Karlsruhe Institute of Technology

Institute for Program Structures
and Data Organization (IPD)

Professor Dr.-Ing. Snelting

Qualitative Modelling of Biological Signalling Pathways using SAT-solving in Prolog

Bachelor Thesis by

Jan-Martin Knorr

at the Department of Informatics

Reviewer: Prof. Dr.-Ing. Gregor Snelting

Advisors: Dr. Olaf Klinke (DKFZ Heidelberg)
Dipl.-Math. Dipl.-Inform. Joachim Breitner

Duration: July 1, 2013 – October 31, 2013

Abstract

This work introduces and investigates a boolean modelling approach for signalling pathways of biological cells. Processes occuring inside one cell are abstracted in the model, where boolean formulae capture the semantics of the presented models. Considering the formulae as a satisfiability instance, its satisfying variable assignments correspond to situations where all modelled processes of the cell are in equilibrium. A Prolog implementation is presented that translates a model into its describing boolean formulae and solves the resulting satisfiability problem.

The major profit of the modelling approach presented here is the verification of circumstances and dependencies presumed in a cell and stated in a model. This verification is possible for cells in equilibrium. There are also limited opportunities for simulating dynamic behaviour of cells, although other approaches are preferable for this task.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text, and have followed the rules of the KIT for upholding good scientific practise.

Karlsruhe, 31st October 2013

.....

(**Jan-Martin Knorr**)

Contents

Chapter 1

Introduction

Signal transductions inside a biological cell are similar to computerised information processing [19]. Regarded as a computational unit, a cell accepts certain inputs and reacts with certain outputs. The processing inside the cell that creates the output for an input is described by signalling pathways.

The inputs are exterior stimuli detected by receptors on the cell wall [18]: "Upon intercellular communication or cellular stress response, the cell senses extracellular signals. They are transformed into intracellular signals and sequences of reactions. Different external changes or events may stimulate signaling. Typical stimuli are hormones, pheromones, heat, cold, light, osmotic pressure, appearance, or concentration change of substances like glucose or K^+, Ca^+, or cAMP."[12, p. 92]

The signals of these exterior stimuli trigger signalling pathways inside the cell. "A biological pathway is a series of actions among molecules in a cell that leads to a certain product or a change in a cell."[18] Thus, pathways control the cellular responses and determine the outputs for the inputs.

As the output "a pathway can trigger the assembly of new molecules, such as a fat or protein. Pathways can also turn genes on and off, or spur a cell to move."[18]

The different components of the cell or events occuring in the cell are referred to as species. This expression needs to be distinguished from biological or chemical species. The meaning of species in this work is more general and allows more abstraction than the standard meaning of chemical species. Modelling the behaviour of and interactions between these species helps to understand how information is transferred in signalling pathways.

Two basic approaches for modelling pathways can be distinguished [21]. In the boolean approach, which is pursued in this work, the species are assumed as either activated (appearing in a high concentration) or not activated (appearing in a low concentration) in the cell. Typically, the change of a species' concentration under a stimulus describes

a sigmoid curve[1] [23, p. 3]. After determining a threshold value, this sigmoid curve can be approximated by a stepfunction, so the simplification of a binary decision is justifiable [ib.]. Furthermore this binary distinction of concentrations appears naturally in experiments where the concentrations of some species vary and the measured result is compared against a reference measurement. Then all species existing in a significantly lower concentration than in the reference are assumed to be not activated and all others to be activated.

A more detailed approach is provided by continuous dynamic models of the kinetic laws between species. Typically differential equations are used. Partial differential equations model the species' location inside the cell over time. For modelling temporal changes only, ordinary differential equations suffice [12, p. 42].

General-purpose computing environments like MATLAB[2] or Mathematica[3] can solve the differential equations of continuous dynamic models numerically [12, p. 527]. Boolean models can be investigated using model checkers such as BIOCHAM [3], PRISM [13] and Antelope [2] which verify that a model meets certain properties.

Although the differential equations approach is more accurate, boolean models are prefered in some cases. Besides the fact that dynamical systems require more computational effort [21, 24], it is not easy to experimentally determine the kinetic parameters needed for differential equations [24]. Moreover, this additional effort is unnecessary when only the cell's states are considered [4]. A more detailed overview and further possibilities for modelling signalling pathways is given in [11].

This work shows how boolean pathway models that claim dependencies between certain species can be verified by measuring the species of a cell which is in equilibrium. The models are translated into satisfiability (SAT) instances that are solved by a SAT solver. Then the measured species can be compared against the solutions of the SAT instances. This idea already appeared in [24].

As the examined model may be erroneous, a measurement may not fit to the satisfying assignments of the SAT instance. In this case a sub-model can be found that claims less dependencies than the initial model, but is compatible with the measurement. Max-SAT solvers can be used to find a sub-model with as few omitted dependencies as possible.

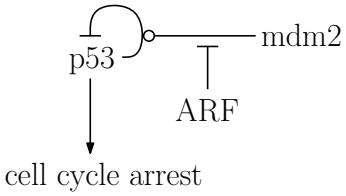
Chapter 2 motivates and formalises the pathway models and points out how these models can be translated into SAT instances. Next, Chapter 3 describes how a pathway model can be encoded and transformed into a corresponding SAT instance using Prolog. Chapter 4 describes Max-SAT solving for these instances in Prolog. Chapter 5 puts the results from the preceeding sections to use by presenting a programme that computes all possible equilibrium situations of a cell according to its model. Chapter 6 exhibits some limitations of the boolean approach when modelling dynamic behaviour inside the cell.

[1] A sigmoid curve is an S-shaped curve. The graph of the hyperbolic tangent describes a sigmoid curve.
[2] `http://www.mathworks.de/products/matlab/`
[3] `http://www.wolfram.com/mathematica/`

Chapter 2

Qualitative Pathway Model

Example 1 *The following simplified p53-pathway will serve as a running example.*



2.1 Motivation and Modelling goals

In this work's modelling approach of signalling pathways, the components of a biological cell and the interactions between these components are all considered as *actions*, processes constantly occurring in the cell. This reflects the fact that in biology everything changes continuously [12, p. 10]. An action describes either a single entity, called *species*, or a composition of a species and another action that indicates the relation between the two. Species denote atomic model elements whereas *composite actions* denote their dependencies.

A pathway model, which is completely defined by its actions, can be illustrated as a directed graph as shown in the example above. Each vertex depicts a species. In the simplified p53-pathway the species are $p53$, $mdm2$, ARF and *cell cycle arrest*. The edges connect the vertices analogous to the composite actions. As the protein p53 triggers the cell cycle arrest, a corresponding arrow points from $p53$ to *cell cycle arrest*. Since composite actions may affect other composite actions, edges pointing at other edges are allowed here.

It depends on the research interests which components and interactions are abstracted as actions in the model [19]. "An abstraction – a mapping from a real-world domain to a mathematical domain – highlights some essential properties while ignoring other,

complicating ones."[19] Actions can cover different levels of abstraction. An action may model that the concentration of a certain molecule is high or low inside the cell or that one enzyme has a catalyzing effect on a certain reaction. But actions can also represent abstract events like 'cell division' or 'cell cycle arrest' involving many different types of molecules. Nevertheless these events may themselves be investigated further using another more detailed model.

In this qualitative model the continuous biological processes are simplified by approximating boolean model actions. Every action is either activated or not activated and can also be regarded as turned on resp. off. For a species representing the concentration of some molecule this means for instance whether the concentration is relatively high or low. A composite action which is activated implies that the action takes effect.

The following actions are supported:

- s: The process denoted by the species s is activated or not activated.
- $on \multimap T$: Some process (not further modelled) turns on T.
- $off \multimap T$: Some process (not further modelled) turns off T.
- $s \to T$: s enhances T.
- $s \dashv T$: s inhibits T.
- $s \multimap T$: s controls T enhancing.
- $s \multimap\!\!\bullet\ T$: s controls T inhibiting.
- $s -\!\!? T$: s unspecifically affects T. Either $s \to T$ or $s \dashv T$ holds.

