectively, and subsets of these of external input and output ports,* $Inputs^{ext}_{name}$ *and* $Outputs^{ext}_{name}$*, respectively, and a function, called* port map*,* $Port_{name}$ *mapping the input port names to the rule denotation corresponding to it. As opposed to the vocabulary, these items, of course, depend* *on the actual schema parameters* $(t, v_1, ..., v_n)$*, which define the interpretation* $\hat{\mathfrak{I}}$ *as described above.*

$$\begin{aligned}
Inputs^{ext}_{name}(t, v_1, ..., v_n) &= \bigcup_{a \in \{1,...,i\}} [iset_a]_{(\hat{\mathfrak{I}}, \mathbf{V}_\perp)} \\
Inputs_{name}(t, v_1, ..., v_n) &= Inputs^{ext}_{name}(t, v_1, ..., v_n) \\
&\quad \cup \bigcup_{a \in \{1,...,ip\}} [ipset_a]_{(\hat{\mathfrak{I}}, \mathbf{V}_\perp)} \\
Outputs^{ext}_{name}(t, v_1, ..., v_n) &= [oset]_{(\hat{\mathfrak{I}}, \mathbf{V}_\perp)} \\
Outputs_{name}(t, v_1, ..., v_n) &= Outputs^{ext}_{name}(t, v_1, ..., v_n) \cup [opset]_{(\hat{\mathfrak{I}}, \mathbf{V}_\perp)} \\
Port_{name}(t, v_1, ..., v_n) &: s \mapsto \langle [p_r, t_r, v_r] : rulebody_r \ \underline{\mathbf{end}} \rangle \\
&\quad \mathit{for} \quad s \in [iset_r]_{(\hat{\mathfrak{I}}, \mathbf{V}_\perp)}, s \in [ipset_r]_{(\hat{\mathfrak{I}}, \mathbf{V}_\perp)}
\end{aligned}$$

Finally, the schema definition also defines an initialization rule $Init_{name}$ *as follows:*

$$Init_{name} = \langle [] rulebody \ \underline{\mathbf{end}} \rangle$$

The vocabulary is only well-defined if all function symbols appear at most once in the function definitions. Likewise, all values of the $iset_i$ terms of input port names must be pairwise disjoint.

We also need to bring the output produced by a rule (cf. Section 4.7.1), which is a map from output ports to A^*, into the tuple format we need to describe actor behavior. According to Def. 4 P^{out}_a is the set of output ports of actor a. A rule returns a function P mapping ports to output sequences. A function $Output_a$ transforms such a function into the appropriate output tuple for actor $\mathcal{A}_a$ as follows:

$$\begin{aligned}
& Output_a : P \mapsto \omega_a s \\
& \text{with} \quad s \quad \text{such that} \\
& \qquad \forall p \in P^{out}_a : p\, s = Pp, \\
& \qquad \forall p \in P^{out} \setminus P^{out}_a : p\, s = \lambda
\end{aligned}$$

Now we can define the semantics of such an actor schema definition in terms of the creation function it denotes.

Def. 37: (Denotation of actor schema definition) *Given the schema definition in Fig. 30. It defines a function* $Create_{name}$ *such that*

$$Create_{name}(t, v_1, ..., v_n)$$

returns the largest set of structures (a, N, c^+, c^-)*, such that the following holds for each of its elements:*

- *a is an actor identifier from the reserve, $\mathcal{A}_a$ is a discrete-event component, such that*

$$A_a = (\Sigma_{\mathcal{V}_{name}}, \mathbf{V}_0, (P_{\mathbf{V}})_{\mathbf{V} \in \Sigma_{\mathcal{V}_{name}}}, (\tau_{\mathbf{V}})_{\mathbf{V} \in \Sigma_{\mathcal{V}_{name}}}, p_{fire}, p_{sched}, t_0)$$

- $(\mathbf{V}_0, P_0, c^+, c^-, N) \in Init_{name}(\hat{\mathfrak{I}}_a, \mathbf{V}_\perp)$*, i.e. the initialization rule of the schema creates the initial valuation as well as the connection sets and the set of new actors.*

- $p_{fire} = \Pi^{in}(a, "fire"), p_{sched} = \Pi^{out}(a, "schedule")$

- *The prefix functions $P_{\mathbf{V}}$ are as defined in Def. 10 (since the actor is a single-token actor).*

- *If $P_0(\Pi^{out}(a, "schedule")) = \lambda$, then $t_0 = \infty$, else $P_0(\Pi^{out}(a, "schedule")) = wa$ (with $w \in A^*$ and $a \in A$) and then $t_0 = \theta\ a$.*

- *Finally, the transition functions $\tau_{\mathbf{V}}$ have to be defined as follows:*

$$\begin{aligned} \tau_{\mathbf{V}}(\lambda_{i-1}, a, \lambda_{n-i}) = \{(&\mathbf{V}', Output_a(P), c^+, c^-, N) \mid \\ &(\mathbf{V}', P, c^+, c^-, N) \in Port_{name}(p)(\hat{\mathfrak{I}}_a, \mathbf{V}, p, \theta a, \nu a), \\ &p = \alpha_a \circ \pi_i\} \end{aligned}$$

The key elements in this definition are the ones concerned with initialization and state transition. Note that the initialization rule may produce output, but all of it is discarded except for the output at the *schedule* output port. This determines the scheduled firing time—if the initialization rule does not produce output at this port, it becomes ∞, i.e. the actor is unscheduled. If the initialization rule produces output, it is only the time stamp of the last token produced that will determine the scheduled firing time of the actor.

This concludes our presentation of the ASM-based description of discrete-event components (or rather component schemata). What is left is to relate a parametric component schema definition to the notion of visual language interpreter schema developed in the last chapter.

4.10 Interpreter schemata and graph-defined schemata

In Def. 19 in the last chapter we defined an *interpreter schema* for a visual language $\mathcal{L}(\mathbf{P})$ to be a function

$$\mathcal{I}(t, G, v_1, ..., v_n) = \{(a_k, N_k, c_k^+, c_k^-) \mid k \in K\}$$

where t is a time tag, G a graph in that language, the v_i are some values, and the i is the actor identifier. In other words, it takes a time, a graph, and some parameters, and returns a set of structures containing an actor identifier, a set of further new actors, and connection sets. The actor creation function developed and defined in the last section, had the following form:

$$Create_{schemaname}(t, v_1, ..., v_n) = \{(a_k, N_k, c_k^+, c_k^-) \mid k \in K\}$$

They differ only in that the interpreter takes a graph as one of its parameters. This means that in order to be able to use ASM component schema definitions as definitions of visual language interpreter schemata, we only need to show how to represent a graph as an object (or rather structure of objects) in our universe.

Recall from Def. 17 that an attributed graph had the following structure:

$$(V, E, s, d, \mu, \overline{\mu})$$

where V and E where sets of vertices and edges, $s, d : E \longrightarrow V$ mapped an edge to its start and end vertex, $\mu : (E \cup V) \times A \longrightarrow \mathcal{U}$ is the attribution function (A is the set of valid attribute names) for all graph objects (edges and vertices), and $\overline{\mu} : A \longrightarrow \mathcal{U}$ is the attribution function for the graph itself. Representing these attribution functions in our universe is straightforward—since any object a is associated with a map $\Phi a : \mathcal{U} \longrightarrow \mathcal{U}$, we could simply require $(\Phi a)(x) = \mu(a, x)$ for graph objects a and $(\Phi g)(x) = \mu x$ for the object g representing the graph itself, for any attribute name x.

The graph structure we can represent by introducing four symbols (into the global interpretation $\overline{\mathfrak{I}}$), say *vertices*, *edges*, *src*, and *dst*, such that the maps associated with the objects bound to them would return (an object representing) the set of vertices and edges (for a graph), and, respectively, the source and destination vertex (for an edge object). This allows us to parameterize a schema for an interpreter of some visual language with a graph and a set of parameters in order to obtain a discrete event component realizing the functionality of the visual program represented by the graph.

Usually, however, we would like to abstract from the concrete visual language used to specify a given component—in fact, it is a key feature of our approach to the interoperability between different visual notations that we can use an actor without knowing what language it was written in. We therefore introduce the notion of a *graph-defined schema*, which is basically an actor schema derived from the schema for a visual language interpreter and a concrete graph:

Def. 38: (Graph-defined schema) *Given a visual language interpreter schema $\mathcal{I}$ and a graph G of that visual language, the* graph-defined schema $\mathcal{C}_G$ *is defined as follows:*

$$\mathcal{C}_G(t, v_1, ..., v_n) = \mathcal{I}(t, G, v_1, ..., v_n)$$

This means that we can view graphs (i.e. visual programs) themselves as representing actor schemata, rather than viewing only interpreter schemata in this way, and graphs only as parameters for these interpreters. In fact, this will be the view of the *user* of a visual language (as opposed to the *author* of that language): ignoring the technicalities of how interpreters are related to graphs, indeed ignoring the very existence of interpreters, users conceive of pictures/visual programs/graphs as denoting actor schemata. We thus have made visual the specification of classes of actors.

4.11 Small examples

With the actor schema definition language fully defined, we can now use it to define the first small examples—Chapter 5 is devoted to further, slightly larger applications. We will use our new description technique first on the two examples we had already defined using plain 'mathematical' notation in Ex. 10 and Ex. 11, viz. the interpreters for finite state machines and Petri nets.

Alg. 1: (Schema definition of an FSM interpreter)

```
1  class FSMInterpreter[tm, G] is
2        input {"input"} : handleInput,
3              {"fire"} : doNothing ;
4        output {"output", "schedule"} ;
5
6        attribute currentState ;
7
8        initialize :
9                do forall e ∈ edges(G) :
10                  if src(e)("type") = "InitMarker" then
11                     currentState := dst(e)
12                  end
13               end
14       end
15
16       rule handleInput[p, t, v] :
17       once
18            choose e ∈ {e | e ∈ edges(G), src(e) =
19                             currentState, e("Input") = v}:
20                  currentState := dst(e),
21                  [OutputPorts(this)("output")] ← e("Output")@t
22            end
23       end
24
25       rule doNothing[p, t, v] :
26            skip
27       end
28 end
```

4.11.1 Example 1: FSM interpreter schema

The full definition of our FSM interpreter as an ASM component is presented in Alg. 1. Recall that it is driven by tokens arriving at some 'data' input (as opposed to the input used by the scheduling mechanism to fire the actor explicitly), which may or may not cause a state transition. Correspondingly, in Line

3 it defines its *fire* input to be associated with a rule of the descriptive name *doNothing*, which in Line 26 does just that.

The vocabulary of an FSM interpreter actor is as simple as one would expect: It consists of exactly one attribute (function of arity zero) containing the *currentState* (Line 6). For simplicity, these states are directly represented by the corresponding vertices of the graph. This can be seen in the initialization rule (Lines 8-14), which iterates over all edges, picks the one (there must be exactly one) with the "InitMarker" vertex at its source, and sets the 'currentState' attribute to the vertex at the other side of the edge.

The actual state transition consists of a simple choose-construct (Lines 18-22). This picks one of the activated transitions (where the activation condition is that it starts at the current state and has the proper "Input" attribute that corresponds to the token value v), updates the state and outputs the "Output" attribute of the chosen transition.

Note that this actor never produces any scheduling messages, and will therefore never receive any firing tokens—nevertheless, in order to make it a legal schedulable actor, we have to include the two ports in its definition.

Alg. 2: (Schema definition of a Petri net interpreter)

```
1  class PNInterpreter[tm, G] is
2      define
3          P = {v | v ∈ vertices(G), v("type") = "Place"},
4          T = {v | v ∈ vertices(G), v("type") = "Transition"} ;
5      input {"fire"} : step ;
6      output {"schedule"} ;
7
8      function M arity 1 ;
9
10     initialize :
11     once
12         [OutputPorts(this)("schedule")] ← ⊥@tm,
13         do forall p ∈ P :
14             M(v) := v("initialTokens")
15         end
16     end
17
18     rule step[p, now, v] :
19     once
20         choose t ∈ {t |t ∈ T,
21                     ∀e ∈ edges(G) :
22                     (dst(e) = t ⇒ e("Weight") ≤ M(src(e)))} :
23
24             do forall e ∈ edges(G) :
25                 if t = dst(e) then
26                     M(src(e)) := M(src(e)) − e("Weight")
27                 end,
28                 if t = src(e) then
29                     M(dst(e)) := M(dst(e)) + e("Weight")
30                 end
31             end,
32             [OutputPorts(this)("schedule")] ← ⊥@now
33         end
34     end
35 end
```

4.11.2 Example 2: Petri net interpreter schema

The Petri net interpreter schema definition in Alg. 2 is only slightly more complicated than the previous example. It contains fewer ports, because it is not driven by any data input, but exclusively by firing messages—consequently, it has to schedule itself—which is what it does in Line 12, by sending a message to its own "schedule" output with time stamp tm, the time parameter of the actor, its creation time.

This schema includes a define-block, which defined the symbols P and T to stand for the sets of ”Place” vertices and ”Transition” vertices of the graph, respectively.

The vocabulary also contains only one function, though this time of arity 1: the function M represents the marking, mapping ”Place” vertices to non-negative integers.

Initialization consists of setting up this marking by iterating over the ”Place” vertices (here we make use of the definition of P) and setting M for each such vertec to the value of its ”initialToken” attribute.

The actual update of the neighboring places is done in an iteration over all edges. If an edge is connected to the selected transition, the marking of the place on the other side of that edge is changed according to the ”Weight” attribute of the edge. Here we must assume that a place and a transition are not connected by more than one edge, otherwise we get inconsistent updates here.

If there is no activated transition the choose-construct will not execute its body (the else part is omitted because it is a skip-rule), thus the actor will not reschedule itself.

Note that, as an actor, such a Petri net is a rather useless thing, since it does not communicate with its environment—other than scheduling itself until it is dead. We will address this issue in Chapter 5.

4.12 Discussion and related work

In this chapter, we have essentially developed an actor description language based on a well-known state-based specification language, Abstract State Machines. We have brought together our actor model, with its notion of composition and communication, and ASM, which are focussed on the definition of discrete state transition behavior. Many approaches to defining languages for describing concurrent systems have approached the problem from another angle, by extending a *functional* language with notions of component, channel, communication etc. Examples of this include Erlang [10], Obliq [35], Facile [47], and Pict [86]. The first two are essentially actor languages (with Erlang having roots in Prolog, despite being essentially functional in flavor), based on objects for keeping the state of the actors and direct addressing of actors as communication mechanism, while the latter two are based on process algebraic notions.

The main focus of our model of computation, apart from communication between components, is the state transition behavior of individual components and the structure of their state. This is why we started from an essentially state-based language that had a very general concept of state structure, and provided a powerful set of constructs for manipulating the state. In order to accommodate the features of our model, we have added a number of constructions to basic Abstract State Machines:

- A notion of component, and a component schema from which to instantiate/create components.

- Input ports, by which tokens could enter a component and trigger the execution of a rule, and output ports by which a rule could output data to other components. Along with these we needed constructs to output tokens and to connect and disconnect ports.

- Binding of rules to input ports. In this way, rules were used to describe the component's reaction to a token.

Traditional Abstract State Machines have been defined in [51, 53] as a way to describe algorithms at arbitrary levels of abstraction while maintaining an essentially operational style of description. In contrast to other state-based specification techniques (such as Schönhage's storage modification machines [40] and random access machines [18]), the state of an ASM is an algebra (which is why initially ASM were known as 'evolving algebras'), a very general concept facilitating the representation of complex structures without the need for much 'coding'.

ASM have been used as a specification and modeling device in many concrete case studies (e.g. the railroad crossing problem [14, 55], the group membership protocol [56], a production cell [23], the Broy-Lamport problem [63]).

A very important part of the ASM work is the application of this technique to the specification of programming language semantics. This includes a variety of imperative languages (C [54], C++ [109], Cobol [105], Java [27, 26], Modula-2 [83], Oberon [73]), the hardware description language VHDL [22, 94], the parallel programming language Occam [21, 57], functional languages [104], and a large body on work in logic programming languages (e.g. Prolog [19, 20], CLP(R) [25]), including the definition of the ISO standard of Prolog [24].

ASM were applied to the specification of visual notations, such as SDL [49] and to Petri nets [48]. However, all these applications focused on the description of one particular visual notation (the same applies to the specification of textual languages cited before), not on the mapping of a range of languages to an underlying common model of computation.

In the context of textual languages, there is some work on general frameworks for their syntax and semantics description where the latter is based on ASM—most notably [87] and [72]. Here, the focus is on a uniform specification technique, mostly (but not exclusively) for imperative languages, and the automatic generation of tools (compilers, interpreters, debuggers) from such a specification.

Abstract State Machines in their most basic form do not include a composition technique. Most work addressing this problem focused on refinement of ASM descriptions from high-level specifications to efficiently executable implementations [15, 23]. Another line of research developed concurrent composition of ASM [50, 53], where ASM *agents* (which can be thought of as threads of control) communicated essentially via a global state (a common valuation of a vocabulary)—despite being called 'distributed ASM'.

Recently, work has been done on incorporating composition concepts of object-oriented languages into ASM, e.g. [8, 112] and our own previous work on Object-based ASM [67] and Mapping Automata [66]. While these approaches do address some issues arising in composition of ASM (encapsulation, locality of state, threads of control, transfer of control between agents etc.) to varying degree, they provide no formal interface to a more generic model of computation—i.e. they define the model of computation in their own terms, rather than being themselves embedded into a more general framework. Furthermore, their focus is on state and how this evolves over time, rather than dataflow and communication between agents—for instance, none of the above approaches would explicitly represent the communication structure of the system, or provide a notion of 'port'.

Traditionally, the focus in ASM research is on the modeling aspect, and on the expressiveness of the language, on ways to improve or demonstrate its applicability to the specification of some system or language. In recent years, however, progress has been made in the area of formal proofs based on ASM descriptions [95, 96] and on model checking ASM [111]. Of course, not all ASM can be subject to automatic verification by these methods, but extending their applicability certainly adds a very important incentive to use ASM in the first place. Being able to automatically derive interesting properties from an ASM description of an actor would, of course, significantly enhance the value of such a description.

5

Applications

In this chapter we will apply the concepts and the language developed in the previous chapters to the definition of a few visual languages. In this, we will focus on the semantics, and will describe the syntax only insofar as attributes of vertices and edges are needed in the definition of interpreters. Syntactical constraints (syntax predicates) will only be formulated informally.

