**Diplomarbeit**

# About
# Expanding Spiking Neural P Systems

Daniel Schröder

Daniel.Schroeder@mailbox.tu-dresden.de

Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
Lehrstuhl für Automatentheorie

Betreuerin: **Dr.-Ing. Monika Sturm**
Verantwortlicher Hochschullehrer: **Prof. Dr.-Ing. Franz Baader**

Eingereicht am 8. Februar 2011

*"The creation of something new is not accomplished by the intellect but by the play instinct acting from inner necessity. The creative mind plays with the objects it loves."*

– Carl Jung (1875 - 1961)

Technische Universität Dresden 02.08.2010
Institut für Theoretische Informatik
Lehrstuhl für Automatentheorie

## Aufgabenstellung Diplomthema

## Bearbeiter: Daniel Schröder (Informatik)

Auf dem Gebiet des Membrane-Computing haben sich in den letzten Jahren die so genannten SNP-Systeme (*Spiking Neural P-Systeme*) wissenschaftlich etabliert. Die Besonderheit dieser Systeme besteht darin, nicht nur Zellhierarchien zu betrachten, wie das üblicherweise bei P-Systemen gemacht wird, sondern auch andere Membranorganisationen mittels P-Systeme abzubilden. Ein Vorschlag ist dabei, P-Systeme zu modellieren, in denen die Membranen netzförmige Strukturen bilden. In der Theorie der Spiking Neural P-Systeme wird als Netz ein Neuronennetz verwendet. In der Diplomarbeit ist die Modellierung der SNP-Systeme vorzustellen und ein bisher offenes Problem aus der aktuellen Forschung zu lösen. Letztere Aufgabe soll den eigenen wissenschaftlichen Beitrag in der Diplomarbeit darstellen.

Auf folgende Punkte soll in der Arbeit eingegangen werden:

- Definition der Spiking Neural P-Systeme,
- Erweiterung der Modellierung mit der Zielstellung, einen exponentiell großen Space durch das Modell selbst aufzubauen,
- Vergleich mit anderen bekannten Ansätzen,
- Modellierung einer konkreten Anwendung (z. B. Entscheidung eines NP- oder PSPACE-Problems),
- Zusammenfassung wissenschaftlicher Fragestellungen auf diesem Gebiet.

Die Bearbeitung des Themas setzt ein intensives Literaturstudium voraus, insbesondere folgender Quellen:

- Alberto Leporati, Claudio Zandron, Claudio Ferretti and Giancarlo Mauri. *Solving Numerical NP-Complete Problems with Spiking Neural P Systems*. Membrane Computing, Lecture Notes in Computer Science, Volume 4860, 336-352, 2007.
- Gheorghe Păun. *Spike trains in spiking neural P systems*. International Journal of Foundations of Computer Science, 17(4):975, 2006.
- Haiming Chen, Mihai Ionescu, Tseren-Onolt Ishdorj, Andrei Păun, Gheorghe Păun and Mario J. Pérez-Jiménez. *Spiking neural P systems with extended rules: universality and languages*. Natural Computing, Volume 7, Number 2, 147-166, 2008.

Organisation und Termine:
Betreuerin: Dr.-Ing. Monika Sturm
Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. Franz Baader
Bearbeitungszeitraum: 02.08.2010–02.02.2011
Zwischenbericht: 10/2010

## Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Zuhilfenahme der angegebenen Literatur verfasst habe.

Daniel Schröder
Dresden, 8. Februar 2011

# Contents

# 1. Introduction

The scientific approach of studying information and data processing found in nature to gain new insights into the nature of computing and computing problems is known as *Natural computing*. Based on the results, new nature inspired computing models, computing paradigms or problem-solving techniques are found, which may have benefits over conventional methods or even solve conventionally-hard problems efficiently. Some well known natural computing based fields of research include *evolutionary algorithms*, *neural networks*, *DNA computing* and *membrane computing*.

Most natural computing based models fall into one of the following three categories:

**In-vivo models** characterize processes in living organisms. For example the process of reordering of DNA strands in living cells. Another example are the cell divisions based ESN P systems introduced in this diploma thesis.

**In-vitro models** characterize natural processes applied outside of living organisms. For example, in-vitro models are used in DNA computing, where DNA molecules are manipulated in test tubes to perform conventionally-hard computations efficiently.

**In-silico models/algorithms** target conventional computer architectures. Natural processes are either simulated or adapted. Evolutionary algorithms are an example where the natural process of evolution is adapted to solve optimization problems.

*Membrane computing* is a natural computing based field of computer science. In membrane computing in-vivo computation models based on processes in biological cells are developed. The most important model is the *P system* conceived by (and named after) Gheorghe Păun. P systems were first mentioned in [PRS98]. A P system models a tree structure of membranes (i. e. a single root membrane contains other membranes which in turn may contain further membranes etc.). Rules then define the transport of different types of objects between the membranes of the system or even into the environment. This is inspired by object transport processes occurring in biological cells.

Based on P systems, the *spiking neural P systems* (in short, *SN P systems*) were introduced as a new in-vivo membrane computing model in [IPY06]. SN P systems adapt the tree-like hierarchy of membranes in P systems to a net-like structure of membranes—now representing neurons—inspired by the information processing of the brain. Unlike object transport in P systems, only one type of objects called *spike*—representing one unit of information—is transported between neurons of SN P systems. This as well is inspired by the brain in which all electrical impulses transmitted by synapses between neurons are almost identical. SN P systems can be utilized in various ways as number generating devices, accepting devices or even function computing devices as for example shown in [IPY06] and [PP07]. In all three cases, SN P systems were found to be computationally complete.

Since neurons of SN P systems work in parallel to a high degree, the usage of these SN P systems to efficiently solve conventionally-hard problems is the focal point of current research. Until now, two possible approaches were published. On the one hand, nondeterministic SN P systems are used. On the other hand, deterministic SN P systems are used, which, however, depend on a very large pre-compiled workspace (i. e. a very large number of neurons and synapses given before computation starts). Efficient solutions to the well known problems `SAT` and `Subset Sum` (which are **NP**-complete) using the nondeterministic approach were introduced in [LMZ$^+$09]. Deterministic systems efficiently solving `SAT` are shown in [IL08] and will also be covered in more detail in Chapter 3 of this diploma thesis. Furthermore, a deterministic SN P system efficiently solving the **PSPACE**-complete problem `QBF` is shown in [ILP$^+$10]. Although these deterministic systems solve the problems in a polynomial or even constant number of steps—depending on how the problem instance is injected into the system—the required pre-compiled workspace needs to be exponentially large with respect to the problem size. In fact, a deterministic SN P system cannot solve an **NP**-hard problem efficiently unless it employs an exponentially large pre-compiled workspace as proved in [Nea08] (assuming **NP** $\neq$ **P**).

Unfortunately, the dependency on exponentially large pre-compiled workspaces is undesirable. The reason is that the question of if and how such a workspace can be built efficiently in time is not answered by the system itself. However, pre-compiled exponential workspaces can be avoided by extending the standard SN P system. Although it is unavoidable to eventually generate an exponentially large workspace—which is necessary to solve the problem efficiently in time—, the workspace can be build during computation by the system itself, starting from a non-exponential workspace. The basic idea to achieve this is to add the ability to grow new neurons and synapses in a structured way during the actual computation, using the high degree of parallelism in SN P systems. In this case, the resulting exponential workspace is indeed not pre-compiled anymore.

One such extension is developed in [WIP09]. There, additional *budding rules* allow the creation of new synapses and neurons based on the structural environment of a neuron but independent of spikes contained in the system. In this diploma thesis an alternative extension to SN P systems is proposed, which allows to create neurons as well as synapses based on the amount of spikes contained in a neuron. The proposed extension introduces two additional types of rules: *expanding rules* create new neurons by dividing existing neurons, whereas *connecting rules* create synapses between existing neurons. The two new types of rules are biologically inspired by cell division and dendritic growth. SN P systems using these additional rules are named *expanding spiking neural P systems* (in short, *ESN P systems*). Although their primary purpose is the same as with the budding rule extension—i. e. avoiding exponentially large pre-compiled workspaces—, we are confident that this extension is also of interest for other essential applications.

To show the important properties of this new extension, an application is shown: A deterministic ESN P system capable of efficiently (in time) solving the **PSPACE**-complete problem `QBF`. To achieve that, the system builds an exponentially large workspace in only a polynomial number of steps. Further good properties distinguishing this system from other such systems are the just polynomially large initial configuration (this includes the number of neurons, spikes and synapses) and the polynomial number of rules the system uses.

The remainder of this diploma thesis is structured in the following way: In Chapter 2, fundamental definitions are recalled. SN P systems are introduced and formally defined in the subsequent Chapter 3. This includes an overview of previous findings as well as current research in this field. Furthermore, an application of SN P systems solving the `QBF` problem is given. This system is the foundation for the following chapters. Chapter 4 introduces and formally defines the proposed extension to SN P systems, which is then applied in Chapter 5 to solve `QBF` with better properties. In Chapter 6, similar approaches are introduced and compared to the extension proposed in this diploma thesis. Finally, the relevance of this solution is discussed in Chapter 7.

# 2. Preliminaries

This chapter briefly recalls basic mathematical definitions as well as definitions for formal languages, regular expressions and the `quantified Boolean formula problem`. The reader is assumed to have basic knowledge of complexity theory and furthermore to be familiar with propositional logic and P systems. An introduction to complexity theory can be found in [WP10] and an introduction to propositional logic in [Sch10]. Finally, P systems are introduced in [PRS98] and [Pau02].

## 2.1. Mathematical Prerequisites

The usual definitions of natural numbers $\mathbb{N}$ (including 0), sets, relations and functions, as used for example in [Ihr98], are assumed in this diploma thesis. $\mathbb{N}_{\geq i}$ denotes the set of all natural numbers greater or equal to $i \in \mathbb{N}$. The composition of two relations (denoted by $\circ$), as used in this diploma thesis, is defined by:

**Definition 2.1.** If $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ are two binary relations, then their *composition* $S \circ R$ is the relation

$$S \circ R = \{(x, z) \in X \times Z \mid \exists y \in Y \colon (x, y) \in R \wedge (y, z) \in S\} \subseteq X \times Z \ . \qquad \diamond$$

## 2.2. Formal Languages and Regular Expressions

The formal languages and regular expressions used in this diploma thesis are defined as follows:

**Definition 2.2.** An *alphabet* is a finite nonempty set. $\diamond$

**Definition 2.3.** Let $\Sigma$ be an alphabet. Then the set $\Sigma^*$ of all *words* over $\Sigma$ is defined as the set of all finite sequences (or *strings*) over $\Sigma$. Words are denoted by listing the elements of the sequence. The length of a word $w = a_1 a_2 \cdots a_k$ with $k \geq 0$, denoted by $|w|$, is defined as $|w| := k$. The empty word is the word of length $0$ and is denoted by $\varepsilon$. $\diamond$

**Definition 2.4.** Let $\Sigma$ be an alphabet. A *formal language* $L$ over $\Sigma$ is a subset of $\Sigma^*$, i. e. a set of words over $\Sigma$. $\diamond$

**Definition 2.5.** Let $\Sigma$ be an alphabet and $L_1$, $L_2$ two formal languages over $\Sigma$. Then the two operations *concatenation* (denoted by $\cdot$) and *Kleene star* (denoted by $^*$), are defined as follows:

$$L_1 \cdot L_2 := \{u \cdot v \mid (u \in L_1) \wedge (v \in L_2)\} \quad \text{where} \quad u \cdot v := uv \ ,$$

$$\begin{aligned} L^0 &:= \{\varepsilon\} \ , \\ L^{n+1} &:= L^n \cdot L \ , \\ L^* &:= \bigcup_{n \geq 0} L^n \ . \end{aligned}$$ ◇

**Definition 2.6.** Let $\Sigma$ be an alphabet. The set of *regular expressions* over $\Sigma$, denoted by $Reg_\Sigma$, is the smallest set $Reg'_\Sigma$ such that:

- $\{\emptyset, \varepsilon\} \subseteq Reg'_\Sigma$ ,
- $\Sigma \subseteq Reg'_\Sigma$ ,
- $\forall n \in \mathbb{N}_{\geq 0} \colon \forall r, s \in Reg'_\Sigma \colon (r+s), (r \cdot s), r^*, r^+, r^n \in Reg'_\Sigma$ . ◇

Priorities for the operators of regular expressions are defined as follows: $\cdot^*$, $\cdot^+$ and $\cdot^n$ have the highest priority, followed by $(\cdot)$. The operator $(+)$ has the lowest priority. Considering these priorities, unnecessary parentheses may be omitted. The operator $(\cdot)$ may be omitted as well. For example, the regular expression $((x \cdot y) + (a^2 \cdot b))$ can also be written as $xy + a^2b$. Note that conventional definitions of $Reg_\Sigma$ do not include the operators $\cdot^+$ and $\cdot^n$. In fact, expressions using these operators can equivalently be substituted by expressions using the standard operators (see Definition 2.7). However, the additional operators allow more compact expressions and are therefore used throughout this diploma thesis.

**Definition 2.7.** Let $\Sigma$ be an alphabet, $r, s, t \in Reg_\Sigma$, $a \in \Sigma$ and $n \in \mathbb{N}_{\geq 0}$. The formal language defined by the regular expression $t$, denoted by $\mathrm{L}(t)$, is defined inductively:

- $\mathrm{L}(\emptyset) := \emptyset, \quad \mathrm{L}(\varepsilon) := \{\varepsilon\}, \quad \mathrm{L}(a) := \{a\}$ ,
- $\mathrm{L}(r+s) := \mathrm{L}(r) \cup \mathrm{L}(s), \quad \mathrm{L}(r \cdot s) := \mathrm{L}(r) \cdot \mathrm{L}(s)$ ,
- $\mathrm{L}(r^*) := \mathrm{L}(r)^*, \quad \mathrm{L}(r^+) := \mathrm{L}(r \cdot r^*)$ ,
- $\mathrm{L}(r^n) := \mathrm{L}(\underbrace{r \cdot r \cdot \ldots \cdot r}_{n \text{ times}})$ . ◇

## 2.3. The Quantified Boolean Formula Problem

The well known **PSPACE**-complete `quantified Boolean formula problem` (`QBF`) asks whether a given fully quantified Boolean formula evaluates to `true` or `false`. Without loss of generality, we assume that those formulas are given in conjunctive normal form (CNF) and with no literal occurring twice within the same clause. Furthermore, we assume that a literal $x$ and its complement $\neg x$ never appear in the same clause (e. g. $x \vee \neg x$). Therefore, every formula with $n$ variables and $m$ clauses is of the form $\gamma_{n,m} = Q_1 x_1 Q_2 x_2 \dots Q_n x_n (C_1 \wedge C_2 \wedge \dots \wedge C_m)$ where $n \in \mathbb{N}_{\geq 0}$, $m \in \mathbb{N}_{\geq 1}$, $Q_1, \dots, Q_n \in \{\exists, \forall\}$ and where the clauses $C_1, \dots, C_m$ are disjunctions of literals using the variables $x_1, \dots, x_n$. $QBF(n,m)$ denotes the set of all problem instances (formulas) of `QBF` consisting of $n$ variables and $m$ clauses in the presented form. For each instance $\gamma_{n,m}$ in $QBF(n,m)$ there exists an equivalent formula of the form $\gamma'_{2n,m} = \forall x'_1 \dots \forall x'_n \exists x'_{n+1} \dots \exists x'_{2n} (C'_1 \wedge C'_2 \wedge \dots \wedge C'_m)$ where for all $j \in \{1, \dots, m\}$, the clause $C'_j$ is obtained from $C_j$ by replacing $x_i$ by $x'_i$ if $Q_i = \forall$ or by $x'_{i+n}$ if $Q_i = \exists$ for all $i \in \{1, \dots, n\}$. For example, an equivalent (of the afore mentioned form) of the formula $\forall x_1 \exists x_2 ((\neg x_1 \vee x_2) \wedge (\neg x_2))$ is $\forall x_1 \forall x_2 \exists x_3 \exists x_4 ((\neg x_1 \vee x_4) \wedge (\neg x_4))$. The set of all instances of this form consisting of $y = 2n$ variables and $m$ clauses is denoted by $QBF_{\mathrm{S}}(y = 2n, m)$. Note that this form always consists of an even amount of variables where the index of a variable fully determines its quantification. As one can easily see, the transformation of any given instance of $QBF(n,m)$ into an equivalent instance of $QBF_{\mathrm{S}}(y = 2n, m)$ is polynomial in time.

# 3. Spiking Neural P Systems

In this chapter spiking neural P systems are covered in detail. Since an extension to this model is proposed in Chapter 4, the term *traditional SN P systems* will be used synonymously for the model, introduced in the current chapter, whenever it is necessary to distinguish between both models. Section 3.6 concludes this chapter by presenting an efficient solution to `QBF`, utilizing an SN P system and a pre-compiled workspace.

SN P systems are derived from P systems. The hierarchy of membranes in P systems is always a tree structure. Membranes contain other membranes which may in turn contain further membranes. Removing this restriction—and therefore allowing arbitrary relations between membranes—leads to spiking neural P systems. These systems no longer model membrane structures but neural networks (consisting of neurons and connecting synapses instead of membranes). Consequently, only a single type of object (called spike) is transported between neurons. These spikes represent the electrical impulses transmitted between biological neurons. Like the P systems they are based on, spiking neural P systems can be utilized to efficiently solve conventionally-hard problems, because of their high degree of parallelism. Spiking neural P systems and possible extensions are defined in numerous publications (first in [IPY06]) and often in slightly different ways. The definition used throughout this diploma thesis is based on [ILP$^+$10] and uses so called extended firing rules[1].

## 3.1. Formal Definition and Representation

In essence, SN P systems consist of a number of *neurons*. Each such neuron contains a number of objects (called *spikes* and represented by the symbol $a$) of the same type and a set of *extended firing rules*. Furthermore, neurons can be connected to each other by directed *synapses*. This structure of an SN P system is formally defined as a tuple in the following way (see [ILP$^+$10]):

**Definition 3.1.** A *spiking neural P system* is a tuple

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out)$$

where

1. $O = \{a\}$ is the alphabet with only one element $a$ called *spike*,

[1] As the name "extended firing rules" suggests, there are other variants using so called "standard rules". Standard rules are a more restricted form of the extended firing rules used throughout this diploma thesis. The exact nature of standard rules is covered in Section 3.5. Furthermore, extended firing rules are part of the traditional SN P systems and should not be confused with the new types of rules introduced in Chapter 4 as an extension of the traditional SN P systems.

2. $\sigma_1, \dots, \sigma_m$ are *neurons* of the form $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$ where
   a) $n_i \geq 0$ is the *initial number of spikes* contained in $\sigma_i$,
   b) $R_i$ is a finite set of *extended firing rules* of the form $E/a^c \to a^p; d$ where $E$ is a regular expression over $O$, and $c \geq 1$, $c \geq p$, $p \geq 0$, $d \geq 0$ ,
3. $syn \subseteq \{1, \dots, m\} \times \{1, \dots, m\}$, with $(i,i) \notin syn$ for every $1 \leq i \leq m$, is the directed graph of *synapses* between neurons, and
4. $in, out \in \{1, \dots, m\}$ indicate the *input neuron* and the *output neuron* of $\Pi$. $\diamond$

As a graphical representation of SN P systems, neurons are drawn as nodes of a graph and synapses as directed edges of the same graph. Rules and spikes are listed inside of the neuron shapes. To indicate input and output neurons, "pseudo edges" are used. Each of them connects one neuron with the environment—rather than connecting two neurons as normal edges do—as shown in Figure 3.1.

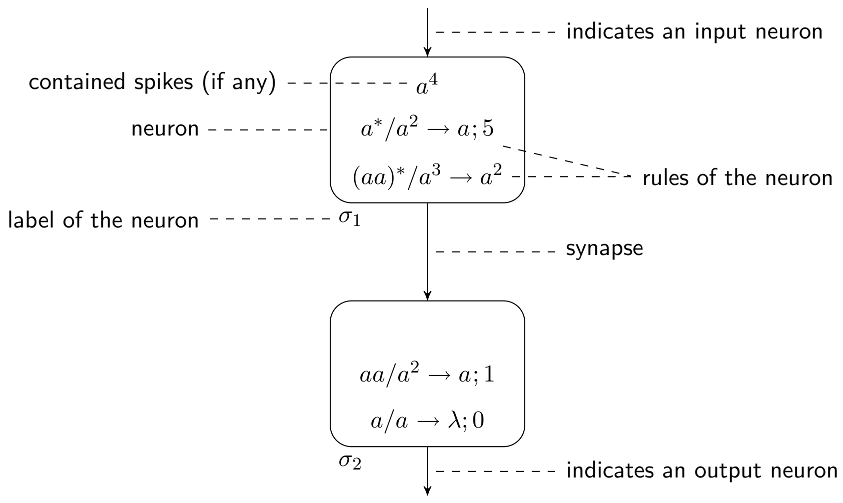


**Figure 3.1.:** *Graphical representation of an SN P system.*

## 3.2. Operation

The dynamic of an SN P system lies in the amount of spikes and their distribution over the neurons of the system. By application of extended firing rules, this distribution is altered. The exact syntax of these rules is simple: $E/a^c \to a^p; d$. Table 3.1 describes the meaning of the four elements of a rule. Under certain conditions, rules may be abbreviated: A regular expression equivalent to $a^c$ can be omitted as well as a delay of 0. Thus, $a^c \to a^p$ is equivalent to $a^c/a^c \to a^p; 0$. Furthermore, rules with $p = 0$ are additionally also called *extended forgetting rules* and may as well be written in the form $E/a^c \to \lambda; d$.

A rule may only be applied on a neuron if the following three conditions are met:

| Elements of the rule $E/a^c \to a^p; d$ | Description |
|---|---|
| $E$ | a regular expression over the symbol $a$ |
| $c$ | the number of spikes to be consumed |
| $p$ | the number of spikes to be generated |
| $d$ | the delay |

**Table 3.1.:** *The elements of rules of SN P systems.*

1. The rule is contained in the neuron it is to be applied to.

2. The neuron contains at least as many spikes as the rule specifies to be consumed.

3. The neuron contains an amount of spikes (interpreted as word over $O = \{a\}$), which is in the language, described by the associated regular expression of the rule. For example, a rule with the expression $(aa)^*$ may be applicable on a neuron containing two, four or any other even number of spikes but not for example on a neuron containing three spikes.

As effect of the application of a rule, the amount of spikes given by the rule is *consumed* (subtracted from contained spikes of the neuron). Furthermore, the application generates a number of new spikes. The amount is again given by the rule. These new spikes are not directly stored in the neuron but transmitted along each outward-directed synapse of the neuron to connected neurons. That means each connected neuron receives as many spikes as the applied rule specifies to be generated. Spikes just received and spikes already stored are accumulated in the connected neurons. Although rules consuming no spikes are not possible, there is a subset of rules which consume spikes but do not generate any new spikes in the process. These rules are called *extended forgetting rules* and may use the special syntax shown above.

In SN P systems, the described application of rules always happens synchronously. This means that a global clock is assumed which defines time steps. During each step, every neuron must apply exactly one contained rule, except if no rule is applicable or the neuron is *closed* (closed neurons are explained later on). If at a time more than one rule is applicable, then one of the applicable rules is chosen nondeterministically for application during that time step. Therefore, a system is *deterministic* if and only if for every neuron of the system at any given time at most one of its rules is applicable. Otherwise, it is called *nondeterministic*.

Finally, each rule comprises a *delay* which defines how many time steps are necessary to apply the rule. If an applied rule has an associated delay greater than zero, then the corresponding neuron is *closed* for that many steps and opens again afterwards. While the neuron is closed, no rules are applied, and incoming spikes are ignored instead of stored during time steps. Only at the end of the last closed time step, spikes are consumed and transmitted as specified by the delayed rule to end the application of this rule. A delay of zero causes the neuron to never close while applying the rule. Consequently, spikes are consumed, transmitted and stored in the same time step as the rule is applied. In the following time step, the neuron may already apply a rule again.

By indicating one neuron as *input neuron*, a predefined number of spikes—given as input to the system beforehand—can be stored in this neuron in each time step. This models a stimulation of the system from outside via synapses between the environment and the special input neuron. For example, prior to execution one may define the input of a system to be three spikes in the first time step, two spikes in the next time step and none in all other steps. Such sequences of amounts of spikes are called *spiketrains*. An input of an SN P system is therefore given as spiketrain. Similarly, a neuron indicated as *output neuron* produces an amount of spikes in each time step and therefore a spiketrain as output. To denote a spiketrain, a sequence of spike symbols $a$ per time step is used. Each spike symbol is associated with a number indicating the amount of spikes of this step. In case the amount of spikes is exactly one, the number may be omitted. An example is $a^2a^{10}aaa^0a^0a$. The empty spiketrain is the spiketrain of length zero, denoted by $\varepsilon$. A formal definition of spiketrains is given later in Section 4.2.

The distribution of spikes, the remaining input spiketrain, the so far computed output spiketrain, and the remaining delay of applied but delayed rules define a *configuration* of an SN P system. With each time step (where rules are applied and input/output is calculated as explained above) the system passes from one configuration to another configuration. This is called a *transition* between configurations of the system. The initial distribution of spikes (as defined by the SN P system), a chosen input spiketrain and the empty output spiketrain form an *initial configuration* for the system. A configuration where no further rules can be applied and no input is left is called a *halting configuration*. A sequence of transitions beginning with an initial configuration and reaching a halting configuration—which must be the only halting configuration of the sequence—is called a *computation*. The output spiketrain of the halting configuration is considered as the result of the computation.

## 3.3. An Example SN P System

Figure 3.2 shows an example SN P system. Note that there is no neuron marked as input neuron for the system. This was omitted because the system does not need any input. One may equivalently define any arbitrary neuron as input neuron and use the spiketrain of length zero as input. An output neuron exists. The interpretation of output spiketrains used in this example is the following: Every spiketrain produced by this system has more than zero spikes only at exactly two different time steps. A spiketrain is interpreted as the number of time steps elapsed between these two time steps. I. e. a natural number $n \in \mathbb{N}_{\geq 1}$. Since the system is nondeterministic (see rules $a^2 \to a^2$ and $a^2 \to a$ in $n_1$), different computations exist, possibly resulting in different spiketrains which in turn may represent different natural numbers. The given system therefore characterizes a set of natural numbers, the set of all numbers represented by possible output spiketrains.