Here s is always a species and T may be any action.

First of all, the model can simulate enhancing ($\to$) and inhibiting ($\dashv$) actions. It means that if the species on the left-hand side of the action is on, then the action on the right-hand side is turned on resp. off. This only holds once the enhancing or inhibiting action itself is on, as composite actions can be inactive as well.

Whenever it is unknown whether an enhancing or an inhibiting action applies[1], this uncertainty can be expressed using a dummy action $s -\!\!? T$ which means that either $s \to T$ or $s \dashv T$ holds.

Additionally the controlling actions $\multimap$ and $\multimap\!\!\bullet$ are supported. The action $s \multimap T$ means that once this enhancing control action is on, s and T are either both on or both off. With this control action, complexes consisting of various components can be simulated. In this case a complex is controlled by each of its components. In contrast the action $s \multimap\!\!\bullet\ T$ expresses that s or T is on while the other is off, provided that the inhibiting control action is on. The control actions are conducive to the modelling of not only signalling

[1]The protein interactions in the STRING database [22] are not explicitly enhancing or inhibiting.

pathways but also metabolism. The chemical transformations referred to as metabolism gain energy or construct components for the cell [12, p. 83 f.]. While signalling pathways means transfer of information, metabolism means transfer of mass [12, p. 92].

Turning actions on or off is again expressed by actions. With $on \multimap T$ and $off \multimap T$ the action T can be turned on resp. off. Consider that *on* and *off* can be regarded as species which are always on resp. off and thus $on \multimap$ and $off \multimap$ actions can be regarded as special $\multimap$ actions.

Mapping the species and composite actions to on resp. off provides a *state* of the whole model, an on/off-configuration that indicates which actions are activated and which are not.

As a first step this work shows how biological models can be verified by comparing measured species' concentrations in fixed states against calculated possible outcomes. In a fixed state the cell reached an equilibrium in the sense that every action stays activated resp. not activated forever[2]. This is the case when the on/off-configuration is consistent with the model's composite actions.

After formalising the model syntax in 2.2 and its semantics in 2.3, Section 2.4 outlines some differences of this semantics as compared to other commonly used semantics. Section 2.5 describes how model actions can be expressed in SAT clauses with a variable for every action. There is an analogy between on/off-configurations of the actions and truth-assignments to the corresponding variables in the SAT instance. Then a fixed state is characterised as an assignment which satisfies all clauses.

In order to find all fixed states of a model, not only one satisfying truth assignment is necessary, but all of them. Since this is not required in the standard satisfiability problem, not every SAT solver is able to find all satisfying assignments. Prolog is used for the implementation here because its built-in backtracking algorithm is suitable for this requirement. Chapter 3 describes this implementation.

2.2 Model Syntax

A pathway model consists of actions. An *action* is either a species or a composite action, i.e. a relation between a species, *on* or *off* and another action.

The formal language of actions is defined by the following production rules:

$$\begin{aligned} \forall s \in S: \quad T \quad \longrightarrow \quad & s \mid on \multimap T \mid off \multimap T \mid s \to T \mid s \dashv T \mid \\ & s \multimap T \mid s \multimap\!\bullet\, T \mid s \multimap\!?\, T \end{aligned}$$

where S is a set of species indentifiers with $on \notin S$ and $off \notin S$. The action s of a single species only indicates that the species s is part of the model. Recall that to describe that a species s is on or off, the rules $on \multimap s$ or $off \multimap s$ are used.

[2]This is a simplification. As every natural object constantly changes, there are only quasi-steady states [12, p. 10].

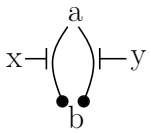


Figure 2.1: invalid model with the duplicate action $a \multimap\!\bullet\, b$

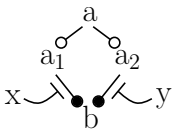


Figure 2.2: possible valid modelling of the duplicate action

A *pathway model* M is a set of actions, where any action appearing in one of these actions are also part of the model. This means that

$$\left.\begin{array}{l} on \multimap T \in M \\ off \multimap T \in M \end{array}\right\} \Rightarrow T \in M \qquad \text{and} \qquad \left.\begin{array}{c} s \to T \in M \\ s \dashv T \in M \\ s \multimap T \in M \\ s \multimap\!\bullet\, T \in M \\ s -\!? T \in M \end{array}\right\} \Rightarrow s, T \in M.$$

Since the dummy action implies an enhancing or an inhibiting action, both possibilities need to be covered by the model as well, so in addition

$$s -\!? T \in M \;\Rightarrow\; s \to T, s \dashv T \in M.$$

The model for the p53-pathway is $M = \{ARF \dashv (mdm2 \multimap (p53 \dashv p53)),\ mdm2 \multimap (p53 \dashv p53),\ p53 \dashv p53,\ p53 \to \textit{cell cycle arrest},\ p53,\ mdm2,\ ARF,\ \textit{cell cycle arrest}\}$.

If a composite action affects a species and has the form $off \multimap s'$ or $s \to s'$ for example, it is called *simple*. In contrast, a *nested* action is a composite action affecting another composite action, like $s \dashv (s' \multimap s'')$ or $on \multimap (s \dashv s')$. In the p53-pathway model the simple actions are $p53 \dashv p53$ and $p53 \to \textit{cell cycle arrest}$ whereas the nested actions are $ARF \dashv (mdm2 \multimap (p53 \dashv p53))$ and $mdm2 \multimap (p53 \dashv p53)$.

The requirements for correct models entail that only the outermost actions, which are not affected by another action, need to be given, in order to fully describe a model. These actions are called *free*. Also a $s -\!? T$ action is free when unaffected by other actions, while the corresponding $s \to T$ and $s \dashv T$ actions are not free. The model contains all non-free actions implicitly. Also the Prolog programme presented in this work uses just the free actions as a reduced description of a model. The free actions of the simplified p53-pathway are $ARF \dashv (mdm2 \multimap (p53 \dashv p53))$ and p53 $\to$ *cell cycle arrest*.

As a model is a set of actions, every action is unique. So, as an example, $x \dashv (a \multimap\!\bullet\, b)$ and $y \dashv (a \multimap\!\bullet\, b)$ both affect the same action $a \multimap\!\bullet\, b$. Although, a could control b in two different ways, where one is inhibited by x and the other one by y. In order to express that x and y inhibit different actions $a \multimap\!\bullet\, b$ (depicted in Figure 2.1) in a valid model, additional actions are required. Figure 2.2 shows how this can be achieved with auxiliary species and composite actions.

2.3 Model Semantics

The result of an experimental measurement of species' concentrations at equilibrium can be expressed as an on/off-configuration of the model. This allocation is mathematically described by a mapping from the set of actions to the boolean values `true` and `false`. Such a mapping defines a *state* of the model. If the model describes the considered biological cells at equilibrium adequately, the state has to be consistent with all actions. That means that the requirements resulting from the action's meanings motivated in Section 2.1 are met. A state that fulfills these requirements is called a *fixed state*. Otherwise, if there are unsatisfied actions, this suggests the measured cell was not in equilibrium or that the model or the measured configuration itself is erroneous. The latter can occur when mapping measurements of the real cell to an on/off-configuration of the model.