In choosing the examples we were guided by two considerations: to cover a reasonable variety of visual notations with different characteristics (from state-based to purely dataflow-oriented notations), and to address the main architectural issues arising in the design of interpreters. The following prototype solutions address the following issues:

- computation inside the model, i.e. the evaluation of 'expressions' inside the model (usually formulated as text) and its integration into the state transition mechanism of the interpreter
- adding a notion of time, i.e. non-instantaneous temporal behavior, involving scheduling of a component at a proper time
- interpreters with a variable number of input/output ports depending on the actual graph
- creation and connection of other components
- connection/disconnection of components during runtime (i.e. dynamic network structures)

These concepts will be presented in a series of increasingly complex interpreter schemata. We will start by extending the finite state machine interpreter schema from the end of the previous chapter.

5.1 State-based languages: extended FSM

The state machine language for which we presented the interpreter at the end of the previous chapter realizes a rather restricted kind of state machine. We will extend this language in this section in two steps.

5.1.1 Adding computation

One way to make this language more expressive is to allow actual computation to be performed in the course of a state transition, which may operate on the input token consumed by the transition, and may modify a number of 'variables' that can contain arbitrary pieces of information.[1] Whether a transition between states is possible may depend on the value of the input as well as the values assigned to those variables. The same holds for the output computed and the new variable values. We will call this kind of state machine an *extended FSM*.

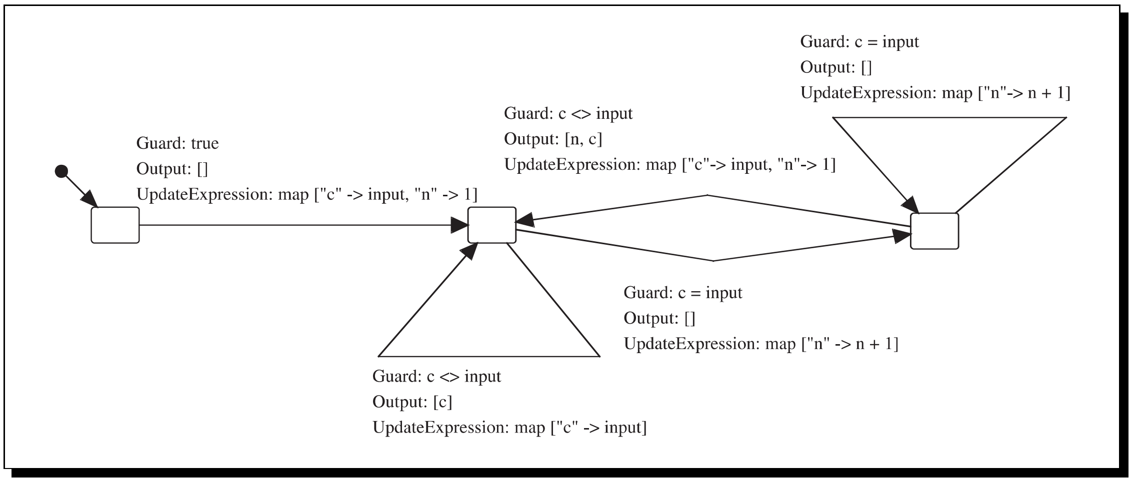

Fig. 31: A simple run-length coder as an extended FSM.

Fig. 31 shows an extended FSM that implements a very simple form of run-length coding. Each transition is inscribed with three attributes, all of them expressions (cf. App. D for a summary of the syntax of the expression language). The *Guard* attribute contains an expression that evaluates to *true* if the transition is possible, and *false* otherwise. It is evaluated in an environment that contains the variables of the state machine (n and c in this example) as well as the symbol *input*, which is bound to the respective input token that the state machine is about to fire on. The *Output* attribute contains an expression computing the output tokens, and the *UpdateExpression* attribute contains an expression that computes a map of new assignments to the variables to the state machine.

Assuming initially c and n to be undefined, the first transition from the initial state sets c to the input token received, and n to 1, no output is being produced. Now in the middle state, either of the two transitions leading away from it may

[1]This means of course that strictly speaking the state space of such an automaton may no longer be finite.

be activated, depending on whether the input token is equal to c. If it is not, the state transition will lead back to the middle state, setting c to the new input token and outputting the old value of c. If it is equal, the right state is entered, no output is produced, but n is incremented by one. Similarly, we remain in the right state as long as the input token remains the same, incrementing n. As soon as we encounter a new input token, we change back to the middle state, setting c to the new token, n to 1, and writing out two tokens, viz. the number of identical tokens we encountered on the input, and the token itself.

In addition to the *Guard*, *Output*, and *UpdateExpression* attributes of edges, the graph itself has an *InitialValues* and a *Parameters* attribute. The former is an expression that results in a map mapping the variable names to their initial values. The *Parameters* attribute is a list of formal parameter names, which get bound to actual parameter values passed when creating an extended FSM actor.

The following Alg. 3 shows an interpreter schema for our extended FSM notation. In Line 3 the local environment, i.e. the one resulting from binding the formal parameter names to the actual parameter values is defined—we assume a predefined function *createMap* that takes two lists and returns a map that maps each element of the first list to the element of the second list at the corresponding position, or $\perp$ if the second list is shorter than the first.

The component has two input ports (*input* for the data tokens, *fire* for fire messages from the scheduling mechanism, which are ignored by these actors) and two output ports (*output* for data generated during a transition, *schedule* for schedule messages, which these actors do not generate). In addition to the actor schema for simple FSMs, the state has an attribute *currentVars* which contains a map mapping the variable names of the actor to their current values (Line 9).

As for the simple FSM, the initialization 'looks' for the state connected to the initial state marker and sets this as the initial value of the *currentState* attribute (Lines 12-16). Additionally, it computes the initial mapping of its variables (Line 17)—here we assume the existence of an evaluation function *eval*, that takes an expression (here the graph attribute *InitialValues*) and an environment (in this case the one resulting from the binding of the parameters, *localEnv*) and returns the value of that expression. This will be a map, which is stored in the *currentVars* attribute.

The transition behavior of the extended FSM is defined by the *handleInput* rule, which does the following:

1. Select a transition from the set of possible transitions (as defined by current state and guard of the transitions)—Lines 25-27. Note how the environment for the evaluation of the guard is constructed (Line 23): to the local environment (the bound formal parameters) we 'add' the current assignment of the local variables and the mapping of the symbol "input" to the input value v. We assume that in this addition values $\neq \perp$ of a map supersede values for the same point of maps to the left of it, i.e. local variables override parameters of the same name.

2. Assign the next state to *currentState* (Line 28).

3. Update the assignment of variables by evaluating the *UpdateExpression* attribute of the chosen transition and adding it to the current assignment (using the overriding behavior of the add-operation for maps just mentioned)—Lines 29-30.

4. Finally, write the result of the evaluation of the *Output* attribute of the transition to the *output* port (Lines 31-32).

Alg. 3: (An extended FSM interpreter schema.)

```
1  class XFSMInterpreter[tm, G, pars] is
2      define
3          localEnv = createMap(G("Parameters"), pars) ;
4      input {"input"} : handleInput,
5            {"fire"} : doNothing ;
6      output {"output", "schedule"} ;
7
8      attribute currentState ;
9      attribute currentVars ;
10
11     initialize :
12             do forall e ∈ edges(G) :
13                 if src(e)("type") = "InitMarker" then
14                     currentState := dst(e)
15                 end
16             end,
17             currentVars :=
18                 eval(G("InitialValues"), localEnv)
19     end
20
21     rule handleInput[p, t, v] :
22     once
23         let env = localEnv+currentVars
24                             +["input" ↦ v]:
25             choose e ∈ {e |e ∈ edges(G),
26                             src(e) = currentState,
27                             eval(e("Guard"), env) ≠ ⊥}:
28                 currentState := dst(e),
29                 currentVars := currentVars+
30                     eval(e("UpdateExpression"), env),
31                 [OutputPorts(this)("output")] ←
32                     eval(e("Output"), env)@t
33             end
34         end
35     end
36
```

```
38        rule doNothing[p, t, v] :
39            skip
40        end
41    end
```

5.1.2 Adding a notion of time

The next addition to the extended FSM visual language is a concept of time. Up to now, all state transition occur instantaneously, and the FSM/extended FSM interpreters did not interact with the scheduler. Now we want to create an extended FSM that 'consumes' time while firing.

There are several possibilities to include such behavior. Of particular importance are the following questions: (a) How is the choice of the transition to be taken influenced by the time model? (b) Where does the delay come from and how is it specified? (c) What happens to input tokens that arrive during the occurrence of a transition?

We will choose a very simple design that answers these questions as follows: (a) Not at all. (b) From an expression attribute of the transition. (c) They are discarded. Intuitively, each transition is attributed with an expression that is supposed to compute a delay, i.e. the time that this expression will take when it is chosen to fire. When an expression is chosen, its delay is evaluated and the actual update of state and variables, as well as the output take place at the end of the transition, i.e. when the delay is expired.

This means that we have two rules participating in the firing: The first selects a transition in response to an input token and schedules the second rule (which is bound to the *fire* input instead of the *doNothing* rule in the previous example). This actually updates state and variables according to the transition taken. It is important that the first rule recognizes that it has already scheduled the actual state transition, so that it can discard tokens arriving in the meantime.

Alg. 4 shows how this can be realized in our framework. It has a few more attributes to hold the new values of the variables, as well as the output values while the transition is in process. We will use the value of the *transition* attribute as an indicator of whether the extended FSM is currently idle or is performing a transition—if it is $\neq \perp$, a transition is in progress, and is contained in the attribute.

The main differences to Alg. 3 is the division of responsibility between the *handleInput* and the *doTransition* rules. As defined above, the former does nothing if the *transition* attribute is $\neq \perp$. If it is $= \perp$, however, and input arrives, the *handleInput* rule picks a transition like before, but instead of executing it, the transition edge is stored in the *transition* attribute (Line 32). Then, the new variable assignment and the output value are stored in *newVars* and *outValue*, respectively (Lines 33-35). Finally, a scheduling message is written to the *schedule* output port for the current time + the result of evaluating the *Delay* expression (Lines 36-37).

Firing this actor at the scheduled time then results in the relevant updates being peformed (*currentState* and *currentVars*) as well as the precomputed output being written (Lines 46-48). At the same time, *transition* is reset to ⊥, so a new input may be processed.

Alg. 4: (A timed FSM interpreter schema.)

```
1  class TimedFSMInterpreter[tm, G, pars] is
2      define
3          localEnv = createMap(G("Parameters"), pars) ;
4      input {"input"} : handleInput,
5            {"fire"} : doTransition ;
6      output {"output", "schedule"} ;
7
8      attribute currentState ;
9      attribute currentVars ;
10     attribute transition ;
11     attribute newVars ;
12     attribute outValue ;
13
14     initialize :
15             do forall e ∈ edges(G) :
16                 if src(e)("type") = "InitMarker" then
17                     currentState := dst(e)
18                 end
19             end,
20             currentVars :=
21                 eval(G("InitialValues"), localEnv)
22     end
23
24     rule handleInput[p, t, v] :
25     once
26         if transition = ⊥ then
27           let env = localEnv+currentVars
28                             +["input" ↦ v, "now" ↦ t]:
29             choose e ∈ {e |e ∈ edges(G),
30                             src(e) = currentState,
31                             eval(e("Guard"), env) ≠ ⊥}:
32                 transition := e,
33                 outValue := eval(e("Output"), env),
34                 newVars := currentVars+
35                     eval(e("UpdateExpression"), env)
36                 [OutputPorts(this)("schedule")] ←
37                         ⊥@t + eval(e("Delay"), env)
38             end
39           end
40         end
```

```
    end

    rule doTransition[p, t, v] :
        if transition ≠ ⊥ then
            transition := ⊥,
            currentState := dst(transition),
            currentVars := newVars,
            [OutputPorts(this)("output")] ← outVal@t
        end
    end
end
```

5.2 Concurrency and infinite-state systems: Petri nets

Now we will turn to the visual language of Petri nets. A basic variant of Petri nets was defined in the interpreter schema Alg. 2 in the previous chapter. In this section we extend it also in two steps, and we will further expand its capabilities in Sect. 5.4.

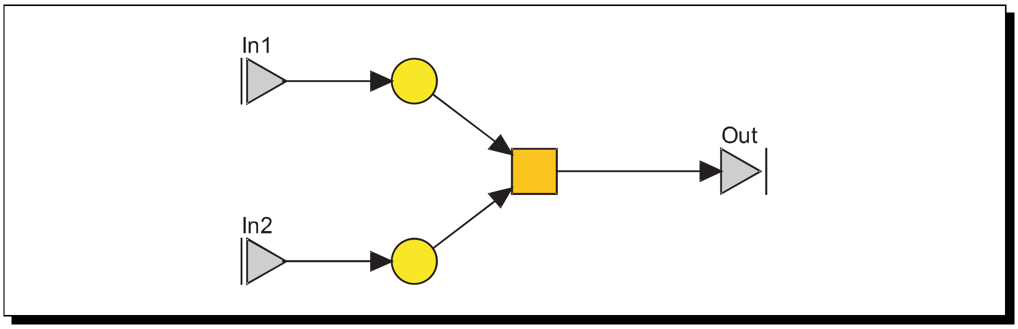

Fig. 32: A Petri net component.

5.2.1 Petri net components

One of the most fundamental shortcomings of the Petri nets as we defined them in Alg. 2 is that there is no way to connect them to any other components—they have no input or output ports, except for the *fire* and *schedule* ports. We will now add input and output ports to these Petri nets in the way depicted in Fig. 32 (which shows a simplified version of the network in Fig. 10 on page 14). Each input port will be represented by a vertex of type *InputPort*, similarly each output port is represented by a vertex of type *OutputPort*. These vertices have a *Name* attribute containing a string that becomes the name of the respective port. (All *Weight* attributes of the arcs are 1, so they have been omitted in the figure.)

Correspondingly, in addition to the usual Petri net arcs between places and transitions, we have now arcs going from input port vertices to places and from transitions to output port vertices. The interpreter schema for this kind of Petri net component is shown in Alg. 5. It begins by defining variables containing the different sets of vertices (input ports, output ports, places, transitions—Lines 2-5), and then the sets of arcs corresponding to the different kinds of connections we allow (places to transitions, transitions to places, input ports to places, transitions to output ports—Lines 6-13).

A crucial feature of this visual language is that the input/output signature (i.e. the number of ports and their names) depends on the graph to be interpreted—as opposed e.g. to the FSM examples, where there were always the same input/output ports irrespective of the graph that was interpreted. This means that the declarations of ports must somehow refer to the graph structure. In Line 16 we declare the set of 'data' input names as those that occur as values of the *Name* attribute of *InputPort* vertices. These are associated with the *handleInput* rule, which is of course different from the *step* rule associated with the *fire* input port. Similarly, the set of output port names is computed in Line 17.

The vocabulary and the initialization are identical to Alg. 2.

The *handleInput* rule is fired when some token (other than a firing message) arrives at some port. Intuitively, the intention is that this token will be put onto every place connected to the corresponding input port vertex. The *handleInput* rule realizes this by iterating over those edges in the *IP* set (the input-to-place edges) that originate from an input port vertex with the correct *Name* attribute. The marking of each place connected to an input port is incremented by one (Lines 31-33), and eventually the actor is rescheduled at the current time (because the new token might result in a transition to be 'firable' with the net previously dead) in Line 34.

The first part of the *step* rule is very similar to the *step* rule in Alg. 2, except that the iterations are structured here according to the edge-subset that they use (whereas in Alg. 2 the iteration went over all edges). After having chosen the transition t from the set of activated transitions (Lines 39-41), we compute the effect of its firing in three steps:

1. Iterate over the place-to-transition edges that end at t, subtracting the weight of the edge from the marking of the place it comes from (Lines 43-45).

2. Iterate over the transition-to-place edges originating at t, adding the weight of the edge to the marking of the place it points to (Lines 46-48).

3. Iterate over the transition-to-output-port edges originating at t, sending a $\perp$ token from each port that is pointed to (Lines 49-52).

Finally, the *step* rule reschedules the actor, so it could fire again (Line 53).

Alg. 5: (An interpreter of Petri nets with ports.)

```
class PNComponentsInterpreter[tm, G] is
    define I = {v | v ∈ vertices(G), v("type") = "InputPort"},
           O = {v | v ∈ vertices(G), v("type") = "OutputPort"},
           P = {v | v ∈ vertices(G), v("type") = "Place"},
           T = {v | v ∈ vertices(G), v("type") = "Transition"},
           PT = {e | e ∈ edges(G),src(e)("type") = "Place",
                                  dst(e)("type") = "Transition"},
           TP = {e | e ∈ edges(G),src(e)("type") = "Transition",
                                  dst(e)("type") = "Place"},
           IP = {e | e ∈ edges(G),src(e)("type") = "InputPort",
                                  dst(e)("type") = "Place"},
           TO = {e | e ∈ edges(G),src(e)("type") = "Transition",
                                  dst(e)("type") = "OutputPort"} ;

    input {"fire"} : step,
          {v("Name") | v ∈ I} : handleInput ;
    output {"schedule"} ∪ {v("Name") | v ∈ O} ;

    function M arity 1 ;

    initialize :
    once
        [OutputPorts(this)("schedule")] ← ⊥@tm,
        do forall p ∈ P :
           M(v) := v("initialTokens")
        end
    end

    rule handleInput[p, now, v] :
    once
        do forall e ∈ {e | e ∈ IP, src(e)("Name") = p} :
           M(dst(e)) := M(dst(e)) + 1
        end,
        [OutputPorts(this)("schedule")] ← ⊥@now
    end

    rule step[p, now, v] :
    once
        choose t ∈ {t |t ∈ T,
                    ∀e ∈ PT :
                    (dst(e) = t ⇒ e("Weight") ≤ M(src(e)))} :

              do forall e ∈ {e | e ∈ PT, t = dst(e)} :
                 M(src(e)) := M(src(e)) − e("Weight")
```

end,
do forall $e \in \{e \mid e \in TP, t = src(e)\}$:
    $M(dst(e)) := M(dst(e)) + e(\text{"}Weight\text{"})$
end,
do forall $e \in \{e \mid e \in TO, t = src(e)\}$:
    $[OutputPorts(this)(dst(e)(\text{"}Name\text{"}))] \leftarrow$
        $\perp @now$
end,
$[OutputPorts(this)(\text{"}schedule\text{"})] \leftarrow \perp @now$
end
end
end

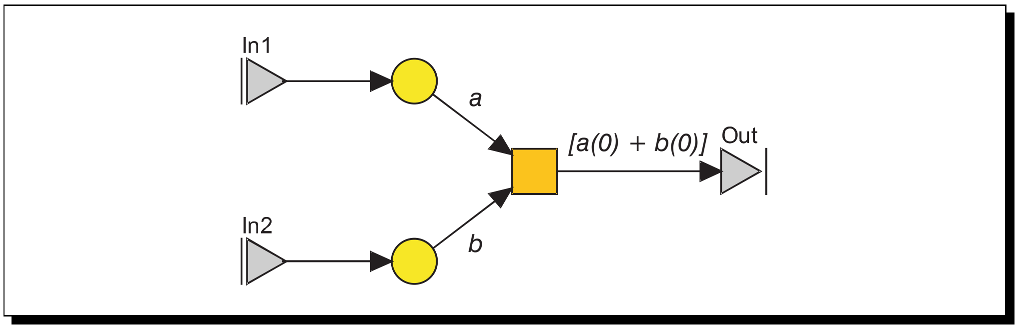

Fig. 33: A high-level Petri net component.