To find the characterized set of natural numbers, consider the possible computations: Initially, only the neurons $n_1$ and $n_4$ contain any spikes (two spikes each). In time step

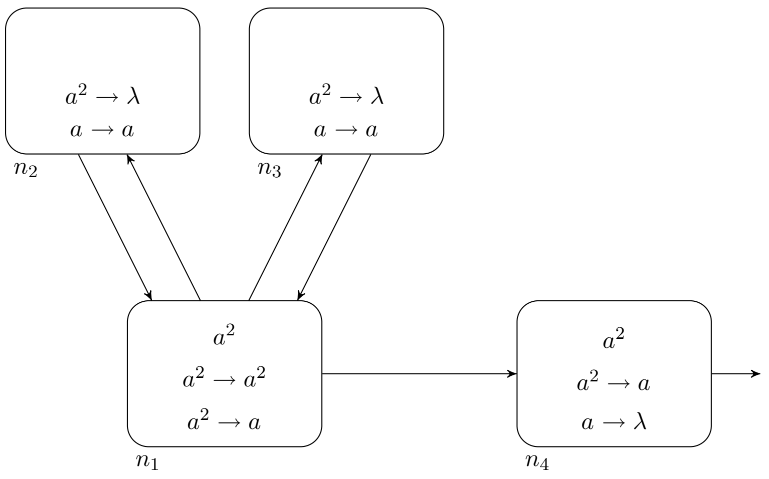


**Figure 3.2.:** *An example SN P system.*

1, the neuron $n_4$ applies the rule $a^2 \to a$. This generates the first spike in the output spiketrain. During the same time step, the neuron $n_1$ decides nondeterministically between applying $a^2 \to a^2$ and $a^2 \to a$. If $a^2 \to a^2$ is applied, the neurons $n_2, n_3$ and $n_4$ receive two spikes each. In the following time step 2, the neurons $n_2$ and $n_3$ delete their spikes, while $n_4$ produces the second spike in the output spiketrain. The computation halts, since no spikes are left in the system. The represented number is $1$, since the two spikes happened at time steps 1 and 2 with exactly one time step elapsed. If instead the rule $a^2 \to a$ is applied for $n_1$ at time step 1, the neurons $n_2, n_3$ and $n_4$ receive only one spike each. The neuron $n_4$ then deletes this spike in the following time step 2. In the same time step, the neurons $n_2$ and $n_3$ apply the rule $a \to a$. Two spikes are received by $n_1$. In time step 3, the neuron $n_1$ decides nondeterministically between applying $a^2 \to a^2$ and $a^2 \to a$ again. As long as $a^2 \to a$ is chosen, the same configuration as prior to time step 3 is reached again two steps later, i. e. at time steps $2m + 1$ for $m \in \mathbb{N}_{\geq 1}$. If the rule $a^2 \to a^2$ is chosen for application by $n_1$ at any time step $2m + 1$ with $m \in \mathbb{N}_{\geq 1}$, the neuron $n_4$ receives two spikes and applies $a^2 \to a$ one time step later ($2m + 2$), creating the second spike of the spiketrain. The system halts and the represented number is $2m + 1$, since $2m + 1$ time steps have elapsed between steps $1$ and $2m + 2$.

Considering that computations for $1$ and for $2m + 1$ for all $m \in \mathbb{N}_{\geq 1}$ are possible, the characterized set is $\{2m + 1 \mid m \in \mathbb{N}_{\geq 0}\}$, i. e. the set of all odd numbers.

## 3.4. Utilization and Computational Completeness

SN P systems can be utilized in various ways, depending on the interpretation of input and output spiketrains. There are four important utilizations which are regularly employed and well understood:

- as *number generating device*,

- as *accepting device*,
- as *language generating device* and
- as *function computing device*.

SN P systems as number generating devices and accepting devices were introduced in [IPY06]. In both cases, spiketrains are interpreted as numbers. The number a spiketrain represents is the number of time steps elapsed between the first two time steps of the spiketrain with more than zero spikes. A number generating device receives no input and produces an output spiketrain per computation which can be interpreted as a number, as explained above. For the same initial configuration of a system, more than one computation is possible (if the system is nondeterministic). The system therefore generates a set of numbers: the set of all numbers represented by the output spiketrains of all possible computations. The example system given in Section 3.3 works in this way. An SN P system as accepting device, on the other hand, takes a number (encoded as spiketrain) as input but produces no output. Instead, the number is accepted if a halting configuration can be reached. Otherwise, the number is not accepted. In both cases, SN P systems are computationally complete. This is proved in [IPY06][2]. The basic idea is to construct an SN P system for an arbitrary given register machine such that the constructed SN P system simulates this register machine. This system is then extended such that the input of the SN P system, encoded as described above, is translated to a register value before the simulation starts, to obtain an accepting device. Alternatively, it is extended such that the value of the output register is encoded as spiketrain after the simulation ends, to obtain a number generating device.

A third utilization of SN P systems is the language generating device. This works similarly to the utilization as number generating device. However, an output spiketrain does not represent a number but a word of the generated language. Each time step of the spiketrain encodes exactly one symbol of the word. This means all possible amounts of spikes in a time step of the output spiketrain are mapped to one symbol of the alphabet. However, as a result of this encoding, the system needs to output one symbol in every time step. This is often referred to as the *restricted case*, since this heavily restricts the languages that can be characterized by such a system. SN P systems in this mode are not computationally complete (see [CII$^+$08] for more details). The introduction of time steps which do not produce a symbol (for example by encoding $\varepsilon$, which is not in the alphabet) leads to a characterization of recursive enumerable languages (proved in [CII$^+$08]). This is referred to as the *unrestricted case*.

To use SN P systems as function computing devices, the parameters of the function are encoded as spiketrain and injected into the system via input neuron. A spiketrain is produced as output and encodes the result. This is similar to a combination of generating and accepting devices. An encoding as spiketrain can be chosen freely. A binary encoding of input parameters is, for example, used in [PP07]. SN P systems

[2]Actually, computational completeness is proven for SN P systems using standard rules (see Section 3.5). However, these standard rules are a subset of the extended rules used for SN P systems in this diploma thesis. Every SN P system using standard rules is therefore an SN P system using extended rules, too. For this reason, the computational completeness result holds for SN P systems using extended rules.

used as function computing devices are still computationally complete. This is easy to see, since all input parameters can be mapped to a single natural number and then used as input for the simulated register machine. By not reaching a halting configuration for certain input spiketrains, partial functions can be characterized. Consequently, the other utilizations are special cases of the utilization as function computing device.

## 3.5. Variants of SN P Systems

Besides the introduced standard variant of SN P Systems, other variants of SN P Systems can be derived by restricting the allowed rules or other properties of the model. It is of interest how far SN P Systems can be restricted while still retaining computational completeness. This section gives a short overview of some notable variants and their properties.

**SN P Systems with Standard Rules:** Standard rules are a more restricted form of the extended firing rules used in this diploma thesis. On the one hand, there are *standard firing rules* (or just *firing rules*) which are almost identical to extended firing rules, except that exactly one spike must be produced by the application of these rules (for instance $(aa)^*/a^4 \to a; 5$). On the other hand, there are *forgetting rules*. These are just differently restricted firing rules. Forgetting rules must always consume all spikes of a neuron and may not produce any new spikes. Furthermore, forgetting rules may not define delays greater than zero. Therefore, the syntax of forgetting rules is always $a^c \to \lambda$. Additionally, firing rules and forgetting rules may never be applicable at the same time. That means, nondeterminism is only allowed within the same type of rule.

In spite of all these restrictions, SN P systems with standard rules are still computationally complete as shown in [IPY06].

As the name suggests, standard rules were used in the very first definitions of SN P systems. Only later, firing rules and forgetting rules were generalized to extended firing rules. While extended firing rules add nothing to the computational power of the model[3], they are much more elegant (as only a single type of rule is necessary) and allow much smaller systems, as shown in [PP07] and [CII$^+$08]. For these reasons, extended firing rules are used throughout this diploma thesis instead of standard rules.

**SN P Systems with Restricted Standard Rules:** As it turns out, standard rules can be restricted even further without loss of computational power. As summarized in [GAPRPS08], any one of the following features may be removed (yet not necessarily combinations of them):

- unbounded indegree of neurons,

[3]However, that is only if the SN P system is utilized as function computing device, since both variants are computationally complete. If used as language generating device, extended rules allow a characterization of all finite languages even in the restricted case. SN P systems with standard rules can not characterize all finite languages in the restricted case. See [CII$^+$08] for further details.

- unbounded outdegree of neurons,
- delays, and
- forgetting rules.

Certain combinations can even be removed simultaneously: for instance delays and forgetting rules.

**Asynchronous SN P Systems:** In standard (synchronized) SN P systems, every neuron applies a rule in every time step (except for neurons which are closed or where no rule is applicable). In asynchronous SN P systems, a neuron with an applicable rule may chose to apply the rule or not at any time step as long as the rule is applicable. Since the neuron still receives spikes a rule may even become inapplicable again without being applied. Such systems are studied in [CEI$^+$08].

If extended rules are used, asynchronous SN P systems are computationally complete (shown in [CEI$^+$08]). However, it remains an open problem whether asynchronous SN P systems using standard rules are computationally complete as well or not.

**Sequential SN P Systems:** A sequential SN P system applies at most one rule per time step. If more than one neuron has one or more applicable rules, one neuron is non-deterministically chosen and then one of its applicable rules is non-deterministically chosen and applied.

Sequential SN P systems are computationally complete using standard rules or using extended rules (see [IW07]).

## 3.6. A Solution to `QBF` Utilizing SN P Systems

In this section, it is shown how SN P systems are applied to solve the `quantified Boolean formula problem`, as introduced in Section 2.3. Since the `QBF` problem is in the focus of this section, the terms "polynomial" and "exponential" relate to the size (which is the number of variables) $n$ of the `QBF` instance that is to be solved, unless stated otherwise. Recall that $m$ denotes the number of clauses in an instance of `QBF` and $y = 2n$ the number of variables of the equivalent instance of $QBF_{\mathrm{S}}(y = 2n, m)$. These variables will be used throughout the section with these meanings.

The solution to `QBF` given in this section is an *exp-uniform solution*. That means a family of SN P systems parametrized by the problem size is defined. For the size of any given problem instance a deterministic Turing machine constructs the corresponding SN P system of the family in exponential time with respect to problem size (this is called pre-compilation). Then a polynomial number of spikes is introduced to the input neuron in form of a spiketrain. Subsequently, The SN P system computes a result in polynomial time.

Since for every instance of $QBF(n, m)$ an equivalent instance of $QBF_{\mathrm{S}}(y = 2n, m)$ can be computed efficiently, a family of SN P systems with parameters $n$ and $m$ efficiently solving instances of $QBF_{\mathrm{S}}(y = 2n, m)$ is sufficient to efficiently solve `QBF`. The general idea for such systems is simple: For each possible assignment of variables of

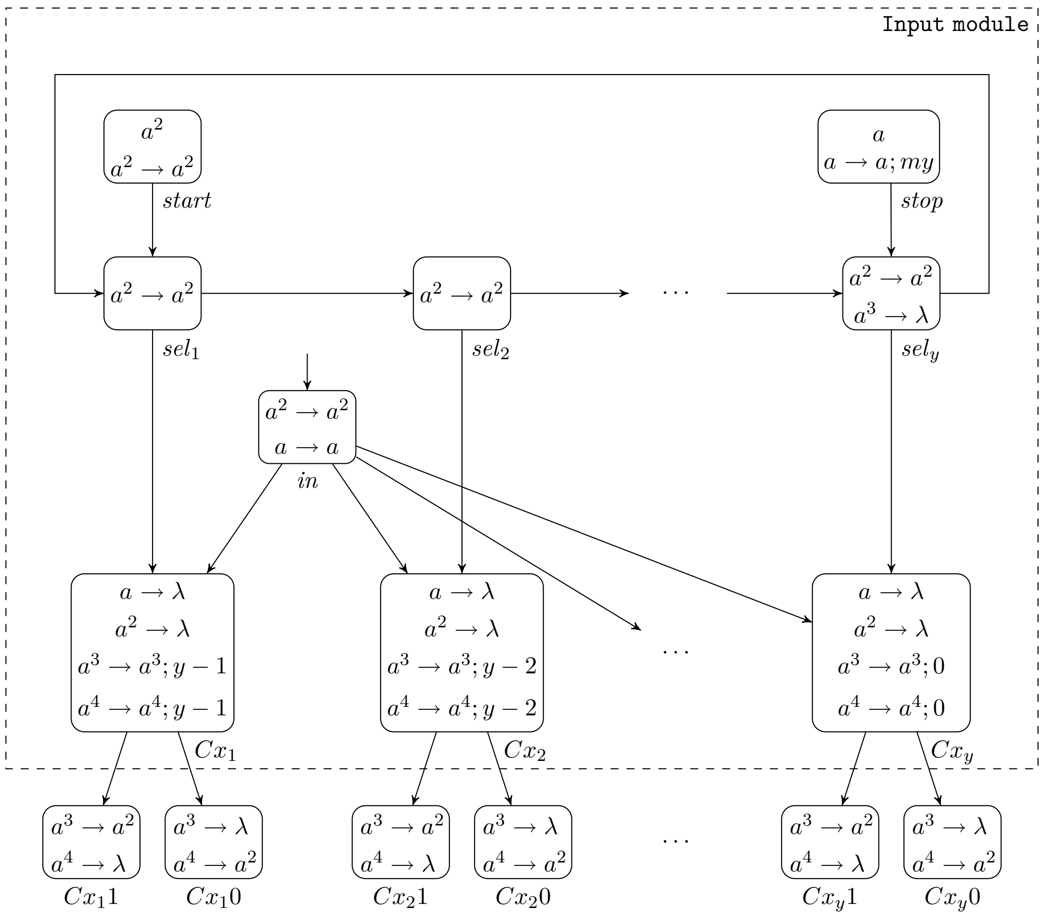


**Figure 3.3.:** *The input module used by the traditional SN P system family in Figure 3.4 to solve instances of $QBF_S(y = 2n, m)$.*

the formula, a dedicated neuron exists within the system to compute the evaluation of the consumed formula (at this point ignoring any quantifications of variables). Further neurons perform logical and-operations on these evaluations to account for ∀-quantified variables. If this finally yields true, then the formula is satisfiable. Since all neurons apply their rules in parallel, this is efficient in time. However, an exponential number of neurons is necessary to represent all variable assignments. Due to the fact that all these neurons and their synapses must be part of the systems specification, this is called an *exponential pre-compiled workspace*.

Figures 3.3 and 3.4 show an implementation of this idea as a family of SN P systems parametrized by the number of variables and clauses. The shown systems are a modification of similar systems introduced in [ILP$^+$10] solving the same problem. The reason for modifying these systems is to use a slightly different form of formulas as input (instances of $QBF_S(y = 2n, m)$ as constructed above), since this is more suitable for later extension.

The systems of that family work as function computing devices. One input neu-

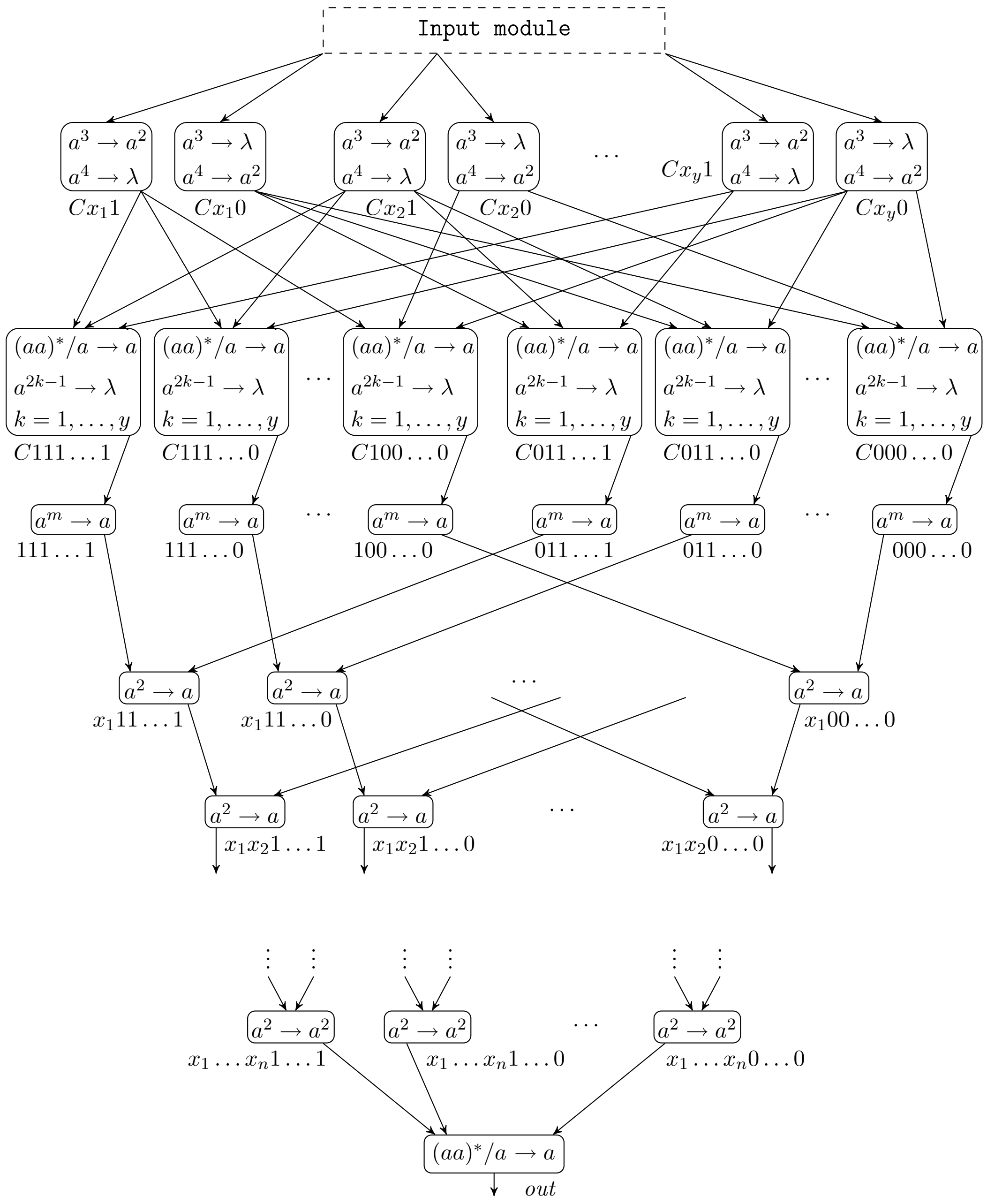


**Figure 3.4.:** *A traditional SN P system family solving instances of $QBF_{\mathrm{S}}(y = 2n, m)$ using a pre-compiled workspace similar to the system given in [ILP+10] including the input module given in Figure 3.3.*

ron is responsible for consuming the spiketrain-encoded formula. A problem instance given as instance of $QBF_\mathrm{S}(y = 2n, m)$ is encoded as input spiketrain by the function $\mathrm{enc}_{n,m}$ in the following way: Quantifications of variables are fixed in instances of $QBF_\mathrm{S}(y = 2n, m)$ and are therefore not encoded at all. All clauses are encoded sequentially in $y$ time steps of the spiketrain per clause—hence $y \cdot m$ time steps are necessary to consume the complete formula. Within each clause all $y$ possible variables of the formula are sequentially encoded by one time step per variable and three possible amounts of spikes transmitted during the step.

- If the variable does not appear in the clause, no spikes are transmitted.
- If the variable occurs as positive literal, one spike is transmitted, and
- if the variable occurs as negative literal (i. e. $\neg x$), two spikes are transmitted.

As an example consider the formula $\forall x_1 \forall x_2 \exists x_3 \exists x_4 ((\neg x_1 \vee x_4) \wedge (\neg x_4))$. This formula has four variables and two clauses. The first clause $(\neg x_1 \vee x_4)$ contains the first variable as negative literal, therefore encoded as $a^2$. The second and third variables do not occur in the clause and are encoded as $a^0$ each. The last variable occurs as positive literal and is encoded as $a^1$. Hence, the spiketrain encoding the complete first clause is $a^2a^0a^0a^1$. Similarly, the second clause is encoded as $a^0a^0a^0a^2$ because it only consists of the fourth variable as negative literal. The complete encoding of the formula is $a^2a^0a^0a^1a^0a^0a^0a^2$.

One output neuron is responsible to publish the result. This output neuron signals the satisfiability by generating a spike at one point in time, if the formula is satisfiable, or never generating a spike, if not.

To give a detailed description of the computation, the neurons of Figure 3.4 are grouped into different layers of neurons. Thereby, each layer is a set of all neurons which are aligned in the same horizontal line of neurons in the graphical representation; e. g. the neurons $C_{x_1}1, \ldots, C_{x_y}0$ form a layer.

The input neuron $in$ consumes the problem instance encoded by $\mathrm{enc}_{n,m}$. The input module processes this in the following way: Every time after consuming the spikes of $y$ time steps—describing one complete clause—the input module sends spikes to the next layer of neurons (the *input bridge neurons* $(Cx_11), (Cx_10), \ldots, (Cx_y1), (Cx_y0)$). This is done in a way so that for every variable $x_i$ the neuron $(Cx_i1)$ will produce two spikes if and only if the literal $x_i$ occurred in the clause; and $(Cx_i0)$ will produce two spikes if and only if the literal $\neg x_i$ occurred. The neurons of this layer fire always simultaneously and at most one time per clause—always one time step after a complete clause is consumed. This represents the formerly serialized clause now within one time step.

Of the neurons $(C11\ldots1), (C11\ldots0), \ldots, (C00\ldots0)$—one layer below—each one represents one of the possible $2^y$ assignments for the $y$ variables. Depending on the assignment they represent, these *assignment neurons* are connected by a synapse to either $(Cx_i1)$ or $(Cx_i0)$ for every variable $x_i$ occurring in the formula. An assignment neuron fires and produces a spike transmitted to the next layer of neurons if it receives two or more spikes in one time step. This is the case if at least one literal of the clause is evaluated to `true` and therefore the whole clause is evaluated to `true` by the represented assignment. For example the assignment neuron $(C01\ldots1)$—which assigns `false` to $x_1$—is connected to $(Cx_10)$ and therefore receives two spikes from it if a clause contains $\neg x_1$—which evaluates to `true` under the represented assignment—but no spikes if the clause contains $x_1$—which would evaluate to `false`—since it is not connected to $(Cx_1 1)$. Note that $(C01 \dots 1)$ would also receive two or more spikes from corresponding neurons if at least one of the other variables occurs as positive literal, since `true` is assigned to all other variables by the represented assignment.

The next layer (denoted by *sat-layer*) consists of $2^y$ *sat neurons*. Each one of them is connected to exactly one of the assignment neurons. The sat neurons accumulate spikes produced by the corresponding assignment neurons. A sat neuron fires when it stores exactly $m$ spikes. This happens if the corresponding assignment neuron fires once for each clause, i. e. $m$ times. That in turn means, the represented assignment satisfies all $m$ clauses and therefore the whole formula (at this point ignoring any quantifications of variables).

Let $n_1$ and $n_2$ be neurons of the sat-layer representing the same assignments except for the first variable $x_1$, which differs in value under these assignments. Let $n_3$ be a neuron performing a logical `and`-operation on $n_1$ and $n_2$; i. e. $n_3$ fires only if $n_1$ and $n_2$ fire both within one time step. It is easy to see that $n_3$ represents an assignment identical to $n_1$ and $n_2$ except for the first variable which is $\forall$-quantified. I. e. $n_3$ fires only if the represented assignment yields `true` for the input formula for both possible assigned values of $x_1$. We call this a $\forall_1$-assignment. The neurons of the first layer below the sat-layer represent all $2^{y-1}$ such $\forall_1$-assignments. Therefore, the layer is denoted by $\forall_1$-layer.

Performing a similar `and`-operation on two neurons of the $\forall_1$-layer—which differ only in the assigned value of the variable $x_2$—yields the $2^{y-2}$ neurons of the $\forall_2$-layer below the $\forall_1$-layer. Naturally, they represent all variable assignments where $x_1$ and $x_2$ are $\forall$-quantified. More $\forall_i$-layer follow for $i \leq n$, constructed in the same way. The last such layer is the $\forall_n$-layer which represents the $2^n$ assignments where the first $n$ variables are $\forall$-quantified.

At this point, the problem is reduced to test whether at least one $\forall_n$-assignment exists such that the formula is evaluated to `true`. The output neuron $out$ simply performs a logical `or`-operation on all $\forall_n$-layer neurons to achieve this.

The two interesting properties of these systems are the maximum number of time steps necessary to solve an instance of $QBF_S(y = 2n, m)$ and the number of neurons necessary for the initial workspace of the system corresponding to the problem instance.

As one can see in Figure 3.3 and Figure 3.4, the number of neurons is independent of the number of clauses. It depends only on the number of variables used by the problem instance. The following neurons are necessary (grouped by modules or layers):

- input module: $2y + 3$ neurons,
- input bridge neurons: $2y$ neurons,
- assignment neurons: $2^y$ neurons,
- sat neurons: $2^y$ neurons,
- $\forall_i$-layer for all $i \in \{1, \dots, n\}$: $2^{2n-1} + 2^{2n-2} + \dots + 2^n = 2^{2n} - 2^n$ neurons, and
- the output neuron: $1$ neuron.

The amount of neurons necessary for the initial workspace is indeed exponential in the number of variables used by the problem instance.

The number of of steps necessary to solve an instance of $QBF_{\mathrm{S}}(y = 2n, m)$ can easily be found. Note that the used SN P systems are designed to work similar to a pipeline. Input is collected in the input module and subsequently forwarded to the next layer. In all following time steps it is then deleted or forwarded one layer further while the next element of the input spiketrain is already "fetched" by the input module. After $m \cdot y$ time steps, the input module receives the last element of the input spiketrain. The last clause of the formula is complete and the input module sends its representation to the input bridge neurons exactly 2 time steps later. There are $2 + n$ layers (assignment, sat, $\forall_1$, . . . , $\forall_n$) as well as the output neuron left to pass. Thus, the output neuron receives the results of the last clause of the problem instance after $m \cdot y + n + 5$ time steps and might fire in the following time step. This implies that the output neuron fires within the first $m \cdot y + n + 6$ time steps or never. A decision is therefore reached in polynomial time.