In detail, a fixed state f must satisfy the boolean formulae below.

$$
\begin{aligned}
f(on \multimap T) &\;\Rightarrow\; \big[f(T)\big] \\
f(off \multimap T) &\;\Rightarrow\; \big[\neg f(T)\big] \\
f(s \rightarrow T) &\;\Rightarrow\; \big[f(s) \Rightarrow f(T)\big] \\
f(s \dashv T) &\;\Rightarrow\; \big[f(s) \Rightarrow \neg f(T)\big] \qquad (*) \\
f(s \multimap T) &\;\Rightarrow\; \big[f(s) \Leftrightarrow f(T)\big] \\
f(s \multimap\!\bullet\, T) &\;\Rightarrow\; \big[f(s) \Leftrightarrow \neg f(T)\big] \\
f(s \,-\!?\, T) &\;\Rightarrow\; \big[f(s \rightarrow T) \text{ XOR } f(s \dashv T)\big]
\end{aligned}
$$

All these rules for composite actions are implications with the configuration for the action on the left-hand side and a boolean formula capturing the meaning of the action on the right-hand side. Since only composite actions have effects on other actions, no rule exists for species.

The set $\mathcal{F}$ of all fixed states is defined as

$$\mathcal{F} \;:=\; \big\{f : \mathcal{T} \to \{\texttt{true}, \texttt{false}\} \mid f \text{ is consistent with } (*)\big\},$$

where $\mathcal{T} := \big\{T \mid T \text{ action}\big\}$ is the set of all actions.

For a model $M \subset \mathcal{T}$, any fixed state $f \in \mathcal{F}$ restricted to M is a fixed state of the model.

$$\mathcal{F}_M \;:=\; \big\{f|_M : M \to \{\texttt{true}, \texttt{false}\} \mid f \in \mathcal{F}\big\}.$$

Conversely, every fixed state of M is also part of some fixed state in $\mathcal{F}$ which is shown in Lemma 1. The concistency with $(*)$ of restricted state functions $f|_M$ is well-defined since every rule for an action in $(*)$ refers to its sub-actions and these are contained in the model M by definition.

The set $\mathcal{F}_M$ can be determined by generating a SAT instance that describes M which will be explained in Section 2.5. Every possible solution of this SAT instance describes one fixed state in $\mathcal{F}_M$.

Lemma 1 *For any pathway model M, the set $\mathcal{F}_M$ contains all fixed states of the model.*

Proof. Let $f : M \to \{\texttt{true}, \texttt{false}\}$ be a fixed state of M. Define the state $f' : \mathcal{T} \to \{\texttt{true}, \texttt{false}\}$ as the extension that maps any action $T \in \mathcal{T} \backslash M$ to `false`. Thereby $f' \in \mathcal{F}$ because mapping an action to `false` cannot produce inconsistencies to $(*)$. Since $f' \in \mathcal{F}$ and $f'|_M = f$, one has $f \in \mathcal{F}_M$. □

Lemma 2 *Every model has a fixed state.*

Proof. The mapping $f(T) = \texttt{false}$ for all actions $T \in M$ is a fixed state for any model M, since every implication in $(*)$ is trivially satisfied if its left-handed side is `false`. □

When creating a composite action which is not affected by any other action, the modeller probably wants to express that this composite action always applies. So this action should always be activated intuitively. From now on, all (outermost) free actions shall be on in a valid state. This way the trivial fixed state where all actions are off is avoided.

The next lemma shows that the number of dummy $-?$ actions should be low when verfying models. The rules for a $-?$ action in $(*)$ can always be satisfied and thus these actions are not excluding any states to be fixed states.

Lemma 3 *Let M be a model where all free composite actions are unspecified $-?$ actions.*[3] *Let $M' \subset M$ be the model without these free $-?$ actions and their corresponding $\to$ and $\dashv$ actions. All fixed states of M' can be extended to fixed states of M, even if all free $-?$ actions are activated in M.*

Proof. Let $f' : M' \to \{\texttt{true}, \texttt{false}\}$ be a fixed state of M'. Then f' can be extended to a state f of M while choosing $f(s -? T) = \texttt{true}$ for any free composite action $s -? T \in M$. It remains to show that the $\to$ and $\dashv$ actions, which are implied by the free $-?$ actions in M, have an on/off-configuration that is consistent with the requirements in $(*)$.

Let $s -? T$ be a free action in M that is thus not included in M'. Since $f(s -? T) = \texttt{true}$, $f(s \to T)$ XOR $f(s \dashv T)$ must hold, according to $(*)$. If $f'(T) = f(T) = \texttt{true}$, setting $f(s \to T) = \texttt{true}$ and $f(s \dashv T) = \texttt{false}$ fulfills the requirements for a fixed state. Otherwise $f'(T) = f(T) = \texttt{false}$ holds; then a configuration with $f(s \to T) = \texttt{false}$ and $f(s \dashv T) = \texttt{true}$ meets the rules of $(*)$. □

Given any model with only $-?$ composite actions (and the accordingly implied $\to$ and $\dashv$ actions) this lemma implies that there is a fixed state for every on/off-configuration of the species. Thus the models in the STRING database [22], already mentioned in Section 2.1 and consisting of $-?$ actions only, are not suitable for the verification with the approach presented here. Nevertheless it can help to specify these models. If a free action $s -? T$ appears in a model and the species s is activated in some measurement of the cell at equilibrium, then $s -? T$ can be resolved to $s \to T$ resp. $s \dashv T$, depending on whether T is activated resp. deactivated. Thus a measurement of species at equilibrium can possibly eliminate some free $-?$ actions.

[3] Recall that the $\to$ and the $\dashv$ action implied by a $-?$ action are non-free actions.

Figure 2.3: negative autoregulation

Figure 2.4: positive autoregulation

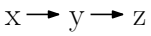


Figure 2.5: positive cascade

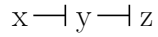


Figure 2.6: negative cascade

Figure 2.7: double-negative-feedback loop

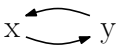


Figure 2.8: double-positive-feedback loop

Figure 2.9: negative-feedback loop

2.4 Network Motifs

Network motifs are patterns that occur in many pathways. The figures above show seven motifs as they are commonly expressed in other pathway models, like in [20]. These examples demonstrate that modelling in this approach differs from other approaches due to different semantics. Hence the translation between different graphical representations of models must be done carefully.

Autoregulation motifs as depicted in Figures 2.3 and 2.4 have effects on the dynamic behaviour of a cell, but not on its equilibrium situations. An autoregulated protein slows down or speeds up its own response time [20, 1]. These dynamics are difficult to capture in boolean models. In the models presented here, a negative autoregulation forces the species involved to be off in a fixed state, since $\big[f(x) \Rightarrow \neg f(x)\big]$ is true only for $f(x) = \texttt{false}$. On the other hand, a positive autoregulation has no influence on a model at all, because $\big[f(x) \Rightarrow f(x)\big]$ is a tautology and always true. The dynamics of a negative autoregulation are discussed further in Section 6.2.

A positive cascade describes a sequential activation of species which is again mainly important for dynamic considerations. The cascade model in Figure 2.5 can be simplified by the single composite action $x \to z$ omitting the species y. Negative cascades as depicted in Figure 2.6 are intended to express an alternating activation and deactivation of the species involved [20]. If x is activated, y shall be deactivated and z be activated again in a fixed state. This semantic is actually captured by the $\multimap$ actions in the approach of this paper. Thus the negative cascade in Figure 2.6 is appropriately expressed by the actions $x \multimap y$ and $y \multimap z$.

The double-negative-feedback loop in Figure 2.7 uses this different semantics for $\dashv$ as well and the adequate model would be given by the actions $x \multimap y$ and $y \multimap x$. Double-negative-feedback loops forces one of the involved species to be activated and the other one to be deactivated. In contrast, the species in double-positive-feedback loops are both either activated or deactivated. The common representation of this circumstance is depicted in Figure 2.8 which is also in this work's semantics an appropriate model.

The last example motif in Figure 2.9 shows a negative-feedback loop. A negative-feedback loop leads to oscillations or occasionally pulses of the species concentrations

[20, 1]. It prevents the cell from reaching an equilibrium. To avoid fixed states, the adequate model is given by the free actions $x \rightharpoonup y$ and $y \multimap x$. Further explanations about this motif can be found in Section 6.2.

2.5 Expressing Model Semantics as a SAT instance

The boolean satisfiability problem (SAT) asks for a satisfying variable assignment of a boolean formula, usually given in conjunctive normal form.