5.2.2 Adding computation

Another important extension to Petri nets is shown in Fig. 33 (which implements the same functionality as the network in Fig. 10 on page 14, though we have rearranged the attributes somewhat to simplify the interpreter) is the ability to embed computation in it, very much like we did for extended FSM in the previous section. This means that the tokens in a Petri net need to represent objects, and in firing a transition we need to be able to express computation on these objects.

As is usually the case in the design of a visual notation, there are various ways to add a capability for manipulating objects to the Petri net language. Here, we have chosen one that facilitates a simple description and still illustrates the key design elements. The token objects will be stored in the markings of the places in the order in which they 'arrived' at the place, i.e. our marking (the state of the Petri net) maps each place to a finite sequence of objects. As can be seen in Fig. 33, the arcs going from places to transitions have an attribute (called *Var*) containing a symbol, and the arcs going from transitions to places have one (called *Values*) containing an expression. (As in the last examples, all weights are 1 in Fig. 33, and therefore have been omitted for clarity.)

When a transition fires, it takes for each incoming arc the sublist of tokens of the length corresponding to its *Weight* attribute from the head of the sequence of tokens residing on the place the arc is coming from. This is then bound to the symbol contained in the *Var* attribute of that arc. For instance, if in the example if the figure, the upper place would contain the token sequence $[1, 2, 3]$ and the lower place the token sequence $[4, 5, 6]$, a would be bound to the sequence $[1]$ and b to the sequence $[4]$. The resulting environment is then used to evaluate the expressions in the *Values* attributes of the outgoing arcs. In the example, this would be $[a(0) + b(0)]$, where we employ the convention that a sequence is a map from indices (starting at 0) to the corresponding elements, so that $a(0)$ and $b(0)$ select the first element in their corresponding sequence (which in this case contains only one element anyway, of course). The result of the expression is a sequence $[5]$, which is then written to the output port in the example.

Alg. 6 shows an interpreter schema for this visual notation. It begins like the schema in Alg. 5 by declaring sets of the different kinds of structural elements, and also a *localEnv* map representing the environment resulting from binding the formal parameter symbols to the actual parameter values (Lines 2-14). Additionally, it declares a map *createEnv* of two parameters, a transition vertex and a map containing the current marking (Lines 15-15).[2] The notation we use to specify this function follows the λ-calculus, using $\lambda(t, m).e$ as a shortcut for $\lambda t.(\lambda m.e)$. The result of applying this function to a transition and a marking is a map which maps the symbols of the *Var* attributes of the incoming arcs of that transition to the initial segments of the sequences of the corresponding places—in other, words, it constructs the bindings that will be needed when firing the transition. It assumes the existence of a predefined function *head* that is applied to a sequence and a number and returns the initial segment of that sequence of the specified length.

Initialization is very similar to Alg. 5, except that the *InitialTokens* attribute of a place is now expected to contain an expression (rather than a number), which is evaluated in the local environment to yield the initial list of tokens (Line 29). Similarly, when adding a token coming in at an input port (Line 36), we do not just add 1 to the number of tokens as in the previous example, but instead add the token itself to the lists of tokens at the markings of the corresponding places.

The firing of a transition is also similar in structure to the rule in Alg. 5. However, choosing a transition involves evaluating its guard (not shown in the example, since here it is always true), which is an expression bound to the transition vertex' *Guard* attribute and must be $\neq \perp$ if the transition is allowed to fire (Lines 44-48). Once a transition is chosen, tokens are removed from the marking of those places from which arcs go to the transition (Lines 50-52), then tokens (computed as the result of evaluating the *Values* attribute of the corresponding arc) are added to the marking of those places to which arcs lead from the transition (Lines 53-56), and finally tokens are written to those ports

[2]It is necessary to pass the marking as an object because a function cannot, of course, depend on the valuation of a vocabulary—only rules can.

to which arcs lead from the transition (Lines 57-60). In removing tokens, we assume a predefined function *tail*, which is the dual of *head* in that it takes a sequence and a number and return the rest of the sequence after removing that number of elements from its head.

Alg. 6: (An interpreter of high-level Petri net components.)

```
1  class HLPNInterpreter[tm, G, pars] is
2      define I = {v | v ∈ vertices(G), v("type") = "InputPort"},
3          O = {v | v ∈ vertices(G), v("type") = "OutputPort"},
4          P = {v | v ∈ vertices(G), v("type") = "Place"},
5          T = {v | v ∈ vertices(G), v("type") = "Transition"},
6          PT = {e | e ∈ edges(G),src(e)("type") = "Place",
7                                 dst(e)("type") = "Transition"},
8          TP = {e | e ∈ edges(G),src(e)("type") = "Transition",
9                                 dst(e)("type") = "Place"},
10         IP = {e | e ∈ edges(G),src(e)("type") = "InputPort",
11                                dst(e)("type") = "Place"},
12         TO = {e | e ∈ edges(G),src(e)("type") = "Transition",
13                                dst(e)("type") = "OutputPort"},
14         localEnv = createMap(G("Parameters"), pars),
15         createEnv =λ(t, m).localEnv+
16                 [e("Var") ↦ head(m(src(e)), e("Weight")) |
17                 e ∈ PT, dst(e) = t] ;
18
19     input {"fire"} : step,
20         {v("Name") | v ∈ I} : handleInput ;
21     output {"schedule"} ∪ {v("Name") | v ∈ O} ;
22
23     function M arity 1 ;
24
25     initialize :
26     once
27         [OutputPorts(this)("schedule")] ← ⊥@tm,
28         do forall p ∈ P :
29             M(v) := eval(v("InitialTokens"), localEnv)
30         end
31     end
32
33     rule handleInput[p, now, val] :
34     once
35         do forall e ∈ {e | e ∈ IP, src(e)("Name") = p} :
36             M(dst(e)) := M(dst(e)) + [val]
37         end,
38         [OutputPorts(this)("schedule")] ← ⊥@now
39     end
```

```
      rule step[p, now, v] :
      once
            let m = [p ↦ M(p) | p ∈ M] :
                choose t ∈ {t |t ∈ T,
                                    ∀e ∈ PT :
                                       (dst(e) = t ⇒
                                          e(”Weight”) ≤ length(M(src(e)))),
                                    eval(t(”Guard”), createEnv(t, m)) ≠ ⊥} :

                        do forall e ∈ {e | e ∈ PT, t = dst(e)} :
                            M(src(e)) := tail(M(src(e)), e(”Weight”))
                        end,
                        do forall e ∈ {e | e ∈ TP, t = src(e)} :
                            M(dst(e)) :=M(dst(e))+
                                              eval(e(”Values”), createEnv(t, m))
                        end,
                        do forall e ∈ {e | e ∈ TO, t = src(e)} :
                            [OutputPorts(this)(dst(e)(”Name”))] ←
                                  eval((e(”Values”), createEnv(t, m))@now
                        end,
                        [OutputPorts(this)(”schedule”)] ← ⊥@now
                end
            end
      end
end
```

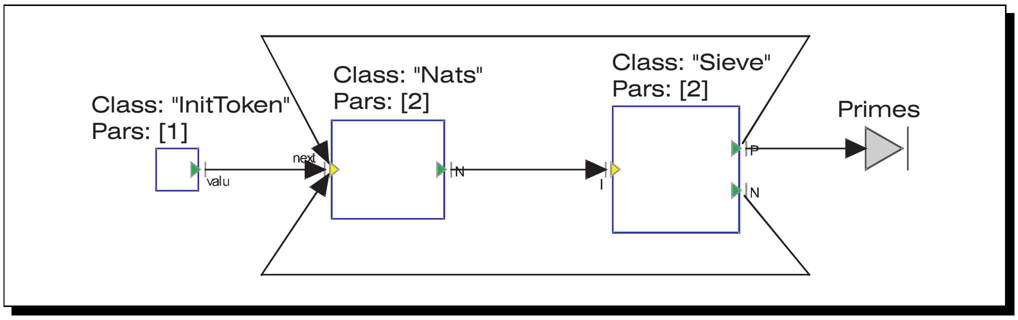

Fig. 34: Asynchronously communicating processes.

5.3 Networks of communicating actors

So far, we have described visual languages that allow us to express computation, describe the structure of the state of an actor and how it changes, and to define actors that have input and output ports. However, none of these languages provided a facility to combine actors, to describe networks of actors, and define their communication structure. The language we are now about to describe is just the opposite: Actors defined in that language never change state (they only have one), and all these actors do is, in fact, create other actors and establishing a communication structure between them, as well as acting as an 'interface' between the network they contain and the outside world.

Fig. 34 (a slight adaptation of Fig. 11 on page 14 with attribution a little more explicit) shows an example of such a network of communicating actors. The contained actors are connected to each other as well as to the input and output ports of the containing actor (though the one in the example only has an output port).

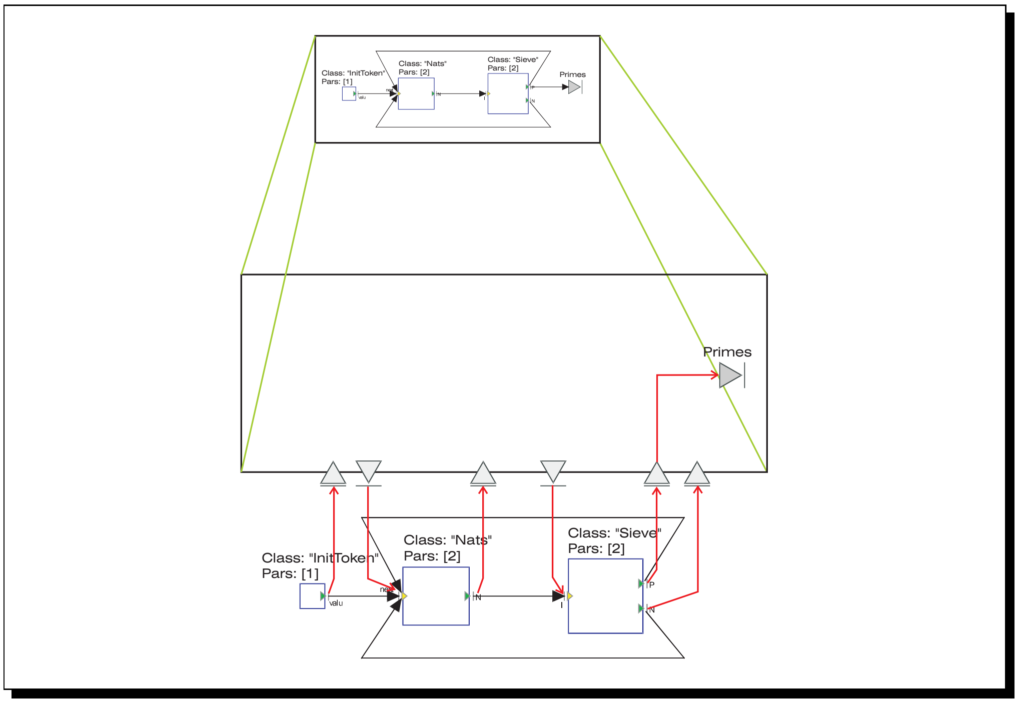

Fig. 35: Flattening the 'hierarchy' in Fig. 34.

Note, however, that our model of computation does not provide an explicit concept of hierarchy—all actors, in principle, can be connected to all other actors, there is no notion of one actor being 'above' or 'around' another actor, or 'containing' another actor. The only thing we can do to two actors, the only relation they can be in, is to be connected to each other, to that tokens from the outputs of one flow into the inputs of the other.

In order to represent an actor network as in Fig. 34, we have to 'flatten' the hierarchy, connecting the seemingly 'contained' actors to their 'container' in some appropriate manner, so that this container can provide the interfaces and functionality we expect from the apparent containment. In order to do this, the container has to have special ports, one for each input and output port of its contained actors, so that they can be connected to their container. Fig. 35 illustrates this 'flattening' operation, which connects the 'inner' actors (the *InitTokens*, the *Nats*, and the *Sieve* actors) to their containing actor by means of those special ports, which we call *private* ports (introduced in Section 4.9). Although otherwise completely normal input and output ports, by convention these ports are not meant to be connected with other ('outside', i.e not contained) actors. These private ports are drawn along the bottom of the enclosing actor. The only functionality provided by it (after it has established the contained actors and their connection) is to 'catch' the tokens leaving the rightmost actor at its upper output port and to forward them to its own *Primes* output port. In general, such an actor does not perform any state transitions, or schedule itself—it only provides interface functionality to and from the enclosed actors.

In the case of the vertices representing embedded components, arcs are connected to them at so-called *connectors*, which are graphically represented by little triangles pointing into or out of the vertex they belong to. In the abstract syntax, each vertex has two attributes, *InConnectors* and *OutConnectors*, containing the sets of input and output connectors, respectively. (Basically, the little triangles can be considered graphical representations of these sets.) In general, any vertex may have connectors, as permitted, required or forbidden by the syntax rules of the language. Edges may be connected to 'just the vertex', or to a particular connector of the vertex. In the abstract syntax, an edge has two attributes *ConnectorA* and *ConnectorB*, representing the name of the connector of its source vertex and its destination vertex, respectively. If these are $= \perp$, it means that the edge is connected to 'just the vertex'. Otherwise, it is connected to the corresponding vertex connector.

Alg. 7 shows an interpreter schema for this visual language. As usual, it starts with a list of definitions of sets of vertices and edges, as well as a local environment binding the parameters (Lines 2-13). This time there are three sets of vertices, viz. those representing input ports, output ports, and embedded components. These are followed by four sets of edges: Edges connecting input ports to input ports of embedded components (*IC*), edges connecting output ports of embedded components to input ports of embedded components (*CC*), edges directly connecting an input port vertex to an output port vertex (*IO*), and finally edges connecting output ports of embedded components to output port vertices (*CO*).

The actor network actor will represent component-to-component connections directly by connecting the two components in question, thus freeing itself from having to handle the tokens flowing between them. The other connections, *IC*, *IO*, and also *CO*, involve a token flowing over an input port of the containing actor, i.e. they involve the execution of a rule. Note that this also holds for *CO* connections—as can be seen from Fig. 35, these are realized via an intermediate private input port.

The set of external (i.e. non-private) input ports is defined as in the previous examples as the *fire* input port (associated with an empty rule) plus an input port for each *I* vertex (Lines 15-16). Then follows the declaration of the private input ports (Lines 17-19). Here we want to construct one for each output port of each embedded component that an arc might be connected to. The private input port corresponding to the component output port c of the component represented by the vertex v is given the 'name' (v, c). In this way, we construct a private input port for each *OutConnector* in the graph.

Similarly, the external output ports are generated from the output port vertices, while each private output port corresponds exactly to one *InConnector* in the graph (Lines 20-22).

The vocabulary contains just one unary function, which eventually maps each *Comps* vertex to the component that was created for it (Line 24). This information is used during setup.

In the initialization, in a first step we create the components corresponding to each *Comps* vertex (Lines 28-33). Then we 'wire' them up: First the connections among the embedded components themselves (Lines 34-40), then we connect the private output ports to the embedded components' input ports, (Lines 42-45), and finally the private input ports to the embedded components' output ports (Lines 46-49).

In handling incoming tokens, we distinguish between those tokens coming from an external input port and those arriving at a private input port. External input is handled by the rule *handleInput*, which simply forwards the tokens along each arc leading to a component input port (which, of course, is represented by the corresponding private output port—Lines 55-58), and along each arc leading directly to an output port (Lines 59-61). The rule handling input at private input ports is even simpler: It just needs to forward the tokens along arcs connecting the corresponding embedded component output with an output port (Lines 65-68).

Alg. 7: (An interpreter schema of networks of communicating actors.)