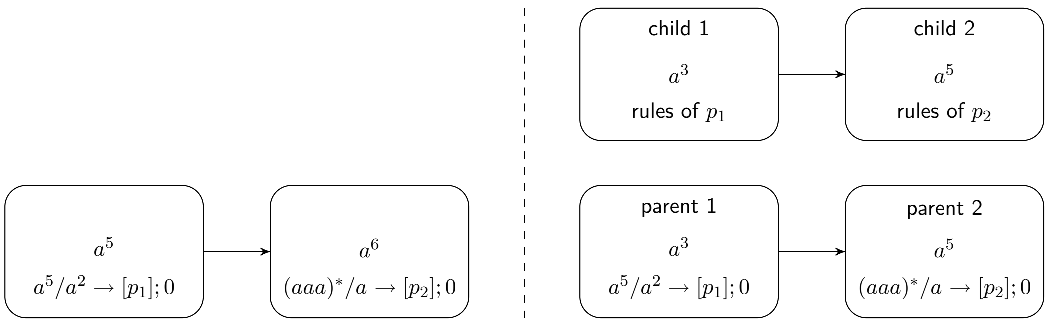
# 4. Expanding Spiking Neural P Systems

## 4.1. Overview

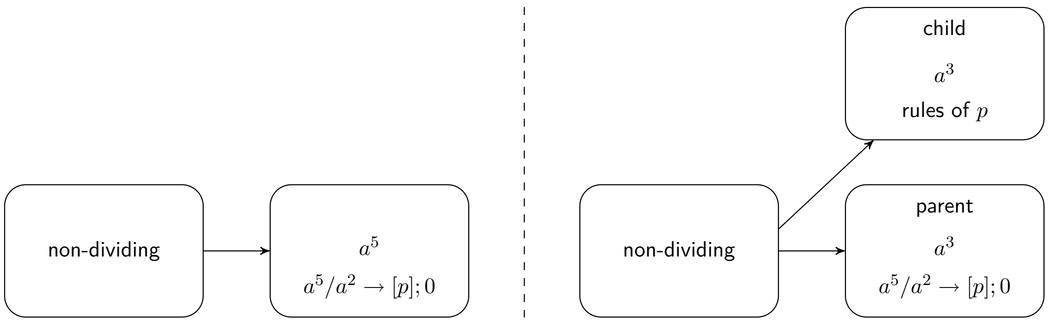
As motivated in the introduction, this diploma thesis aims to extend the traditional spiking neural P systems, shown in the previous section, in order to add capabilities to grow new neurons and synapses during a computation. The extended model will be compatible to the traditional spiking neural P systems—i. e. firing rules (now called spiking rules) are retained and behave like in traditional systems. Therefore, the new capability can be used to efficiently grow a (possibly exponential) workspace first but then proceed the computation like a traditional SN P system afterwards. To achieve such a system, it is sufficient to extend the traditional SN P systems by two new types of rules in addition to the spiking rules. These new types of rules are called *expanding rules* and *connecting rules*. The respective purpose and behavior of the three types of rules is as follows:

**Spiking rules** have the form $E/a^c \rightarrow a^g; d$. If a spiking rule is applied on a neuron (which is therefore the *spiking neuron*), then $c$ spikes from the spiking neuron are consumed while $g$ new spikes are generated and sent to all neurons connected by a synapse originating from the spiking neuron. Therefore, spiking rules are similar to the rules of traditional SN P systems. Note that there is no explicit rule to delete spikes needed. Deletion of spikes can be accomplished by a rule of the form $E/a^c \rightarrow a^0; d$, equivalently written as $E/a^c \rightarrow \lambda; d$, which indeed consumes spikes without creating new spikes.

**Expanding rules** have the form $E/a^c \rightarrow [p]; d$. Such a rule also consumes $c$ spikes, but instead of creating new spikes, a new neuron is created by dividing the neuron that applies the rule. The neuron applying the rule is called *parent neuron*. The newly created neuron is called *child neuron*. Both neurons have the same amount of spikes after the divide, which is the amount of spikes the parent neuron retains after consuming $c$ spikes. The prototype definition associated with $p$—as given by the applied rule—specifies the rules of the child neuron. Furthermore, all incoming and outgoing synapses of the parent neuron are cloned for the child neuron. Since a synapse always connects two neurons, there are two cases to be considered. If both neurons of a synapse apply an expanding rule at the same time (i. e. in the same time step), the synapse between the two parent neurons is reproduced in the same direction between the respective child neurons. Otherwise, if only one of both neurons of a synapse applies an expanding rule, there is a non-dividing neuron, a parent neuron and a child neuron involved. In this case, the synapse between the non-dividing and the parent neuron is reproduced in the same direction between the non-dividing and the child neuron. Figure 4.1 shows



**(a)** *Both neurons are divided. The synapse between the parents is replicated between the children.*



**(b)** *Just one neuron is divided. The synapse between the non-dividing neuron and the parent neuron is replicated between the non-dividing neuron and the child.*

**Figure 4.1.:** *ESN P systems before neuron division on the left side and after, on the right side.*

both cases. If the non-dividing neuron applies a spiking rule in the same time step, spikes are transmitted along both synapses (between the non-dividing neuron and the parent neuron and between the non-dividing neuron and the child neuron), assuming the synapse is directed from the non-dividing neuron to the parent neuron.

**Connecting rules** have the form $E/a^c \to \mathrm{out}; d$ or $E/a^c \to \mathrm{in}; d$. By using this type of rules, synapses connecting already existing neurons can be created. In each step, synapses are created from all neurons applying an *out-connecting rule* to all neurons applying an *in-connecting rule* in the same step as shown in Figure 4.2. Just as with the other types of rules, the application of a connecting rule consumes $c$ spikes.

Generally, rules (regardless of type) are only applicable to a neuron if the amount of spikes stored in the neuron is in the language defined by the regular expression $E$ and is greater or equal to $c$. All three types of rules are delayed if $d \neq 0$. In this case, the neuron is closed for $d$ time steps before the rule is finally applied and the neuron is opened again in this same step. While a neuron is closed, all incoming spikes are ignored and no rule can be applied. To ease the definition of ESN P systems, certain rule abbreviations are possible: $a^1$ can be abbreviated to $a$, $E/$ can be omitted if $E = a^c$, and $; d$ can be omitted if $d = 0$.

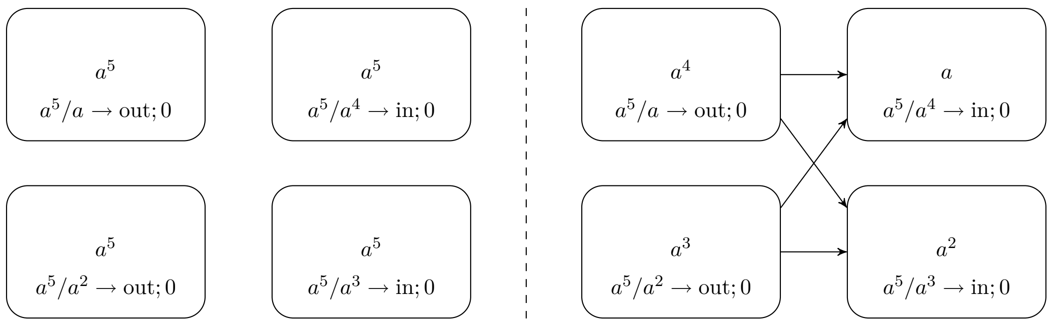

$a^5$
$a^5/a \to \text{out}; 0$
$a^5$
$a^5/a^4 \to \text{in}; 0$
$a^4$
$a^5/a \to \text{out}; 0$
$a$
$a^5/a^4 \to \text{in}; 0$
$a^5$
$a^5/a^2 \to \text{out}; 0$
$a^5$
$a^5/a^3 \to \text{in}; 0$
$a^3$
$a^5/a^2 \to \text{out}; 0$
$a^2$
$a^5/a^3 \to \text{in}; 0$

**Figure 4.2.:** *ESN P system before applying connecting rules (shown on the left side) and after applying (shown on the right side).*

All these definitions extend traditional SN P systems consistently to ESN P systems. That means, delays and regular expressions of all three types of rules are handled in exactly the same way as delays and regular expressions of the rules in traditional SN P systems.

## 4.2. Formal Definition

The formal definition of ESN P systems is structured as follows: In Definitions 4.1–4.7, basic components of ESN P systems (like labels, rules and spiketrains) and operations on these components are introduced. In the subsequent Definitions 4.8–4.11, ESN P systems and configurations of such systems are covered. Finally, Definitions 4.12–4.23 introduce state transitions between configurations, and describe computations of SN P systems.

**Definition 4.1.** A set $P$ of *prototype labels* (or *prototypes* for short) is a finite, non-empty set. ◇

**Definition 4.2.** Let $P$ be a set of prototype labels. A *rule over* $P$ is a construct of the form $E/a^c \to \mathit{result}; d$ where

- $E$ is a regular expression over a singleton alphabet $O = \{a\}$ ,
- $c \geq 1$ ,
- $d \geq 0$ and
- $\mathit{result}$ is one of the following:
  (1) $\mathit{result} = a^g$ with $0 \leq g \leq c$ (then the rule is called *spiking rule*),
  (2) $\mathit{result} = [p]$ with $p \in P$ (then the rule is called *expanding rule*),
  (3) $\mathit{result} \in \{\text{in}, \text{out}\}$ (then the rule is called *connecting rule*).

Furthermore, the set of all such rules over $P$ is denoted by $\text{Rules}_P$. ◇

The set of all rules over $P$ comprises all possible spiking, expanding and connecting rules. All rules define an amount of spikes $c$ to be consumed and a delay $d$. Spiking rules define a number of spikes to be generated. Expanding rules specify a prototype

$p$ that is to be used for a newly created neuron. This must be a prototype out of $P$. Connecting rules specify if created synapses are inward-directed or outward-directed.

**Definition 4.3.** Let $r = (E/a^c \to \mathit{result}; d) \in \mathrm{Rules}_P$ be a rule over $P$. Then, projections to the components of $r$ are defined by

- $r_{|\mathrm{E}} \coloneqq E$ ,
- $r_{|\mathrm{c}} \coloneqq c$ ,
- $r_{|\mathrm{g}} \coloneqq \begin{cases} g & \text{if } \mathit{result} = a^g \text{ (and therefore } r \text{ is a spiking rule)} \\ \perp & \text{otherwise,} \end{cases}$
- $r_{|\mathrm{p}} \coloneqq \begin{cases} p & \text{if } \mathit{result} = [p] \text{ with } p \in P \text{ (and therefore } r \text{ is an expanding rule)} \\ \perp & \text{otherwise,} \end{cases}$
- $r_{|\delta} \coloneqq \begin{cases} \mathit{result} & \text{if } \mathit{result} \in \{\mathrm{in}, \mathrm{out}\} \text{ (and therefore } r \text{ is a connecting rule)} \\ \perp & \text{otherwise,} \end{cases}$
- $r_{|\mathrm{d}} \coloneqq d$ .

Furthermore, a function $\mathrm{decrease}\colon \mathrm{Rules}_P \to \mathrm{Rules}_P$ is defined by

$$\mathrm{decrease}\colon r \mapsto \begin{cases} E/a^c \to \mathit{result}; (d-1) & \text{if } d > 0 \\ E/a^c \to \mathit{result}; 0 & \text{otherwise.} \end{cases} \qquad \diamond$$

**Definition 4.4.** A set $N$ of *neuron labels* (or *neurons* for short) is a finite, non-empty set such that there exists an alphabet $\Sigma$ with $\Sigma \cap \{:\} = \emptyset$ and $N \subseteq \Sigma^*$. $\diamond$

**Definition 4.5.** Let $N$ be a set of neuron labels and $e \in \mathbb{N}_{\geq 0}$. The *set of extended neuron labels based on $N$ restricted by $e$*, denoted by $\mathrm{Ext}_e(N)$, is defined as

$$\begin{aligned} \mathrm{Ext}_0(N) &\coloneqq N \ , \\ \mathrm{Ext}_e(N) &\coloneqq \mathrm{Ext}_{e-1}(N) \cdot (\varepsilon \cup \{:\} \cdot \{e\}) \ . \end{aligned}$$

The set of *all extended neuron labels based on $N$*, denoted by $\mathrm{Ext}(N)$, is defined as

$$\mathrm{Ext}(N) \coloneqq \bigcup_{e \geq 0} \mathrm{Ext}_e(N) \ . \qquad \diamond$$

Since an ESN P system can grow neurons using expanding rules of already existing neurons, a unique label for each of these new neurons must be generated. This is simply done by appending a number onto the label of the parent neuron that generates a new neuron label (using a ":"-symbol as separation). The exact strategy to construct such new labels is defined later on. However, the set of extended neuron labels restricted by $e$ comprises all labels the ESN P system $\Pi$ may generate in $e$ or less steps from labels of $N$. A subset of this generally describes existing neurons after $e$ time steps. The set of all extended neuron labels $\mathrm{Ext}(N)$ comprises all labels an ESN P system based on neuron labels $N$ may generate.

**Definition 4.6.** Let $N$ be a set of neuron labels and $e \in \mathbb{N}_{\geq 1}$. Then, the function $\mathrm{child}_{e-1}\colon \mathrm{Ext}_{e-1}(N) \to \mathrm{Ext}_e(N)$ is defined by

$$\mathrm{child}_{e-1}\colon n \mapsto (n{:}e)\ .$$

Furthermore, the function $\mathrm{parent}\colon \mathrm{Ext}(N) \setminus N \to \mathrm{Ext}(N)$ is defined by

$$\mathrm{parent}\colon (n{:}i) \mapsto n$$

where $n \in \mathrm{Ext}(N)$ and $i \in \mathbb{N}_{\geq 0}$. ◇

This defines parent-child relationships between neuron labels. Note that for all $e \geq 0$, $n_1, n_2 \in \mathrm{Ext}_e(N)$:

- $n_1 \neq n_2 \implies \mathrm{child}_e(n_1) \neq \mathrm{child}_e(n_2)$ and
- $\mathrm{child}_e(n_1), \mathrm{child}_e(n_2) \notin \mathrm{Ext}_e(N)$.

Because of these two properties, every new neuron label generated by a system—as defined later—is unique (i. e. not already present).

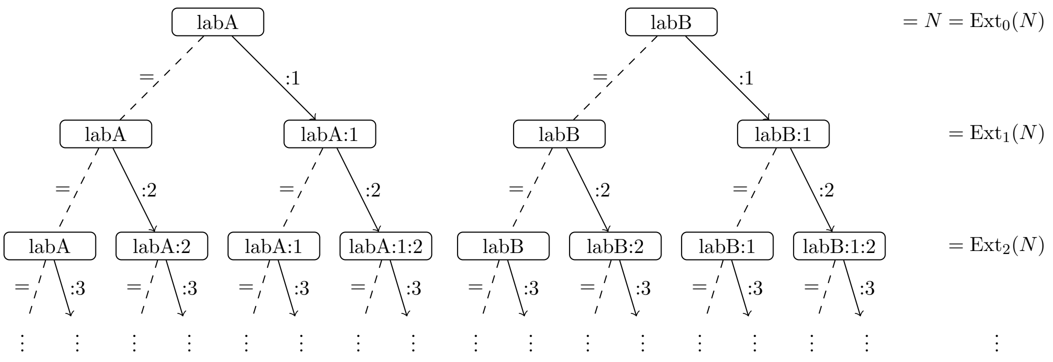


**Figure 4.3.:** *Construction of extended neuron labels.*

**Definition 4.7.** A *spiketrain* of length $l \in \mathbb{N}_{\geq 0}$ is a sequence denoted by $a^{x_1} \cdots a^{x_l}$, where $x_1, \ldots, x_l \in \mathbb{N}_{\geq 0}$. The symbol $\varepsilon$ refers to the spiketrain of length 0, which is the empty sequence. ◇

A spiketrain represents the flow of spikes along a certain synapse. That is the number of spikes transmitted over that synapse on each of a number of consecutive time steps. Reading direction is from left to right, which means the leftmost element of the spiketrain sequence represents the oldest time step. For example, $a^3a^{20}a^0a^{10}$ means that 3 spikes were transmitted before 20 spikes in the following step and so on. Since the input of an ESN P system is transmitted from the environment to the input neuron and the output from the output neuron into the environment, spiketrains are suitable to represent input as well as output of an ESN P system.

**Definition 4.8.** An *expanding spiking neural P system* (ESN P system, for short) is a tuple

$$\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$$

where:

1. $O = \{a\}$ is the alphabet with only one element $a$ called *spike*.
2. $P$ is a set of prototype labels.
3. $N$ is a set of neuron labels.
4. $ru\colon P \to \{R \mid R \subseteq \text{Rules}_P\}$ maps a prototype label to a *prototype definition*, i. e. a set of rules over $P$.
5. $spn\colon N \to \mathbb{N}_{\geq 0}$ maps a neuron label to the initial number of spikes contained in that neuron.
6. $proto\colon N \to P$ maps a neuron label to the associated prototype of that neuron.
7. $syn \subseteq N \times N$, such that for all $n \in N\colon (n, n) \notin syn$, is a relation defining the *initial synaptic connections* between neuron labels.
8. $n_{\text{in}} \in N$ is the label of the so called *input neuron* and
9. $n_{\text{out}} \in N$ is the label of the so called *output neuron*. ◇

In ESN P systems, *initial neurons* are represented by the set of neuron labels $N$. Every such neuron is mapped to a prototype as well as to the amount of spikes initially contained in the neuron. Associated prototype and initial number of spikes are called *attributes* of the neuron. Furthermore, prototypes are mapped to sets of rules. Therefore, each neuron is indirectly associated with a set of rules as well.

A slightly different graphical representation of neurons is used in contrast to the representation of SN P systems. The new representation is shown in Figure 4.4. A prototype of the system is completely defined by the representation of a neuron using that prototype. In case there are prototypes which are not used by any initial neuron of a system, a neuron prototype is represented independently of neurons, as in Figure 4.5. Figure 4.6 uses neuron and prototype representations to show an example of an ESN P system. Initial synapses between neurons are hereby presented as directed edges.

In the remainder of this diploma thesis, the rule abbreviations—introduced in Section 4.1—are considered to be synonymous to the respective long forms.

**Definition 4.9.** An ESN P system $\Pi$ is *deterministic* if and only if for every possible choice of two rules of the same prototype definition of $\Pi$, the two rules do not conflict. Two rules *conflict* if and only if the two languages defined by the regular expression components of the rules are not disjunct. ◇

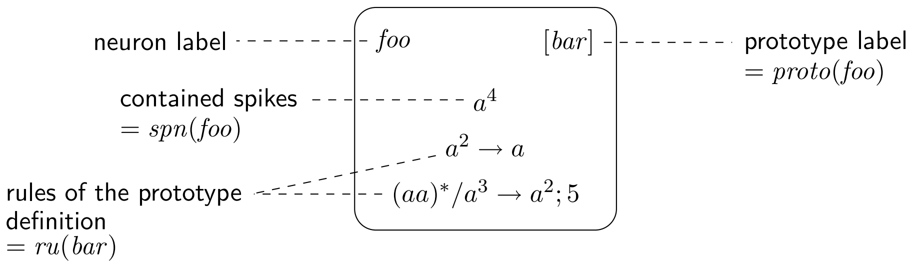


**Figure 4.4.:** *Graphical representation of a neuron and its prototype.*

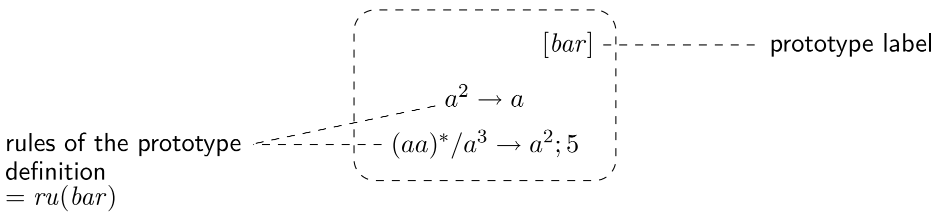

$[bar]$
prototype label
$a^2 \rightarrow a$
rules of the prototype definition
$= ru(bar)$
$(aa)^*/a^3 \rightarrow a^2; 5$

**Figure 4.5.:** *Graphical representation of a prototype.*

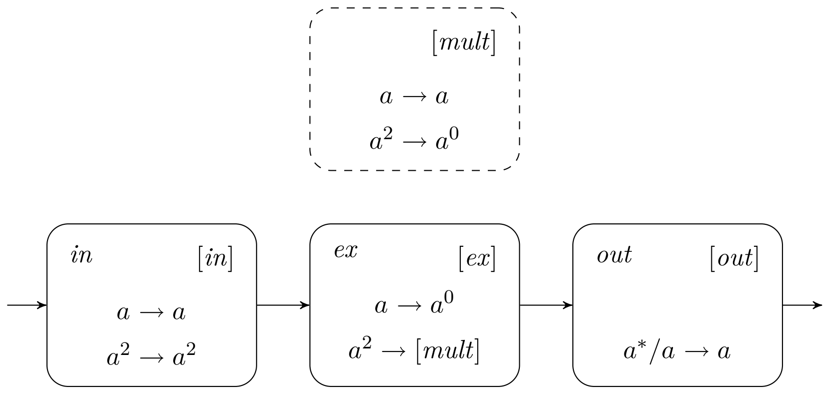


**Figure 4.6.:** *A simple unary multiplication ESN P system. Example for $4 \cdot 2$: An input of $a^2a^2a^2a^2aa$ yields an output of $a^0a^0a^0a^0a^0a^0a^0aaaaaaaa$. The resulting spiketrain is interpreted as number by counting the spikes.*

**Definition 4.10.** Let $\mathit{\Pi} = (O, P, N, ru, spn, proto, syn, n_{\mathrm{in}}, n_{\mathrm{out}})$ be an ESN P system. Then the *set of schedules for a prototype* $p \in P$, denoted by $\mathrm{Sched}_{\mathit{\Pi}}(p)$, is defined as the smallest set $S$ such that:

1. $\bot \in S$ ,
2. $ru(p) \subseteq S$ ,
3. $\forall r \in S \setminus \{\bot\}\colon \ \mathrm{decrease}(r) \in S$ .

The projections of Definition 4.3 are extended for $\bot$, such that

$$\bot_{|\mathrm{E}} \coloneqq \bot_{|\mathrm{c}} \coloneqq \bot_{|\mathrm{g}} \coloneqq \bot_{|\mathrm{p}} \coloneqq \bot_{|\delta} \coloneqq \bot_{|\mathrm{d}} \coloneqq \bot \ .$$

Finally, the set of all schedules of the ESN P system $\mathit{\Pi}$, denoted by $\mathrm{Sched}_{\mathit{\Pi}}$, is defined as

$$\mathrm{Sched}_{\mathit{\Pi}} \coloneqq \bigcup_{p \in P} \mathrm{Sched}_{\mathit{\Pi}}(p) \ . \qquad \diamond$$

The set of schedules of a prototype comprises all rules of the prototype definition (i. e. set of rules) associated by $ru$. Additionally, it comprises all variants of these rules with delays lower than the delays the original rules have. These schedules are called *derived* from their corresponding prototype definition rule. Schedules represent the application progress of a rule on a neuron at one point in time by counting down the remaining delay.

**Definition 4.11.** Let $\mathit{\Pi} = (O, P, N, ru, spn, proto, syn, n_{\mathrm{in}}, n_{\mathrm{out}})$ be an ESN P system. A *configuration of* $\mathit{\Pi}$ is a tuple

$$\mathit{Conf} = (e, N', spn', proto', scd, syn', in, out)$$

where:

1. $e \in \mathbb{N}_{\geq 0}$ is the *restriction* of extended neuron labels in this configuration.
2. $N' \subseteq \mathrm{Ext}_e(N)$ is the set of extended neuron labels restricted by $e$ the net consists of in this configuration. It must adhere to $N \subseteq N'$.
3. $spn'\colon N' \to \mathbb{N}_{\geq 0}$ maps an extended neuron label of the configuration to the number of spikes contained by that neuron in this configuration.
4. $proto'\colon N' \to P$ maps an extended neuron label of the configuration to the associated prototype of that neuron. It must adhere to $\forall n \in N\colon proto'(n) = proto(n)$.
5. $scd\colon N' \to \mathrm{Sched}_{\mathit{\Pi}}$ maps an extended neuron label to an *active schedule* describing the state of rule application of the neuron. It must adhere to $\forall n \in N'\colon scd(n) \in \mathrm{Sched}_{\mathit{\Pi}}(proto'(n))$.
6. $syn' \subseteq N' \times N'$ such that $\forall n \in N'\colon (n, n) \notin syn'$. This relation defines the synaptic connections between the extended neuron labels of this configuration. It must adhere to $syn \subseteq syn'$.

7. $in$ is a spiketrain.

8. $out$ is a spiketrain of length $e$.

The number of spikes, the associated prototype and the active schedule of a neuron are called *state attributes* of this neuron. For all $n \in N'$, the state attributes $r = scd(n)$ and $p = proto'(n)$ must adhere to the restriction $r \in \mathrm{Sched}_\Pi(p)$, i. e. an active schedule of a neuron $n$ must be derived from a rule of the prototype definition of the neuron $n$. If a schedule is the active schedule in a configuration, then the rule of the prototype definition it is derived from is said to be *scheduled*. The set of all configurations of $\Pi$ is denoted by $\mathrm{Config}_\Pi$ and $out$ is called the *output* of $Conf$. ◇

A configuration describes the state of an ESN P system after a number of time steps and is similar in definition to the ESN P system itself. However, the set of neuron labels in a configuration is a subset of all extended neuron labels. It is not necessarily a subset of the set of neurons labels $N$, because there may be neurons present that were not present in the description of the system, but were generated later. Note that a restriction for label extensions is present in a configuration. This ensures that a new label—i. e. a label that is not already used—can be derived by adding a number greater than the restriction to an already used label.

Another difference is the additional attribute of a configuration: the active schedule. This attribute describes the progress of rule application, since rules may need more than one step to be applied. If no rule is currently applied—i. e. $\perp$ is the active schedule—the neuron is *open*. An open neuron may select a rule of its prototype definition as new active schedule in the next step. Otherwise the neuron is *closed*.

Figure 4.7 shows how this additional attribute is represented graphically: The active schedule is underlined, or nothing is underlined if $\perp$ is the active schedule. Note that in a neuron representation only rules of the associated prototype definition are shown. This ensures that the prototype definition of the represented neuron can always be derived. However, by definition of $\mathrm{Sched}_\Pi$, a schedule may define a delay lower than the rule of the prototype it is derived from. In case such a schedule is the active schedule of a neuron, the rule the schedule is derived from is underlined instead. The actual delay of the active schedule is given in addition. As an example, consider the rule $(aa)^*/a^3 \to a^2; 5$. If this rule is part of the prototype definition of a neuron and active schedule, then the corresponding representation is $\underline{(aa)^*/a^3 \to a^2; 5}$. Since $(aa)^*/a^3 \to a^2; 5$ is in $\mathrm{Sched}_\Pi$, the schedule $(aa)^*/a^3 \to a^2; 2$ is in $\mathrm{Sched}_\Pi$ as well. If $(aa)^*/a^3 \to a^2; 2$ becomes the active schedule, then the corresponding representation is $\underline{(aa)^*/a^3 \to a^2; 5}\ \langle 2 \rangle$ to indicate that $(aa)^*/a^3 \to a^2; 2$ is not part of the prototype definition but active schedule in this configuration.