In order to create a SAT instance for a given model M, a SAT variable is introduced for every action in M. If T is an action, $\widetilde{T}$ denotes the variable for T. The requirements for composite actions in $(*)$ can be directly converted into boolean formulae of the corresponding variables. These formulae brought into conjunctive normal form make up the SAT instance. Let $[\![T]\!]$ denote the boolean formula capturing the semantic of the composite action $T \in M$. As motivated in Section 2.3 there are no formulae for species.

$$
\begin{aligned}
[\![on \multimap T]\!] &= \left(\widetilde{on \multimap T} \Rightarrow \widetilde{T}\right) \\
&= \left(\neg\widetilde{on \multimap T} \vee \widetilde{T}\right) \\
[\![off \multimap T]\!] &= \left(\widetilde{off \multimap T} \Rightarrow \neg\widetilde{T}\right) \\
&= \left(\neg\widetilde{off \multimap T} \vee \neg\widetilde{T}\right) \\
[\![s \to T]\!] &= \left(\widetilde{s \to T} \Rightarrow (\widetilde{s} \Rightarrow \widetilde{T})\right) \\
&= \left(\neg\widetilde{s \to T} \vee \neg\widetilde{s} \vee \widetilde{T}\right) \\
[\![s \dashv T]\!] &= \left(\widetilde{s \dashv T} \Rightarrow (\widetilde{s} \Rightarrow \neg\widetilde{T})\right) \\
&= \left(\neg\widetilde{s \dashv T} \vee \neg\widetilde{s} \vee \neg\widetilde{T}\right) \\
[\![s \multimap T]\!] &= \left(\widetilde{s \multimap T} \Rightarrow (\widetilde{s} \Leftrightarrow \widetilde{T})\right) \\
&= \left(\neg\widetilde{s \multimap T} \vee \widetilde{s} \vee \neg\widetilde{T}\right) \wedge \left(\neg\widetilde{s \multimap T} \vee \neg\widetilde{s} \vee \widetilde{T}\right) \\
[\![s \rightharpoonup T]\!] &= \left(\widetilde{s \rightharpoonup T} \Rightarrow (\widetilde{s} \Leftrightarrow \neg\widetilde{T})\right) \\
&= \left(\neg\widetilde{s \multimap T} \vee \widetilde{s} \vee \widetilde{T}\right) \wedge \left(\neg\widetilde{s \multimap T} \vee \neg\widetilde{s} \vee \neg\widetilde{T}\right) \\
[\![s -? T]\!] &= \left(\widetilde{s -? T} \Rightarrow (\widetilde{s \to T} \text{ XOR } \widetilde{s \dashv T})\right) \\
&= \left(\neg\widetilde{s -? T} \vee \widetilde{s \to T} \vee \widetilde{s \dashv T}\right) \wedge \\
&\quad \left(\neg\widetilde{s -? T} \vee \neg\widetilde{s \to T} \vee \neg\widetilde{s \dashv T}\right)
\end{aligned}
$$

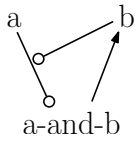


Figure 2.10: AND-Pathway

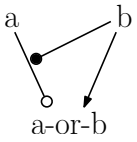


Figure 2.11: OR-Pathway

Figure 2.12: NOT-Pathway

For the example p53-pathway the SAT instance is given by the boolean formulae

$$\begin{aligned}
[\![ARF \overset{\gamma}{\dashv} (mdm2 \overset{\beta}{\multimap} (p53 \overset{\alpha}{\dashv} p53))]\!] &= \left(\neg\widetilde{\gamma} \vee \neg\widetilde{ARF} \vee \neg\widetilde{\beta}\right), \\
[\![mdm2 \overset{\beta}{\multimap} (p53 \overset{\alpha}{\dashv} p53)]\!] &= \left(\neg\widetilde{\beta} \vee \widetilde{mdm2} \vee \neg\widetilde{\alpha}\right) \wedge \left(\neg\widetilde{\beta} \vee \neg\widetilde{mdm2} \vee \widetilde{\alpha}\right), \\
[\![p53 \overset{\alpha}{\dashv} p53]\!] &= \left(\neg\widetilde{\alpha} \vee \neg\widetilde{p53} \vee \neg\widetilde{p53}\right), \\
[\![p53 \overset{\delta}{\to} cell\ cycle\ arrest]\!] &= \left(\neg\widetilde{\delta} \vee \neg\widetilde{p53} \vee cell\ \widetilde{cycle}\ arrest\right),
\end{aligned}$$

where the composite actions are abbreviated by the greek letters above the actions.

Apart from $\multimap\!\bullet$ and $-?$, all composite actions translate to Horn-clauses [8] which are clauses with at most one positive literal. Since Horn-SAT is solvable in linear time [6], one possible fixed state for a model without $\multimap\!\bullet$ and $-?$ actions can be found in linear time as well.

On the other hand, finding a fixed state for a model which includes also $\multimap\!\bullet$ actions can be as complex as solving an arbitrary SAT instance. Figures 2.10, 2.11 and 2.12 depict pathway models for the boolean functions $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT). For any configuration of the species a and b, the other species' truth-value in each model is given by the corresponding boolean function, provided that the model reached a fixed state. This can easily be verified with the Prolog programme presented in the following sections. Since the three boolean functions are the operations of a boolean algebra, arbitrary boolean functions can be expressed in pathway models by combining copies of the models in Figures 2.10, 2.11 and 2.12. The species representing the output of this function can be enforced to be `true` resp. `false` with *on* $\multimap$ resp. *off* $\multimap$. Thus, every SAT instance can be encoded as a pathway model.

The idea of transforming boolean pathway models into SAT instances already appeared in [24]. In that approach no variables are introduced for the species, but only for reactions which are simple composite actions with possibly several reactants and products. That leads to less variables, but more complex clauses for expressing whether certain species are activated or not. Moreover, nested actions are not supported.

In a nutshell, the SAT instance to a model M is given by the set of variables

$$V = \{\widetilde{T} \mid T \in M\}$$

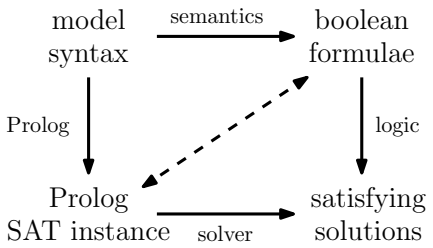
and the set of clauses

$$\begin{aligned} C \;=\; & \{\neg \widetilde{on \multimap T} \vee \widetilde{T} \;\mid\; on \multimap T \in M\} \;\cup \\ & \{\neg \widetilde{off \multimap T} \vee \neg\widetilde{T} \;\mid\; off \multimap T \in M\} \;\cup \\ & \{\neg \widetilde{s \to T} \vee \neg\widetilde{s} \vee \widetilde{T} \;\mid\; s \to T \in M\} \;\cup \\ & \{\neg \widetilde{s \dashv T} \vee \neg\widetilde{s} \vee \neg\widetilde{T} \;\mid\; s \dashv T \in M\} \;\cup \\ & \{\neg \widetilde{s \multimap T} \vee \widetilde{s} \vee \neg\widetilde{T},\; \neg \widetilde{s \multimap T} \vee \neg\widetilde{s} \vee \widetilde{T} \;\mid\; s \multimap T \in M\} \;\cup \\ & \{\neg \widetilde{s \bullet\!\!\multimap T} \vee \widetilde{s} \vee \widetilde{T},\; \neg \widetilde{s \bullet\!\!\multimap T} \vee \neg\widetilde{s} \vee \neg\widetilde{T} \;\mid\; s \bullet\!\!\multimap T \in M\} \;\cup \\ & \{\neg \widetilde{s -\!\!? T} \vee \widetilde{s \to T} \vee \widetilde{s \dashv T},\; \neg \widetilde{s -\!\!? T} \vee \neg\widetilde{s \to T} \vee \neg\widetilde{s \dashv T} \;\mid\; s -\!\!? T \in M\}. \end{aligned}$$

Chapter 3

Model-to-SAT transformation in Prolog

This section shows the correctness of the model-to-SAT transformation in Prolog. This proves that the boolean formulae resulting from the model syntax are captured by the SAT instance encoding. Thus, the following diagram commutes, assuming that the SAT solver works properly.