1 <u>**class**</u> $ProcNetInterpreter[tm, G, pars]$ <u>**is**</u>
2 <u>**define**</u> $I = \{v \mid v \in vertices(G), v(\text{"}type\text{"}) = \text{"}InputPort\text{"}\}$,
3 $O = \{v \mid v \in vertices(G), v(\text{"}type\text{"}) = \text{"}OutputPort\text{"}\}$,
4 $Comps = \{v \mid v \in vertices(G), v(\text{"}type\text{"}) = \text{"}Component\text{"}\}$,
5 $IC = \{e \mid e \in edges(G), src(e)(\text{"}type\text{"}) = \text{"}InputPort\text{"}$,
6 $dst(e)(\text{"}type\text{"}) = \text{"}Component\text{"}\}$,
7 $CC = \{e \mid e \in edges(G), src(e)(\text{"}type\text{"}) = \text{"}Component\text{"}$,
8 $dst(e)(\text{"}type\text{"}) = \text{"}Component\text{"}\}$,
9 $IO = \{e \mid e \in edges(G), src(e)(\text{"}type\text{"}) = \text{"}InputPort\text{"}$,
10 $dst(e)(\text{"}type\text{"}) = \text{"}OutputPort\text{"}\}$,
11 $CO = \{e \mid e \in edges(G), src(e)(\text{"}type\text{"}) = \text{"}Component\text{"}$,
12 $dst(e)(\text{"}type\text{"}) = \text{"}OutputPort\text{"}\}$,
13 $localEnv = createMap(G(\text{"}Parameters\text{"}), pars)$ <u>**;**</u>
14
15 <u>**input**</u> $\{\text{"}fire\text{"}\} : doNothing$,
16 $\{v(\text{"}Name\text{"}) \mid v \in I\} : handleInput$ <u>**;**</u>
17 <u>**private**</u> <u>**input**</u>
18 $\{(v, c) \mid v \in Comps, c \in v(\text{"}OutConnectors\text{"})\}$

```
            : handleInternalInput ;
output {"schedule"} ∪ {v("Name") | v ∈ O} ;
private output
            {(v, c) min v ∈ Comps, c ∈ v("InConnectors")} ;

function C arity 1 ;

initialize :
once
      do forall v ∈ Comps :
          import proc = component
                        v("Class")(eval(v("Pars"), localEnv)) :
                C(v) := proc
          end
      end ;
      do forall e ∈ CC :
          let v1 = src(e), c1 = e("ConnectorA"),
              v2 = dst(e), c2 = e("ConnectorB") :
              [OutputPorts(C(v1))(c1)]
               ⟶ [InputPorts(C(v2))(c2)]
          end
      end,
      do forall v ∈ Comps :
          do forall c ∈ v("InConnectors") :
              [OutputPorts(this)((v, c))]
               ⟶ [InputPorts(C(v))(c)]
          end,
          do forall c ∈ v("OutConnectors") :
              [OutputPorts(C(v))(c)]
               ⟶ [InputPorts(this)((v, c))]
          end
      end
end

rule handleInput[p, now, val] :
once
      do forall e ∈ {e | e ∈ IC, src(e)("Name") = p} :
          [OutputPorts(this)((dst(e), e("ConnectorB")))]
           ← val@now
      end,
      do forall e ∈ {e | e ∈ IO, src(e)("Name") = p} :
          [OutputPorts(this)(dst(e)("Name"))] ← val@now
      end
end
```

```
rule handleInternalInput[p, now, val] :
    do forall e ∈ {e | e ∈ CO,
                          p = (src(e)("Name"), e("ConnectorA"))} :
        [OutputPorts(this)(dst(e)("Name"))] ← val@now
    end
end

rule doNothing[p, t, v] :
once
    skip
end
end
```

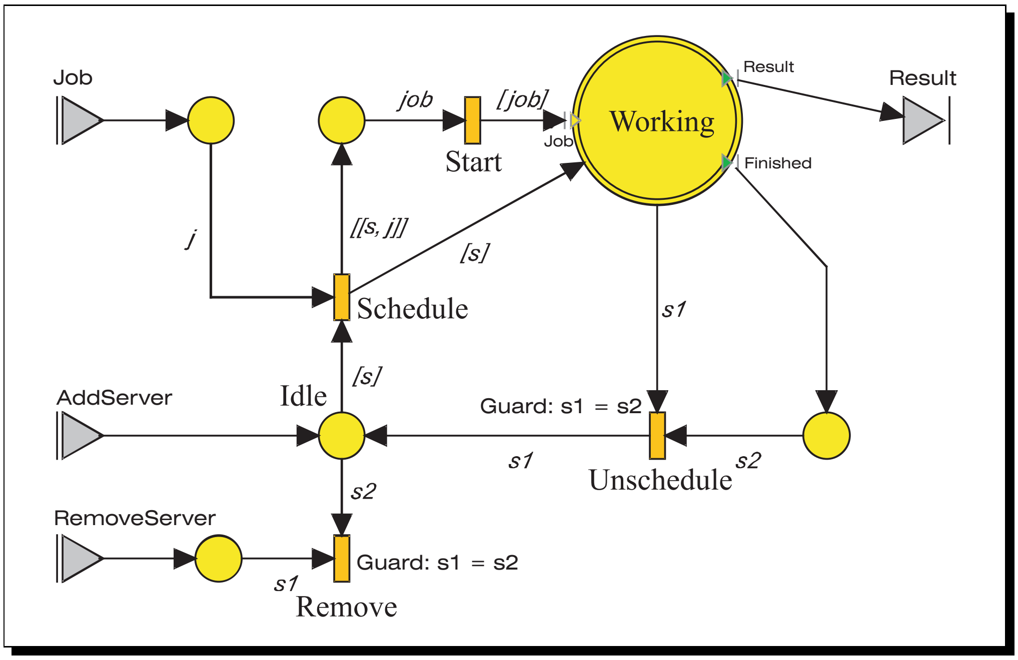

Fig. 36: A dynamic Petri net.

5.4 Petri nets revisited: dynamic Petri net structures

Now we will turn again to Petri nets, incorporating some of the techniques from the previous example into them, creating a new form of Petri nets that support dynamic network structures. A more complete description of its semantics can be found in [65], applications of this visual notation are presented in [45, 68].

Fig. 36 shows a simple example of such a 'dynamic' Petri net (we have annotated some places and vertices with 'non-functional' names in a serif typeface, to aid our explanation). The basic function of this component is to distribute 'jobs' to an arbitrary number of other actors—indeed, this number is allowed to change over time! Jobs are represented by tokens arriving at the *Job* input port. We assume that there are a number of tokens representing actors (i.e. elements of I) on the place named *Idle*. In that case, the transition called *Schedule* can fire, picking an actor from the *Idle* place and the new job and placing the actor on the place labeled *Working*, and the tuple consisting of that actor and the job token on the place above it, activating the transition *Start*.

The *Working* place (Fig. 37) looks somewhat different—most importantly, it has input and output connectors attached to it, similar to those representing embedded components in the previous example. Also, some arcs (such as the one from the *Start* transition) go to the input connectors, while other arcs (such as the one from the *Schedule* transition) go to the place 'directly'—and the same applies to arcs leaving the place. The intended meaning is this: Arcs directly connected to the place itself work just like for any other place in a Petri net, i.e. tokens created by the expression associated with an incoming arc are put on the place, and tokens bound to the variable symbol associated with an outgoing arc are removed from the place. This was the case, e.g., when the *Schedule* transition placed the selected actor on the *Working* place.

However, arcs connecting to a place via one of its connectors do not transport tokens to or from the place itself—instead, they connect to the components residing on the place (which may be any number, including zero). In other words, when the firing of the *Start* transition produces a token on this arc, this token is not placed on the *Working* place, but instead is fed to the *Job* input ports of all actors residing on that place (assuming they do have such an input port). Similarly, the arcs starting at the *Result* and *Finished* connectors collect tokens from the correspondingly named output ports of the actors residing on the *Working* place.

The technique that is employed to realize this in the following interpreter schema for this visual language (Alg. 8) will be similar to the one used in the previous example (Alg. 7)—for each connector at each place vertex we create a private input or output port, interpreting the arcs connected to a connector as leading to or from the corresponding port (Fig. 38).

The most important new feature compared to the previous example is the fact that the connections of other actors with these private ports change over time. When the actor was moved onto the *Working* place by the *Schedule* transition, the connections were established. Now when the *Start* transition fires it sends the tuple consisting of the actor identifier and the job token to the *Job* input port of all actors residing on the *Working* place. This is, by the way, the reason why we included the actor identifier in the message—this way, the actors are able to distinguish those messages intended for them and ignore those that do not carry their own identifier.[3]

Now the embedded actor works on the job, producing result tokens at its *Result* output port—which are directly forwarded to the *Result* output port of

[3]Other visual notations might provide better ways to address the destination actor directly, of course.

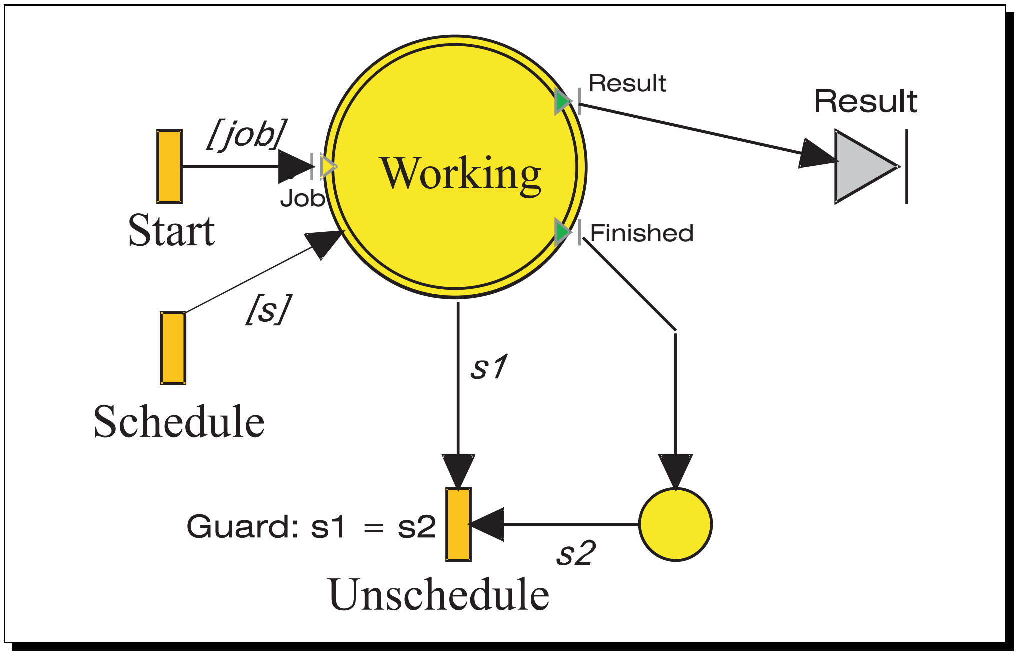

Fig. 37: A place vertex with connectors.

the dynamic Petri net. Eventually, it produces a token at its *Finished* output port, which is required to be its own actor identifier. This token is placed on the place that is connected to the *Finished* connector of the *Working* place. This activates the *Unschedule* transition, which eventually removes the actor from the *Working* place and puts it on the *Idle* place again.

Finally, the *AddServer* and *RemoveServer* input ports allow other actors to put new server actors into the pool of idle actors (on the *Idle* place), or have them removed from there.

The interpreter schema for dynamic Petri nets is presented in Alg. 38. As before, it starts with defining names for sets of different kinds of vertices and edges in the graph. While the vertex sets are the usual input ports, output ports, places, and transitions (Lines 2-5), there are nine different kinds of edges, depending on the entities connected by them (Lines 6-30). The sets are named according to the type of things they connect, where "Pc" denotes the connector on a place, so that *PcO* are edges going from such an output connector to an output port vertex, while *TPc* are edges going from a transition to an input connector.

The definition of the local environment and the *createEnv* function creating the environment for a given transition and a given marking (Lines 31-31) are identical to the way this was done in the high-level Petri nets (Alg. 6). The definition of the input and output ports is similar to the definition in Alg. 7—external ports for the input/output port vertices, a private port for each connector of a place (Lines 36-43).

The vocabulary consists of the usual unary marking function *M* (Line 45). For simplicity, we initialize it with an empty list for all places (Line 49).

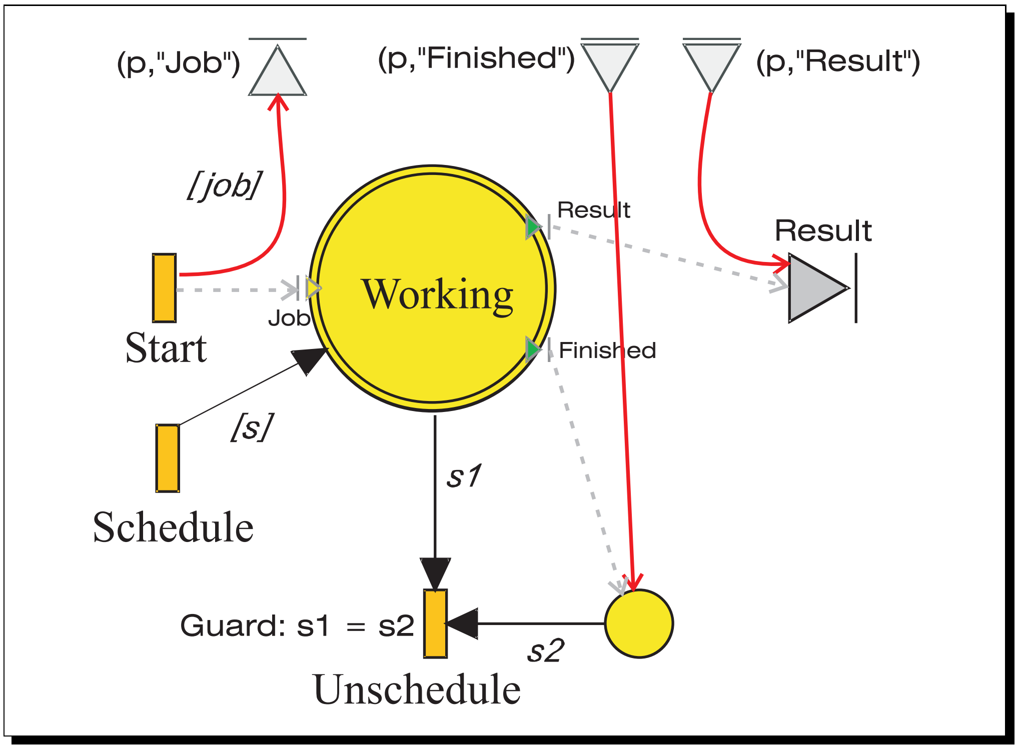

Fig. 38: ... and its translation into the actor model.

As for the visual language of actor networks (Alg. 7), there are three rules: *handleInput* describes the reaction to tokens arriving at the external input ports, while *handleInternalInput* defines what happens when a token arrives at one of the private input ports. Finally, *step* performs the state transition when a transition fires.

The *handleInput* rule first iterates over those edges connecting the input port vertex corresponding to the input port at which the current token is coming in to a place (Lines 54-66). In addition to adding the token to the list of tokens that constitute the marking of that place, the input/output ports of the corresponding actor must be connected to the respective private output/input ports of this actor.[4] This is done in Lines 57-65. Lines 67-69 forward the input along edges leading to embedded component inputs, and finally the component schedules itself.

Tokens arriving at an internal input (i.e. tokens coming from an embedded component) are dealt with by the rule *handleInternalInput*. They can go to three different destinations: output ports, places, or other embedded components' input ports. These three cases are dealt with in the rule in that order. First, the token is forwarded along edges leading to output port vertices (Lines 75-78), then the token is added to the marking of places connected to the input

[4]If the token does not, in fact, represent an actor, the $InputPorts(val)(conn)$ etc. expressions will not yield a port, and then by the definition of the connection/disconnection rules (Sect. 4.7.1), nothing happens.

(Lines 79-92), and finally the token is forwarded to connected input ports of embedded components (Lines 93-97). Note that adding a token to the marking of a place again involves connecting it to the corresponding private input/output ports (Lines 82-91).

Finally, the *step* rule, like for high-level Petri nets (Alg. 6), first picks a transition it can take (Lines 103-107). As in the high-level Petri net interpreter schema, firing the selected transition essentially consists of three steps: removing the 'consumed' tokens (Lines 109-124), adding the 'produced' tokens (Lines 125-140), and writing out output (Lines 141-148), the latter consisting in writing out tokens to external output ports (Lines 141-144), and to internal output ports, i.e. to embedded component input ports (Lines 145-148). Removing consumed tokens from a place involves disconnecting their ports from this actor, just as adding them involves connecting their ports to the corresponding private ports. Here, we use the function *elementsOf* to produce the set of objects contained in a list.

Alg. 8: (An interpreter of dynamic Petri nets)

1 **class** $DynamicPNInterpreter[tm, G, pars]$ **is**

2 **define** $I = \{v \mid v \in vertices(G), v(\text{"type"}) = \text{"InputPort"}\},$