Finally, the configuration contains two spiketrains. These represent not yet consumed input and already generated output respectively.

**Definition 4.12.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$ be an ESN P system and $in$ a spiketrain. The *initial configuration with $in$ of $\Pi$* is defined as

$$Conf_{\text{init}} = (0, N, spn, proto, scd, syn, in, \varepsilon)$$

where $scd \colon n \mapsto \perp$. ◇

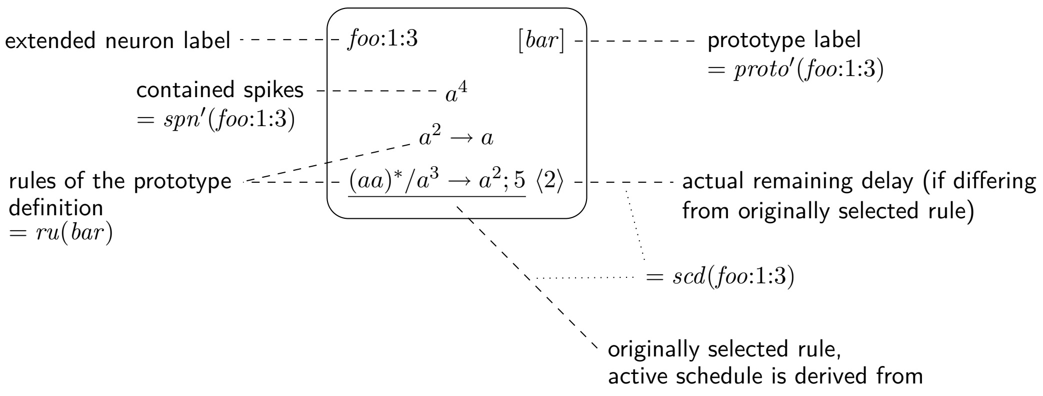

extended neuron label
contained spikes = $spn'(foo{:}1{:}3)$
rules of the prototype definition = $ru(bar)$
$foo{:}1{:}3$
$[bar]$
$a^4$
$a^2 \to a$
$\underline{(aa)^*/a^3 \to a^2; 5}$ $\langle 2 \rangle$
prototype label = $proto'(foo{:}1{:}3)$
actual remaining delay (if differing from originally selected rule)
$= scd(foo{:}1{:}3)$
originally selected rule, active schedule is derived from

**Figure 4.7.:** *Graphical representation of a neuron configuration.*

**Definition 4.13.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$ be an ESN P system, $Conf = (e, N', spn', proto', scd, syn', in, out)$ a configuration of $\Pi$ and $n \in N'$ an extended neuron label of the configuration. A rule $r \in \text{Rules}_P$ is called *applicable* on the neuron $n$ in $Conf$, denoted by $r \overset{Conf}{\text{app}} n$, if and only if:

1. $r \in ru(proto'(n))$ ,
2. $spn'(n) \geq r_{|\text{c}}$ ,
3. $a^{spn'(n)} \in \text{L}(r_{|\text{E}})$ , and
4. $scd(n) = \perp$ . ◇

That means, a rule is applicable on a neuron $n$ if and only if

1. the rule is part of the prototype definition of $n$,
2. the rule does not consume more spikes than $n$ contains,
3. the number of spikes contained in $n$ matches against the regular expression of the rule, and
4. $n$ is open.

**Definition 4.14.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$ be an ESN P system and $Conf = (e, N', spn', proto', scd, syn', in, out)$ a configuration of $\Pi$. A function $sel \colon N' \to \text{Sched}_\Pi$ is called a *selection for $Conf$* if and only if $\forall n \in N'$:

1. $(\exists r \in ru(proto'(n)) \colon r \overset{Conf}{\text{app}} n) \implies sel(n) \overset{Conf}{\text{app}} n$ and
2. $(\nexists r \in ru(proto'(n)) \colon r \overset{Conf}{\text{app}} n) \implies sel(n) = \perp$ . ◇

A selection represents the choice of rules which are to be applied in the next time step. For each neuron of the configuration, exactly one applicable rule must be chosen as long as there is at least one rule applicable. No rule may be chosen (expressed by $\perp$) otherwise.

**Definition 4.15.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$ be an ESN P system. A binary relation $\text{Rescheduled}_\Pi \subseteq \text{Config}_\Pi \times \text{Config}_\Pi$ is defined as follows: Let

$$Conf_1 = (e, N', spn', proto', scd_1, syn', in, out) \text{ and}$$
$$Conf_2 = (e, N', spn', proto', scd_2, syn', in, out)$$

be configurations of $\Pi$. Then, $(Conf_1, Conf_2) \in \mathrm{Rescheduled}_\Pi$ if and only if there exists a selection $sel$ for $Conf_1$ such that

$$\forall n \in N' \colon scd_2(n) = \begin{cases} sel(n) & \text{if } sel(n) \neq \bot \\ scd_1(n) & \text{otherwise.} \end{cases} \qquad \diamond$$

A configuration of $\Pi$ can be *rescheduled* with a selection (for that configuration) to a new configuration in the following way: For every neuron of $\Pi$, if a rule is selected for a neuron, the selected rule is written into the state of that neuron as new schedule. However, if no rule ($\bot$) is selected for that neuron, the state attributes of that neuron remain unchanged.

The relation $\mathrm{Rescheduled}_\Pi$ relates two configurations $Conf_1$ and $Conf_2$ if and only if there exists a selection for $Conf_1$ such that $Conf_1$ can be rescheduled to $Conf_2$.

**Definition 4.16.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\mathrm{in}}, n_{\mathrm{out}})$ be an ESN P system and $Conf = (e, N', spn', proto', scd, syn', in, out)$ a configuration of $\Pi$. Then the function $\mathrm{grow}_\Pi \colon \mathrm{Config}_\Pi \to \mathrm{Config}_\Pi$ is defined by

$$\mathrm{grow}_\Pi \colon Conf \mapsto (e+1, N' \cup N'_{\mathrm{new}}, spn'', proto'', scd', syn'', in, out)$$

where

$$N'_{\mathrm{new}} = \{\mathrm{child}_e(n) \mid n \in N',\ scd(n)_{|\mathrm{p}} \neq \bot,\ scd(n)_{|\mathrm{d}} = 0\}\ ,$$

$$spn'' \colon n \mapsto \begin{cases} spn'(n) & \text{if } n \in N' \\ spn'(\mathrm{parent}(n)) & \text{if } n \in N'_{\mathrm{new}}\ , \end{cases}$$

$$proto'' \colon n \mapsto \begin{cases} proto'(n) & \text{if } n \in N' \\ scd(\mathrm{parent}(n))_{|\mathrm{p}} & \text{if } n \in N'_{\mathrm{new}}\ , \end{cases}$$

$$scd' \colon n \mapsto \begin{cases} scd(n) & \text{if } n \in N' \\ scd(\mathrm{parent}(n)) & \text{if } n \in N'_{\mathrm{new}}\ , \end{cases}$$

$$\begin{aligned} syn'' = syn' \\ &\cup\ \{(n_1, n_2) \mid n_1, n_2 \in N'_{\mathrm{new}},\ (\mathrm{parent}(n_1), \mathrm{parent}(n_2)) \in syn'\} \\ &\cup\ \{(n_1, n_2) \mid n_1 \in N',\ \mathrm{child}_e(n_1) \notin N'_{\mathrm{new}}, \\ &\qquad n_2 \in N'_{\mathrm{new}},\ (n_1, \mathrm{parent}(n_2)) \in syn'\} \\ &\cup\ \{(n_1, n_2) \mid n_2 \in N',\ \mathrm{child}_e(n_2) \notin N'_{\mathrm{new}}, \\ &\qquad n_1 \in N'_{\mathrm{new}},\ (\mathrm{parent}(n_1), n_2) \in syn'\}\ . \qquad \diamond \end{aligned}$$

The function $\mathrm{grow}_\Pi$ computes the effects of application of expanding rules. New labels are generated for every neuron with a scheduled expanding rule and zero delay. All state attributes—except for the corresponding prototype—of the new neurons are cloned from the parent neurons. The corresponding prototype of new neurons is a component of the rule dividing the parent neuron. New synapses are added as was shown in Figure 4.1.

**Definition 4.17.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\mathrm{in}}, n_{\mathrm{out}})$ be an ESN P system and $Conf = (e, N', spn', proto', scd, syn', in, out)$ a configuration of $\Pi$. Then the function $\mathrm{connect}_\Pi \colon \mathrm{Config}_\Pi \to \mathrm{Config}_\Pi$ is defined by

$$\mathrm{connect}_\Pi \colon Conf \mapsto (e, N', spn', proto', scd, syn'', in, out)$$

where

$$\begin{aligned} syn'' = syn' \cup \{(n_1, n_2) \mid n_1, n_2 \in N', &\\ & scd(n_1)_{|\delta} = \mathrm{out},\ scd(n_2)_{|\delta} = \mathrm{in},\\ & scd(n_1)_{|\mathrm{d}} = scd(n_2)_{|\mathrm{d}} = 0\}\ . \end{aligned} \qquad \diamond$$

The function $\mathrm{connect}_\Pi$ computes the effects of applications of connecting rules. New synapses are added for each pair of neurons applying an out-connecting rule with a delay of zero and an in-connecting rule with a delay of zero.

**Definition 4.18.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\mathrm{in}}, n_{\mathrm{out}})$ be an ESN P system and $Conf = (e, N', spn', proto', scd, syn', in, out = a^{y_1} \cdots a^{y_e})$ a configuration of $\Pi$. Then the function $\mathrm{spike}_\Pi \colon \mathrm{Config}_\Pi \to \mathrm{Config}_\Pi$ is defined by

$$\mathrm{spike}_\Pi \colon Conf \mapsto (e, N', spn'', proto', scd, syn', in, out')$$

where

$$out' = \begin{cases} a^{y_1} \dots a^{y_e} a^i & \text{if } scd(n_{\mathrm{out}})_{|\mathrm{d}} = 0,\ i \neq \perp \text{ where } i = scd(n_{\mathrm{out}})_{|\mathrm{g}} \\ a^{y_1} \dots a^{y_e} a^0 & \text{otherwise.} \end{cases}$$

$$spn'' \colon n \mapsto \begin{cases} spn'(n) + \sum\limits_{\hat{n} \in N'} \mathrm{income}(\hat{n}, n) & \text{if } scd(n)_{|\mathrm{d}} \in \{\perp, 0\} \\ spn'(n) & \text{otherwise} \end{cases}$$

where $\mathrm{income} \colon N' \times N' \to \mathbb{N}_{\geq 0}$ is defined by

$$\mathrm{income} \colon (\hat{n}, n) \mapsto \begin{cases} scd(\hat{n})_{|\mathrm{g}} & \text{if } scd(\hat{n})_{|\mathrm{g}} \neq \perp,\ scd(\hat{n})_{|\mathrm{d}} = 0 \text{ and } (\hat{n}, n) \in syn' \\ 0 & \text{otherwise.} \end{cases} \qquad \diamond$$

The function $\mathrm{spike}_\Pi$ computes the effects of application of spiking rules. Open neurons receive and accumulate spikes from neurons applying a spiking rule with delay zero if a synapse from the applying neuron to the receiving neuron exists. Neurons applying a rule with a delay of zero are considered as open since they will be open as soon as the time step is completed. If the output neuron generates spikes, the current output spiketrain is modified accordingly.

**Definition 4.19.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$ be an ESN P system and $Conf = (e, N', spn', proto', scd, syn', in, out)$ a configuration of $\Pi$. Then the function $\text{tick}_\Pi\colon \text{Config}_\Pi \to \text{Config}_\Pi$ is defined by

$$\text{tick}_\Pi\colon Conf \mapsto (e, N', spn'', proto', scd', syn', in, out)$$

where

$$spn''\colon n \mapsto \begin{cases} spn'(n) - scd(n)_{|\text{c}} & \text{if } scd(n)_{|\text{d}} = 0 \\ spn'(n) & \text{otherwise,} \end{cases}$$

$$scd'\colon n \mapsto \begin{cases} \bot & \text{if } scd(n)_{|\text{d}} \in \{\bot, 0\} \\ \text{decrease}(scd(n)) & \text{otherwise.} \end{cases} \qquad \diamond$$

The function $\text{tick}_\Pi$ ends application of rules in the current time step. All schedules with a remaining delay greater than zero are decreased by one. Furthermore, schedules with a delay of zero are removed from their neurons. All neurons removing a schedule consume an amount of spikes according to the applied rule.

**Definition 4.20.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$ be an ESN P system and $Conf = (e, N', spn', proto', scd, syn, in = a^{x_1} \cdots a^{x_j}, out)$ a configuration of $\Pi$ with $j \in \mathbb{N}_{\geq 0}$. Then the function $\text{stim}_\Pi\colon \text{Config}_\Pi \to \text{Config}_\Pi$ is defined by

$$\text{stim}_\Pi\colon Conf \mapsto (e, N', spn'', proto', scd, syn, in', out)$$

where

$$spn''\colon n \mapsto \begin{cases} spn'(n) + x_1 & \text{if } n = n_{\text{in}},\ scd(n)_{|\text{d}} = \bot \text{ and } j > 0 \\ spn'(n) & \text{otherwise,} \end{cases}$$

$$in' = a^{x_2} \ldots a^{x_j}\ . \qquad \diamond$$

The function $\text{stim}_\Pi$ ends a time step by computing the effects of input. The number of spikes specified as input for the current time step are added to the spikes of the input neuron (if that neuron is open). The name stim is derived from "stimulating" the net.

**Definition 4.21.** Let $\Pi = (O, P, N, ru, spn, proto, syn, n_{\text{in}}, n_{\text{out}})$ be an ESN P system and $Conf = (e, N', spn', proto', scd, syn', in, out)$ a configuration of $\Pi$. Then, $Conf$ is called a *halting configuration* of $\Pi$ if and only if:

1. $\forall n \in N'\colon \nexists r \in ru(proto(l))\colon r \overset{Conf}{\text{app}} n$ ,
2. $\forall n \in N'\colon scd(n) = \bot$ , and
3. $in = \varepsilon$ . $\diamond$

A configuration is a halting configuration if no rules are scheduled, no rules are applicable and there is no more input to be consumed.

**Definition 4.22.** Let $\Pi$ be an ESN P system and $\circ$ denote the composition of relations. Then the *transition relation of $\Pi$* $\vdash_\Pi \subset \mathrm{Config}_\Pi \times \mathrm{Config}_\Pi$ is defined by

$$\vdash_\Pi \quad = \quad (\mathrm{stim}_\Pi \circ \mathrm{tick}_\Pi \circ \mathrm{spike}_\Pi \circ \mathrm{connect}_\Pi \circ \mathrm{grow}_\Pi \circ \mathrm{Rescheduled}_\Pi) \; . \qquad \diamond$$

A transition between configurations as specified by the relation $\vdash_\Pi$ defines time steps of the system. Note that $\mathrm{Select}_\Pi$ is a relation but usually not a function. Therefore, $\circ$ denotes the composition of relations (right hand side relation first, as defined in the preliminaries), with $\mathrm{stim}_\Pi$,$\mathrm{tick}_\Pi$, $\mathrm{spike}_\Pi$, $\mathrm{connect}_\Pi$ and $\mathrm{grow}_\Pi$ interpreted as relations. Since the composition of functions is obtained as a special case of the composition of relations, $\circ$ can equally be interpreted as function composition in all cases where $\mathrm{Rescheduled}_\Pi$ is a function (for example in deterministic ESN P systems).

**Definition 4.23.** Let $\Pi$ be an ESN P system and $in$ a spiketrain. Then $Comp = (Conf_0, \dots, Conf_z)$ is a *computation with $in$ of $\Pi$* if and only if

1. $Conf_i \in \mathrm{Config}_\Pi$ for all $i \in \{0, \dots, z\}$ ,
2. $Conf_i \vdash_\Pi Conf_{i+1}$ for all $i \in \{0, \dots, z-1\}$ ,
3. $Conf_0$ is the initial configuration with $in$ of $\Pi$ ,
4. $Conf_i$ is not a halting configuration for all $i \in \{0, \dots, z-1\}$ ,
5. $Conf_z$ is a halting configuration of $\Pi$ .

The *output of the computation $Comp$* is defined as the output of $Conf_z$. $\diamond$

A computation with input of an ESN P system defines a series of time steps, beginning at the initial configuration (corresponding to the input) and ending at the first halting configuration yielding the output.

If an ESN P system is deterministic, there is exactly one selection for any given configuration of this system. Therefore, $\mathrm{Rescheduled}_\Pi$, is a function and only one state transition for a configuration is possible. Since there is exactly one initial configuration for each input spiketrain, at most one computation of the system for that input exists.

# 5. Polynomial ESN P Systems Solving `QBF`

In this chapter, an application of ESN P systems is demonstrated: A solution to `QBF` which is deterministic and efficient in time without relying on pre-compiled exponential workspaces.

Section 2.3 presents the `QBF` problem, and furthermore a polynomial transformation to instances of $QBF_{\mathrm{S}}(y = 2n, m)$. This was used in Section 3.6 to define a family of SN P systems solving the problem efficiently in time by solving instances of $QBF_{\mathrm{S}}(y = 2n, m)$ efficiently.

The same conventions as in Section 3.6 are used throughout this chapter: The terms "polynomial" and "exponential" relate to the size (which is the number of variables) $n$ of the `QBF` instance that is to be solved. Furthermore, $m$ denotes the number of clauses in an instance of `QBF` and $y = 2n$ the number of variables of the equivalent instance of $QBF_{\mathrm{S}}(y = 2n, m)$.

Although, the systems given in Section 3.6 are deterministic, they depend on pre-compiled exponential workspaces. To eliminate the need for this pre-compiled exponential workspace, a new family of ESN P systems is constructed based on the original family. Every ESN P system constructed in that way (as described below) will be called *extended system*. Every extended system is based on an SN P system of the original family, called its *base system*. For every system in the original family, one extended system is constructed which has the same function but does not rely on a pre-compiled workspace.

The intended principle of operation of extended systems is summarized as follows:

1) An extended system of the new family starts on a polynomial workspace. This workspace is then expanded in a polynomial number of time steps (via expanding and connecting rules) until the (originally pre-compiled) exponential workspace of its base system is built. These time steps form the *expansion phase*.

2) After the original workspace is reached, an extended system will not apply any expanding or connecting rules in future time steps. Therefore, only spiking rules are applied (like in an SN P system) and the execution from here on is identical to the base system. Naturally, this means, the extended system yields the same result as the base system (if the input problem instance is identical). These time steps form the *computation phase*.

To facilitate 2), the basic structure with all its neurons and synapses of the base system is kept, although not necessarily present in the initial workspace but generated later during the expansion phase. Additionally, the spiking rules of these neurons are mostly kept as well. Because of that, the explanations on the structure of the base systems in Section 3.6 still apply, and these inherited parts in an extended system are not discussed any further in this chapter. Nor is the computation phase of the extended systems discussed, as this process is identical to the execution of the base system for the same reason. Although, input processing and input encoding still follow the same strategy, too, some aspects are modified to account for the delay of input processing caused by the addition of an expansion phase to the net. This is addressed later on in connection with the input module. On the other hand, output interpretation is unchanged. The system is still used as function computing device in the same way (again already covered by Section 3.6).

To facilitate 1), the initial workspace must be reduced and new rules must be added to the neurons of the original structure. To retain the determinism of this system, the new rules must not conflict with the spiking rules kept from the base system. This chapter covers the implementation of the expansion phase and avoidance of rule conflicts with inherited spiking rules.

The initial workspace of the new extended systems is shown in Figure 5.1. Rules and initial spikes of the neurons are omitted from the figure for clarity. However, they are subject of the following sections and can also be gathered from the formal definition of the ESN P system family given in Appendix A.

In comparison to the base systems of Section 3.6, some neurons are renamed (or rather have different labels) but the structural and functional relationship is easy to see. Similar to the base systems, the neurons of each extended system are grouped into modules (*input module*, *assignment module* and *quantification module*), and additionally the *input bridge neurons* which are labeled $1x_1$, $0x_1$, $1x_2$, $0x_2$, $\ldots$, $1x_y$, $0x_y$.[1] Neurons of each module in turn are grouped by horizontal layers. While the input module and the input bridge neurons are completely retained from the base system, the other two modules are reduced to one neuron per layer. Missing neurons of a layer will be generated by the existing neuron of the layer or one of its descendants during execution. Synapses need to be generated as well.

In the base systems, the input module drives the rest of the net. That means, there are no neurons outside of the input module having spikes in their initial configuration. None of these neurons can apply any rules, before the input module sends spikes for the first time. This allows to add initial spikes and matching rules to these neurons without causing side effects, if the additional rules are carefully chosen such that:

- No spikes are generated.
- All initial spikes are consumed, before spikes arrive for the first time.
- None of the additional rules may become applicable again, after all initial spikes are consumed.

[1]Actually, neurons of the SN P systems of Section 3.6 are only grouped into neurons of the input module and all other neurons. For the extended systems this grouping is retained for the input module, while the other neurons are separated into the two new modules and the input bridge neurons.

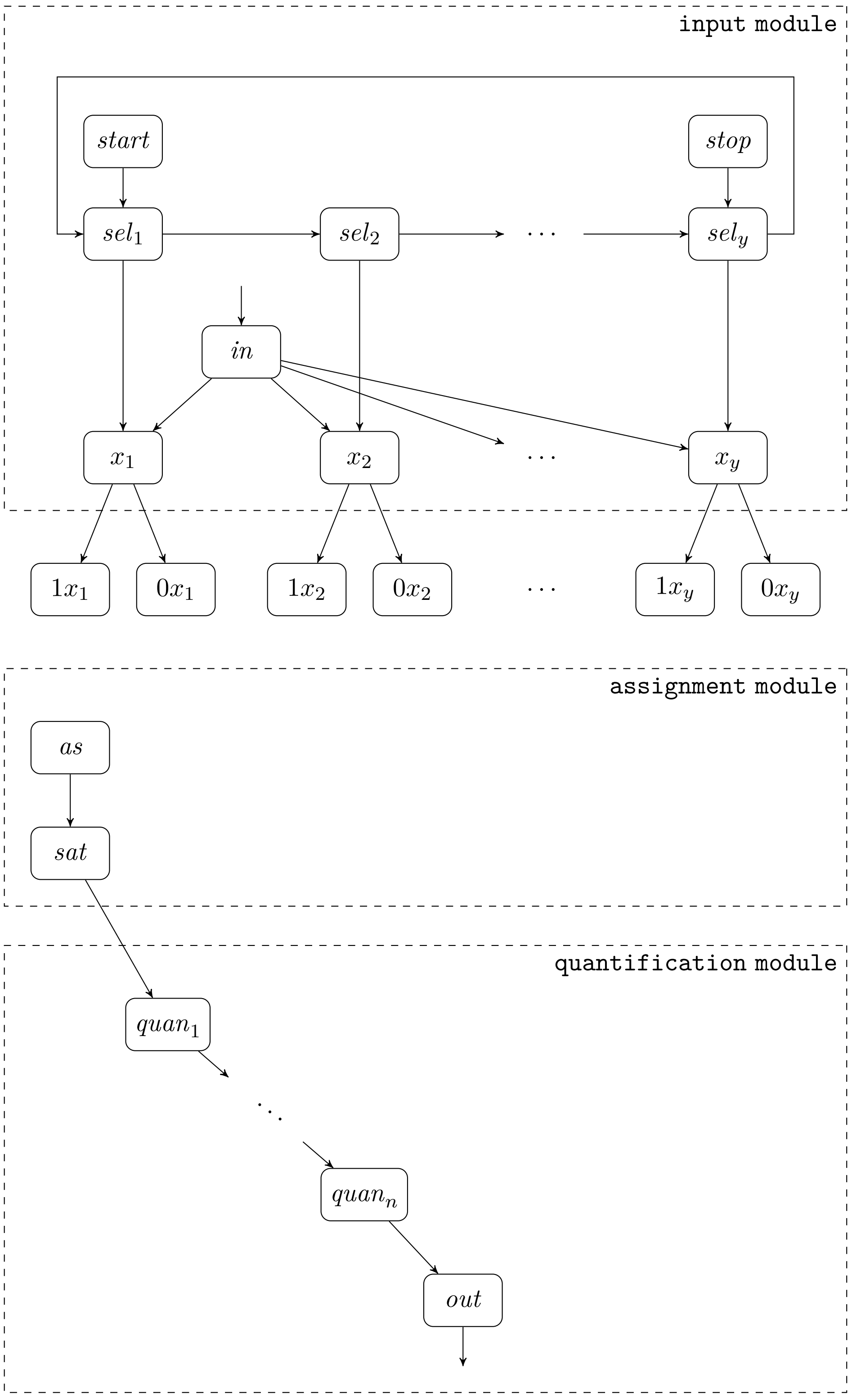


**Figure 5.1.:** *Family of ESN P systems solving instances of* $QBF_{\mathrm{S}}(y = 2n, m)$

- The additional rules may not conflict with the spiking rules already present, to retain determinism.

By adding rules under these restrictions to the reduced workspace of an extended system, the original workspace of its base system can be generated during the expansion phase. The subsequent computation phase is not influenced by the workspace expansion at all and therefore indeed identical to the base systems. However, to make this possible, the input module needs to delay the start of the computation phase until the expansion phase is finished. This requires some modifications on the input module and on the input encoding.

A detailed description of necessary rule additions and the strategies involved in handling the expansion phase without conflicting with spiking rules is given separately for each module in the following sections. The last section of this chapter discusses the properties of the constructed systems. The complete formal definition of the constructed systems is given in Appendix A. Finally, a step by step example, showing the complete expansion phase of such an extended system, can be found in Appendix B.

## 5.1. Input Encoding and Input Module

The amount of neurons necessary for the input module of the base systems in Section 3.6 depends linearly on the problem size (to be specific, $2y + 3$ neurons are necessary). For this reason, the initial structure of the input module of the base SN P system does not need to be reduced. Its structure is completely retained, although neurons are slightly modified (as seen in a moment), without causing an exponential workspace. Since the input module is retained, the input encoding of the problem instance ($enc$) is reused, too. However, it needs an additional step to adapt to the input module changes detailed in the following.