3.1 Encoding pathway models

Species are encoded as Prolog atoms which are set in single quotation marks or start with a lower-case character.

Five new Prolog operators are defined by

```
:- op(500, xfy, '-->').
:- op(500, xfy, '--/').
:- op(500, xfy, '-<>').
:- op(500, xfy, '->-<').
:- op(500, xfy, '--?').
```

in order to encode the actions

Turn on	$(on \multimap a)$	`on-<>a`,
Turn off	$(off \multimap a)$	`off-<>a`,
Enhancement	$(a \rightarrow b)$	`a-->b`,
Inhibition	$(a \dashv b)$	`a--/b`,
Control enhancing	$(a \multimap b)$	`a-<>b`,
Control inhibiting	$(a \multimap\!\bullet\ b)$	`a->><b`,
Dummy	$(a -\!\!? b)$	`a--?b`.

These operators are right-associative, e.g. `a-->b--/c` is equivalent to `a-->(b--/c)`.

A pathway model is encoded as a list of all free actions appearing in the model. The implicitly contained actions and species will be determined by the programme. Hypothetically, also species' encodings may appear in the model's list. But either the species already appears in a composite action and is thus implicitly contained already without explicitly mentioning. Or the species has no connection to the rest of the model and whether it is on or off is irrelevant. In both cases adding a species to the model's list of actions is futile.

3.2 Encoding SAT instances

A SAT instance is encoded as a list of clauses together with a list of variables. All variables appearing in one of the clauses must be contained in the variables list. The variables as well as the clauses are labelled. The label indicates what action the variable resp. clause represents.

A labelled variable has the representation

`label:Variable`.

A clause is encoded in Prolog as

`label:[pol1-Var1, pol2-Var2, ... , polN-VarN]`.

The literals of a clause are represented by the Prolog variables `Var1`, ..., `VarN` together with a polarity `pol1`, ..., `polN` indicating whether the literal is positive or negative. A polarity is either `true` or `false`, where the former stands for a positive literal and the latter for a negative one. This encoding is adopted from the Prolog SAT solver implementation in [9], which is utilised in this work's programme.

The encoded actions are transferred into encoded SAT clauses analogously to their semantics presented in Section 2.5. For an encoded action T, $(\!|T|\!)$ denotes the encoded SAT clauses capturing the meaning of T.

$$
\begin{aligned}
[\![on \multimap T]\!] &= \left(\neg \widetilde{on \multimap T} \vee \widetilde{T}\right) \\
(\!|\texttt{on -<> } T|\!) &= \texttt{on-<>}T\texttt{:[false-}\widetilde{on \multimap T}\texttt{,true-}\widetilde{T}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
[\![off \multimap T]\!] &= \left(\neg \widetilde{off \multimap T} \vee \neg\widetilde{T}\right) \\
(\!|\texttt{off -<> } T|\!) &= \texttt{off-<>}T\texttt{:[false-}\widetilde{off \multimap T}\texttt{,false-}\widetilde{T}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
[\![s \to T]\!] &= \left(\neg \widetilde{s \to T} \vee \neg\widetilde{s} \vee \widetilde{T}\right) \\
(\!|s \texttt{ --> } T|\!) &= s\texttt{-->}T\texttt{:[false-}\widetilde{s \to T}\texttt{,false-}\widetilde{s}\texttt{,true-}\widetilde{T}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
[\![s \rightarrowtail T]\!] &= \left(\neg \widetilde{s \rightarrowtail T} \vee \neg\widetilde{s} \vee \neg\widetilde{T}\right) \\
(\!|s \texttt{ --/ } T|\!) &= s\texttt{--/}T\texttt{:[false-}\widetilde{s \rightarrowtail T}\texttt{,false-}\widetilde{s}\texttt{,false-}\widetilde{T}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
[\![s \multimap T]\!] &= \left(\neg \widetilde{s \multimap T} \vee \widetilde{s} \vee \neg\widetilde{T}\right) \wedge \left(\neg \widetilde{s \multimap T} \vee \neg\widetilde{s} \vee \widetilde{T}\right) \\
(\!|s \texttt{ -<> } T|\!) &= s\texttt{-<>}T\texttt{:[false-}\widetilde{s \multimap T}\texttt{,true-}\widetilde{s}\texttt{,false-}\widetilde{T}\texttt{]}\texttt{,} \\
&\quad\ s\texttt{-<>}T\texttt{:[false-}\widetilde{s \multimap T}\texttt{,false-}\widetilde{s}\texttt{,true-}\widetilde{T}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
[\![s \multimap\!\!\bullet\ T]\!] &= \left(\neg \widetilde{s \multimap\!\!\bullet\ T} \vee \widetilde{s} \vee \widetilde{T}\right) \wedge \left(\neg \widetilde{s \multimap\!\!\bullet\ T} \vee \neg\widetilde{s} \vee \neg\widetilde{T}\right) \\
(\!|s \texttt{ -><} T|\!) &= s\texttt{-><}T\texttt{:[false-}\widetilde{s \multimap\!\!\bullet\ T}\texttt{,true-}\widetilde{s}\texttt{,true-}\widetilde{T}\texttt{]}\texttt{,} \\
&\quad\ s\texttt{-><}T\texttt{:[false-}\widetilde{s \multimap\!\!\bullet\ T}\texttt{,false-}\widetilde{s}\texttt{,false-}\widetilde{T}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
[\![s -\!\!? T]\!] &= \left(\neg \widetilde{s -\!\!? T} \vee \widetilde{s \to T} \vee \widetilde{s \rightarrowtail T}\right) \wedge \left(\neg \widetilde{s -\!\!? T} \vee \neg\widetilde{s \to T} \vee \neg\widetilde{s \rightarrowtail T}\right) \\
(\!|s \texttt{ --?} T|\!) &= s\texttt{--?}T\texttt{:[false-}\widetilde{s -\!\!? T}\texttt{,true-}\widetilde{s \to T}\texttt{,true-}\widetilde{s \rightarrowtail T}\texttt{]}\texttt{,} \\
&\quad\ s\texttt{--?}T\texttt{:[false-}\widetilde{s -\!\!? T}\texttt{,false-}\widetilde{s \to T}\texttt{,false-}\widetilde{s \rightarrowtail T}\texttt{]}
\end{aligned}
$$

Here $\widetilde{T}$ stands for the SAT variable as well as for the encoding's Prolog variable of the action T. The Prolog variables for the SAT instance encoding will be generated automatically by Prolog's unification algorithm and have undefined names.