3 $O = \{v \mid v \in vertices(G), v(\text{"type"}) = \text{"OutputPort"}\},$

4 $P = \{v \mid v \in vertices(G), v(\text{"type"}) = \text{"Place"}\},$

5 $T = \{v \mid v \in vertices(G), v(\text{"type"}) = \text{"Transition"}\},$

6 $PT = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Place"},$

7 $dst(e)(\text{"type"}) = \text{"Transition"},$

8 $e(\text{"ConnectorA"}) = \bot\},$

9 $TP = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Transition"},$

10 $dst(e)(\text{"type"}) = \text{"Place"},$

11 $e(\text{"ConnectorB"}) = \bot\},$

12 $IP = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Input"},$

13 $dst(e)(\text{"type"}) = \text{"Place"},$

14 $e(\text{"ConnectorB"}) = \bot\},$

15 $TO = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Transition"},$

16 $dst(e)(\text{"type"}) = \text{"Output"}\},$

17 $IPc = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Input"},$

18 $dst(e)(\text{"type"}) = \text{"Place"},$

19 $e(\text{"ConnectorB"}) \neq \bot\},$

20 $PcO = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Place"},$

21 $dst(e)(\text{"type"}) = \text{"Output"}\},$

22 $PcP = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Place"},$

23 $dst(e)(\text{"type"}) = \text{"Place"},$

24 $e(\text{"ConnectorB"}) = \bot\},$

25 $PcPc = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Place"},$

26 $dst(e)(\text{"type"}) = \text{"Place"},$

27 $e(\text{"ConnectorB"}) \neq \bot\},$

28 $TPc = \{e \mid e \in edges(G), src(e)(\text{"type"}) = \text{"Transition"},$

```
                                  dst(e)("type") = "Place",
                                  e("ConnectorB") ≠ ⊥},
        localEnv = createMap(G("Parameters"), pars),
        createEnv = λ(t, m).localEnv+
                    [e("Var") ↦ head(M(src(e)), e("Weight")) |
                    e ∈ PT, dst(e) = t] ;

input {"fire"} : step,
      {v("Name") | v ∈ I} : handleInput ;
private input
      {(v, c) | v ∈ P, c ∈ v("OutConnectors")}
      : handleInternalInput ;
output {"schedule"} ∪ {v("Name") | v ∈ O} ;
private output
      {(v, c) min v ∈ P, c ∈ v("InConnectors")} ;

function M arity 1 ;

initialize :
once
    do forall v ∈ P : M(v) := [] end
end

rule handleInput[p, now, val] :
once
    do forall e ∈ {e | e ∈ IP, src(e)("Name") = p} :
      M(dst(e)) := M(dst(e)) + val,
      let place = dst(e) :
        do forall conn ∈ place("InConnectors") :
          [OutputPorts(this)((place, conn))]
           ⟶ [InputPorts(val)(conn)]
        end,
        do forall conn ∈ place("OutConnectors") :
          [OutputPorts(val)(conn)] ⟶
          [InputPorts(this)((place, conn))]
        end
      end
    end,
    do forall e ∈ {e | e ∈ IPc, src(e)("Name") = p} :
      [Output(this)((dst(e), e("ConnectorB")))] ← val@now
    end,
    [OutputPorts(this)("schedule")] ← ⊥@now
end

rule handleInternalInput[p, now, val] :
```

```
once
    do forall e ∈ {e | e ∈ PcO,
                    p = (src(e)("Name"), e("ConnectorA"))} :
        [OutputPorts(this)(dst(e)("Name"))] ← val@now
    end,
    do forall e ∈ {e | e ∈ PcP,
                    p = (src(e)("Name"), e("ConnectorA"))} :
        M(dst(e)) := M(dst(e)) + val,
        let place = dst(e) :
            do forall conn ∈ place("InConnectors") :
                [OutputPorts(this)((place, conn))]
                 ⟶ [InputPorts(val)(conn)]
            end,
            do forall conn ∈ place("OutConnectors") :
                [OutputPorts(val)(conn)] ⟶
                [InputPorts(this)((place, conn))]
            end
        end
    end,
    do forall e ∈ {e | e ∈ PcPc,
                    p = (src(e)("Name"), e("ConnectorA"))} :
        [OutputPorts(this)(((dst(e), e("ConnectorB"))))]
         ← val@now
    end
end

rule step[p, now, v] :
once
    let m = [p ↦ M(p) | p ∈ M] :
        choose t ∈ {t |t ∈ T,
                    ∀e ∈ PT :
                      (dst(e) = t ⇒
                        e("Weight") ≤ length(M(src(e)))),
                    eval(t("Guard"), createEnv(t, m)) ≠ ⊥} :

            do forall e ∈ {e | e ∈ PT, t = dst(e)} :
                let place = src(e) :
                    do forall comp ∈ elementsOf(head(M(place),
                                                  e("Weight"))):
                        do forall conn ∈ place("InConnectors") :
                            [OutputPorts(this)((place, conn))]
                             ⟶̸ [InputPorts(comp)(conn)]
                        end,
                        do forall conn ∈ place("OutConnectors") :
                            [OutputPorts(comp)(conn)] ⟶̸
```

```
                          [InputPorts(this)((place, conn))]
                        end
                      end
                    end,
                    M(src(e)) := tail(M(src(e)), e("Weight"))
                  end,
                  do forall e ∈ {e | e ∈ TP, t = src(e)} :
                    let place = dst(e),
                      res = eval(e("Fire"), createEnv(t, m)) :
                      M(dst(e)) := M(dst(e)) + res,
                      do forall comp ∈ elementsOf(res) :
                        do forall conn ∈ place("InConnectors") :
                          [OutputPorts(this)((place, conn))]
                           ⟶ [InputPorts(comp)(conn)]
                        end,
                        do forall conn ∈ place("OutConnectors") :
                          [OutputPorts(comp)(conn)] ⟶
                          [InputPorts(this)((place, conn))]
                        end
                      end
                    end
                  end,
                  do forall e ∈ {e | e ∈ TO, t = src(e)} :
                    [OutputPorts(this)(dst(e)("Name"))]
                     ← eval((e("Values"), createEnv(t, m))@now
                  end,
                  do forall e ∈ {e | e ∈ TPc, t = src(e)} :
                    [OutputPorts(this)((dst(e), e("ConnectorB")))]
                     ← eval((e("Values"), createEnv(t))@now
                  end,
                  [OutputPorts(this)("schedule")] ← ⊥@now,
             end
           end
     end
```

5.5 Primitive components

In this section we give a short impression of how to use ASM component schemata to specify the behavior of *primitive components*, which we understand to be components that do not interpret a graph of a visual language (and are therefore not parameterized by a graph), but rather perform some specific 'built-in' task. We take this short departure from the main topic of this work to demonstrate how the language we have developed can be put to use to solve more traditional programming problems.

As an example we describe a component that generates prime numbers by the method known as the *Sieve of Erathostenes*. It has an input port and an output port, and for each input token (whose value is irrelevant) it produces the next prime number, starting from 2.

The component schema in Alg. 9 implements this. After the declaration of the input and output ports (Lines 2-4), binding the *fire* input port to a *doNothing* rule and the *input* input port to the rule that actually computes the next prime, we declare two attributes to be the vocabulary (Lines 6-7). The attribute *n* holds the next prime (after the last that has been output), while the attribute *sieve* contains an object representing the set of all primes output so far. The initialization sets them to initially 2 and the empty set (which is represented by an object associated with a map that is $\perp$ everywhere), respectively (Lines 9-12).

Executing the *handleInput* rule happens in two phases, one basic rule that is executed once, and one that is iterated until its update set is empty. In the first phase (Lines 16-18), the next prime (the one in *n*) is output, added to the set of primes written out and then *n* is incremented.

In the second phase (Lines 20-22), the next prime is computed iteratively. The technique employed here relies on the definition of the choose-construct (page 93), which specifies that the body of the choose is not executed if and only if the set which is chosen from is empty (in that case, if an else-clause is present, this would be executed instead, but here the empty else-clause is omitted). With the current value of n (i.e. the prime which was output before plus one), we pick some number from the set of all primes so far which evenly divide the current n.[5] Of course, this set is empty exactly if n contains the next prime, in which case the choose-construct has no effect and the iteration stops, with n containing the next prime. Otherwise, if that set was not empty, we increment n. This makes the update set non-empty, and consequently makes the iteration proceed, until an *n* is found which is not evenly divided by any of the primes found so far.

Alg. 9: (Sieve of Erathostenes)

```
1  class Sieve[] is
2      input {"input"} : handleInput,
3            {"fire"} : doNothing ;
4      output {"output", "schedule"} ;
5
6      attribute n ;
7      attribute sieve ;
8
9      initialize :
10             sieve := [],
```

[5]Obviously, we could improve the efficiency of an implementation of this algorithm if we chose from a smaller set, viz. $\{k \mid k \in sieve, k * k \leq n, n \mod k = 0\}$.

```
                n := 2
        end

        rule handleInput[p, t, v] :
        once
            [OutputPorts(this)("output")] ← n@t,
            sieve := sieve + [n ↦ true],
            n := n + 1
        then
            choose a ∈ {k | k ∈ sieve, n mod k = 0} :
                n := n + 1
            end
        end

        rule doNothing[p, t, v] :
            skip
        end
end
```

6

Conclusion

This work presented a comprehensive approach to the specification of graph-like visual notations for the description of discrete-event systems. Starting from a very general concept of an actor, we described a discrete-event model of computation using a special class of these actors, a notion of dynamically changing network structure, and an abstract concept of time.

Then we presented a notion of abstract visual syntax, which provided us with a straightforward way of representing visual programs as mathematical structures. Our approach of specifying visual language semantics is to define mechanisms (which we called 'schemata') that generate an actor from a given visual program (i.e. a graph). This actor was then said to be the *interpreter* of that program.

Seeing that the specification of these actors directly in the formal terms of their definition was somewhat cumbersome, we then developed a more concise and better structured language for the specification of actors. This language was based on concepts developed for *Abstract State Machines*—a simple but very general notion of state, and a powerful concept of atomic state change. We added some structuring facilities and extensions to accommodate the special features of our actor model, in particular input/output ports, and a dynamically changing network structure between them. The language is relatively small, is open to extensions by arbitrarily powerful side-effect-free constructions, and has a fully formal semantics.

This language we then applied to writing interpreters for a variety of visual notations. The resulting specifications were relatively compact. They are operational and executable, and there exists a prototype implementation generating code from such specifications suitable for execution in the MOSES environment (App. E). Building an actor language on the basis of Abstract State Machines was an appropriate choice for a language intended to describe the semantics of discrete-event state-based notations.

There are a number of directions in which this work could be extended and improved.

- The first class of extensions concern the syntactical framework presented here. Attributed graphs as an abstract syntax may not be the most appropriate structure to represent some visual notations in. In particular, hierarchical graph structures, where vertices contain sub-graphs, might better be represented in some other abstract syntax [58, 59].

  On the other hand, we might like to be able to work with visual notations other than graph-like diagrams—diagrams that interpret visual features such as size, color, relative placement (containment, overlapping, touching, crossing of areas or lines—cf. [37]). Or one might want to extend the framework developed here towards the definition of textual languages, incorporating the Montages approach to describing the operational textual language semantics in an ASM framework [72].

  It seems that rather than designing a new abstract syntax concept for each of those special cases, a somewhat broader notion is needed to be able to represent the necessary abstract syntax structures in a common framework.

- Another possible direction of work would be the improvement of our specification technique and its actor language. So far, we have not addressed the issue of compositionality of a specification itself. Clearly, as we have seen in Chap. 5, many visual language semantics can be considered to consist of a set of 'ingredients', such as computation, temporal behavior, parameters, dynamic network structures and others. One interesting line of research would explore whether these could in fact be abstracted into reusable parts that could then be put together in new interpreter specifications, so that constructing a visual language semantics would consist of selecting and configuring standard library components and possibly handling their specific interactions in any given situation.

  Obviously, a language facilitating such a semantics description must exhibit a high degree of compositionality, something that our current rule language fails to address. Developing composition constructs for the rule language itself would therefore be an important contribution towards this goal. Another approach would be to use the semantics framework developed in Chap. 2 and embed another concrete actor language into it, possibly by adapting an existing one. In fact, given the variation in visual language semantics, it might turn out that description techniques other than the ASM-based one we proposed in this work might be more suitable for some languages. In these cases, it would certainly be most useful to have a tool box of actor languages available from which to choose.

  Finally, it might be interesting to actually visualize the actor language itself, i.e. to develop a visual notation for the definition of visual language interpreters.

- An important line of research concerns the use of the semantics specifications. Their most straightforward application is as abstract yet executable specifications of behavior, from which prototype implementations may be generated. However, recent work on verification of Abstract State Machines (using proof techniques and model checking) could be put to use in proving properties of interpreters, e.g. their equivalence to some formal model of computation, fairness, determinacy, equivalence of some transformation of the visual program etc.

  Similarly, there has been some research in efficient code generation by stepwise transformation of ASM specification into a target language. Being able to generate an efficient interpreter from an ASM specification would certainly add considerable value to the language and the method.

The basis for much of this further research would be the experience gained by using the current language in a wide spectrum of applications. So far, the experiences with rather general purpose modeling languages have been encouraging. The development of more specialized, more intricate visual notations will be the litmus test for the viability of this approach.

A

Denotation of deterministic actors

Here we will discuss one important property of deterministic actors, viz. their *denotation* in terms of a function on streams. The reason for this is that this is one of the major contributions in the actor model in [77] that our model is based on. In fact, we will be able to repeat the construction given there for our actors with state, thus generalizing it for our larger class of actors. Like for the stateless actors in [77], a very important result of this construction will be the fact that it coincides with the operational semantics defined in Def. 2. It is this property that makes it possible to analyze, e.g., a network of deterministic actors in terms of their denotations (which makes the network essentially a system of equations), and still implement it operationally without invalidating the results of the analysis.

Even though we are able to formulate a rather straightforward operational interpretation of actors, it may be desirable to consider actors as implementations of *functions* on sequences, effectively operational definitions of such a function. One possible motivation might be to analyze the properties of the functions they denote—for instance, it is well-known that networks of continuous functions are themselves continuous [71]. Therefore, if in a network of actors each one is known to compute a continuous function from its input sequences to its output sequences, this property is then automatically inherited by the network itself. In other words, continuity is a compositional property of actors.

In this section we will construct the denotation of deterministic actors. First though, we have to discuss a few preliminary concepts concerning the partial order of sequences known as the *prefix order*, then we characterize the subclass of deterministic actors, and then we construct their denotation by first showing their existence and then giving a constructive approximation procedure.

A.1 The order structure of sequences

In the definitions below we will need to express the property that one sequence is a prefix of (or equal to) another. In fact, this property defines a partial *prefix order* $\sqsubseteq$ over S and can be defined by using concatenation:

$$s \sqsubseteq s' \iff \exists t \in S : s + t = s' \tag{A.1}$$

The empty sequence λ, being a prefix of every sequence, is the unique smallest element of this partial order, also often referred to as its *bottom* element. This order gives rise to a notion of least upper bound for a set of sequences. Let $X \subseteq S$ be a set of sequences, then $\sqcup X$ denotes its least upper bound in the prefix order, i.e.

$$\forall s \in S, x \in X : s \sqsubseteq x \wedge s \sqsubseteq \sqcup X \implies s = \sqcup X \tag{A.2}$$

Note that not all sets of sequences have a least upper bound, e.g. the sequences $\{aa, ab\}$ do not.

A *chain* in a partial order is a sequence of elements (sequences in our case) such that each following element is greater than or equal to all its predecessors, i.e.

$$(a_i)_{i \in \mathbb{N}} \quad \text{such that} \quad a_i \sqsubseteq a_j \iff i \leq j \tag{A.3}$$

If every chain has a least upper bound, the partial order with a bottom element is called a *complete partial order*, or *cpo*. Examples of cpos would be $(\mathbb{N}, \leq)$ or $(\mathbb{R}_0^+, \leq)$, while neither $(\mathbb{R}, \leq)$ nor $(\mathbb{R}^+, \leq)$ are a cpo.[1] It is easy to see that $(S, \sqsubseteq)$ is a cpo, with λ of course being the bottom element.

In the following, we will mostly deal with tuples of sequences, instead of single sequences. In this, we will assume the prefix order to extend naturally over tuples of sequences, such that for $s, r \in S^n$, with $s = (s_1, ..., s_n)$ and $r = (r_1, ...r_n)$:

$$s \sqsubseteq r \iff \forall i \in \{1, ..., n\} : s_i \sqsubseteq r_i \tag{A.4}$$

Obviously, any $(S^n, \sqsubseteq)$ is also a cpo. We call its least element, the n-tuple of empty sequences, λ_n

One way of discussing the semantics of dataflow actors will be by considering them as *functions* on streams. If it has m inputs and n outputs, then it can be thought of as a function

$$F : S^m \longrightarrow S^n$$

[1] We use $\mathbb{R}^+$ for the strictly positive real numbers, i.e. without 0.

which takes an m-tuple of streams and maps it to an n-tuple of streams. Viewing actors in this way makes it possible to discuss their 'mapping' properties. Two interesting properties in this context are *monotonicity* and *continuity*, which are of course defined based on the prefix order on tuples of streams.

Def. 39: **(Monotonic function)** *A function $F : S^m \longrightarrow S^n$ is called* monotonic *iff*

$$\forall s, r \in S^m : s \sqsubseteq r \Longrightarrow Fs \sqsubseteq Fr \tag{A.5}$$

Intuitively this means that providing more input tokens will only produce more output tokens, which can be considered a kind of causality property. Continuity is defined in terms of the least upper bound of the values of the function on a chain of input sequences as follows.

Def. 40: **(Continuous function)** *A function $F : S^m \longrightarrow S^n$ is called* continuous *iff for any chain X in $(S^m, \sqsubseteq)$, $F(X)$ has a least upper bound $\sqcup F(X)$ and*

$$F(\sqcup X) = \sqcup F(X) \tag{A.6}$$

If we consider a chain of finite sequences as increasingly good approximation of an infinite sequence, then continuity means that we can approximate F's value for an infinite input sequence (which can be infinite itself) arbitrarily 'good' by computing it for longer and longer finite input sequences. This property, if it holds, is central to constructing a denotational semantics for an actor.

A.2 Constructing the denotational semantics

In the following we will construct a denotational semantics for deterministic actors that we have already informally introduced in the above examples. Note that Def. 1 allows for two sources of non-determinism of an actor:

- Given a tuple of input sequences s and a state σ, there may be any number of active prefixes in $P_\sigma s$.
- For any active prefix $p \in P_\sigma s$, the transition function may yield any positive number of successor state/output combinations in $\tau_\sigma p$.

In order to make an actor deterministic, we have to restrict its definition at these two points:

Def. 41: **(Deterministic actor condition)** *We call an actor* deterministic *iff*[2] *it has at most one active prefix for any input in any state and the transition function*

[2]In fact, the following condition could be considered too strong in the sense that it declares some actors nondeterministic whose observable behavior is perfectly determinate, viz. those that have several fully equivalent internal states and a transition function such that these occur as alternative successor states with the same output. Since this is a rather special case, we will not consider this point any further.

always returns singleton sets of possible successor state/output pairs.

$$\forall \sigma \in \Sigma, s \in S^m : \mid P_\sigma s \mid \leq 1$$
$$\forall \sigma \in \Sigma, p \in D_\sigma : \mid \tau_\sigma p \mid = 1$$

For notational convenience in this section, the condition on the uniqueness of τ_σ for deterministic actors allows us to turn it into two functions ν_σ and ϕ_σ defined as follows:

$$\tau_\sigma p = \{(\sigma', v)\} \iff \begin{array}{l} \nu_\sigma p = \sigma' \\ \wedge\, \phi_\sigma p = v \end{array}$$

Now we construct the function (on sequences) computed by an actor, which we will call its *process* following the use of that word in [77]. We do this by defining a recursive function $f_\mathcal{A}$ that takes a state and an input tuple, fires once on the prefix of that input tuple (note that there must be exactly one), and appends its results on the rest of the input and the next state to the output of this firing. Once this function is defined, we only need to 'start' it in the initial state σ_0, which is done by another function $F_\mathcal{A}$. An this simply gives us the actor function:

Def. 42: **(Actor function)** *Given a deterministic actor $\mathcal{A} : S^m \rightsquigarrow S^n$, we define its* process *to be a function $F_\mathcal{A} : S^m \longrightarrow S^n$ as follows:*

$$F_\mathcal{A} : s \mapsto f_\mathcal{A}(\sigma_0, s) \tag{A.7}$$

$$f_\mathcal{A} : (\sigma, s) \mapsto \begin{cases} \phi_\sigma p + f_\mathcal{A}(\sigma', s') & \text{if} \quad p \in P_\sigma s \quad \text{with} \quad p + s' = s, \sigma' = \nu_\sigma p \\ \lambda_n & \text{if} \quad P_\sigma s = \emptyset \end{cases} \tag{A.8}$$

Unfortunately, due to the recursive structure of the definition of $f_\mathcal{A}$ there is no guarantee that there exists a function that fulfills this definition, that it is in fact well-defined.