Since, initially, the input module is already complete, no further neurons or synapses need to be created during the expansion phase. The single reason to modify the input module anyway is to wait for the expansion phase of other modules to finish, before starting the computation phase. As seen later, the maximum number of time steps necessary to finish expanding for all modules is $3 \cdot y$. Both neurons with initial spikes (the neurons $start$ and $stop$) have a delay of $3 \cdot y$ added to their respective rules matching the initial number of spikes. Since all other neurons are driven by these two neurons, the complete input module is automatically delayed by $3 \cdot y$ time steps, too.

An exception to this is the input neuron $in$. The consumption of the input spiketrain by this neuron starts immediately in the first time step and cannot be delayed by a rule (input as spiketrain is associated with time steps, it can be ignored but not delayed). To solve this problem, $3 \cdot y$ arbitrary elements are prepended to the input spiketrain as additional step in the encoding of the problem instance using $enc$. The prepended elements are then consumed completely during the expansion phase, such that only the original encoded spiketrain remains as input at the beginning of the computation phase. The new function to encode a problem instance, working in the described way, is denoted by $enc'$. As an example, consider the formula $\forall x_1 \forall x_2 \exists x_3 \exists x_4 ((\neg x_1 \vee x_4) \wedge (\neg x_4))$ from Section 3.6 again. Encoding it by $enc$ yields the spiketrain $a^2 a^0 a^0 a^1 a^0 a^0 a^0 a^2$. Since there

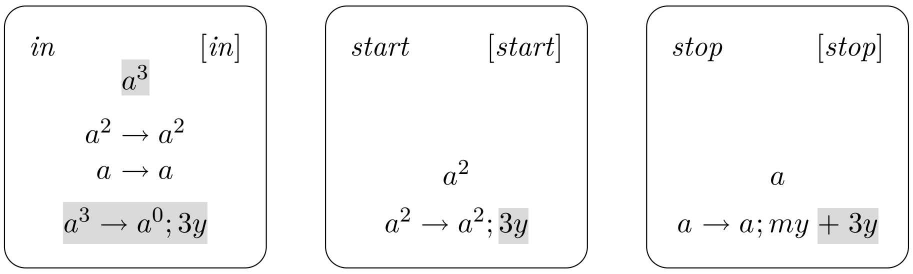


**Figure 5.2.:** *Modifications on the input module. Normal font: retained from base system; gray highlight: additions and modifications.*

are four different variables, 12 time steps are prepended to the result of $enc$. The number of spikes in each prepended step is chosen to be three spikes per step in this example to distinguish the prepended time steps from following steps of the original encoding. The finally resulting spiketrain therefore is $a^3a^3a^3a^3a^3a^3a^3a^3a^3a^3a^3a^3a^2a^0a^0a^1a^0a^0a^0a^2$.

Since the prepended elements of an input encoding are not supposed to influence the net in any way, an additional rule $a^3 \to a^0; 3y$ and 3 initial spikes are added to the neuron $in$. This effectively closes the input neuron during the expansion phase, thereby ignoring any input during these time steps.

Figure 5.2 shows the described modifications on the input module (compared to the base system input module in Figure 3.3).

The delay of the input module ensures that no spikes reach the input bridge neurons in the first $3 \cdot y$ time steps, the assignment-layer neurons in the first $3 \cdot y + 1$ time steps, etc.

## 5.2. Input Bridge Neurons

Like the neurons of the input module, all input bridge neurons are already completely defined in the initial workspace and no new neurons need to be created during the expansion phase. However, synapses to neurons in the assignment-layer need to be created. This is achieved by adding an out-connecting rule and a single initial spike to each input bridge neuron of an extended system. The new connecting rule consumes the initial spike and therefore leaves the neuron with zero spikes (as in the base system) before the computation phase starts. Since input bridge neurons of the base systems have no initial spikes and can never receive only a single spike (to trigger the connecting rule again), this does not interfere with the execution of the computation phase. The delay of the added connecting rule is chosen such that for every $1 \leq i \leq y$ the neuron $1x_i$ applies its connecting rule in time step $1$ and executes its effects in time step $3i - 1$. Similarly, for every $1 \leq i \leq y$ the neuron $0x_i$ applies its connecting rule in time step $1$ and executes its effects in time step $3i$. This translates to a delay of $3i - 2$ for the connecting rule in $1x_i$ and a delay of $3i - 1$ for the connecting rule in $0x_i$. Note that $1x_i$ always executes its connecting rule one time step before $0x_i$.

The implemented timing of these rules is entirely based on the timing of the corresponding in-connecting rules in the assignment module which is further discussed in the

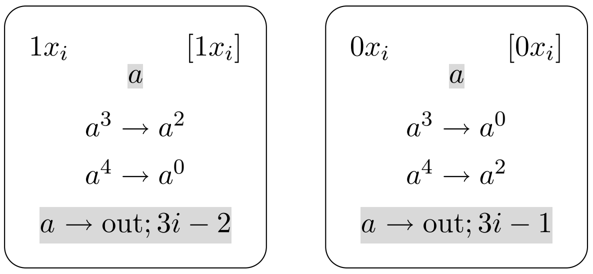


**Figure 5.3.:** *Modifications on the input bridge neurons for $1 \leq i \leq y$. Normal font: retained from base system; gray highlight: additions and modifications.*

following section.

Figure 5.3 shows the additions.

## 5.3. Assignment Module: Assignment-Layer

The assignment module of the base systems consists of 2 layers. Both layers are reduced to a single neuron in the extended systems and must be expanded again during the expansion phase: *Assignment neurons* as descendants of the neuron $as$ in the *assignment-layer* and *sat neurons* as descendants of the neuron $sat$ in the *sat-layer*. The assignment-layer is subject of this section and the sat-layer is subject of the subsequent section.

Recall that assignment neurons represent an assignment of the variables of the input formula. All assignment neurons of the base system use the same set of spiking rules. Therefore, the represented variable assignment of a certain input neuron solely arises from existing synapses between this neuron and the input bridge neurons. Since in extended systems these synapses are created during the expansion phase of the system, the meaning of the assignment neurons changes during that phase. The other way around, the current meaning (i. e. variable assignments) of an assignment neuron describes the set of synapses existing between this neuron and the input bridge neurons. Note that although this representation of variable assignments is ultimately related to the function of this neuron in the computation phase (detailed in Section 3.6), during the expansion phase it has no further meaning than to describe existing synapses.

In the initial configuration of every extended system there exists exactly one assignment neuron ($as$) and no synapses between assignment-layer and input bridge neurons. This neuron represents the one possible variable assignment of zero variables. In $y$ identical *periods*—each one consisting of 3 time steps—the complete assignment-layer is generated. I. e. a layer consisting of $2^y$ assignment neurons representing the $2^y$ possible assignments of the $y$ variables in the input formula. All these neurons are initially present in the corresponding base system. After the first period there are two neurons representing the two possible assignments of the first variable $x_1$. After the second period there are four neurons representing the four possible assignments of the first two variables $x_1$ and $x_2$ etc.

To understand the three time steps of each period, recall that dividing a neuron also clones all its synapses. In terms of variable assignment, that means, dividing an assignment neuron yields two neurons representing the same variable assignments. Such a division of all existing assignment neurons happens in the first step of each period. At the beginning of a period $i$, neurons representing the $2^{i-1}$ assignments of $x_1, \dots, x_{i-1}$ are already present. The division yields parent neurons still representing all these assignments and new child neurons representing the exact same assignments. In time step 2 of a period $i$, all parent neurons assign `true` to the next unassigned variable, which is $x_i$, by creating a synapse to the corresponding input bridge neuron $1x_i$ via in-connecting rule. The second time step of the period $i$ is the absolute time step $3(i-1)+2 = 3i-1$. Note that this is indeed the timing chosen for the input bridge neuron $1x_i$ to execute an out-connecting rule in the previous section. Finally in time step 3 of period $i$, all child neurons assign `false` to the variable $x_i$ by creating a synapse to the corresponding input bridge neuron $0x_i$. This timing is the same as chosen for the input bridge neuron $0x_i$ to execute its out-connecting rule. The variable $x_i$—that is assigned to in the period $i$—is called the *variable of period $i$*. Figure 5.4 illustrates this strategy.

As an example consider the third period (the first period after the periods shown in Figure 5.4). At this point, four neurons representing all possible assignments of the two variables $x_1$ and $x_2$ already exists. The division in the first step of the period yields two identical sets of four neurons. Each set represents all four possible variable assignments for the variables $x_1$ and $x_2$. In step 2, `true` is assigned to the variable $x_3$ for all variable assignments in the first set. In step 3, `false` is assigned to the variable $x_3$ for all variable assignments in the second set. Combined, this yields all eight possible variable assignments for the variables $x_1$, $x_2$ and $x_3$.

Note that this requires all child neurons created in the first step of any given period to behave differently in comparison to their corresponding parent neurons in the following two time steps. I. e. they apply different rules. However, they do behave identically in all following periods, because they are parents in these periods themselves. Furthermore, all neurons of the same status (either parent or child) always execute the exact same operations during a period.

To implement this behavior, each generation of child neurons is created with an (at this point) unused prototype (associated to the generation). This prototype defines rules to assign `false` to the variable of the associated period and to assign `true` during all following periods to the respective variables of these periods. Since there are $y$ periods, $y+1$ prototypes $p_0, \dots, p_y$ are necessary, where $p_i$ is the prototype of the $i$-th generation, and $p_0$ is the prototype of the initial neuron of the assignment-layer: $as$. Table 5.1 lists the intended behavior of these prototypes. As seen in this table, the prototypes $p_0$ and $p_1$ need rules implementing periods $1, \dots, y$, while $p_2$ only needs rules to implement periods $2, \dots, y$; $p_3$ only needs rules to implement periods $3, \dots, y$ etc., since no neuron of a prototype $p_i$ exists in an earlier period than $i$.

Naturally, spiking rules of the base system are retained for any prototype of the assignment-layer. During the computation phase, this ensures that all assignment neurons generated during the expansion phase behave exactly like the assignment neurons initially present in the base systems. However, this causes the implementation of periods by adding expanding and connecting rules to the new prototypes to be complex. The

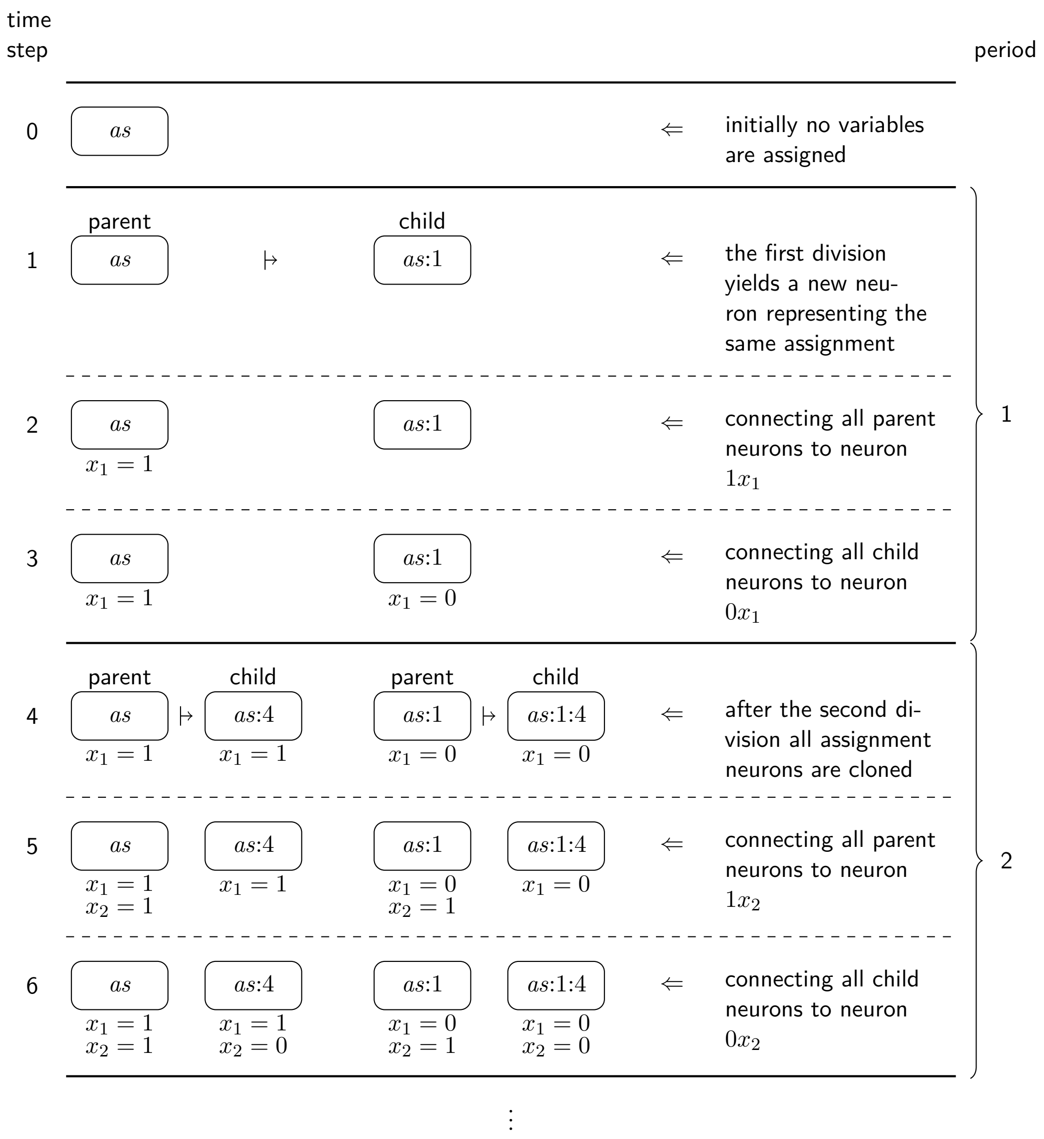

time step
period
0
$as$
initially no variables are assigned
1
parent
child
$as$
$as{:}1$
the first division yields a new neuron representing the same assignment
2
$x_1 = 1$
connecting all parent neurons to neuron $1x_1$
3
$x_1 = 0$
connecting all child neurons to neuron $0x_1$
4
$as{:}4$
$as{:}1{:}4$
after the second division all assignment neurons are cloned
5
$x_2 = 1$
connecting all parent neurons to neuron $1x_2$
6
$x_2 = 0$
connecting all child neurons to neuron $0x_2$
1
2

**Figure 5.4.:** *The strategy employed to create all assignment neurons.*

| | Neuron Prototype, Neurons with this Prototype and Behavior | | | | |
|---|---|---|---|---|---|
| | $[p_0]$ | $[p_1]$ | $[p_2]$ | $[p_3]$ | $\dots$ |
| Period $k$ | $as$ | $as{:}1$ | $as{:}4$, $as{:}1{:}4$ | $as{:}7$, $as{:}1{:}7$, $as{:}4{:}7$, $as{:}1{:}4{:}7$ | $\dots$ |
| 0 | – | – | – | – | $\dots$ |
| 1 | parent | child | – | – | $\dots$ |
| 2 | parent | parent | child | – | $\dots$ |
| 3 | parent | parent | parent | child | $\dots$ |
| 4 | parent | parent | parent | parent | $\dots$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |

| Legend of behavior: | | |
|---|---|---|
| | – | neuron does not exist or assigns no variable during the period |
| | parent | neuron assigns `true` to $x_k$ during the period $k$ |
| | child | neuron assigns `false` to $x_k$ during the period $k$ |

**Table 5.1.:** *The behavior of different prototypes of the assignment module.*

reason for this is that these additional expanding and connecting rules now may only match on uneven amounts of spikes which are greater than $2y-1$. Otherwise, they would conflict with the retained spiking rules. Figure 5.5 shows prototypes $p_0$ and $p_i$ for $1 \leq i \leq y$, retaining the spiking rules and implementing the periods without conflicts. The new rules shown in Figure 5.5 work in the following way:

The neuron $as$ has the prototype $p_0$. It applies $3 \cdot y$ rules ($y$ periods, 3 time steps each) during the expansion phase. Each such rule consumes 2 spikes. At the beginning of a period $k$, the neuron contains exactly $8y+5-6k$ spikes. Consequently, the neuron is initialized with $8y-1$ spikes for the first period. As first step of a period, an expanding rule $a^{8y+5-6k}/a^2 \to [p_k]$ is applied. Thereby, a child neuron is created with a prototype $[p_k]$ corresponding to the current period $k$. The next applicable rule for step two is $a^{8y+5-6k-2}/a^2 \to \mathrm{in}$. Since the input bridge neuron $1x_k$ executes an out-connecting rule in the exact same time step, a new synapse is created. Finally, the rule $a^{8y+5-6k-4}/a^2 \to a^0$ becomes applicable to conclude the period in the third step. This rule does nothing except for consuming two spikes. Now, $8y+5-6(k+1)$ spikes remain, starting the next period $k+1$.

Since the neuron $as$ with its prototype $p_0$ is initially present, it is always the parent neuron. The other prototypes $p_1, \dots, p_y$, however, belong to neurons created at the beginning of one of the periods. Since these are child neurons in this period, they must create a synapse to $0x_i$ instead of $1x_i$ during the period $i$ of their creation. A neuron with prototype $p_i$ is created in the first time step of period $i$ by its parent neuron. Therefore, it inherits the $8y+5-6i-2$ spikes the parent neuron has left at this point. In the following second time step of the period, the rule $a^{8y+5-6i-2}/a^2 \to a^0$ is applied. This rule has no effect except for consuming two spikes. At the same time step, the parent neuron applies a connecting rule and connects to the input bridge neuron $1x_i$. In the third time step, $a^{8y+5-6i-4}/a^2 \to \mathrm{in}$ is applicable. This creates a synapse to

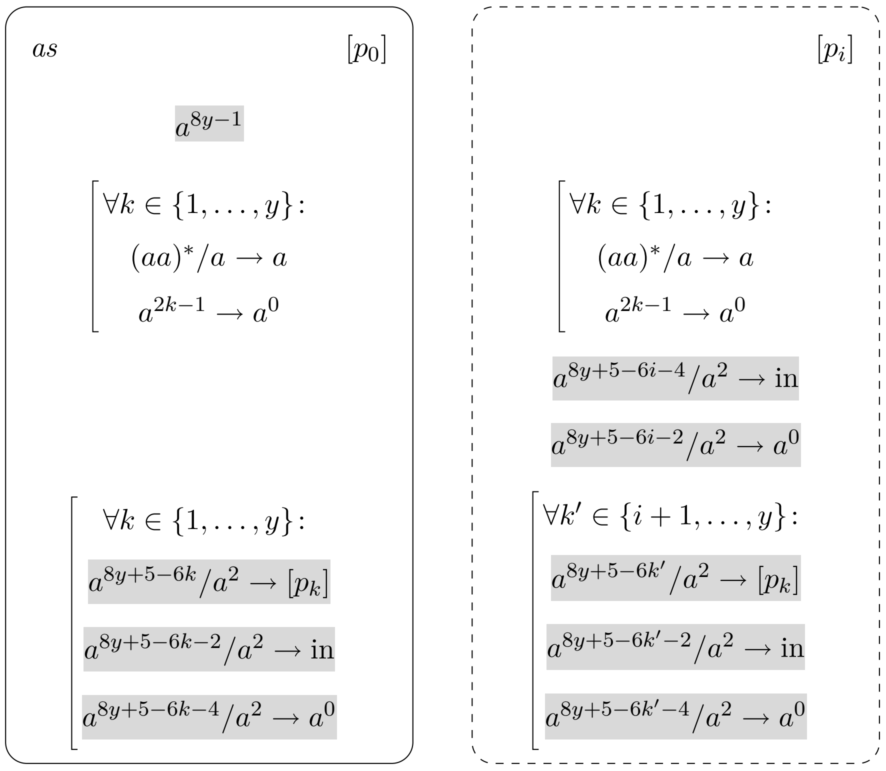

$as$
$[p_0]$
$a^{8y-1}$
$\forall k \in \{1,\ldots,y\}$:
$(aa)^*/a \to a$
$a^{2k-1} \to a^0$
$\forall k \in \{1,\ldots,y\}$:
$a^{8y+5-6k}/a^2 \to [p_k]$
$a^{8y+5-6k-2}/a^2 \to \mathrm{in}$
$a^{8y+5-6k-4}/a^2 \to a^0$
$[p_i]$
$\forall k \in \{1,\ldots,y\}$:
$(aa)^*/a \to a$
$a^{2k-1} \to a^0$
$a^{8y+5-6i-4}/a^2 \to \mathrm{in}$
$a^{8y+5-6i-2}/a^2 \to a^0$
$\forall k' \in \{i+1,\ldots,y\}$:
$a^{8y+5-6k'}/a^2 \to [p_k]$
$a^{8y+5-6k'-2}/a^2 \to \mathrm{in}$
$a^{8y+5-6k'-4}/a^2 \to a^0$

**Figure 5.5.:** *Modifications on assignment-layer prototypes $p_0$ and $p_i$ for $1 \leq i \leq y$. Normal font: retained from base system; gray highlight: additions and modifications*

the input bridge neuron $0x_i$ for the child neuron. Note that this works because $0x_i$ executes an out-connecting rule one time step after $1x_i$. Meanwhile, the parent neuron does nothing but consuming two spikes during this third time step. For further periods $k' \in \{i+1, \dots, y\}$, the prototype $p_i$ has the same rules as $p_0$, since it will be a parent neuron in these periods. There are no rules for periods less than $i$ necessary as explained before.

Since $as$ is initialized with $8y-1$ spikes, $(8y-1)-2(3y) = 2y-1$ spikes remain after execution of all $y$ periods. Similarly, $2y-1$ spikes remain in all other assignment neurons created during the expansion phase. In fact, it is easy to see that all assignment neurons contain the same amount of spikes in every time step of the expansion phase, since upon creation, they inherit the amount of spikes their parent neuron contains, and every assignment neuron consumes 2 spikes in every step of the expansion phase. At $2y-1$ spikes, the spiking rule $a^{2y-1} \to a^0$ (retained from the base system) deletes these remaining spikes. Therefore, all assignment neurons are in the same state as in the base system at the beginning of the computation phase (zero stored spikes and correct synaptic connections to the input bridge neurons).

Furthermore, none of the additional rules can be applied again, after the expansion phase finishes. This is guaranteed because an assignment neuron cannot collect more than $2y$ spikes at any time during the computation phase.

## 5.4. Assignment Module: Sat-Layer

The second layer of the assignment module is the sat-layer. The family of base systems defines exactly one *sat neuron* in this layer for each assignment neuron of the assignment-layer. Furthermore, each sat neuron is connected to its assignment neuron by a synapse. Initially, only one of these sat neurons (labeled $sat$) is present in the extended systems as seen in Figure 5.1. However, generating the missing sat neurons proves to be not complicated.

Recall that all assignment neurons are divided in the first step of each period. Dividing all sat neurons each time at the exact same time step yields one newly generated sat neuron for each newly generated assignment neuron. Since the division of neurons also clones all connected synapses by definition, the necessary synapse between new sat neuron and new assignment neuron is generated automatically in this step by cloning the existing synapse between their parents.

The implementation of this behavior is shown in Figure 5.6. All neurons of the sat-layer use the shown prototype. The initial neuron of the layer (labeled $sat$) is initialized with $m+y+1$ spikes. Therefore, the rule $a^{m+y+1}/a \to [sat]$ is applied in the first time step, which is also the first step of the first period. The neuron is divided and the newly created neuron has the prototype $sat$ as well. After a division in the first step of a period $k$, exactly $m+y+1-k$ spikes are left. Thus, the rule $a^{m+y+1-k}/a \to [sat]; 2$ is applied. The delay of this rule causes the neuron to be closed for steps 2 and 3 of the current period. The division therefore becomes effective in the first time step of the following period. The rules of the prototype are chosen such that this is repeated until a division occurred in every period (recall that the final period $y$ ends after time

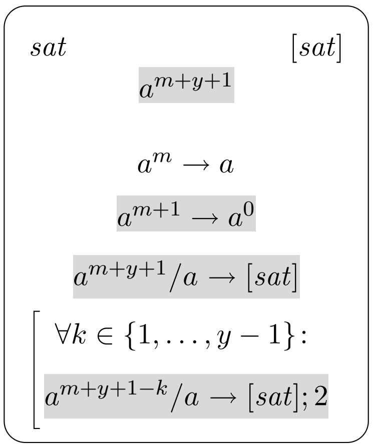


**Figure 5.6.:** *Modifications on the sat prototype. Normal font: retained from base system; gray highlight: additions and modifications.*

step $3 \cdot y$). Since all generated neurons of this layer inherit the same prototype $sat$ and the remaining amount of spikes of their parent, all existing neurons of the layer behave identical in every time step, and therefore are divided once in every period.

After the final period $y$, there are $m+1$ spikes left in every neuron of the layer. At this point, the rule $a^{m+1} \to a^0$ is applied removing all remaining spikes and therefore establishing the same state sat neurons are in initially in the base system. None of the additional rules are applied in future time steps again, since sat neurons cannot collect more than $m$ spikes during the computation phase. The only rule adopted from the base system is $a^m \to a$ which does not conflict with the added rules.

## 5.5. Quantification Module

The final module is the quantification module. Consider the situation shown in Figure 5.7: Between three neurons $n_1, n_2$ and $n_3$ there exist two synapses $(n_1, n_2)$ and $(n_3, n_2)$. To describe such a structure we say: The neuron $n_2$ *links* the neurons $n_1$ and $n_3$. Recall that the quantification module of a base system consists of the output neuron $(out)$ and $n$ neuron layers (named $\forall_1, \ldots, \forall_n$). Since there is only one output neuron in every base system, the neuron is simply retained for the extended systems and no division of the output neuron is necessary. Each of the $n$ neuron layers of the quantification module consists of neurons linking two neurons of the respective layer above. More specifically, for all $i \in \{1, \ldots, n\}$, two neurons of the $\forall_{i-1}$-layer are linked by a neuron of the $\forall_i$-layer if and only if the two neurons differ exclusively in the variable $x_i$ of their respectively represented variable assignments. In case of the $\forall_1$-layer, neurons of the sat-layer are linked, since there is no $\forall_0$-layer.

The fully expanded sat-layer consists of exponentially many neurons, and therefore exponentially many neurons are required for the fully expanded quantification module as well. This can also be observed in the base systems of Figure 3.4. To avoid an exponential amount of neurons in the initial workspace of the extended systems, each

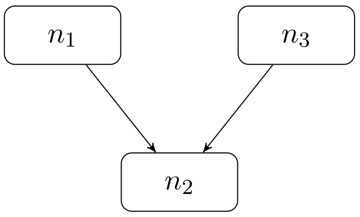


**Figure 5.7.:** *The neuron $n_2$ links the neurons $n_1$ and $n_3$*

layer is reduced to one neuron labeled $quan_i$ and a synapse from $quan_{i-1}$ to $quan_i$ for $1 \leq i \leq n$ (consider $quan_0$ as a synonym to $sat$, see Figure 5.1). Each of these neurons and their descendants will form one complete layer of the quantification module after the expansion phase. Furthermore, there is a synapse from the neuron $quan_n$ to the neuron $out$. In the following, the construction strategy for the $\forall_1$-layer during the expansion phase is explained. Afterwards, it is generalized to a strategy for any layer and finally implemented via expansion rules.