The example of the p53-pathway can be encoded as the following list of labelled clauses. Here the Prolog variables start with the capital letter `V` followed by the identifier for a species and a number for a composite action. The assignment of composite actions to numbered Prolog variables is noted above each action. These notions are not part of the actual encoding.

```
                V1          V2          V3
[  arf--/(mdm2-<>(p53--/p53)):[false-V1,false-Varf,false-V2],
              V2          V3
   mdm2-<>(p53--/p53):[false-V2,true-Vmdm2,false-V3],
              V2          V3
   mdm2-<>(p53--/p53):[false-V2,false-Vmdm2,true-V3],
              V3
   p53--/p53:[false-V3,false-Vp53,false-p53],
              V4
   p53-->cell_cycle_arrest:[false-V4,false-Vp53,true-Vcell_cycle_arrest]  ]
```

3.3 Creating SAT instances

This section proves the correctness of the `create_sat/6` predicate and the other required predicates called during unification. For a given pathway model the `create_sat/6` predicate returns the encoded SAT instance consisting of a clauses list `Clauses` and a variables list `Vars`. The model is given by the list `Free_Actions` which consists of its free actions.

```
create_sat(Free_Actions, Unfolded_Actions, Species_vars, Actions_vars, Vars, Clauses) :-
151     extract_species(Free_Actions, Species),
        create_vars(Species, Species_vars),
153     unfold_actions(Free_Actions, Unfolded_Actions),
        create_vars(Unfolded_Actions, Actions_vars),
155     append(Species_vars, Actions_vars, Vars),
        maplist(create_clauses(Species_vars, Actions_vars),
157             Unfolded_Actions, Clauses0),
        flatten(Clauses0, Clauses), !.
```

In order to create the SAT instance, variables for the actions need to be generated. The variables for the species and composite actions are treated separately for simplicity and efficiency reasons. In the first place the species appearing in the given free actions need to be extracted via the `extract_species/2` predicate.

```
extract_species(Actions, Species) :-
162     extract_species_(Actions, Species0),
        list_to_set(Species0, Species).
```

In `extract_species/2` the auxiliary predicate `extract_species_/2` is called which iterates over the actions list `Actions` and appends every appearing species to the output

list `Species`. This output list may contain duplicates which are removed using the `list_to_set/2` predicate from the `lists` library.

```
extract_species_([on-<>(T)|Actions], Species) :-
165      extract_species_([T|Actions], Species), !.
extract_species_([off-<>(T)|Actions], Species) :-
167      extract_species_([T|Actions], Species), !.
extract_species_([Act|Actions], [S|Species]) :-
169      action(Act, S, T), extract_species_([T|Actions], Species), !.
extract_species_([S|Actions], [S|Species]) :-
171      extract_species_(Actions, Species), !.
extract_species_([], []).
```

The species extraction in `extract_species_/2` (with duplicates) for a model given by the list of free actions `[`T_1`, ..., `T_n`]` shall follow the rules stated below. The Prolog expression τ is converted into $(\!|\tau|\!)$, such that if `Actions` is the list of free actions defining the model, then $(\!|$`Actions`$|\!)$ is the list of the species contained.

$$
\begin{aligned}
(\!|\texttt{[ ]}|\!) &= \texttt{[ ]} \\
(\!|\texttt{[}T\texttt{,}T_1,\ldots,T_k\texttt{]}|\!) &= \texttt{[}(\!|T|\!)\ \texttt{|}\ (\!|\texttt{[}T_1,\ldots,T_k\texttt{]}|\!)\texttt{]}
\end{aligned}
$$

$$
(\!|s|\!) = \texttt{[}s\texttt{]} \qquad \left.\begin{array}{l}(\!|s\ \texttt{-->}\ T|\!)\\ (\!|s\ \texttt{--/}\ T|\!)\\ (\!|s\ \texttt{-<>}\ T|\!)\\ (\!|s\ \texttt{->\textless}\ T|\!)\\ (\!|s\ \texttt{--?}\ T|\!)\end{array}\right\} = \texttt{[}s\ \texttt{|}\ (\!|T|\!)\texttt{]}.
\qquad \left.\begin{array}{l}(\!|\texttt{on -<>}\ T|\!)\\ (\!|\texttt{off -<>}\ T|\!)\end{array}\right\} = (\!|T|\!)
$$

The first two rules of `extract_species_/2` treat `on-<>T` and `off-<>T` actions. Species only appear on the right-hand side `T` in this case. To find the species there, the predicate calls itself with `T` as the first element in the actions list. This recursion similarly works for all other actions in the third rule. The `action/3` predicate ensures that the third rule is only applied on composite actions, i.e. terms of the form `S-->T`, `S--/T`, `S-<>T`, `S->\<T` or `S--?T`. Here the left-hand side of an action is a species which is appended to the output list of species. The recursion stops when there is no action discovered in the third rule and the fourth rule applies. It inserts the species to the output list and recurses on the rest of the actions list. Cuts at the end of every rule ensure correct case differentiation. The last rule terminates the recursion of the predicate when the end of the actions list is reached.

Having the list of species available, the variables for the species can be created.

```
create_var(Label, Label:V) :- nonvar(Label), var(V).
create_vars(As, Vars) :- maplist(create_var, As, Vars).
```

To create a new labelled variable for an action, the `create_var/2` rule assures that the second argument `V` is a free variable, but the first argument `Label` is not. The built-in predicates `var/1` and `nonvar/1` find out whether a term is a free variable. These variables are generated automatically during Prolog's unification algorithm. In order to get the list of labelled variables for a species list, it suffices to apply the map combinator on the species list with the function `create_var/2`. The map combinator is offered by the `maplist/3` predicate from the `apply` library. The `create_vars/2` predicate invokes the map combinator.

The `create_vars/2` rule can be directly applied on the species list. The `Free_Actions` list however includes only free actions. There may be nested actions, i.e. actions affecting other actions. These actions need to be unfolded via `unfold_actions/2` before mapping, such that the implicitly included non-free actions are also part of the list. That way every composite action, regardless of whether free or non-free, can be mapped to a variable.

```
unfold_actions(Actions, Unfolded) :-
     unfold_actions_(Actions, Unfolded0),
     list_to_set(Unfolded0, Unfolded), !.
unfold_actions_([S--?(T)|Actions], [S--?(T),S-->(T),S--/(T)|Unfolded]) :-
     unfold_actions_([T|Actions], Unfolded), !.
unfold_actions_([A|Actions], [A|Unfolded]) :-
     action(A, _S, T), unfold_actions_([T|Actions], Unfolded), !.
unfold_actions_([_|Actions], Unfolded) :-
     unfold_actions_(Actions, Unfolded), !.
unfold_actions_([], []).
```

Unfolding actions has some similarities to the extraction of species. The auxiliary predicate `unfold_actions_/2` iterates over all nested actions and collects every single composite action in a list. The duplicates[1] of this list are removed afterwards with the `list_to_set/2` predicate.

For a model, given by a list of free actions `[`T_1`, ..., `T_n`]`, the list of all composite actions (including possible duplicates) can be obtained by unfolding the free actions according to the following rules, where T shall be converted into $(\!|T|\!)$ by `unfold_actions_/2`. If `Actions` is the list of free actions defining the model, then $(\!|$`Actions`$|\!)$ is the list of all actions of the model.

$$
\begin{aligned}
(\!|\texttt{[ ]}|\!) &= \texttt{[ ]} \\
(\!|\texttt{[}T\texttt{,}T_1,\ldots,T_k\texttt{]}|\!) &= \texttt{[}(\!|T|\!)\ \texttt{|}\ (\!|\texttt{[}T_1,\ldots,T_k\texttt{]}|\!)\texttt{]}
\end{aligned}
$$

[1] Duplicates appear if one action is affected by several other actions, like $a \dashv (x \to y)$ and $b \dashv (x \to y)$.

$$
\begin{aligned}
(\!|s|\!) &= \texttt{[ ]} \\
(\!|\texttt{on -<> } T|\!) &= \texttt{[on-<>}T\texttt{ | } (\!|T|\!)\texttt{]} \\
(\!|\texttt{off -<> } T|\!) &= \texttt{[off-<>}T\texttt{ | } (\!|T|\!)\texttt{]} \\
(\!|s\texttt{ --> } T|\!) &= \texttt{[}s\texttt{-->}T\texttt{ | } (\!|T|\!)\texttt{]} \\
(\!|s\texttt{ --/ } T|\!) &= \texttt{[}s\texttt{--/}T\texttt{ | } (\!|T|\!)\texttt{]} \\
(\!|s\texttt{ -<> } T|\!) &= \texttt{[}s\texttt{-<>}T\texttt{ | } (\!|T|\!)\texttt{]} \\
(\!|s\texttt{ -><} T|\!) &= \texttt{[}s\texttt{-><}T\texttt{ | } (\!|T|\!)\texttt{]} \\
(\!|s\texttt{ --? } T|\!) &= \texttt{[}s\texttt{--?}T\texttt{, }s\texttt{-->}T\texttt{, }s\texttt{--/}T\texttt{ | } (\!|T|\!)\texttt{]}
\end{aligned}
$$

Similarly to `extract_species_/2` the `unfold_actions_/2` predicate recurses on the right-hand side of a composite action, which may be another composite action as well. The first rule treats —? actions separately. Since `S--?T` means that either `S-->T` or `S--/T` holds, variables are needed for both possibilities and `S--?T` as well as `S-->T` and `S--/T` are added to the output list `Unfolded`. Otherwise the action stands only for itself and the second rule appends it to the output list. The recursion over the right-hand sides stops at a species, in the third rule leaving the output list unchanged. The whole predicate terminates when the actions list is empty in the last rule.