However, there is a way of showing that one exists, and even to construct it. This is accomplished by a least fixed point construction, making use of the complete partial order nature of the space of sequences, and functions over them.

The basic idea is to as follows. We define a 'functional' (i.e. a function operating on functions) that takes a 'candidate' f and returns a new function f' (both taking a state and an input tuple, and producing an output tuple) which does the following: f' makes exactly one step of the recursive definition above, i.e. firing the actor exactly once, and then applying f to the rest of the input. Note that if f is well-defined, then so is f', because it is simply the result of the application of the functional to f, and the functional is not recursive.

Def. 43: **(Actor functional)** *Given an actor $\mathcal{A} : S^m \rightsquigarrow S^n$, we define its* actor functional *as follows:*

$$\Phi_{\mathcal{A}} : (\Sigma \times S^m \longrightarrow S^n) \longrightarrow (\Sigma \times S^m \longrightarrow S^n) \tag{A.9}$$

$$\Phi_{\mathcal{A}} f(\sigma, s) = \begin{cases} \phi_\sigma p + f(\sigma', s') & \textit{if} \quad p \in P_\sigma s \quad \textit{with} \quad p + s' = s, \sigma' = \nu_\sigma p \\ \lambda_n & \qquad\qquad P_\sigma s = \emptyset \end{cases} \tag{A.10}$$

The key observation at this point (and in the preservation of well-definedness by $\Phi_{\mathcal{A}}$) is that any *fixed point* of $\Phi_{\mathcal{A}}$, any f that satifies $f = \Phi_{\mathcal{A}}\, f$, automatically is a valid $f_{\mathcal{A}}$ because it satisfies its recursive definition in Def. 42. Obviously, there could in principle be any number of such fixed points, including zero. In the following we will show that a minimal fixed point exists (by virtue of the properties of the actor and the way it consumes and produces tokens), and that it is unique. We will also give a way to actually construct this fixed point, or arbitrarily good approximations to it.

Instead of acting on sequences, the actor functional acts on functions. It will be crucial to our construction that the set of these functions form a complete partial order as well. First we define an order among functions that return tuples of sequences (such as the $f_{\mathcal{A}}$ or the arguments of $\Phi_{\mathcal{A}}$).

Def. 44: **($(\langle A \longrightarrow S^n \rangle, \sqsubseteq)$)** *Given an arbitrary set A and a set of tuples of sequences S^n, the partial order over tuples of sequences induces a partial order $\sqsubseteq$ among functions in $\langle A \longrightarrow S^n \rangle$ as follows:*

$$f \sqsubseteq g \Longleftrightarrow \forall a \in A : fa \sqsubseteq ga$$

In other words, a function is 'greater than or equal to' another function in this order if for the same input it produces at least the same output sequences as the other function, and possibly more. Obviously, if we have the choice between two candidates of $f_{\mathcal{A}}$, we would like to pick the one that produces as little output as possible, which is why we are especially interested in finding a *minimal* fixed point of the actor functional, if it exists.

Th. A.1: **(Functions on cpo form a cpo)** *If $(Y, \sqsubseteq)$ is a cpo, then so is $(\langle X \longrightarrow Y \rangle, \sqsubseteq)$.*

Proof. We have to show two things for $(\langle X \longrightarrow Y \rangle, \sqsubseteq)$:

- the existence of a unique least element and
- that each chain has a least upper bound.

Let us call Λ the function defined by $\Lambda : x \mapsto \bot$, i.e. it maps everything to the smallest element of Y. Since this is smaller than any $y \in Y$, $\Lambda \sqsubseteq f$ for any function $f \in \langle X \longrightarrow Y \rangle$. Therefore, Λ is the least element in $(\langle X \longrightarrow Y \rangle, \sqsubseteq)$.

For any chain $(f_i)_{i \in \mathbb{N}}$ we construct its least upper bound by pointwise taking the least upper bound of its function values. This is guaranteed to exists since

$(f_i x)_{i\in\mathbb{N}}$ is a chain (because the f_i are a chain) in Y, and Y is a cpo. Thus we can define

$$f : x \mapsto \sqcup(f_i x)_{i\in\mathbb{N}}$$

Obviously f is an upper bound to all f_i (since it is pointwise bigger than either of them), and it is also the least upper bound, because the existence of some distinct $f' \sqsubseteq f$ would imply the existence of some $x \in X$ such that $f'x \sqsubseteq fx$ and $f'x \neq fx$. However, by construction, $fx = \sqcup(f_i x)_{i\in\mathbb{N}}$, which is a contradiction.

Thus, we have a least element, and every chain has a least upper bound, therefore $(\langle X \longrightarrow Y\rangle, \sqsubseteq)$ is a cpo. □

Cor. A.2: *The actors functions $f_{\mathcal{A}} : \Sigma \times S^m \longrightarrow S^n$ from Def. 42 form a cpo.*

The next theorem will yield the existence of a unique least fixed point of $\Phi_{\mathcal{A}}$. For this, we need to show that it is monotonic.

Th. A.3: (Monotonicity of $\Phi_{\mathcal{A}}$—existence and uniqueness of $\mu\Phi_{\mathcal{A}}$) *For any deterministic actor $\mathcal{A}$, its functional $\Phi_{\mathcal{A}}$ is monotonic, in other words:*

$$\forall f, f' \in \langle \Sigma \times S^m \longrightarrow \Sigma \times S^n\rangle : f \sqsubseteq f' \Longrightarrow \Phi_{\mathcal{A}} f \sqsubseteq \Phi_{\mathcal{A}} f'$$

This implies the existence of a unique minimal fixed point, which we will call $\mu\Phi$.[103]

Proof. Assume we have two $f, f' \in \langle \Sigma \times S^m \longrightarrow S^n\rangle$ such that $f \sqsubseteq f'$. We have to show that $\Phi_{\mathcal{A}} f \sqsubseteq \Phi_{\mathcal{A}} f'$, in other words for all (σ, s) we must have

$$\Phi_{\mathcal{A}} f(\sigma, s) \sqsubseteq \Phi_{\mathcal{A}} f'(\sigma, s) \tag{A.11}$$

We need to distinguish two cases, viz. whether $P_\sigma s$ is empty or not. If it is empty, we have

$$\Phi_{\mathcal{A}} f(\sigma, s) = \Phi_{\mathcal{A}} f'(\sigma, s) = \lambda_n$$

which satisfies (A.11).

If $P_\sigma s \neq \emptyset$, due to the determinacy of the actor it must contain exactly one element, which we call p, and which is a prefix of s, i.e. there exists an s' such that $p + s' = s$. Then with $\sigma' = \nu_\sigma p$ and $a = \phi_\sigma p$ the following is true:

$$\Phi_{\mathcal{A}} f(\sigma, s) = a + f(\sigma', s') \sqsubseteq a + f'(\sigma', s') = \Phi_{\mathcal{A}} f'(\sigma, s)$$

Therefore, in this case, too, (A.11) holds.

Since $\Phi_{\mathcal{A}}$ is monotonic on a complete partial order $\langle \Sigma \times S^m \longrightarrow \Sigma \times S^n\rangle$, it has a unique minimal fixed point $\mu\Phi \in \langle \Sigma \times S^m \longrightarrow \Sigma \times S^n\rangle$. □

Now we have to show that an actor functional is continuous, since continuity will immediately give us a procedure to iteratively construct arbitrarily good approximations to its fixed point.

Th. A.4: **(Continuity of $\Phi_{\mathcal{A}}$)** *For any deterministic actor $\mathcal{A}$, its functional $\Phi_{\mathcal{A}}$ is continuous, in other words for any chain $\mathbf{f} = (f_i)_{i\in\mathbb{N}}$:*

$$\Phi_{\mathcal{A}} \sqcup \mathbf{f} = \sqcup(\Phi_{\mathcal{A}}\mathbf{f})$$

Proof. We have to show that for each (σ, s),

$$\Phi_{\mathcal{A}}(\sqcup\mathbf{f})(\sigma, s) = \sqcup(\Phi_{\mathcal{A}}\mathbf{f})(\sigma, s) \tag{A.12}$$

First note that the construction in the proof of Theorem A.1 of the least upper bound of a chain of functions is as follows:

$$\sqcup(f_i)_{i\in\mathbb{N}}a = \sqcup(f_i a)_{i\in\mathbb{N}}$$

This allows us to formulate (A.12) as follows:

$$\Phi_{\mathcal{A}}(\sqcup\mathbf{f})(\sigma, s) = \sqcup(\Phi_{\mathcal{A}}\mathbf{f}(\sigma, s)) \tag{A.13}$$

Again we distinguish two cases, viz. whether $P_\sigma s$ is empty or whether it contains exactly one element p. First, assume it is empty. Then

$$\sqcup(\Phi_{\mathcal{A}}\mathbf{f}(\sigma, s)) = \sqcup(\Phi_{\mathcal{A}}f_i(\sigma, s))_{i\in\mathbb{N}} = \lambda_n = \Phi_{\mathcal{A}}(\sqcup\mathbf{f})(\sigma, s)$$

since for any f_i we have $\Phi_{\mathcal{A}}f_i(\sigma, s) = \lambda_n$ in this case.

If there is a $p \in P_\sigma s$, and $s = p + s'$, $\sigma' = \nu\sigma p$, $a = \phi\sigma p$, we have

$$\begin{aligned}
&\sqcup\ (\Phi_{\mathcal{A}}\mathbf{f}(\sigma, s)) \\
=&\sqcup\ \{a + f_i(\sigma', s') \mid i \in \mathbb{N}\} = a + \sqcup\{f_i(\sigma', s') \mid i \in \mathbb{N}\} = a + \sqcup\mathbf{f}(\sigma', s') \\
=&\Phi_{\mathcal{A}} \sqcup \mathbf{f}(\sigma, s)
\end{aligned}$$

This concludes the proof that $\Phi_{\mathcal{A}}$ is indeed continuous. □

Cor. A.5: **(Denotational semantics of determinate actors)** *Theorem A.4 gives us the continuity of $\Phi_{\mathcal{A}}$. Since in a cpo, the least minimal fixed point μf of a continuous function f is given by*[3]

$$\mu f = \sqcup(f^i\bot)_{i\in\mathbb{N}}$$

we can therefore conclude that the minimal fixed point of an actor functional, and therefore the actor function, is

$$f_{\mathcal{A}} = \mu\Phi_{\mathcal{A}} = \sqcup(\Phi_{\mathcal{A}}\Lambda)$$

which makes

$$F_{\mathcal{A}}s = f_{\mathcal{A}}\sigma_0 s = \mu\Phi_{\mathcal{A}}(\sigma_0, s) \tag{A.14}$$

[3] Cf. [100].

This denotation of an actor is consistent with its operational semantics (Def. 2). The first few approximation steps in the construction of the fixed point would be the following:

$$
\begin{aligned}
f^0_{\mathcal{A}}\Lambda(\sigma_0, s) &= \Lambda(\sigma_0, s)\\
f^1_{\mathcal{A}}\Lambda(\sigma_0, s) &= \phi_{\sigma 0}p + \Lambda(\sigma_1, s')\\
f^2_{\mathcal{A}}\Lambda(\sigma_0, s) &= \phi_{\sigma 0}p + \phi_{\sigma_1}p' + \Lambda(\sigma_2, s'')\\
f^3_{\mathcal{A}}\Lambda(\sigma_0, s) &= \phi_{\sigma 0}p + \phi_{\sigma_1}p' + \phi_{\sigma_2}p'' + \Lambda(\sigma_3, s''')
\end{aligned}
$$

This is of course exactly the way the result sequence in the run is constructed—more specifically, the operational semantics generates the sequences of approximations to the least fixed point of the actor functional. In other words, for deterministic actors, our denotational semantics is fully abstract [110] with respect to the operational semantics.

In other words, adding a notion of state to dataflow actors with firing retains their property (of stateless actors as defined in [77]) of having a denotation as a function on streams which is compatible with their operational semantics, providing the actor is deterministic.

B

Products

In the definition of actor networks and their structure we made use of a notion of *product* that stems from category theory[1]. In the following we will introduce the basic idea outside of its categorial context, since in this work it is only used in the category of sets.

$$\begin{tikzcd}
 & X \arrow[dl, "f"'] \arrow[d, dashed, "h"] \arrow[dr, "g"] & \\
A & C \arrow[l, "\pi_A"'] \arrow[r, "\pi_B"] & B
\end{tikzcd}$$

Fig. 39: A product and its projections.

Assume we have two sets A and B. We will call a third set C together with two *projections* $\pi_A : C \longrightarrow A$ and $\pi_B : C \longrightarrow B$ a *product* of A and B iff for any set X and two functions $f : X \longrightarrow A$ and $g : X \longrightarrow B$ there is a unique function $h : X \longrightarrow C$ such that the diagram in Fig. 39 commutes,[2] i.e.

$$f = \pi_A \circ h$$
$$g = \pi_B \circ h$$

Often, the set C is simply called $A \times B$, implicitly assuming projections $\pi_A : (a, b) \mapsto a$ and $\pi_B : (a, b) \mapsto b$, but note that the above definition of

[1] cf. [76, 85] for more comprehensive introductions

[2] The uniqueness requirement on h is expressed by the dashed arrow.

product does not require C to be a set of tuples. On the other hand, every product of two sets must be isomorphic to the 'ordinary' cartesian product (this easily follows from the uniqueness of h). Note also that strictly speaking we cannot talk of C as a product without giving the appropriate projections into A and B.

$$\begin{tikzcd}
X \arrow[d, dashed, "h"] \arrow[dr, "f_i"] & \\
\prod\limits_{i\in I} A_i \arrow[r, "\pi_i"] & A_i
\end{tikzcd}$$

Fig. 40: An n-ary product.

Obviously, this can be generalized to any number of sets, as in Fig. 40.

C

Parallel combination of rule results

Here we will define the parallel combination operator for result sets of basic rules that we used in Chapter 4. First, recall that a basic rule r has a denotation of the following form:

$$[r] : (\mathfrak{I}, \mathbf{V}) \mapsto \{(U_k, P_k, c_k^+, c_k^-, N_k) \mid k \in K\}$$

It returns a set of tuples consisting of

- an update set U_k,
- an output function (mapping ports to output sequences) P_k,
- connection sets c_k^+ and c_k^-,
- a set of new actors N_k.

One such tuple we call *rule outcome*. A set of these structures we call a *result set*. Since rules may be arbitrarily non-deterministic, these result sets may be arbitrarily large. The following basic rule produces an infinite result set:

choose $n \in \mathbb{N}$:
    $a := n$
end

Now we want to combine the results of several rules such that they are considered to be executed 'in parallel'. For two rules, say r_1 and r_2, this was written as

$$r_1, r_2$$

The result of executing them in parallel should be a set such that we take all combinations of outcomes from both result sets and 'merge' them. Intuitively, merging two outcomes involves joining their update sets, connection sets, and sets of new actors. Joining their outputs poses a difficulty—we have to allow outputs to be put in any order, i.e. any permutation. For example:

$[p] \leftarrow 1,$
$[p] \leftarrow 2$

produces *two* possible results at the output port p: $[1, 2]$ or $[2, 1]$. From this we can see that our rules are non-deterministic even without using a choose-construct—at least in the output they produce. All other parts of the rule result are, of course, not affected by this.

Unfortunately, the do-forall-construct allows for unbounded, and in principle even infinite, parallel combination of update sets. The following definition handles all these issues by constructing the set Ξ of selector functions, i.e. functions which pick one outcome from each result set. Then it joins the sets and adds the output sequences for each of those selections. The final result is just the union of all results for all selector functions.

Def. 45: **(Parallel combination of result sets)** *Assume a set of result sets*

$$\mathbf{R} = \{R_k \mid k \in K\}$$

indexed by some set K. We define the parallel combination $\Diamond\mathbf{R}$ *of these result sets as follows.*

First let $\Xi = \left\{\xi : K \longrightarrow \bigcup_{k \in K} R_k \mid \xi\, k \in R_k\right\}$ *be the set of choice functions, which pick an outcome out of each result set. We will write* $(U_{\xi k}, P_{\xi k}, c^+_{\xi k}, c^-_{\xi k}, N_{\xi k})$ *for the outcome* ξk*.*

Furthermore, for any set $W \subseteq \mathfrak{U}^*$*, let* $\sum W \subseteq \mathfrak{U}^*$ *be the set of all sequences that arise as the sum of the sequences in* W *in any order.*

Then we define the parallel combination of the results sets in $\mathbf{R}$ *as follows:*

$$\Diamond\mathbf{R} = \left\{\left(\bigcup_{k \in K} U_{\xi k}, P, \bigcup_{k \in K} c^+_{\xi k}, \bigcup_{k \in K} c^-_{\xi k}, \bigcup_{k \in K} N_{\xi k}\right) \mid \xi \in \Xi, Pp \in \sum\{P_{\xi k}(p)\}\right\}$$

Note that this definition implies that $\Diamond\emptyset = \{(\emptyset, P_\lambda, \emptyset, \emptyset, \emptyset)\}$.

D

ELAN—An expression language

D.1 Introduction

The ELAN expression language is a small language designed for use in inscriptions in MOSES models. Its design goals were the following:

- Simple syntactic structure.
- Simple, side-effect free evaluation semantics, as well as simple constructs for side-effects and iteration.
- Interoperability with the underlying host-language, i.e. Java. Java structures should be usable by ELAN constructions, and ELAN objects should have a straightforward Java representation.
- Support for a number of basic and structured types that allows for the construction of practical models without recourse to Java.
- No static typing.

Evaluation efficiency was not the primary design goal, the rationale being that algorithmically complex activities and runtime-critical parts would be done in Java anyway. We had no intention to duplicate existing facilities, only to merge them into MOSES models in a sufficiently convenient manner.

D.2 Preliminaries

The key concept of the language is an *expression*. There are a variety of expressions in ELAN, but their common characteristic is that all of them can be *evaluated* to produce a *value*. This value in general depends on an *environment*, which is simply a (partial) mapping of variable names to objects.

The evaluation of some expressions may also produce *side-effects*. Side-effects come in different forms—some expression may change the environment by substituting the object a variable is bound to for another object, other expression may change an object's internal state. We will hint at possible side-effects whenever they may occur.

In the following, we will denote general expressions by symbols e and e_i, boolean expressions (conditions) by c and c_i, and variable symbols by v and v_i. Operator symbols will be written as $\circ$.

D.2.1 Atomic expressions

Atomic expressions are those containing no other expressions. These are variable symbols and a number of constants of different types.