To "grow" the neurons and synapses of the $\forall_1$-layer, the behavior of the neurons of the sat-layer is exploited. Recall that in every period $i$ new assignment neurons are created. The new assignment neurons of every period inherit the assignments of the variables $x_1, \ldots, x_{i-1}$ from their parents (by means of synapse cloning). `False` is assigned to $x_i$ for the child neurons while `true` is assigned to $x_i$ for the parent neurons. Then `true` is assigned to every still undefined variable $x_{i+1}, \ldots, x_y$ during the future periods for all existing neurons. At the same time, a corresponding sat neuron is created with every new assignment neuron as covered in the last section. These new sat neurons represent the same variable assignments as the corresponding assignment neuron. This implies:

- At the end of the expansion phase, the initial neuron of the sat-layer (labeled $sat$) will represent the assignment of `true` to all variables. Its first child neuron (labeled $sat$:1) will represent the assignment of `false` to $x_1$ and `true` to all other variables. In the base systems, these neurons are linked by a neuron of the $\forall_1$-layer, since they differ only in the represented assignment to $x_1$. To grow this same structure within the extended systems, it is sufficient to not divide $quan_1$ in the first period (in which $sat$ is divided to create $sat$:1). The initially existing synapse from $sat$ to $quan_1$ is cloned to a synapse from $sat$:1 to $quan_1$. As a result, after period 1, all existing sat-layer neurons (that is $sat$ and $sat$:1) are properly linked by a $\forall_1$-layer neuron ($quan_1$).

- The represented variable assignments of two new sat neurons differ exclusively in the assignment of the variable $x_1$ if and only if their parent neurons already differ exclusively in that variable. Assume that all existing sat-layer neurons are properly linked by a $\forall_1$-layer neuron at the beginning of a period $i \geq 2$ (i. e. a period after the first period). Then, after the division of the neurons in the sat-layer, two newly created sat neurons need to be linked by a newly created $\forall_1$-layer neuron if and only if their parent neurons are already linked by an existing $\forall_1$-layer neuron.

> This can be achieved simply by dividing all existing $\forall_1$-layer neurons at the same time the sat-layer neurons are divided. Therefore, after the period, all existing sat-layer neurons are still properly linked by a $\forall_1$-layer neuron.

Using this leads to a simple strategy: The initial neuron of the $\forall_1$-layer is inactive until the first period (in which the sat neurons are divided for the first time) is finished. In all following periods, all existing neurons of the $\forall_1$-layer are divided at the same time step as the neurons of the sat-layer are divided.

As a result, neurons of the $\forall_1$-layer behave similar to neurons of the sat-layer. In each period after the first one, the represented variable assignments are copied. During the period, different values are assigned (or rather are inherited from the layer above) to the variable of the period for parent and child neurons. The represented variable assignments of two new $\forall_1$-layer neurons differ exclusively in the assignment of the variable $x_2$ if and only if their parent neurons already differ exclusively in that variable. Therefore, the same strategy can be used to generate the $\forall_2$-layer, except that the initial neuron of the $\forall_2$-layer is inactive until the second period (in which the $\forall_1$-layer neurons are divided for the first time) is finished.

Further generalized to an arbitrary layer, the resulting strategy is: The initial neuron of the $\forall_i$-layer is inactive until the period $i$—in which the layer above is divided for the first time—is finished. In all following periods, all neurons of the $\forall_i$-layer are divided in the first time step of the period. Figure 5.8 illustrates this.

The implementation of this behavior is shown in Figure 5.9. It employs $n$ different prototypes: $q_1, \dots, q_n$. That is one prototype $q_i$ for each of the $n$ initial neurons $quan_i$ of the $n$ $\forall_i$-layers. In addition to the inherited spiking rule $a^2 \to a$, each prototype defines one rule for each necessary division. An initial neuron $quan_i$ must be inactive until the period $i$ is finished. It must subsequently be divided in each of the $y - i$ periods left. Let $steps_i := y - i$. Then the prototype $q_i$ defines $steps_i$ expanding rules to implement this. Each such rule consumes one spike and creates another neuron of the same prototype. To avoid conflicts, the new rules may only match on three or more spikes. A neuron $quan_i$ (with prototype $q_i$) is initialized with $3 + steps_i$ spikes. Consequently, the rule $a^{3+steps_i}/a \to [q_i]; 3i$ is applicable in the first time step. The delay of $3i$ ensures that the neuron is closed for $i$ periods. In time step 1 of period $i+1$, the division becomes effective. After this initial division, $steps_i - 1$ further divisions are necessary. Let $k$ be the number of divisions already executed by the neuron. Then, $3 + steps_i - k$ spikes are left. The rule $a^{3+steps_i-k}/a \to [q_i]; 2$ is now applicable. Again the neuron is divided. The delay of 2 ensures that the division becomes effective in the first time step of the next period. After $steps_i$ divisions (and therefore at the end of the expansion phase), only three spikes are left. The additional rule $a^3 \to a^0$ deletes these remaining spikes at this point.

Since other neurons of the same layer inherit the prototype and the remaining spikes of their parent neuron upon creation, they behave identically to the initial neuron of the layer in every time step. Therefore, all neurons of the quantification module contain zero spikes at the end of the expansion phase. None of the additional rules are applied in future time steps again, since neurons of the quantification module cannot collect more than two spikes during the computation phase. Furthermore, the only retained

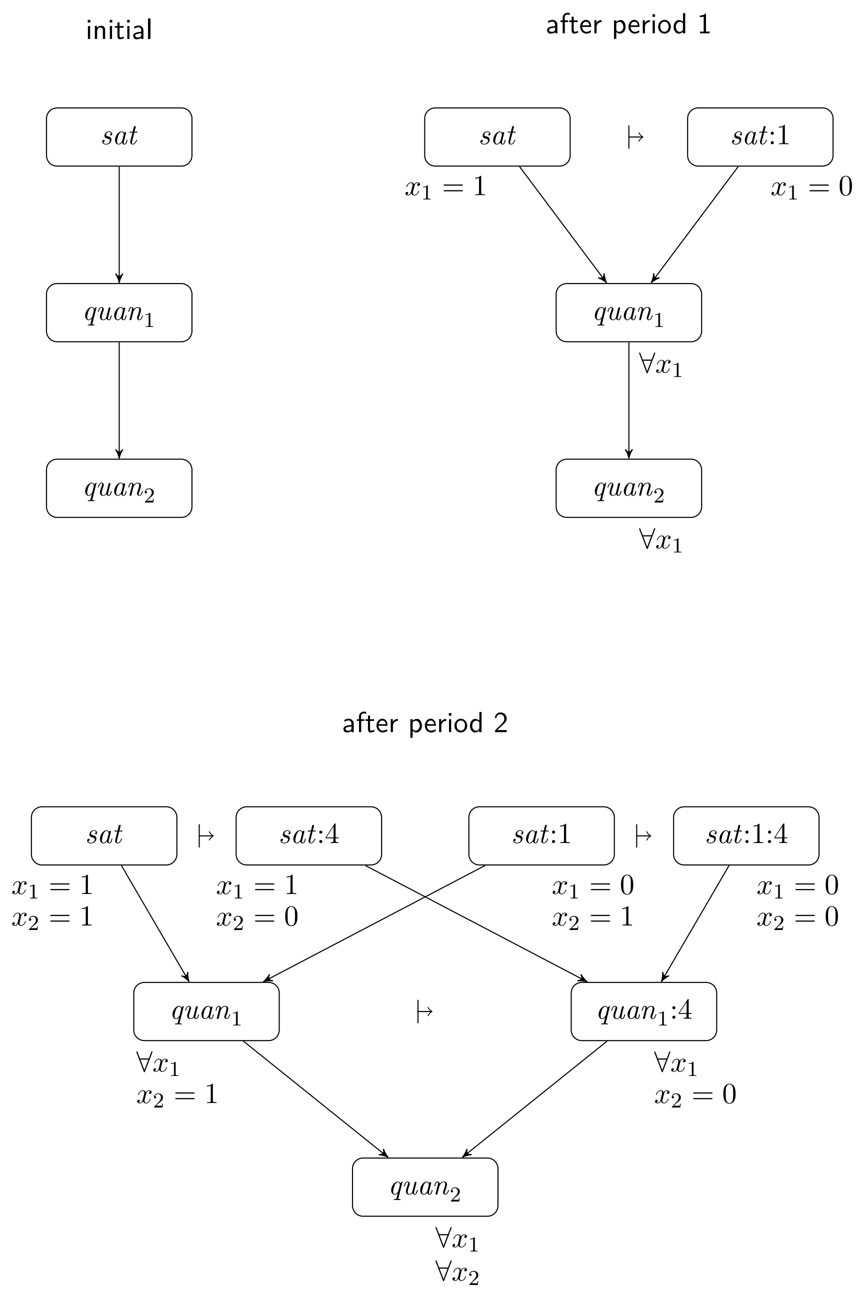


**Figure 5.8.:** *The strategy employed to generate the quantification module.*

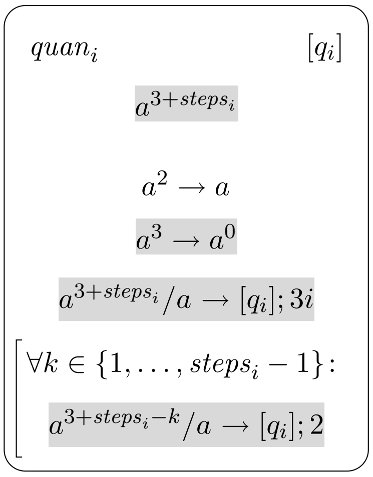

$quan_i$
$[q_i]$
$a^{3+steps_i}$
$a^2 \to a$
$a^3 \to a^0$
$a^{3+steps_i}/a \to [q_i]; 3i$
$\forall k \in \{1, \ldots, steps_i - 1\}:$
$a^{3+steps_i-k}/a \to [q_i]; 2$

**Figure 5.9.:** *Modifications on the prototypes $q_i$ for $1 \leq i \leq n$ and $steps_i = y - i$. Normal font: retained from base system; gray highlight: additions and modifications.*

rule of the base system ($a^2 \to a$) does not conflict with the additional rules.

The synapses of the base system between every neuron of the $\forall_n$-layer and the output neuron $out$ are automatically created. The reason is that on every division of neurons of the $\forall_n$-layer, all newly created neurons clone the already existing synapse between their parent neuron and the output neuron.

## 5.6. Properties

The improvement of properties of the ESN P systems solving QBF, as introduced in this chapter, compared to the properties of their base systems (the SN P systems of Chapter 3) is obvious by now. As rules are carefully chosen, the extended systems are still deterministic. The workspace of an ESN P system solving an instance of $QBF_{\mathrm{S}}(y = 2n, m)$ is reduced to $2y + 3$ neurons of the input module, $2y$ input bridge neurons, $2 + n$ neurons for the following layers (which were all reduced to one neuron) and the output neuron. Clearly, this is a polynomial amount of neurons with respect to the number of variables $y$ used. The number of time steps necessary is increased in comparison to the corresponding base system. The reason for this is the addition of the expansion phase. However, this expansion phase is finished after $3y$ time steps and the following computation phase is identical to the computation of the corresponding base system which is finished after $3y+n+6$. That results in a total of $(3+m)y+n+6$ time steps for the extended systems. Therefore, a decision is still reached in polynomial time. The size of the definition of an ESN P system solving an instance of $QBF_{\mathrm{S}}(y = 2n, m)$ is polynomial in $y$ as well. That means, the number of neurons, synapses and spikes in the initial workspace as well as the number of prototypes and rules defined within the system is polynomial in $y$. Therefore, the presented solution to QBF is an *uniform solution*. In comparison to the exp-uniform solution presented earlier, for any problem instance an ESN System of the family defined in this chapter can be built by a deterministic Turing machine in polynomial time with respect to the problem size.

The expansion phase of the extended systems has further interesting properties to analyze. Naturally, since the extended systems generate an exponential workspace from a polynomial workspace in only a polynomial number of steps, in at least one step an exponential number of neurons must be generated. The same is true for synapses. The production resources for new neurons and synapses are the already existing neurons which do produce in parallel. Therefore, it is of interest whether any neuron of the systems needs to create an exponential amount of new neurons or synapses within a single time step.

By definition of ESN P systems, a neuron cannot create more than one new neuron per time step.

The number of synapses that need to be cloned by a neuron division is not bounded by the definition. However, in the family of ESN P systems given in Figure 3.4, all neurons which are divided in the expansion phase have a number of connected synapses bounded by $y+1$ at any given point in time: Neurons of the assignment-layer define at most $y$ different synapses to input bridge neurons and one synapse to the corresponding sat-layer neuron. Neurons of the sat-layer have a synapse to the respective assignment neuron and another synapse to a neuron of the first quantification layer. Neurons of the quantification module have two synapses to link two neurons of the previous layer and one synapse to a neuron of the subsequent layer.

Synapses can also be created using connecting rules. Connecting rules are only used to grow synapses between input bridge neurons and assignment neurons. An assignment neuron never generates more than one synapse within a single time step using connecting rules. Since there is an exponential number of assignment neurons, the input bridge neurons do create an exponential amount of synapses within a single time step. Improving on this property is an open problem. A solution may require the extension of ESN P systems by additional types of rules. See Chapter 7 for some proposals. However, one can argue that still less synapses are created than neurons are involved in creating them. That is, $i$ assignment neurons create $i$ synapses to one single input bridge neuron. Therefore, the maximum number of synapses created in a single time step (via cloning and via connecting rules) is actually polynomially bounded by the number of existing neurons.

# 6. Similar Extensions

There are two notable similar approaches to extend SN P systems in order to avoid exponential initial workspaces. On the one hand, *SN P systems with budding rules* extend SN P systems by so called budding rules which enable the construction of neurons as well as synapses. On the other hand, *SN P systems with neuron division and budding* modify the budding rules of the former approach and introduce new neuron division rules as well as a synapse dictionary. In both cases the basic mechanics of SN P systems are kept but extended by new types of rules. In this chapter, both systems are introduced shortly and are subsequently compared to the proposed ESN P systems of this diploma thesis.

## 6.1. SN P Systems with Budding Rules

SN P systems with budding rules were introduced in [WIP09]. SN P systems are extended to develop new synapses and neurons based on the environment of the existing neurons. To achieve this, neurons are uniquely identified by a label. These labels are subsequently used by the newly introduced *budding rules*. These rules have the following syntax: $x[\ ]_i \rightarrow y[\ ]_j$. While $i$ and $j$ are neuron labels, $x$ and $y$ describe synapses. The synapse described by $x$ may be any synapse of any direction connected to $i$ (it is also possible to specify $\lambda$, which means no synapse at all). The synapse described by $y$ must be a synapse of any direction between $i$ and $j$.

Application of such a rule is possible if and only if the neuron $i$ is connected to just the synapse $x$ (no less and no more). Clearly, the application of budding rules indeed depends on the environment of a neuron. As effect of the application of a rule $x[\ ]_i \rightarrow y[\ ]_j$, the neuron $j$ and the synapse $y$ between $i$ and $j$ are created. If $j$ already exists, only the synapse is created. If the synapse already exists, too, the rule has no effect. In every time step all applicable budding rules of a neuron are applied in parallel.

Firing rules are retained from SN P systems and work as usual. This means, in particular, a single neuron applies at most one spiking rule per time step. It is worth noting that budding rules are independent of firing rules, since their application does not depend on spikes and they also do not consume spikes. Because of that independence, it is relatively easy to reduce the workspace of an existing SN P system. Rules added to re-expand during computation can never conflict with retained rules.

The disadvantage of this independence is that the application of budding rules cannot be influenced by the input of the system. Therefore, SN P systems with budding rules are not interesting for applications outside of eliminating pre-compiled workspaces. A second problem arises from the fact that a single budding rule can not create more than one neuron and one synapse during the entire computation. Therefore, at least an exponential number of budding rules must be present in the definition of a system generating an exponential workspace.

## 6.2. SN P Systems with Neuron Division and Budding

SN P systems with neuron division and budding were introduced very recently in [PPPJ09]. In this extension, a *synapse directory* is used to create synapses without involving spikes. Additionally, two new types of rules are introduced to create new neurons and synapses based on spikes contained in the neurons of the system.

Neurons are represented by labels again. However, these labels are not unique anymore in this extension (i. e. two neurons with the same label may exist). The synapse dictionary specifies which labels are connected by a synapse and in which direction. It has two functions. First, the synapses defined by the dictionary are used as initial synapses between the initial neurons. Second, whenever new neurons are created, synapses are added to the system according to the labels of the new neurons and the information in the dictionary.

The first new type of rule is the *neuron division rule*. This type of rule uses the following syntax: $[E]_i \rightarrow [\ ]_j \mid\mid [\ ]_k$ where $E$ is a regular expression over $\{a\}$ and $i, j, k$ are neuron labels. Such a neuron division rule is only applicable on a neuron with label $i$ and only if the contained amount of spikes matches the regular expression $E$ (i. e. $a^s \in \mathrm{L}(E)$ where $s$ is the amount of contained spikes). After application, the applying neuron $i$ is replaced by two new neurons $j$ and $k$. Synapses connected to $i$ are inherited by both created neurons. For example, a synapse $(i, h)$ for any neuron $h$ is replaced by two synapses $(j, h)$ and $(k, h)$. The same is true for synapses directed to $i$, as for example $(h, i)$ for any neuron $h$ which is replaced by $(h, j)$ and $(h, k)$[1]. Furthermore, all spikes of $i$ are consumed in the process and $j$ and $k$ are both initialized with zero spikes.

The second new type of rule is the *neuron budding rule*. Although it is derived from the budding rules introduced in the previous section, it works differently. The syntax used for neuron budding rules is the following: $[E]_i \rightarrow [\ ]_i/[\ ]_j$ where $E$ is a regular expression over $\{a\}$ and $i, j$ are neuron labels. Again, such a rule is only applicable on a neuron with label $i$ and only if the contained amount of spikes matches the regular expression $E$. After application, a new neuron $j$ is created as well as a synapse $(i, j)$ connecting the applying neuron with the newly created neuron. While the neuron $i$ keeps all incoming synapses, all outgoing synapses are transferred to the neuron $j$. As an example consider the following situation: There exist three neurons $i$, $h$ and $k$. Furthermore, two synapses $(h, i)$ and $(i, k)$ exist. After applying a neuron budding rule $[E]_i \rightarrow [\ ]_i/[\ ]_j$, a new neuron $j$ and a synapse $(i, j)$ is created. While $(h, i)$ is kept, $(i, k)$ is replaced by $(j, k)$. In the process all spikes contained in $i$ are consumed and both $i$ and $j$ contain zero spikes.

Firing rules are retained from SN P systems and work as usual. Regardless of type, every neuron applies at most one rule per time step. If more then one rule is applicable,

[1]It is worth mentioning that no behavior is defined in a case where two neurons $i$ and $m$ both apply a neuron division rule and a synapse $(i, m)$ exists.

one rule is chosen nondeterministically.

SN P systems with neuron division and budding as well as ESN P systems are both motivated by cell division. Both define a neuron division rule (called expanding rule in ESN P systems) that creates a new neuron and clones connected synapses. However, there are also differences. One disadvantage of SN P systems with neuron division and budding is the consumption of all spikes in the process. Often a combination of auxiliary neurons and the synapse dictionary is necessary to initialize the created neuron with spikes.

There is a second disadvantage of SN P systems with neuron division and budding. Despite two additional types of rules and the synapse dictionary, it not possible to create synapses between existing neurons based on spikes. Both new types of rules involve the creation of a new neuron. The dictionary is not influenced by spikes and can only create synapses for neurons which were created in the same time step. This limits the versatility of these systems for applications outside of reducing initial workspaces (see Chapter 7 for some suggestions of other applications).

The strong point of SN P systems with neuron division and budding are budding rules. Multiple applications of these rules allow to exchange a single neuron of the system for a subnet of neurons without cloning the synapses from or to neighboring neurons. This cannot be done in ESN P systems. Therefore, a type of rule with similar functionality is suggested in Chapter 7 as a possible future extension to ESN P systems.

## 6.3. Comparison to ESN P Systems

ESN P systems and both similar extensions (introduced in the two previous sections) are characterized by different biologically motivated rules to create neurons, synapses or both during computations. Application of these rules is either based on the number of spikes contained in a neuron or on the environment of a neuron. Obviously, systems using the former type of rules can be influenced in their growth by input. Therefore, this type of rule is called *dynamic*, and in the other case it is called *static*. Table 6.1 shows a classification of all introduced systems based on their types of rules.

Since ESN P systems use dynamic rules exclusively, they are very versatile. But, in the shown application of ESN P systems to reduce an otherwise exponentially large workspace, all systems were carefully designed such that the input does not influence the growth of the system. To measure the quality of ESN P systems for this application, other properties (as already shown in Section 5.6) should be compared. No applications of SN P systems with neuron division and budding or SN P systems with budding rules to solve a **PSPACE**-complete problem deterministically and in a polynomial number of time steps from a polynomial initial workspace were shown yet. However, applications solving the **NP**-complete problem SAT were shown for both systems. For this reason these applications are compared to the application of ESN P systems to solve QBF, as shown in Chapter 5. Table 6.2 shows the results. Note that the ESN P system shows better or at least equal properties while solving a harder problem.

| Model | computationally complete | creates additional neurons | creates additional synapses |
|---|---|---|---|
| SN P systems | yes | no | no |
| SN P systems +B | yes | static | static |
| SN P systems +ND +B | yes | dynamic | static |
| ESN P systems | yes | dynamic | dynamic |

**Table 6.1.:** *Properties of the four introduced models: SN P systems, SN P systems with budding rules (+B), SN P systems with neuron division and budding (+ND +B) and ESN P systems.*

| Applied Model | Problem | Steps | Initial Size | Desc Size | New Neurons |
|---|---|---|---|---|---|
| SN P systems | `QBF` | p | exp | exp | 0 |
| SN P systems +B | `SAT` | p | p | exp | p |
| SN P systems +ND +B | `SAT` | p | p | p | 2 |
| ESN P systems | `QBF` | p | p | p | 1 |

| Legend: | | |
|---|---|---|
| | Problem | the problem which is to be solved by an application of the model. |
| | Steps | time steps necessary to solve an instance of the problem |
| | Initial Size | number of neurons and synapses in the initial workspace of the applied model |
| | Desc Size | number of neurons, rules and spikes in the definition of the applied model |
| | New Neurons | maximum number of neurons created per time step per existing neuron. |
| | p | polynomial with respect to the problem size |
| | exp | exponential with respect to the problem size |

**Table 6.2.:** *Properties of the application of the four introduced models to solve `QBF` or `SAT` deterministically and in a polynomial number of time steps.*

# 7. Conclusions and Remarks

In this diploma thesis, SN P systems were extended to so called ESN P systems by two new biologically motivated types of rules: extension rules and connecting rules. The meaningfulness of the additional types of rules was demonstrated by an application: A family of ESN P systems was given, solving the **PSPACE**-complete QBF problem with better properties than possible with SN P systems. With respect to the problem size, SN P systems need an exponential pre-compiled workspace to solve QBF deterministically and in a polynomial number of time steps. ESN P systems only need a polynomial initial workspace. To be more specific, the introduced family of ESN P systems showed the following properties with respect to the input size:

- A computation finishes in a polynomial number of time steps.
- There is a polynomial number of neurons, synapses and spikes in the initial workspace.
- There is a polynomial number of prototypes and rules defined within the system.
- Every existing neuron creates no more than one neuron per time step, and
- the maximum number of synapses created in a single time step is polynomially bounded by the number of existing neurons.

There are two notable other extensions to SN P systems aimed to reduce the initial workspace: SN P systems with budding rules and SN P systems with neuron division and budding. However, for these, no application to solve a **PSPACE**-complete problem deterministically and in a polynomial number of time steps from a polynomial initial workspace was shown so far. Therefore, it is an open problem if these extensions can be used to solve any **PSPACE**-complete problem with similar properties to the application of ESN P systems shown in this diploma thesis. In case of SN P systems with neuron division and budding, this seems very likely.

Of all three compared extensions, only in ESN P systems the creation of both, new neurons with synapses and new synapses between existing neurons, is directly influenced by the spikes in the system and therefore by the input of the system. This versatility of ESN P systems allows applications beyond reducing workspaces of SN P systems. What follows are some suggestions for possible applications and open problems to be investigated in future works:

- It seems to be possible to implement a single ESN P system solving all instances of QBF instead of implementing a parametrized family of systems as done in this diploma thesis. For example, the assignment layer can be implemented by only two prototypes which execute an infinite cycle over the three time steps of a period. This cycle is stopped only by incoming spikes from the input module, after the number of variables is read from the input. Naturally, other parts of the system must be modified as well.

- Which exponential workspaces can be generated efficiently using the introduced connecting rules and expanding rules?
- The possible creation of new neurons during the computation based on the input could allow a simple simulation of a Turing machine by an ESN P system. Each used cell of the tape of the simulated Turing machine can be represented by a neuron. The number of spikes contained in that neuron encodes the symbol in that cell of the tape. If a previously unused cell of the tape is written to, a new neuron is created to represent that cell.
- Are SN P systems, only using connecting rules and expanding rules, computationally complete? Obviously, spiketrains can not be used as output and something else based on the structure of the net must be found.

Some of these applications might only be feasible by further extending ESN P systems. To further adapt properties of biological neurons, some of the following features may be introduced:

- A second type of rule implementing cell division. While the expanding rules of ESN P systems create one new neuron which clones synapses and remaining spikes of its parent, the new type of rule replaces the applying neuron by two new neurons, splitting up spikes and synapses between these two. That is, one child neuron inherits the incoming synapses of its parent while the other neuron inherits the outgoing synapses of its parent. Such a "replacing" rule would allow to replace a neuron in a system by a subnet of neurons without cloning all synapses to and from the neighboring neurons.
- Neurons of ESN P systems live forever. It might be reasonable to define a lifetime for neurons. This lifetime might be based on either time steps of the system or on a maximum number of divisions (aging neurons).
- Similarly, existing synapses in ESN P systems cannot be removed again. A type of rule to accomplish might work similarly to connecting rules.