In the fourth line of the `create_sat/6` predicate variables for the unfolded actions are created. The species variables together with the composite action variables are the variables of the SAT instance. So, for the variable list `Var` the lists of species variables and composite action variables are concatenated in the fifth line using the `append/3` predicate from the `lists` library.

In order to create the clauses for the SAT instance, the clauses expressing the action's semantic (as explained in Section 2.3) need to be created for every composite action. A map combinator with the function `create_clauses/4` applied on the list of unfolded actions does this. Since not every action can be encoded in one clause (disjunction of literals), the `create_clauses/4` predicate returns a list of clauses. To get rid of these lists the flatten combinator is applied on the map result. The flatten combinator is provided by the `flatten/2` predicate from the `lists` library.

```
var_of([X:V|_Vars], X, V) :- !.
var_of([_|Vars], X, V) :- var_of(Vars, X, V), !.

var_of(_Species_vars, Actions_vars, X, V) :- composite_action(X),
                                             var_of(Actions_vars, X, V), !.
var_of(Species_vars, _Actions_vars, X, V) :- var_of(Species_vars, X, V).
```

Every `create_clauses/4` rule treats one composite action type and creates the labelled clauses for this type according to Section 3.2. The variables for the species and composite actions are looked up using the `var_of` predicates. To speed up the lookup, species variables and composite action variables are treated separately. For a given list of labelled

variables and a label the `var_of/3` predicate returns the variable for the label using linear search over the variables list. The `var_of/4` predicate simplifies the lookup in case it is unknown whether the variable of a species or a composite action is required. The `composite_action/1` predicate is used to distinguish between terms of species and composite actions. All `create_clauses/4` rules are given below.

```
% s-->X (s enhances X)
create_clauses(Species_vars, Actions_vars, S-->(X), Clauses) :-
     var_of(Species_vars, S, Var_S),
     var_of(Actions_vars, S-->(X), Var_A),
     var_of(Species_vars, Actions_vars, X, Var_X),
     Clauses = [S-->(X):[false-Var_S,false-Var_A,true-Var_X]].
% a--/X (a inhibits X)
create_clauses(Species_vars, Actions_vars, S--/(X), Clauses) :-
     var_of(Species_vars, S, Var_S),
     var_of(Actions_vars, S--/(X), Var_A),
     var_of(Species_vars, Actions_vars, X, Var_X),
     Clauses = [S--/(X):[false-Var_S,false-Var_A,false-Var_X]].
% on-<>X (X is on)
create_clauses(Species_vars, Actions_vars, on-<>(X), Clauses) :-
     var_of(Actions_vars, on-<>(X), Var_A),
     var_of(Species_vars, Actions_vars, X, Var_X),
     Clauses = [on-<>(X):[false-Var_A, true-Var_X]].
% off-<>X (X is off)
create_clauses(Species_vars, Actions_vars, off-<>(X), Clauses) :-
     var_of(Actions_vars, off-<>(X), Var_A),
     var_of(Species_vars, Actions_vars, X, Var_X),
     Clauses = [off-<>(X):[false-Var_A, false-Var_X]].
% a-<>X (a controls X enhancing)
create_clauses(Species_vars, Actions_vars, S-<>(X), Clauses) :-
     var_of(Species_vars, S, Var_S),
     var_of(Actions_vars, S-<>(X), Var_A),
     var_of(Species_vars, Actions_vars, X, Var_X),
     Clauses = [S-<>(X):[false-Var_A,false-Var_S, true-Var_X],
                S-<>(X):[false-Var_A, true-Var_S,false-Var_X]].
% a-><X (a uncontrols X inhibiting)
create_clauses(Species_vars, Actions_vars, S-><(X), Clauses) :-
     var_of(Species_vars, S, Var_S),
     var_of(Actions_vars, S-><(X), Var_A),
     var_of(Species_vars, Actions_vars, X, Var_X),
     Clauses = [S-><(X):[false-Var_A,true-Var_S, true-Var_X],
                S-><(X):[false-Var_A, false-Var_S,false-Var_X]].
% a--?X (a enhances or inhibits X)
create_clauses(_Species_vars, Actions_vars, S--?(X), Clauses) :-
     var_of(Actions_vars, S--?(X), Var_Aq),
     var_of(Actions_vars, S-->(X), Var_Ae),
     var_of(Actions_vars, S--/(X), Var_Ai),
     Clauses = [S--?(X):[false-Var_Aq,true-Var_Ae,true-Var_Ai],
                S--?(X):[false-Var_Aq,false-Var_Ae,false-Var_Ai]].
```

Chapter 4

Max-SAT solving in Prolog

4.1 Extending a regular SAT solver to a Max-SAT solver

The presented Max-SAT solver uses the SAT solver from [9] at its core. This SAT solver is an implementation of the DPLL [5] algorithm with watched literals [16]. It is written in Prolog as well and can compute not only one but all satisfying assignments for a given SAT instance, utilizing the resatisfiability of predicates in Prolog's backtracking algorithm. The solver is only slightly modified such that it ignores the labels of variables and clauses.

The used SAT solver [9] is called with the `sat/2` predicate. It takes a SAT instance given as a list of clauses in the first argument and a list of variables in the second. The encodings for these lists are given in Section 3.2. The SAT solver establishes a watch for every clause that observes literals of this clause and assures that at least one literal is satisfied. Afterwards the variables are initialised to `true` or `false`.

```
max_sat(Clauses, Vars, Satisfied, Unsatisfied) :-
47      gen_subset(Clauses, Satisfied, Unsatisfied),
        sat_unsat(Vars, Satisfied, Unsatisfied).
```

For a Max-SAT solver, the solver for the regular SAT problem is applied on the subsets of the clauses set.

```
range(X, _Y, X).
60 range(X, Y, Z) :- X>Y, X1 is X-1, range(X1, Y, Z).
range(X, Y, Z) :- X<Y, X1 is X+1, range(X1, Y, Z).
62
subset([],[],[]).
64 subset([E|R], [E|Xs], Ys) :- subset(R, Xs, Ys).
subset([E|R], Xs, [E|Ys]) :- subset(R, Xs, Ys).
66
```

```
gen_subset(R, X, Y) :-
    length(R, K),
    range(K, 0, N),
    M is K - N,
    length(X, N),
    length(Y, M),
    subset(R, X, Y).
```

The `gen_subset/3` predicate, called first in `max_sat/4`, generates all subsets of the given clauses set. The subsets are generated in decreasing order by cardinality, wich is provided by the built-in `length/2` predicate. The initial clauses set in the first argument is partitioned into the generated subset in the second argument and the rest in the third argument. In the `gen_subset/3` predicate, `range/3` binds `N` to the integers from `K` down to 0 and `M` always to `K-N`, where `K` is the cardinality of the initial set. The `range/3` predicate is a generalisation of the built-in `between/3` predicate and can also count down if the first integer argument is larger than the second. Afterwards, `subset/3` generates the partitions of the set `R` into `X` of size `N` and `Y` of size `M`. The `subset/3` predicate can generate all possible partitions of `R`, but as the sizes of `X` and `Y` are restricted beforehand, only the subsets of proper sizes are generated.