Each identifier is a variable symbol, evaluating to the value it is bound to in the current environment.

ELAN recognizes constants of the following types:

- Integers: any sequence of decimal digits optionally prefixed with a sign.
- Real numbers: any sequence of digits with a period, optionally prefixed with a sign and appended with a decimal exponent in the form of an "E" followed by an optionally signed integer.
- Strings: any sequence of characters enclosed in double quotes.
- Boolean: any of the two keywords **true** and **false**.
- The symbol **null**, denoting the null object.

D.2.2 Unary Operators

There are two unary operators.

$$-\,e$$

requires that e evaluates to a number and computes the negative value of that number.

$$\sim\, e$$

requires e to result in a boolean value, and computes the negation of that value.

D.2.3 Binary Operators

The application of a binary operator $\circ$ has the following format:

$$e_1 \ \circ \ e_2$$

ELAN currently does not define precedence rules, so one needs to provide proper parentheses to disambiguate expressions.

There are a number of binary operators to manipulate various kinds of objects. For numerical operands, these are the following:

$$+, -, *, /, \text{mod}, \text{div}$$

with the usual interpretations.

For sets, we have the operators

$$+, -, *$$

denoting union, set difference and intersection, respectively. Additionally, the boolean expression

$$e_1 \text{ in } e_2$$

requires e_2 to evaluate to a set, and then it is true if the value of e_1 is an element of that set and false otherwise.

Lists can be concatenated using the $+$ operator.

Maps may be 'added'using the $+$ operator, where the mappings of the right-hand map override those of the left-hand map. Maps may be concatenated using the $*$ operator.

Boolean values may be manipulated using the operators

$$\&\&, ||$$

denoting conjunction and disjunction, respectively.

Finally, objects may be compared to each other. All objects may be compared for equality or inequality using $=$ or $<>$. Objects that have an order defined on them may be compared using the operators

$$<, <=, >, >=$$

For numbers, the order is the numerical one, for strings it is lexical order, and for sets it is inclusion.

D.2.4 Comprehensions and basic structured data objects

ELAN provides direct support for three kinds of structured data objects: sets, lists, and maps. They are constructed by so-called *comprehensions* in the following manner.

A set comprehension has the form:

$$\{e_1, ..., e_n : \underline{\textbf{for}}\ v_1\ \underline{\textbf{in}}\ e_{1,s}, c_{1,1}, ..., c_{1,k_1}, ..., \underline{\textbf{for}}\ v_m\ \underline{\textbf{in}}\ e_{m,s}, c_{m,1}, ..., c_{m,k_m}\}$$

The constructions $\underline{\textbf{for}}\ v_i\ \underline{\textbf{in}}\ e_{i,s}$ are called *generators*, the boolean expressions $c_{i,j}$ are *filters*. The $e_{i,s}$ must evaluate to sets. Such a set comprehension evaluates by iteratively creating a number of bindings for the v_i, viz. all those in which a variable v_i is bound to an element of the set that results from evaluating $e_{i,s}$ so that all $c_{i,j}$ are true. For each of these bindings, the expressions e_i at the head of the comprehension are evaluated and all resulting objects are added to the set. Examples of set comprehensions:

$$\begin{aligned} \{1, 2, 3, 4\} \\ \{a : \underline{\textbf{for}}\ a\ \underline{\textbf{in}}\ s, a\ \underline{\textbf{mod}}\ 2 = 0\} \\ \{a * b : \underline{\textbf{for}}\ a\ \underline{\textbf{in}}\ s1, \underline{\textbf{for}}\ b\ \underline{\textbf{in}}\ s2\} \\ \{a, b : \underline{\textbf{for}}\ a\ \underline{\textbf{in}}\ s1, \underline{\textbf{for}}\ b\ \underline{\textbf{in}}\ s2\} \end{aligned}$$

The first is a very simple comprehension without generators resulting in a set of four integers, the second assumes the existence of a variable s (bound to a set of numbers) and computes the set of even number in s, the third computes the products of all combinations from number in the sets (bound to) $s1$ and $s2$, and the last one is a roundabout (and inefficient) way to compute the union of $s1$ and $s2$.

The same mechanism can be used for constructing lists and maps. For lists, the general format is the following:

$$[e_1, ..., e_n : \underline{\textbf{for}}\ v_1\ \underline{\textbf{in}}\ e_{1,s}, c_{1,1}, ..., c_{1,k_1}, ..., \underline{\textbf{for}}\ v_m\ \underline{\textbf{in}}\ e_{m,s}, c_{m,1}, ..., c_{m,k_m}]$$

This constructs a list containing the enumerated elements, preserving the order among the e_i in each iteration.

A map is a finite mapping of *keys* to *values*. It is constructed in the following manner:

$$\begin{aligned} \underline{\textbf{map}}[e_{k,1}- > e_{v,1}, ..., e_{k,n}- > e_{v,n} : & \underline{\textbf{for}}\ v_1\ \underline{\textbf{in}}\ e_{1,s}, c_{1,1}, ..., c_{1,k_1}, ..., \\ & \underline{\textbf{for}}\ v_m\ \underline{\textbf{in}}\ e_{m,s}, c_{m,1}, ..., c_{m,k_m}] \end{aligned}$$

Here, the $e_{k,i}- > e_{v,i}$ denote the mapping of the key $e_{k,i}$ to the value $e_{v,i}$. If the comprehension enumerates the same key more than once, the value is chosen non-deterministically.

D.2.5 Closures and closure application

The expression

$$\underline{\textbf{lambda}}\ (v_1, ..., v_n)\ e\ \underline{\textbf{end}}$$

creates an n-ary closure, i.e. a function object encapsulating the current environment which is valid in the context of the evaluation of this expression. Such an object may be applied to a tuple of n actual parameters in an application expression, which has the following form:

$$e_{clos}\ (e_1, ..., e_n)$$

Here, e_{clos} must evaluate to an n-ary closure. The result of this application is that of evaluating the body expression e of the closure in an environment that binds the v_i to the value of e_i.

For example, assume f is bound to the value of **lambda** (x) $2 * x$ **end**, then the expression

$$f(3)$$

will return 6.

D.2.6 Structured expressions

The most basic structured expression is obtained by putting any expression in parentheses:

$$(e)$$

The value of this expression is that of e.

Another fundamental structured expression is the conditional. It has the following form:

if c **then** e_1 **else** e_2 **end**

If the value of c is true, then the value of this expression is the value of e_1, otherwise it is the value of e_2.

Another structured expression allows the definition of local variable bindings. The has the form

let $v_1 = e_1, ..., v_n = e_n : e$ **end**

The value of this expression is the value of e in an environment that binds v_i to the value of e_i.

D.2.7 Quantified expressions

Boolean expressions may be universally quantified over sets in the following manner:

forall v_1 **in** $e_1, ..., v_n$ **in** $e_n : c$ **end**

Here, the e_i are required to evaluate to sets. The result of this expression is true if c is true in all environments resulting from the binding of the v_i to values from the set resulting from e_i. It is false otherwise.

Likewise, a boolean expression may be existentially quantified as follows:

exists v_1 **in** $e_1, ..., v_n$ **in** $e_n : c$ **end**

This expression is true if c evaluates to true in at least one of the environments resulting from the binding to th e v_i to values from the set resulting from e_i. It is false otherwise.

D.2.8 Iterative constructs

Three constructs support iteration in ELAN: assignment, sequential expressions, and loops.

The assignment expression

$$v := e$$

evaluates e and changes the value of the variable v to the value of e. This is also the value of the assignment expression.

Sequential expressions are a way to evaluate a number of expressions after each other:

$$e_1 \,\underline{\mathbf{;}}\ldots\, \underline{\mathbf{;}}\, e_n$$

This evaluates the e_i in the order from left to right. The result of this sequential expression is the value of e_n.

Loops provide a way to iterate an expression until a condition is met:

$$\underline{\textbf{while}}\ c\ :\ e\ \underline{\textbf{end}}$$

This expression evaluates e as long as c evaluates to true. The value of this expression is the value of the last evaluation of e. If c was false at the very beginning, the value of this expression is null.

E

The MOSES Tool Suite

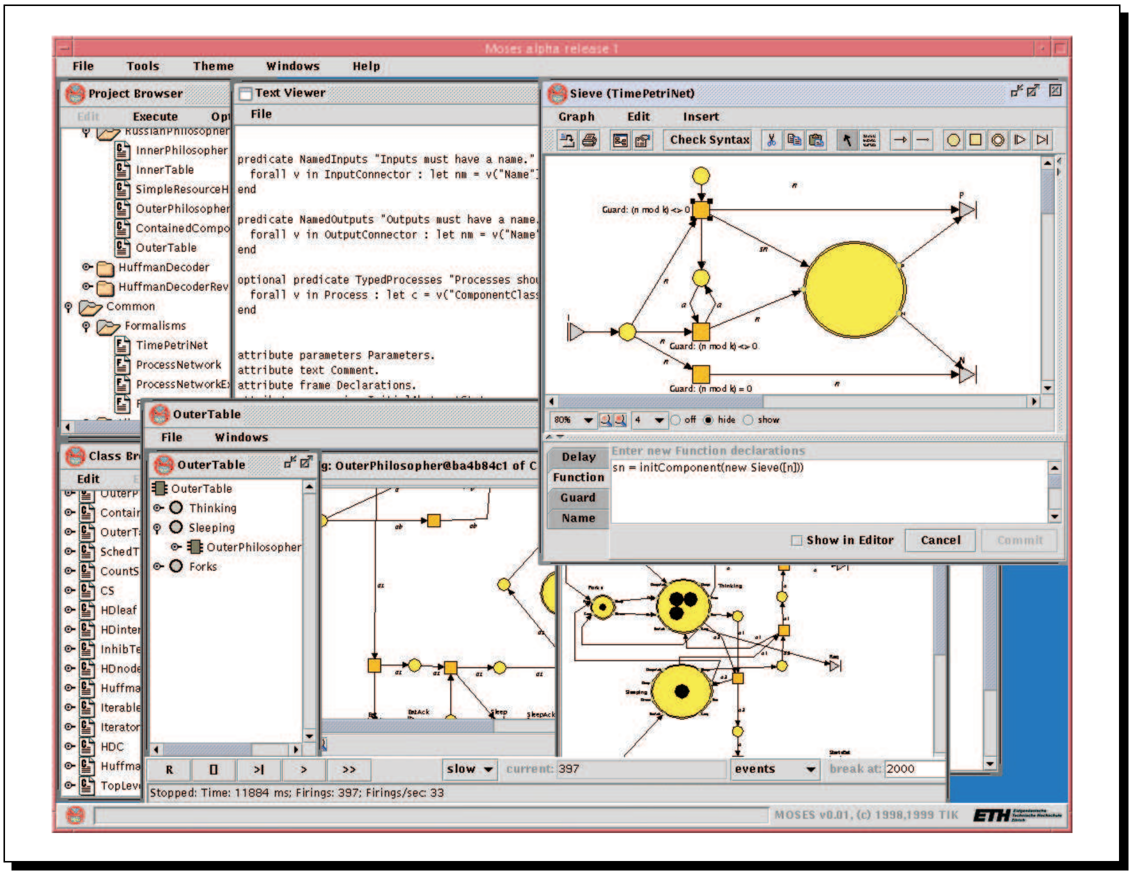

Fig. 41: MOSES Tool Suite, Overview

E.1 Introduction

The MOSES *Tool Suite* is a set of software tools developed in the MOSES Project [1]. It forms the technical environment of this work, and all implementation work has been done in it. The Tool Suite is focused on supporting the following tasks:

- Defining the syntax of new graph-like visual notations, including descriptions of its concrete syntax as well as the definition of syntactical constraints (in the way discussed in Chapter 3).
- Providing an editing environment for these languages which is generated from the syntax descriptions.
- Interpreting graphs on a generic execution platform, so that different visual languages can easily interoperate. Animating visual programs during their run.

In addition to tools directly supporting these tasks (a graph editor, a simulation environment, an animator, a graph translator), there are also management tools, such as a repository for storing and organizing the various files and objects that belong to a modeling and simulation project.

Here we want to give a brief overview of the definition of a visual notation using the Moses Tool Suite and how the resulting specification is used to configure the various parts of the software.

E.2 Syntax definition and editor

The syntax of a visual language is defined textually by enumerating the kinds of vertices and edges that occur in the language and by giving syntax rules for their composition, in the way described in Chapter 3. Section E.4 shows a full specification of a simple Petri net language, in the so-called *Graph-Type Definition Language (GTDL)* [92]. The kind of graph defined by such a description is called a *graph type*.

The specification in E.4 declares the following entities:

- Some kinds of vertices (*vertex types*), including a set of attributes for each (Place, Transition).
- A kind of edge (*edge type*), also along with attributes that describe it (Arc).
- Several syntax predicates that define the well-formedness of a graph (PlaceTransitionConnection, TransitionPlaceConnection, WeightsPositive, NonEmptyPreset).
- The class name of the compiler to be used to translate a graph into executable code (Compiler).

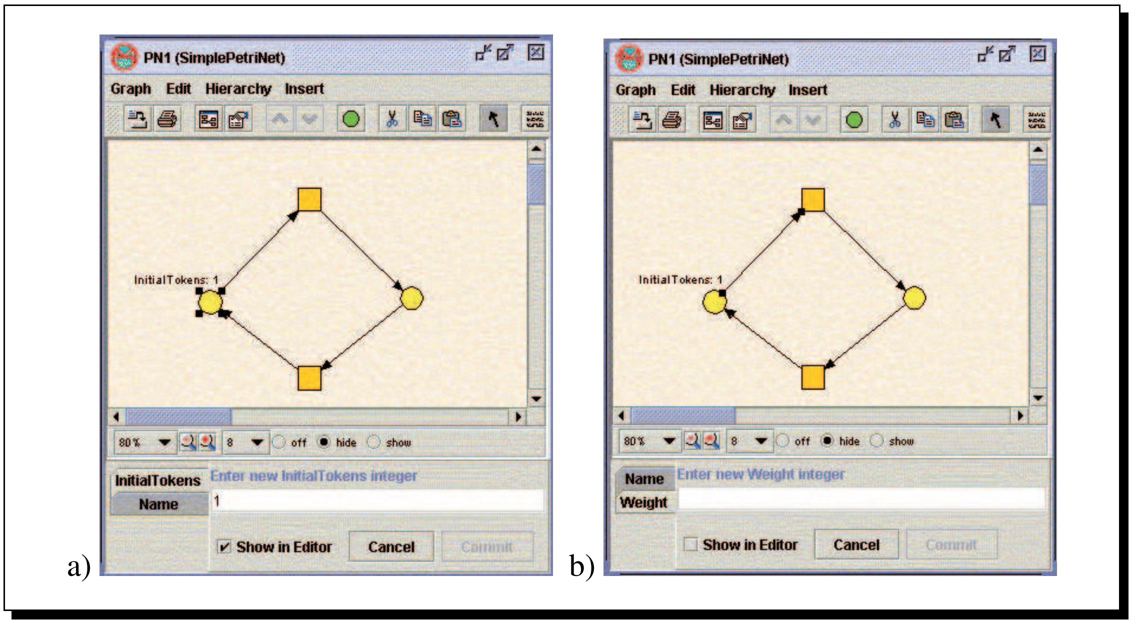

Fig. 42: Object-dependent attribute editors.

The definition of the vertex type (and similarly of an edge type) *Place* looks like this:

$$
\begin{array}{l}
\underline{\textbf{vertex}}\ \underline{\textbf{type}}\ Place(\underline{\textbf{string}}\ Name, \underline{\textbf{integer}}\ InitialTokens)\\
\underline{\textbf{graphics}}(\underline{\textbf{hidden}}\ \underline{\textbf{string}}\ Shape = "Oval",\\
\qquad\qquad \underline{\textbf{hidden}}\ \underline{\textbf{color}}\ Color = "black",\\
\qquad\qquad \underline{\textbf{hidden}}\ \underline{\textbf{color}}\ FillColor = "yellow",\\
\qquad\qquad \underline{\textbf{hidden}}\ \underline{\textbf{integer}}\ ExtentX = 24, \underline{\textbf{hidden}}\ \underline{\textbf{integer}}\ ExtentY = 24).
\end{array}
$$

After the name of the type we have a sequence of *attributes*, defined by a name (Name, InitialTokens) and their *type* (string, integer). This type defines the kind of objects that this attribute must contain. Following this is a similar list of graphical attributes, which define the appearance of a vertex when it is displayed. These are usually marked as 'hidden', which advises any tools processing such a description to hide these attributes from the user.

The most important tool that uses these descriptions is the graph editor. This is an interactive direct-manipulation editor that allows the construction of graphs according to such a graph type specification. It makes use of the definition of vertex and edge types in two ways: it displays instances of these types according to their graphical attributes, and it also provides *attribute editors* which are generated from the attribute definitions associated with a vertex or edge type. These attribute editors are dialogs that allow the user to manipulate the (non-hidden) declared attributes of an edge or a vertex. Fig. 42a shows the attribute editor generated from the definition of the Place vertex type (which is displayed in the lower part of the editor window whenever a Place vertex is

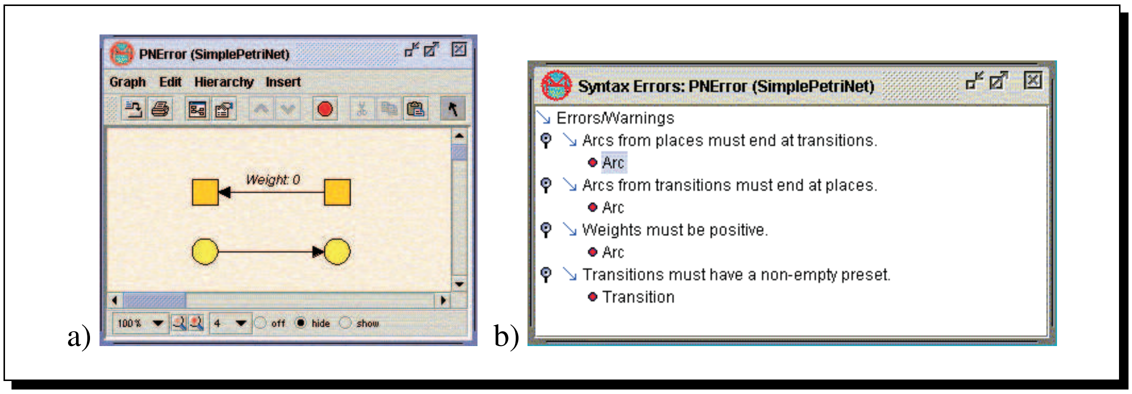

Fig. 43: Error messages generated from syntax predicates.

selected), while Fig. 42b shows the attribute editor displayed when an Arc edge is selected.