# A. Formal Implementation of the ESN P Systems solving QBF

The family of ESN P systems with parameters $n$ and $m$ solving corresponding instances of $QBF_{\mathrm{S}}(y = 2n, m)$, as constructed in Chapter 5, is formally defined as follows:

$$\mathit{\Pi}_{n,m} = (O = \{a\}, P_n, N_n, ru_{n,m}, spn_{n,m}, proto_n, syn_n, in, out)$$

where

$$\begin{aligned} y &\coloneqq 2n \ , \\ H &\coloneqq \{(aa)^*/a \to a\} \\ &\quad \cup \{a^{2k-1} \to a^0 \mid k \in \{1, \ldots, y\}\} \end{aligned}$$

- prototypes:

$$\begin{aligned} P_n = \{&in,\ start,\ stop,\ sel,\ fsel, \\ &x_1, \ldots, x_y,\ 1x_1, \ldots, 1x_y,\ 0x_1, \ldots, 0x_y, \\ &p_0, \ldots, p_y,\ sat,\ q_1, \ldots, q_n,\ out\} \ , \end{aligned}$$

- neuron labels:

$$\begin{aligned} N_n = \{&in,\ start,\ stop,\ sel_1, \ldots, sel_y, \\ &x_1, \ldots, x_y,\ 1x_1, \ldots, 1x_y,\ 0x_1, \ldots, 0x_y, \\ &as,\ sat,\ quan_1, \ldots, quan_n,\ out\} \ , \end{aligned}$$

- prototype definitions:

$$\begin{aligned} ru_{n,m}(in) &= \{a^2 \to a^2,\ a \to a,\ a^3 \to a^0; 3y\} \ , \\ ru_{n,m}(start) &= \{a^2 \to a^2; 3y\} \ , \\ ru_{n,m}(stop) &= \{a \to a; my + 3y\} \ , \\ ru_{n,m}(sel) &= \{a^2 \to a^2\} \ , \\ ru_{n,m}(fsel) &= \{a^2 \to a^2,\ a^3 \to a^0\} \ , \end{aligned}$$

$$\left[\begin{aligned} &\forall i \in \{1, \ldots, y\}\colon \\ &\qquad ru_{n,m}(x_i) &&= \{a \to a^0,\ a^2 \to a^0, \\ &&&\qquad a^3 \to a^3; y-i,\ a^4 \to a^4; y-i\}, \\ &\qquad ru_{n,m}(1x_i) &&= \{a^3 \to a^2,\ a^4 \to a^0,\ a \to \text{out}; 3i-2\}\ , \\ &\qquad ru_{n,m}(0x_i) &&= \{a^3 \to a^0,\ a^4 \to a^2,\ a \to \text{out}; 3i-1\}\ , \end{aligned}\right.$$

$$\begin{aligned} ru_{n,m}(p_0) = H \\ &\cup \{a^{8y+5-6k}/a^2 \to [p_k],\ a^{8y+5-6k-2}/a^2 \to \text{in}, \\ &\quad a^{8y+5-6k-4}/a^2 \to a^0 \mid k \in \{1, \ldots, y\}\}\ , \end{aligned}$$

$$\left[\begin{aligned} &\forall i \in \{1, \ldots, y\}\colon \\ &\qquad ru_{n,m}(p_i) &&= H \\ &&&\quad \cup \{a^{8y+5-6i-2}/a^2 \to a^0,\ a^{8y+5-6i-4}/a^2 \to \text{in}\} \\ &&&\quad \cup \{a^{8y+5-6k}/a^2 \to [p_k],\ a^{8y+5-6k-2}/a^2 \to \text{in}, \\ &&&\qquad a^{8y+5-6k-4}/a^2 \to a^0 \mid k \in \{i+1, \ldots, y\}\}\ , \end{aligned}\right.$$

$$\begin{aligned} ru_{n,m}(sat) &= \{a^m \to a,\ a^{m+1} \to a^0\} \\ &\cup \{a^{m+y+1}/a \to [sat]\} \\ &\cup \{a^{m+y+1-k}/a \to [sat]; 2 \mid k \in \{1, \ldots, y-1\}\}\ , \end{aligned}$$

$$\left[\begin{aligned} &\forall i \in \{1, \ldots, n\}\colon \\ &\qquad ru_{n,m}(q_i) &&= \{a^2 \to a,\ a^3 \to a^0\} \\ &&&\quad \cup \{a^{3+steps_i}/a \to [q_i]; 3i\} \\ &&&\quad \cup \{a^{3+steps_i-k}/a \to [q_i]; 2 \mid k \in \{1, \ldots, steps_i - 1\}\} \end{aligned}\right.$$

where $steps_i = y - i$ ,

$$ru_{n,m}(out) = \{(aa)^*/a \to a\}\ ,$$

- neuron attributes (initial spikes and prototype):

$$(spn_{n,m}(in),\ proto_n(in)) = (3,\ in)\ ,$$

$$(spn_{n,m}(start),\ proto_n(start)) = (2,\ start)\ ,$$

$$(spn_{n,m}(stop),\ proto_n(stop)) = (1,\ stop)\ ,$$

$$\left[\begin{aligned} &\forall i \in \{1, \ldots, y-1\}\colon \\ &(spn_{n,m}(sel_i),\ proto_n(sel_i)) = (0,\ sel)\ , \end{aligned}\right.$$

$$(spn_{n,m}(sel_y),\ proto_n(sel_y)) = (0,\ fsel)\ ,$$

$$\left[\begin{array}{ll}\forall i \in \{1,\dots,y\}: & \\ (spn_{n,m}(x_i),\ proto_n(x_i)) & = (0,\ x_i)\ , \\ (spn_{n,m}(0x_i),\ proto_n(0x_i)) & = (1,\ 0x_i)\ , \\ (spn_{n,m}(1x_i),\ proto_n(1x_i)) & = (1,\ 1x_i)\ ,\end{array}\right.$$

$$(spn_{n,m}(as),\ proto_n(as)) = (8y-1,\ p_0)\ ,$$

$$(spn_{n,m}(sat),\ proto_n(sat)) = (m+y+1,\ sat)\ ,$$

$$\left[\begin{array}{ll}\forall i \in \{1,\dots,n\}: & \\ (spn_{n,m}(quan_i),\ proto_n(quan_i)) & = (3+steps,\ q_i)\end{array}\right.$$

$$\text{where } steps_i = y - i\ ,$$

$$(spn_{n,m}(out),\ proto_n(out)) = (0,\ out)\ ,$$

- synapses:

$$\begin{aligned} syn_n = {} & \{(in,\ x_k) \mid k \in \{1,\dots,y\}\} \\ & \cup \{(sel_{k-1},\ sel_k) \mid k \in \{2,\dots,y\}\} \\ & \cup \{(sel_y,\ sel_1),\ (start,\ sel_1),\ (stop,\ sel_y)\} \\ & \cup \{(sel_k,\ x_k) \mid k \in \{1,\dots,y\}\} \\ & \cup \{(x_k,\ ix_k) \mid i \in \{0,1\},\ k \in \{1,\dots,y\}\} \\ & \cup \{(as,\ sat),\ (sat, quan_1),\ (quan_n,\ out)\} \\ & \cup \{(quan_{k-1},\ quan_k) \mid k \in \{2,\dots,n\}\}. \end{aligned}$$

# B. Expansion Phase of an ESN P System Solving an Instance of QBF

In the following Figures B.1–B.8, the expansion phase of an ESN P system solving instances of $QBF_{\mathrm{S}}(2, 2)$ is demonstrated. Note that in the shown configurations, rules are already scheduled for the current time step. That means, a rule was already chosen for application where possible (and therefore marked in the graphical representation) but not yet executed. Newly generated neurons of the previous step are bold. Changes in the active schedule or amount of stored spikes are marked by a grey background.

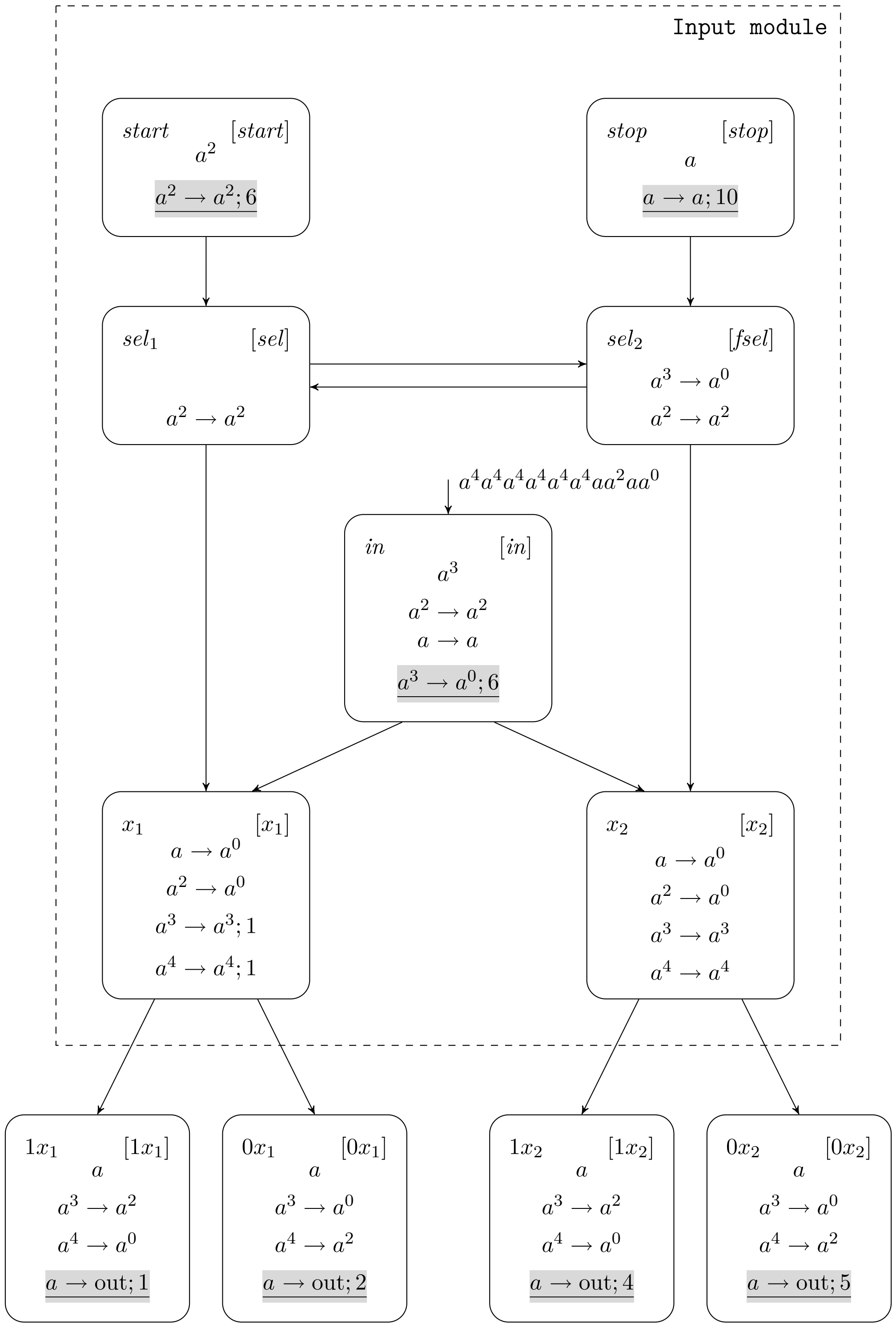

Input module
$start$ $[start]$ $a^2$ $a^2 \to a^2; 6$
$stop$ $[stop]$ $a$ $a \to a; 10$
$sel_1$ $[sel]$ $a^2 \to a^2$
$sel_2$ $[fsel]$ $a^3 \to a^0$ $a^2 \to a^2$
$a^4a^4a^4a^4a^4a^4aa^2aa^0$
$in$ $[in]$ $a^3$ $a^2 \to a^2$ $a \to a$ $a^3 \to a^0; 6$
$x_1$ $[x_1]$ $a \to a^0$ $a^2 \to a^0$ $a^3 \to a^3; 1$ $a^4 \to a^4; 1$
$x_2$ $[x_2]$ $a \to a^0$ $a^2 \to a^0$ $a^3 \to a^3$ $a^4 \to a^4$
$1x_1$ $[1x_1]$ $a$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 1$
$0x_1$ $[0x_1]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 2$
$1x_2$ $[1x_2]$ $a$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 4$
$0x_2$ $[0x_2]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 5$

**Figure B.1.:** *Initial configuration of the input module for $QBF_S(2n, m)$ for $n = 1$ and $m = 2$. As input example the spiketrain $a^4a^4a^4a^4a^4a^4aa^2aa^0$ is chosen which encodes the $QBF_S(2, 2)$ formula $\forall x_1 \exists x_2((x_1 \vee \neg x_2) \wedge (x_1))$. Rules are already selected (highlighted in gray).*

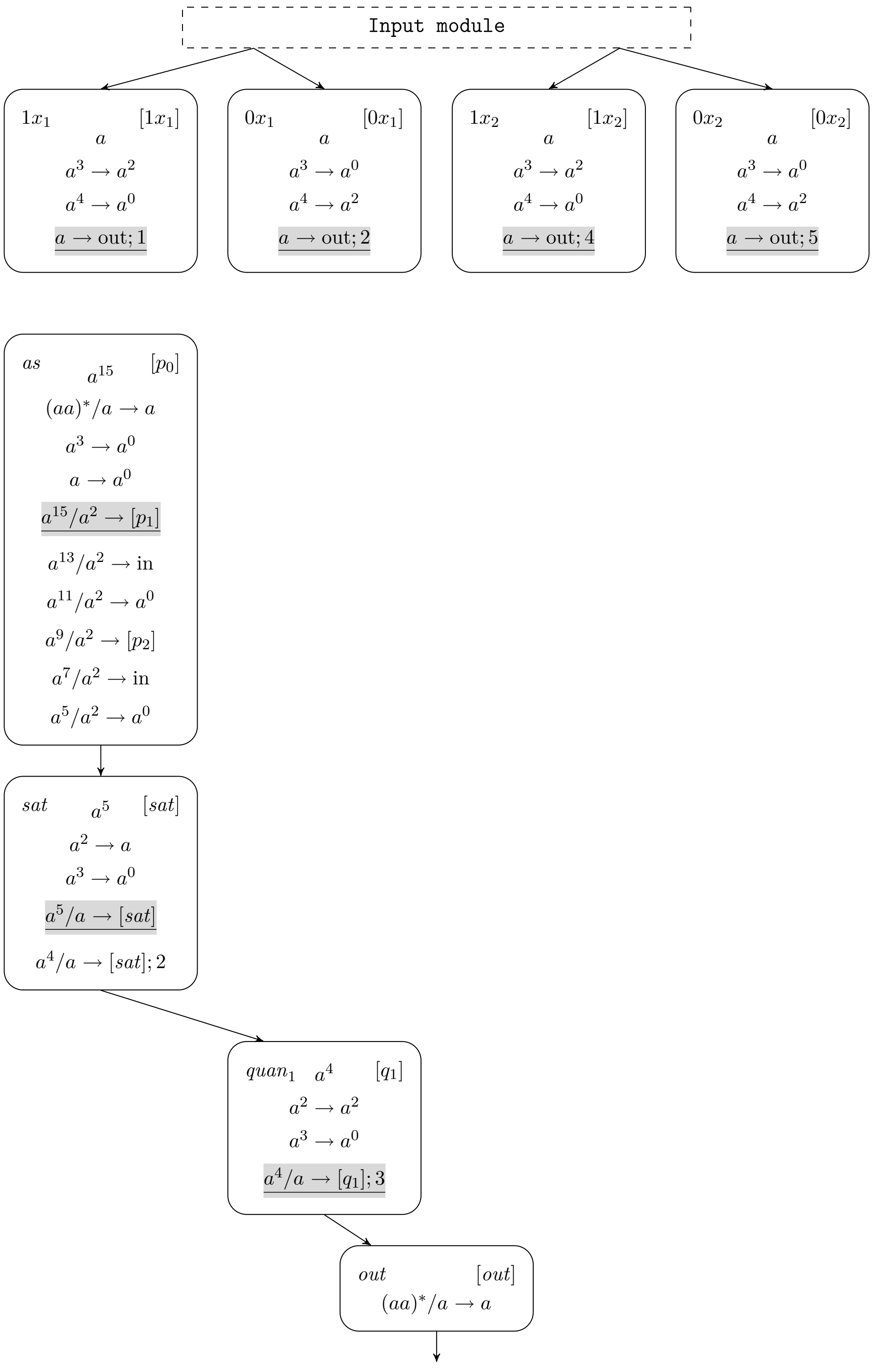

Input module
$1x_1$ $[1x_1]$ $a$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 1$
$0x_1$ $[0x_1]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 2$
$1x_2$ $[1x_2]$ $a$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 4$
$0x_2$ $[0x_2]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 5$
$as$ $a^{15}$ $[p_0]$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{15}/a^2 \to [p_1]$ $a^{13}/a^2 \to \text{in}$ $a^{11}/a^2 \to a^0$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $a^5/a^2 \to a^0$
$sat$ $a^5$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$quan_1$ $a^4$ $[q_1]$ $a^2 \to a^2$ $a^3 \to a^0$ $a^4/a \to [q_1]; 3$
$out$ $[out]$ $(aa)^*/a \to a$

**Figure B.2.:** *Initial configuration for $QBF_{\mathrm{S}}(2n, m)$ with $n = 1$ and $m = 2$ after selecting rules for the next time step (rescheduled). Changes and additions in bold or highlighted in gray.*

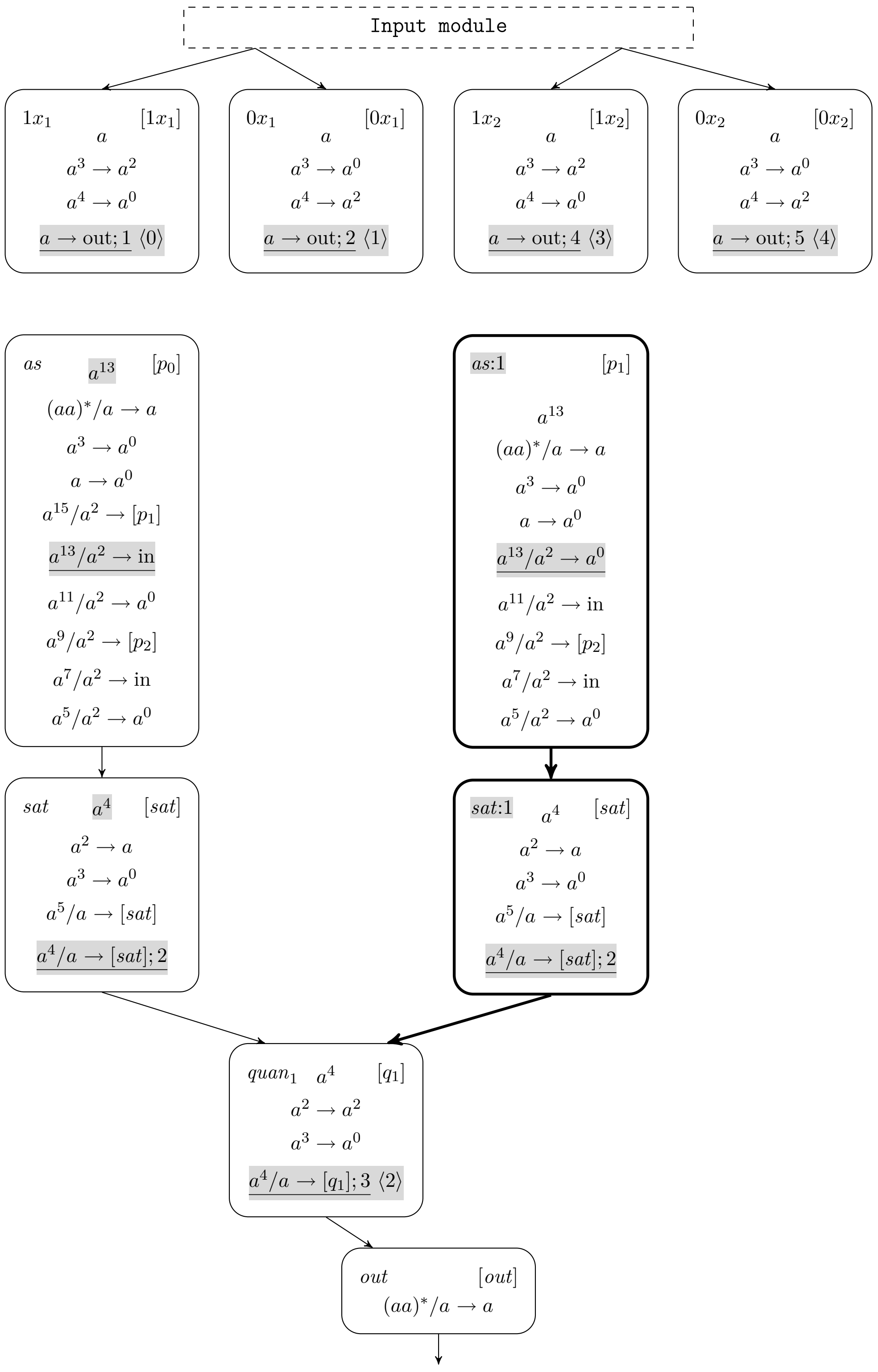

Input module
$1x_1$ $[1x_1]$ $a$ $a^3 \to a^2$ $a^4 \to a^0$ $\underline{a \to \text{out}; 1}$ $\langle 0 \rangle$
$0x_1$ $[0x_1]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $\underline{a \to \text{out}; 2}$ $\langle 1 \rangle$
$1x_2$ $[1x_2]$ $a$ $a^3 \to a^2$ $a^4 \to a^0$ $\underline{a \to \text{out}; 4}$ $\langle 3 \rangle$
$0x_2$ $[0x_2]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $\underline{a \to \text{out}; 5}$ $\langle 4 \rangle$
$as$ $a^{13}$ $[p_0]$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{15}/a^2 \to [p_1]$ $\underline{a^{13}/a^2 \to \text{in}}$ $a^{11}/a^2 \to a^0$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $a^5/a^2 \to a^0$
$as$:1 $[p_1]$ $a^{13}$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $\underline{a^{13}/a^2 \to a^0}$ $a^{11}/a^2 \to \text{in}$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $a^5/a^2 \to a^0$
$sat$ $a^4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $\underline{a^4/a \to [sat]; 2}$
$sat$:1 $a^4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $\underline{a^4/a \to [sat]; 2}$
$quan_1$ $a^4$ $[q_1]$ $a^2 \to a^2$ $a^3 \to a^0$ $\underline{a^4/a \to [q_1]; 3}$ $\langle 2 \rangle$
$out$ $[out]$ $(aa)^*/a \to a$

**Figure B.3.:** *Configuration for $QBF_S(2n, m)$ with $n = 1$ and $m = 2$ after one step and selecting rules for the next time step. Changes and additions in bold or highlighted in gray.*

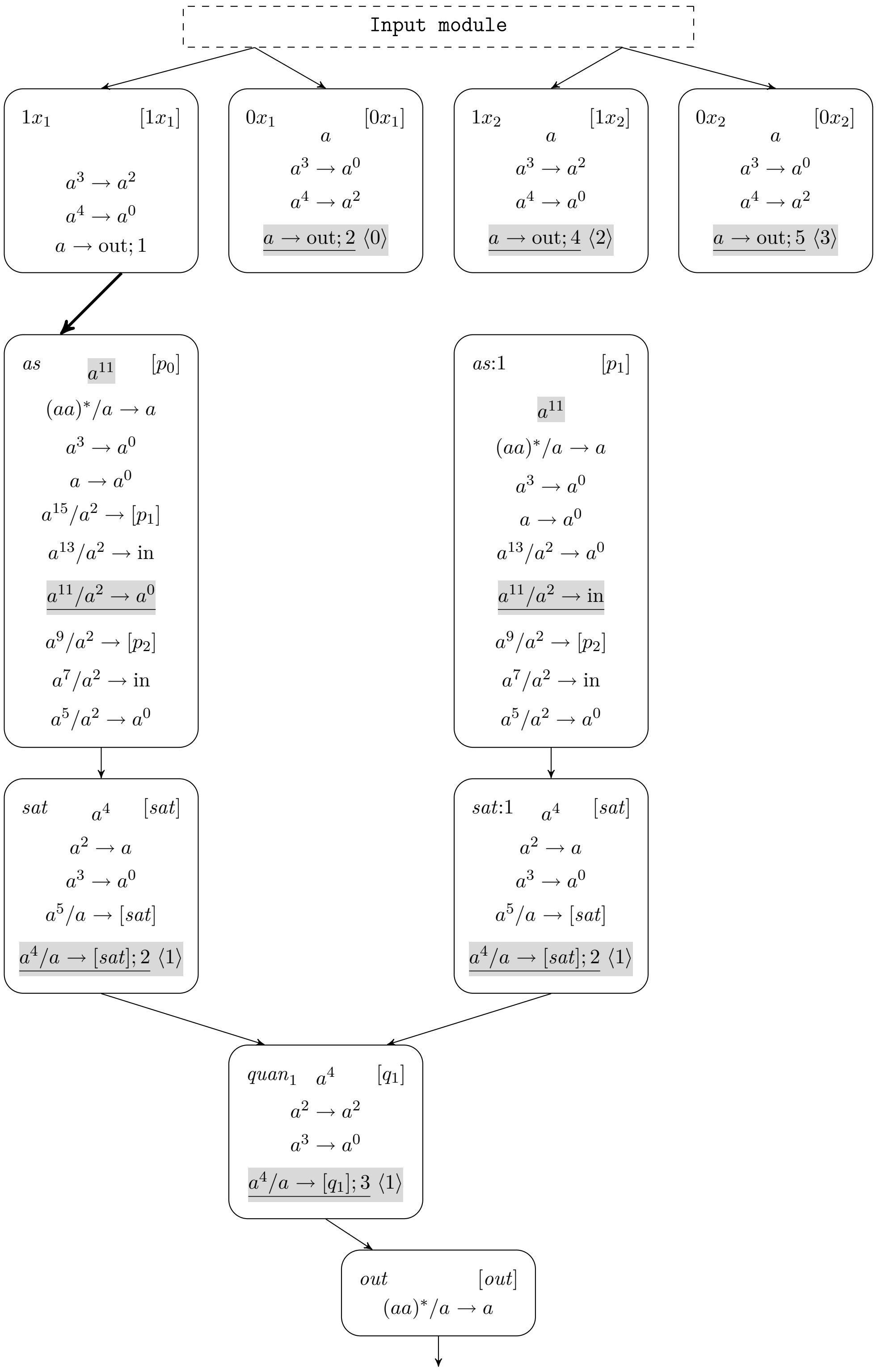

Input module
$1x_1$ $[1x_1]$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 1$
$0x_1$ $[0x_1]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 2\ \langle 0 \rangle$
$1x_2$ $[1x_2]$ $a$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 4\ \langle 2 \rangle$
$0x_2$ $[0x_2]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 5\ \langle 3 \rangle$
$as$ $a^{11}$ $[p_0]$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{15}/a^2 \to [p_1]$ $a^{13}/a^2 \to \text{in}$ $a^{11}/a^2 \to a^0$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $a^5/a^2 \to a^0$
$as{:}1$ $[p_1]$ $a^{11}$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{13}/a^2 \to a^0$ $a^{11}/a^2 \to \text{in}$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $a^5/a^2 \to a^0$
$sat$ $a^4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2\ \langle 1 \rangle$
$sat{:}1$ $a^4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2\ \langle 1 \rangle$
$quan_1$ $a^4$ $[q_1]$ $a^2 \to a^2$ $a^3 \to a^0$ $a^4/a \to [q_1]; 3\ \langle 1 \rangle$
$out$ $[out]$ $(aa)^*/a \to a$

**Figure B.4.:** *Configuration for $QBF_{\mathrm{S}}(2n, m)$ with $n = 1$ and $m = 2$ after two steps and selecting rules for the next time step. Changes and additions in bold or highlighted in gray.*

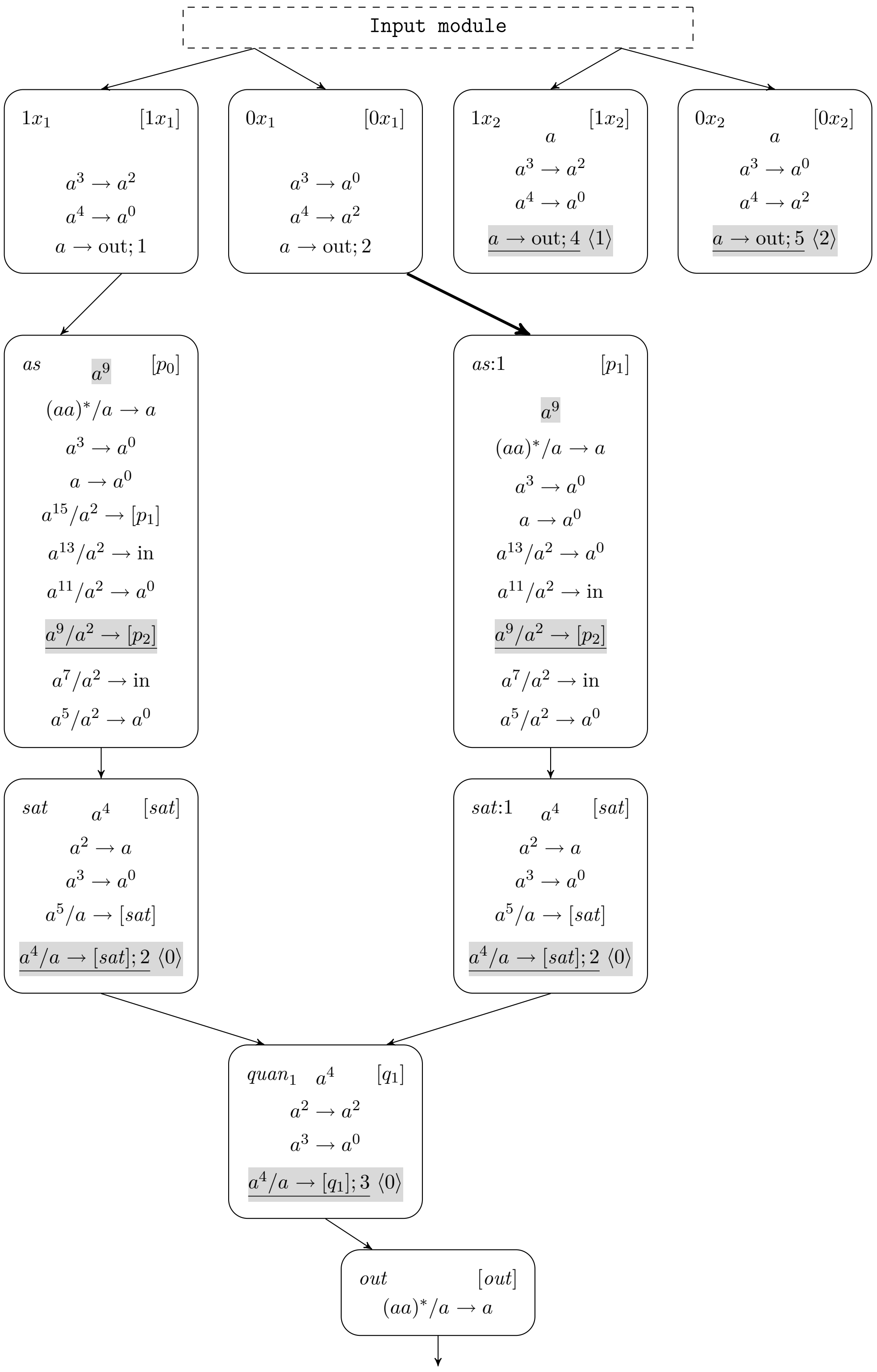


**Figure B.5.:** *Configuration for $QBF_S(2n, m)$ with $n = 1$ and $m = 2$ after three steps and selecting rules for the next time step. Changes and additions in bold or highlighted in gray.*

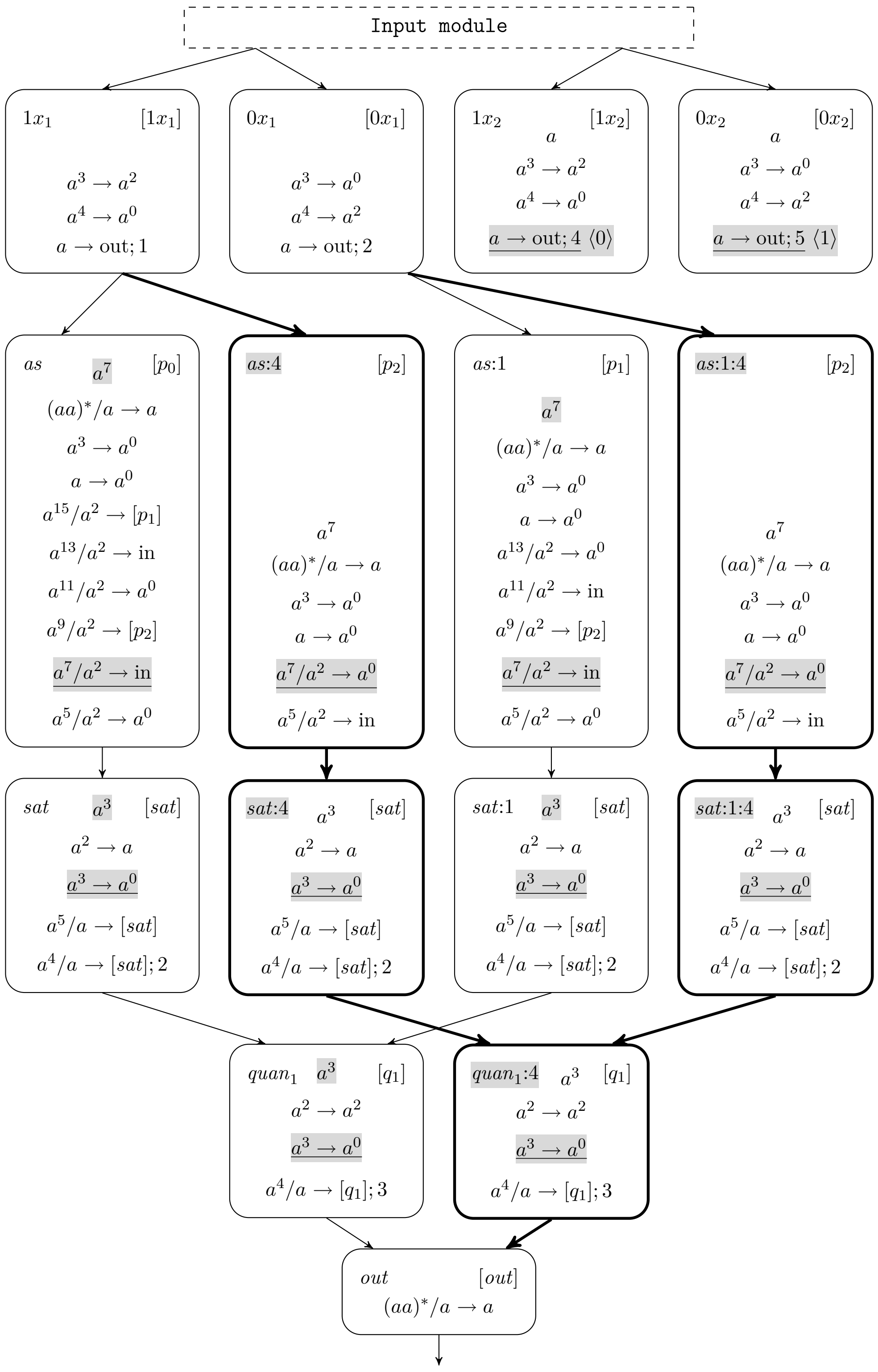
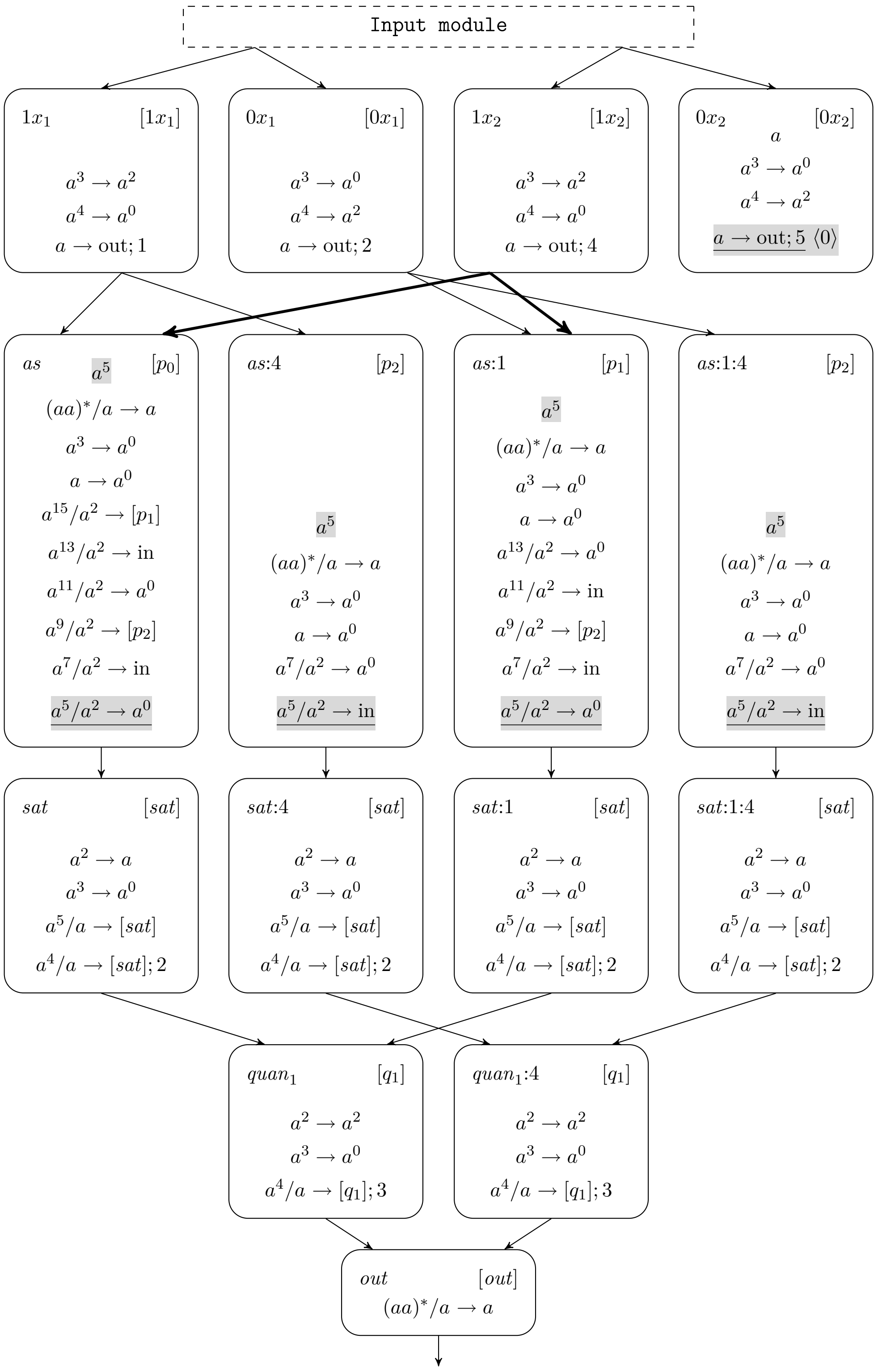

Input module
1x1 [1x1] a^3 → a^2 a^4 → a^0 a → out; 1
0x1 [0x1] a^3 → a^0 a^4 → a^2 a → out; 2
1x2 [1x2] a a^3 → a^2 a^4 → a^0 a → out; 4 ⟨0⟩
0x2 [0x2] a a^3 → a^0 a^4 → a^2 a → out; 5 ⟨1⟩
as a^7 [p0] (aa)*/a → a a^3 → a^0 a → a^0 a^15/a^2 → [p1] a^13/a^2 → in a^11/a^2 → a^0 a^9/a^2 → [p2] a^7/a^2 → in a^5/a^2 → a^0
as:4 [p2] a^7 (aa)*/a → a a^3 → a^0 a → a^0 a^7/a^2 → a^0 a^5/a^2 → in
as:1 [p1] a^7 (aa)*/a → a a^3 → a^0 a → a^0 a^13/a^2 → a^0 a^11/a^2 → in a^9/a^2 → [p2] a^7/a^2 → in a^5/a^2 → a^0
as:1:4 [p2] a^7 (aa)*/a → a a^3 → a^0 a → a^0 a^7/a^2 → a^0 a^5/a^2 → in
sat a^3 [sat] a^2 → a a^3 → a^0 a^5/a → [sat] a^4/a → [sat]; 2
sat:4 a^3 [sat] a^2 → a a^3 → a^0 a^5/a → [sat] a^4/a → [sat]; 2
sat:1 a^3 [sat] a^2 → a a^3 → a^0 a^5/a → [sat] a^4/a → [sat]; 2
sat:1:4 a^3 [sat] a^2 → a a^3 → a^0 a^5/a → [sat] a^4/a → [sat]; 2
quan1 a^3 [q1] a^2 → a^2 a^3 → a^0 a^4/a → [q1]; 3
quan1:4 a^3 [q1] a^2 → a^2 a^3 → a^0 a^4/a → [q1]; 3
out [out] (aa)*/a → a

**Figure B.6.:** *Configuration for $QBF_{\mathrm{S}}(2n, m)$ with $n = 1$ and $m = 2$ after four steps and selecting rules for the next time step. Changes and additions in bold or highlighted in gray.*



Input module
$1x_1$ $[1x_1]$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 1$
$0x_1$ $[0x_1]$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 2$
$1x_2$ $[1x_2]$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 4$
$0x_2$ $[0x_2]$ $a$ $a^3 \to a^0$ $a^4 \to a^2$ $\underline{a \to \text{out}; 5}$ $\langle 0 \rangle$
$as$ $a^5$ $[p_0]$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{15}/a^2 \to [p_1]$ $a^{13}/a^2 \to \text{in}$ $a^{11}/a^2 \to a^0$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $\underline{a^5/a^2 \to a^0}$
$as{:}4$ $[p_2]$ $a^5$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^7/a^2 \to a^0$ $\underline{a^5/a^2 \to \text{in}}$
$as{:}1$ $[p_1]$ $a^5$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{13}/a^2 \to a^0$ $a^{11}/a^2 \to \text{in}$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $\underline{a^5/a^2 \to a^0}$
$as{:}1{:}4$ $[p_2]$ $a^5$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^7/a^2 \to a^0$ $\underline{a^5/a^2 \to \text{in}}$
$sat$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$sat{:}4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$sat{:}1$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$sat{:}1{:}4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$quan_1$ $[q_1]$ $a^2 \to a^2$ $a^3 \to a^0$ $a^4/a \to [q_1]; 3$
$quan_1{:}4$ $[q_1]$ $a^2 \to a^2$ $a^3 \to a^0$ $a^4/a \to [q_1]; 3$
$out$ $[out]$ $(aa)^*/a \to a$

**Figure B.7.:** *Configuration for $QBF_{\mathrm{S}}(2n, m)$ with $n = 1$ and $m = 2$ after five steps and selecting rules for the next time step. Changes and additions in bold or highlighted in gray.*

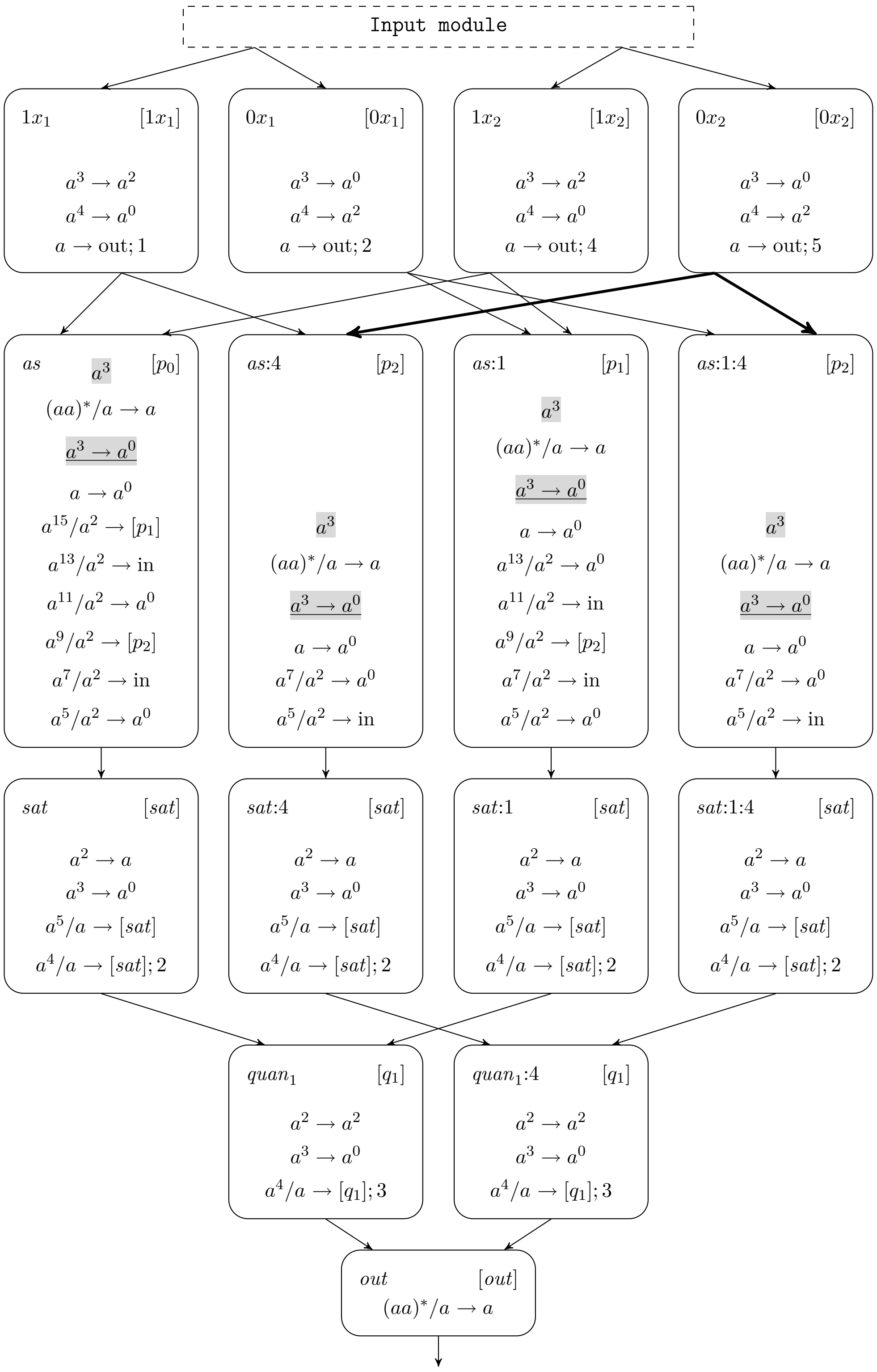

Input module
$1x_1$ $[1x_1]$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 1$
$0x_1$ $[0x_1]$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 2$
$1x_2$ $[1x_2]$ $a^3 \to a^2$ $a^4 \to a^0$ $a \to \text{out}; 4$
$0x_2$ $[0x_2]$ $a^3 \to a^0$ $a^4 \to a^2$ $a \to \text{out}; 5$
$as$ $a^3$ $[p_0]$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{15}/a^2 \to [p_1]$ $a^{13}/a^2 \to \text{in}$ $a^{11}/a^2 \to a^0$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $a^5/a^2 \to a^0$
$as{:}4$ $[p_2]$ $a^3$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^7/a^2 \to a^0$ $a^5/a^2 \to \text{in}$
$as{:}1$ $[p_1]$ $a^3$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^{13}/a^2 \to a^0$ $a^{11}/a^2 \to \text{in}$ $a^9/a^2 \to [p_2]$ $a^7/a^2 \to \text{in}$ $a^5/a^2 \to a^0$
$as{:}1{:}4$ $[p_2]$ $a^3$ $(aa)^*/a \to a$ $a^3 \to a^0$ $a \to a^0$ $a^7/a^2 \to a^0$ $a^5/a^2 \to \text{in}$
$sat$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$sat{:}4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$sat{:}1$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$sat{:}1{:}4$ $[sat]$ $a^2 \to a$ $a^3 \to a^0$ $a^5/a \to [sat]$ $a^4/a \to [sat]; 2$
$quan_1$ $[q_1]$ $a^2 \to a^2$ $a^3 \to a^0$ $a^4/a \to [q_1]; 3$
$quan_1{:}4$ $[q_1]$ $a^2 \to a^2$ $a^3 \to a^0$ $a^4/a \to [q_1]; 3$
$out$ $[out]$ $(aa)^*/a \to a$

**Figure B.8.:** *Configuration for $QBF_{\mathrm{S}}(2n, m)$ with $n = 1$ and $m = 2$ after six steps and selecting rules for the next time step. Changes and additions in bold or highlighted in gray.*

# List of Figures

# List of Tables

# Bibliography

[CEI+08] Matteo Cavaliere, Omer Egecioglu, Oscar Ibarra, Mihai Ionescu, Gheorghe Păun, and Sara Woodworth. Asynchronous spiking neural p systems: Decidability and undecidability. In Max Garzon and Hao Yan, editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 246–255. Springer Berlin / Heidelberg, 2008.

[CII+08] Haiming Chen, Mihai Ionescu, Tseren-Onolt Ishdorj, Andrei Păun, Gheorghe Păun, and Mario Pérez-Jiménez. Spiking neural p systems with extended rules: universality and languages. *Natural Computing*, 7:147–166, 2008. 10.1007/s11047-006-9024-6.

[GAPRPS08] Marc García-Arnau, David Pérez, Alfonso Rodríguez-Patón, and Petr Sosík. On the power of elementary features in spiking neural p systems. *Natural Computing*, 7:471–483, 2008. 10.1007/s11047-008-9082-z.

[Ihr98] Thomas Ihringer. *Diskrete Mathematik. Eine Einführung in Theorie und Anwendungen*. Teubner B.G. GmbH, 12 1998.

[IL08] Tseren-Onolt Ishdorj and Alberto Leporati. Uniform solutions to sat and 3-sat by spiking neural p systems with pre-computed resources. *Natural Computing*, 7:519–534, 2008. 10.1007/s11047-008-9081-0.

[ILP+10] Tseren-Onolt Ishdorj, Alberto Leporati, Linqiang Pan, Xiangxiang Zeng, and Xingyi Zhang. Deterministic solutions to qsat and q3sat by spiking neural p systems with pre-computed resources. *Theor. Comput. Sci.*, 411:2345–2358, May 2010.

[IPY06] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural p systems. *Fundam. Inf.*, 71:279–308, February 2006.

[IW07] Oscar Ibarra and Sara Woodworth. Spiking neural p systems: Some characterizations. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *Fundamentals of Computation Theory*, volume 4639 of *Lecture Notes in Computer Science*, pages 23–37. Springer Berlin / Heidelberg, 2007.

[LMZ+09] Alberto Leporati, Giancarlo Mauri, Claudio Zandron, Gheorghe Păun, and Mario Pérez-Jiménez. Uniform solutions to sat and subset sum by spiking neural p systems. *Natural Computing*, 8:681–702, 2009. 10.1007/s11047-008-9091-y.

[Nea08] Turlough Neary. On the computational complexity of spiking neural p systems. In *Proceedings of the 7th international conference on Unconventional Computing*, UC '08, pages 189–205, Berlin, Heidelberg, 2008. Springer-Verlag.

[Pau02] Gheorghe Paun. *Membrane Computing: An Introduction (Natural Computing Series Natural Computing)*. Springer, Berlin, 1 edition, 12 2002.

[PP07] Andrei Paun and Gheorghe Paun. Small universal spiking neural p systems. *Biosystems*, 90(1):48 – 60, 2007.

[PPPJ09] Linqiang Pan, Gheorghe Paun, and Mario J. Pérez-Jiménez. Spiking neural p systems with neuron division and budding. *7th Brainstorming Week on Membrane Computing*, II:151–168, 02/02/2009 2009.

[PRS98] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *DNA Computing: New Computing Paradigms (Texts in Theoretical Computer Science. An EATCS Series)*. Springer, 9 1998.

[Sch10] Uwe Schöning. *Logic for Computer Scientists (Modern Birkhauser Classics)*. Birkhäuser Boston, reprint of the 1989 ed. edition, 6 2010.

[WIP09] Jun Wang, Tseren-Onolt Ishdorj, and Linqiang Pan. About the efficiency of spiking neural p systems. *7th Brainstorming Week on Membrane Computing*, II:235–252, 02/02/2009 2009.

[WP10] Ingo Wegener and R. Pruim. *Complexity Theory*. Springer Berlin Heidelberg, 1 2010.