The editor also makes use of the syntax predicates in a graph type specification. For example, the syntax predicate that requires all Transition vertices to have a non-empty preset (i.e. there must be at least one Arc edge ending at each Transition vertex) has the following form:

$$
\begin{array}{l}
\underline{\textbf{predicate}}\ NonEmptyPreset \\
\qquad "Transitions\ must\ have\ a\ non-empty\ preset." \\
\qquad \underline{\textbf{forall}}\ t\ \underline{\textbf{in}}\ Transition: \\
\qquad\qquad \underline{\textbf{exists}}\ e\ \underline{\textbf{in}}\ Arc:\ dst(e) = t\ \underline{\textbf{end}} \\
\qquad \underline{\textbf{end}}
\end{array}
$$

Apart from the predicate name (NonEmptyPreset), a predicate has two parts: A message that explains the syntax constraint, and a logical expression that is true if it is fulfilled by the graph. Note that in the above example, variables are quantified over sets called *Transition* and *Arc*—each vertex type and edge type automatically defines a set of that name that contains exactly the vertices/edges of the corresponding type.

Fig. 43a shows a rather non-well-formed Petri net, that violates all four of the predicates in the example in Section E.4. This violation is indicated by the red button just above the main editing area. That button is green for syntactically correct graphs (such as the ones in Fig. 42, and it turns red when an editing operation makes at least one predicate evaluate to false.

In that case, clicking on the button makes a more detailed error report available, as shown in Fig. 43b. This dialog shows the predicates violated (by their clear-text predicate message), and also the vertices or arcs where they failed (clicking on the corresponding red dot would select the culprit).

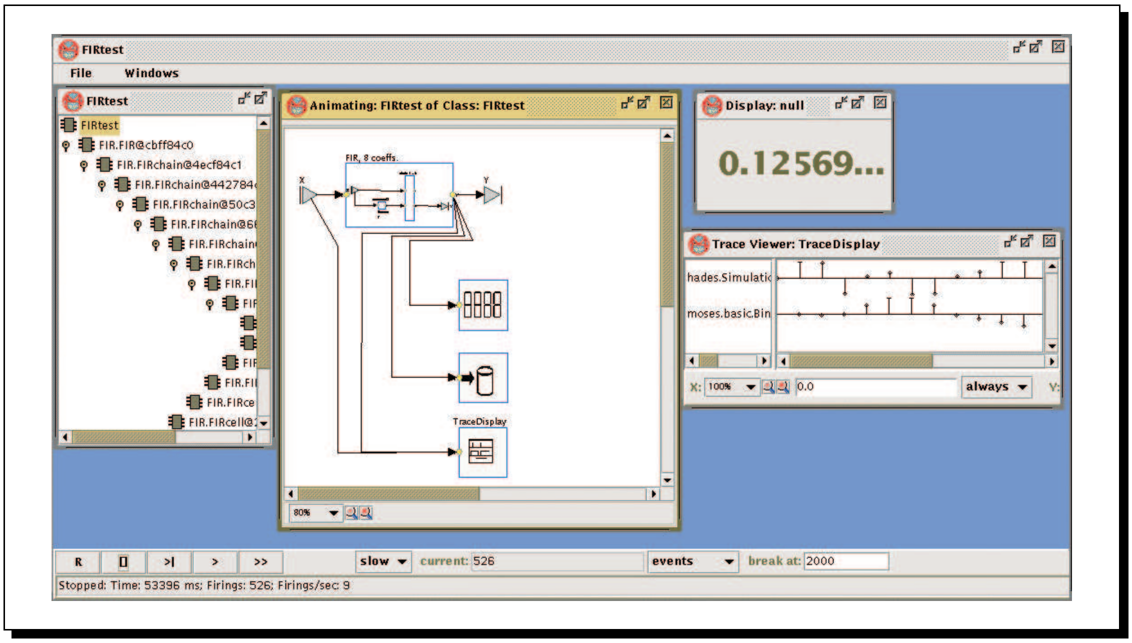

Fig. 44: The simulation environment.

E.3 Execution and animation

Once a graph has been entered and found to be syntactically correct, it can be compiled into executable code and then executed. Although a model may be executed as a standalone program, the Moses Tool Suite provides an environment that facilitates stepwise execution, animation, and inspection of the model state. The interface of this environment is shown in Fig. 44.

The main window features a set of controls at its bottom and a number of internal windows. The controls are used to run and stop the model, a given number of steps or until a certain point in (virtual) time, with or without animation etc. The internal window on the left depicts the containment hierarchy of the model—this is dynamic, so the corresponding tree view may change during the execution of the model.

The big central window shows a top-level view of the model itself—a similar window may be opened for any component in the containment hierarchy. One can also see 'into' contained components by making their container 'transparent' (as shown for the topmost component in that window). Other internal windows may show aspects of the state of the model, show a history of states or output tokens produced etc.

E.4 Sample GTDL Specification

```
graph type SimplePetriNet{

    vertex type Place(string Name, integer InitialTokens)
    graphics(hidden string Shape = "Oval",
             hidden color Color = "black",
             hidden color FillColor = "yellow",
             hidden integer ExtentX = 24, hidden integer ExtentY = 24).

    vertex type Transition(string Name)
    graphics(hidden string Shape = "Rectangle",
             hidden color Color = "black",
             hidden color FillColor = "orange",
             hidden integer ExtentX = 24, hidden integer ExtentY = 24).

    edge type Arc(string Name, integer Weight)
    graphics(hidden string Head = "ClosedTriangle",
             hidden color Color = "black",
             hidden color FillColor = "black").

    predicate PlaceTransitionConnection
             "Arcs from places must end at transitions."
             forall e in Arc :
                    if src(e) in Place then
                      dst(e) in Transition
                    else true end
             end

    predicate TransitionPlaceConnection
             "Arcs from transitions must end at places."
             forall e in Arc :
                    if src(e) in Transition then
                      dst(e) in Place
                    else true end
             end

    predicate WeightsPositive
             "Weights must be positive."
             forall e in Arc :
                    if e("Weight") <> null then e("Weight") > 0
                    else true end
             end
```

predicate $NonEmptyPreset$
$"Transitions\ must\ have\ a\ non-empty\ preset."$
forall t **in** $Transition:$
exists e **in** $Arc:\ dst(e) = t$ **end**
end

const $Compiler = "moses.models.translator.simplepn.PNCompiler".$
}

Bibliography

[1] The Moses Project. Computer Engineering and Communications Laboratory, ETH Zurich ($http : //www.tik.ee.ethz.ch/ \sim moses$).

[2] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1999.

[3] S. Abramsky, S. J. Gay, and R. Nagarayan. Interaction categories and the foundations of typed concurrent programming. In M. Broy, editor, *Deductive Program Design: Proc. 1994 Marktoberdorf International Summer School*, NATO ASI Series F. Springer-Verlag, 1995.

[4] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, 1986.

[5] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1993.

[6] R. M. Amadio. Translating core facile. Technical Report ECRC-1994-3, European Computer-Industry Research Centre, 1994.

[7] M. Andries, G. Engels, and J. Rekers. How to represent a visual program? In *Proc. International Workshop on Theory of Visual Languages, Gubbio, Italy*, 1996.

[8] Matthias Anlauff. XASM—an extensible, component-based abstract state machines language. In Yuri Gurevich, Phillipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines — ASM 2000*, volume 87 of *TIK-Report*. ETH Zurich, 2000.

[9] G. Arango. Domain analysis: From art form to engineering discipline. In *Fifth International Workshop on Software Specification and Design*, volume 14 of *ACM SIGSOFT Engineering Notes*, pages 152–159, May 1989.

[10] J. Armstrong, Virding R, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[11] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In *Handbook on Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[12] Roswitha Bardohl and Gabriele Taentzer. Defining visual languages by algebraic specification techniques and graph grammars. In *TVL87*, 1997.

[13] H. P. Barendregt. *The Lambda Calculus—its Syntax and Semantics*. North-Holland, 2nd edition, 1984.

[14] D. Bèauquier and A. Slissenko. The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *LNCS*, pages 201–212. Springer, 1997.

[15] C. Beierle, E. Börger, I. Durdanovic, U. Glässer, and E. Riccobene. Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in LNCS, pages 62–78. Springer, 1996.

[16] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proc. IEEE*, 79(9):1270–1282, 1991.

[17] Alan F. Blackwell. Metacognitive theories of visual programming: What do we think we are doing? In *VL96*, pages 240–246, 1996.

[18] A. Blass and Y. Gurevich. The Linear Time Hierarchy Theorems for Abstract State Machines. *Journal of Universal Computer Science*, 3(4):247–278, 1997.

[19] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer, 1990.

[20] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer, 1990.

[21] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: Simple Mathematical Interpreters. In U. Montanari and E. R. Olderog, editors, *Proc. PROCOMET'94 (IFIP*

Working Conference on Programming Concepts, Methods and Calculi), pages 489–508. North-Holland, 1994.

[22] E. Börger, U. Glässer, and W. Müller. The Semantics of Behavioral VHDL'93 Descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE CS Press.

[23] E. Börger and L. Mearelli. Integrating ASMs into the Software Development Life Cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997.

[24] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[25] E. Börger and R. Salamone. CLAM Specification for Provably Correct Compilation of CLP($\mathcal{R}$) Programs. In E. Börger, editor, *Specification and Validation Methods*, pages 97–130. Oxford University Press, 1995.

[26] E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1998.

[27] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[28] P. Bottoni, M. F. Costabile, S. Levialdi, and P. Mussio. Formalising visual languages. In *Proc. 11th IEEE Symposium on Visual Languages*, pages 45–52, 1995.

[29] Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinements of Petri nets. In *Advances in Petri Nets*.

[30] Manfred Broy. Advanced component interface specification. In Takaysau Ito and Akinori Yonezawa, editors, *International Workshop on Theory and Practice of Parallel Programming*. Springer-Verlag, 1995.

[31] Manfred Broy. Towards a mathematical concept of a component and its use. In *Proceedings Components' Users Conference CUC'96*, 1996.

[32] Manfred Broy and Gheorghe Stefanescu. The algebra of stream processing functions. Technical Report TUM-I9620, TU München, Institut für Informatik, May 1996. SFB-Bericht 342/11/96 A.

[33] D. Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In S. Kamin, editor, *DSL97*, pages 17–35, 1998.

[34] Margaret M. Burnett and Marla J. Baker. A classification system for visual programming languages. *JVLC*, 5:287–300, 1994.

[35] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation, Systems Research Center, 1994.

[36] Noam Chomsky. *Aspects of the Theory of Syntax*. MIT Press, 1965.

[37] W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In *IEEE Symp. on Visual Languages*, pages 208–215, 1995.

[38] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[39] J. B. Dennis. First version data flow procedure language. Technical Memo MAC TM 61, MIT Lab. Comp. Sci., May 1975.

[40] S. Dexter, P. Doyle, and Y. Gurevich. Gurevich Abstract State Machines and Schönhage Storage Modification Machines. *Journal of Universal Computer Science*, 3(4):279–303, 1997.

[41] Martin Erwig. Abstract visual syntax. In *Proc. 13th IEEE Symposium on Visual Languages*, pages 15–25, 1997.

[42] Martin Erwig. Semantics of visual languages. In *Proc. 13th IEEE Symposium on Visual Languages*, pages 300–307, 1997.

[43] Martin Erwig. Abstract syntax and semantics of visual languages. *Journal of Visual Languages and Computing*, 9:461–483, 1998.

[44] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, ETH Zurich, 1996.

[45] Robert Esser and Jörn W. Janneck. Exploratory performance evaluation using dynamic and parametric Petri nets. In Adrian Tentner, editor, *Proceedings of the HPC 2000*, pages 357–364. Society for Computer Simulation, April 2000.

[46] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1995.

[47] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[48] U. Glässer. Combining Abstract State Machines with Predicate Transition Nets. In F. Pichler and R. Moreno-Díaz, editors, *Computer Aided Systems Theory–EUROCAST'97 (Proc. of the 6th International Workshop on Computer Aided Systems Theory, Las Palmas de Gran Canaria,*

Spain, Feb. 1997), volume 1333 of *LNCS*, pages 108–122. Springer, 1997.

[49] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[50] P. Glavan and D. Rosenzweig. Communicating Evolving Algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 182–215. Springer, 1993.

[51] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[52] Y. Gurevich. Evolving Algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 423–427, Elsevier, Amsterdam, the Netherlands, 1994.

[53] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[54] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[55] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.

[56] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 295–328. Oxford University Press, 1995.

[57] Y. Gurevich and L. Moss. Algebraic Operational Semantics and Occam. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'89, 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 176–192. Springer, 1990.

[58] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[59] David Harel. On visual formalisms. *Communications of the ACM*, 11(5):514–530, 1988.

[60] R. Helm and Kim Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2:311–331, 1991.

[61] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[62] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[63] J. Huggins. Broy-Lamport Specification Problem: A Gurevich Abstract State Machine Solution. Technical Report CSE-TR-320-96, EECS Dept., University of Michigan, 1996.

[64] Charles E. Hughes. The equivalence of vector addition systems to a subclass of post canonical systems. *Information Processing Letters*, 7(4):201–204, June 1978.

[65] Jörn W. Janneck. Compositional Petri net structures. Technical Report 60, Computer Engineering and Networks Laboratory, ETH Zürich, November 1998.

[66] Jörn W. Janneck and Phillipp Kutter. Mapping Automata: Simple Abstract State Machines. TIK-Report 49, Swiss Federal Institute of Technology (ETH) Zurich, June 1998.

[67] Jörn W. Janneck and Phillipp Kutter. Object-based Abstract State Machines. TIK-Report 47, Swiss Federal Institute of Technology (ETH) Zurich, 1998.

[68] Jörn W. Janneck and Martin Naedele. Modeling hierarchical and recursive structures using parametric Petri nets. In Adrian Tentner, editor, *Proceedings of the HPC '99*, pages 445–452. Society for Computer Simulation, 1999.

[69] M. Jantzen. The large markings problem. *Petri Net Newsletter*, (14), 1983.

[70] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts of *EATCS Monographs in Computer Science*. Springer-Verlag, 1992.

[71] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*. North-Holland Publishing Co., 1974.

[72] P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

[73] P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[74] Charles A. Lakos. From coloured Petri nets to object Petri nets. In *Proceedings of the 15th International Conference on Applications and Theory of Petri Nets*, Lecture Notes in Computer Science, 1995.

[75] Leslie Lamport. Time, clocks, and the ordering of events in a dsitributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[76] F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics*. Cambridge University Press, 1998.

[77] Edward A. Lee. A denotational semantics for dataflow with firing. Technical Report UCB/ERL M97/3, EECS, University of California at Berkeley, January 1997.

[78] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[79] Kim Marriott, Bernd Meyer, and K. Wittenburg. A survey of visual language specification and recognition. In Kim Marriott and Bernd Meyer, editors, *Visual Language Theory*, pages 5–85. Springer Verlag, 1998.

[80] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Transactions on Communications*, 24(9), September 1976.

[81] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[82] R. Milner, J. G. Parrow, and D. J. Walker. A calculus of mobile processes (parts i and ii). Technical Report ECS-LFCS-89-85, -86, University of Edinburgh, 1989.

[83] J. Morris. *Algebraic Operational Semantics and Modula-2*. PhD thesis, University of Michigan, Ann Arbor, Michigan, 1988.

[84] Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[85] Benjamin C. Pierce. *Basic category theory for computer scientists*. MIT Press, 1998.

[86] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, 1995.

[87] A. Poetzsch-Heffter. Prototyping Realistic Programming Languages Based On Formal Specifications . *Acta Informatica* , 34:737–772, 1997.

[88] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[89] Wolfgang Reisig. *Elements of Distributed Algorithms*. Springer Verlag, 1998.

[90] J. Rekers and A. Schürr. A graph grammar approach to graphical parsing. In *Proc. 11th IEEE Symposium on Visual Languages*, pages 195–202, 1995.

[91] J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *Proc. 12th IEEE Symposium on Visual Languages*, pages 148–155, 1996.

[92] Jö rn W. Janneck. Graph-type definition language (gtdl)—specification. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, 2000.

[93] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1998.

[94] H. Sasaki, K. Mizushima, and T. Sasaki. Semantic Validation of VHDL-AMS by an Abstract State Machine. In *Proceedings of BMAS'97 (IEEE/VIUF International Workshop on Behavioral Modeling and Simulation)*, pages 61–68, Arlington, VA, October 20-21 1997.

[95] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[96] A. Schönegge. Extending Dynamic Logic for Reasoning about Evolving Algebras. Technical Report 49/95, Universität Karlsruhe, Fakultät für Informatik, 1995.

[97] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[98] D. B. Skillcorn. Stream languages and data-flow. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

[99] D. C. Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser, Basel, Stuttgart, 1977.

[100] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[101] G. Taentzer, C. Ermel, and M. Rudolf. The AGG approach: Language and tool environment. In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[102] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophicae*, 1:261–405, 1936. English translation in A. Tarski. Logic, Semantics, Metamathematics. Oxford University Press.

[103] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[104] H. Tonino and J. Visser. Stepwise Refinement of an Anstract State Machine for WHNF-Reduction of λ-Terms. Technical Report 96-154, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1996.

[105] M. Vale. The Evolving Algebra Semantics of COBOL. Part I: Programs and Control. Technical Report CSE-TR-162-93, EECS Dept., University of Michigan, 1993.

[106] Rüdiger Valk. On processes of object Petri nets. Technical Report 185, Fachbereich Informatik, Universität Hamburg, 1996.

[107] A. van Deursen and P Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–98, 1998.

[108] Walter Vogler. Behaviour preserving refinements of Petri nets. In G. Tinhofer and G. Schmidt, editors, *Graph-Theoretic Concepts in Computer Science*, number 246 in Lecture Notes in Computer Science, pages 82–93. Springer-Verlag, 1987.

[109] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[110] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, 1993.

[111] K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[112] A. Zamulin. Object-oriented Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

[113] Bernhard P. Zeigler. *Theory of Modelling and Simulation*. John Wiley and Sons, 1976.