Since Prolog's backtracking algorithm tries to resatisfy the last goal first, all subsets of size `N` are generated before decreasing `N`. Hence the `gen_subset/3` predicate generates subsets in decreasing order of cardinality and works as requested.

```
unsat(true-Var) :- Var = false.
unsat(false-Var) :- Var = true.
unsatisfied(_Label:Clause) :- maplist(unsat, Clause).

sat_unsat(Vars, Satisfied, Unsatisfied) :-
    maplist(unsatisfied, Unsatisfied),
    sat(Satisfied, Vars).
```

After a subset of the clauses set is generated in `max_sat/4`, the `sat_unsat/3` predicate is called which finds the assignments satisfying exactly the clauses in the `Satisfied` list, but not the ones in `Unsatisfied`. A clause is unsatisfied by an assignment if none of its literals appear in that assignment. This is the case if the polarity of every variable appearing in the clause differs from the truth-value assigned to that variable. This property is enforced for every variable of a given labelled clause by the `unsatisfied/1` rule using the built-in `maplist/2` predicate. In `sat_unsat/3` the `unsatisfied/1` predicate is applied to every clause in `Unsatisfied` in order to instantiate the variables appearing in these clauses such that the resulting assignment unsatisfies these clauses if possible. After that, the `sat/2` predicate from [9] tries to find a satisfying assignment for the clauses in `Satisfied`. Note that the variables already instantiated by the `unsatisfied/1` predicate keep their truth-values and are not varied during the unification of `sat/2`. However,

apart from these variables the SAT solver starts from scratch for every subset of the clauses.

All in all, during backtracking of the `max_sat/4` predicate Prolog first tries to find another variable assignment that satisfies the current subset of clauses, while at the same time unsatisfies the remaining clauses. Second, it tries to find another clauses subset of the same cardinality and finally, it decreases the number for the subset cardinality.

Since the number of subsets grows exponentially in the cardinality of the initial set, the Max-SAT solver has an exponential worst case running time for finding one solution, even if the SAT solver finds solutions in polynomial time as with Horn-SAT[1]. It is known that the maximisation problem for Horn-SAT is NP-hard [10]. Although in the case of application which is considered in this work, at most a small number of unsatisfiable clauses is likely to occur. When verifying pathway models, only a few actions should be incorrect. If a pathway model contains too many contradictory actions, there is no good opportunity that the model can be corrected.

4.2 Simplifying SAT assignments

With the `max_sat/4` predicate it takes several redo instructions in order to find all satisfying variable assignments for a current subset of clauses. The `max_sat_minimized/5` predicate presents all these assignments more clearly in the `Sat_Assignments` argument by grouping up all satisfying variable assignments for one subset and by merging these assignments for a compressed output.

```
max_sat_minimized(Clauses, Vars, Satisfied, Unsatisfied, Sat_Assignments) :-
51      gen_subset(Clauses, Satisfied, Unsatisfied),
        findall(Vars,
53              sat_unsat(Vars, Satisfied, Unsatisfied),
                Sat_Assignments_),
55      Sat_Assignments_ \== [],
        simplify_assignments(Sat_Assignments_, Sat_Assignments).
```

In `max_sat_minimized/5` subsets of the clauses set are generated in decreasing order of cardinality using the `range/3` and `subset_gen/3` predicates just like before. For one clauses subset all assignments of variables in the `Vars` list, that satisfy exactly this subset, are collected with the built-in `findall/3` predicate. The result `Sat_Assignments_` is a list of variable lists, i.e. unified instances of the `Vars` list. If no matching instantiation of `Vars` can be found, `Sat_Assignments_` is the empty list which is precluded, since only matching assignments are desired.

[1]The SAT instance for a pathway model without ⊸ and −? actions consists of Horn clauses only, see Section 2.5.

```
simplify_assignments(Assignments, Simplified) :-
    merge_assignments(Assignments, Merged),
    del_implied_vars(Merged, Simplified_),
    list_to_set(Simplified_, Simplified).
```

The `simplify_assignments/2` rule tries to simplify this list of solutions by repeatedly merging two assignments (`Vars` list instantiations) that differ from each other in exactly one position. Once two assignments differ only at one variable, this variable can be set arbitrarily and it suffices to specify the assignment for all other variables. This idea is known from the Quine-McCluskey algorithm [15] for minimising boolean functions.

```
merge_vars([Label:Var|Assignments1], [Label:Var|Assignments2], [Label:Var|Merged]) :-
    !, merge_vars(Assignments1, Assignments2, Merged).
merge_vars([Label:_|Merged], [Label:_|Merged], [Label:dontcare|Merged]) :- !.
```

Two assignments that can be merged fulfil the `merge_vars/3` predicate. The third argument `Merged` of this predicate is the merging result for the assignments in the first two arguments `Assignment1` and `Assignment2`. The first rule of `merge_vars/3` iterates over the elements that the assignments have in common. This suffices to compare the assignments because of the fact that the order of the actions variables in a `Vars` list is fixed. Once the first rule fails, `Assignment1` and `Assignment2` differ in the currently considered position. If the rests of the assignments are identical, `Assignment1` and `Assignment2` can be merged and a `dontcare` atom marks the action at this position in the `Merged` list. The `dontcare` atom indicates that the corresponding action variable can be chosen arbitrarily. The second rule captures this case. Otherwise, merging is not possible and the predicate fails.

```
merge_assignments([], []) :- !.
merge_assignments(Assignments, Merged) :-
    merge_assignments_(Assignments, Merged_Step),
    merge_assignments(Merged_Step, Merged_Rest),
    append(Assignments, Merged_Rest, Merged).
merge_assignments_([], []) :- !.
merge_assignments_([X|Assignments], Merged) :-
    findall(X_Y,
            (member(Y, Assignments), merge_vars(X, Y, X_Y)),
            Merged_Current),
    merge_assignments_(Assignments, Merged_Rest),
    append(Merged_Current, Merged_Rest, Merged).
```

As a first step in `simplify_assignments/2`, the `merge_assignments/2` predicate repeatedly adds the representations of merged assignments given by `merge_vars/3` to the assignments list, until there are no two assignments left to merge. The helping predicate `merge_assignments_/2` returns a list of all possible merged assignments in the second

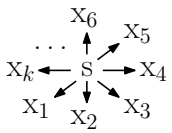
argument which can be obtained from the assignments given in the first argument. The `merge_assignments/2` predicate calls the helping predicate and recurses on the result of this merging step in order to further merge. The outcome of this recursion is appended to the assignments and returned as the predicate's second argument. The predicate stops recursing at the empty list.

To find all possible pairs of assignments which can be merged, the helping predicate `merge_assignments_/2` iterates over all assignments in the list given in the first argument. For any assignment it searches the remaining list for all other assignments that can be merged with the former one and thus fulfils the `merge_vars/3` predicate. The built-in `findall/3` predicate collects the newly merged assignments in the list `Merged_Current`. Together with the results of the recursion in `Merged_Rest`, this list forms the output for the predicate in the second argument. The exit condition for the recursion is again an empty list.

```
del_implied_vars([M|Merged], [M|Simplified]) :-
108      forall(member(M0, Merged), not(merge_vars(M, _, M0))),
         !, del_implied_vars(Merged, Simplified).
110  del_implied_vars([_|Merged], Simplified) :-
         !, del_implied_vars(Merged, Simplified).
112  del_implied_vars([], []).
```

Since the result of the `merge_assignments/2` predicate still contains the assignments that are implied by the merged assignments added, these implied assignments are removed using `del_implied_vars/2` when simplifying with `simplify_assignments/2`. The `del_implied_vars/2` predicate iterates over the list of assignments given in the first argument and removes all assignments implied by a merged one. The output is returned in the second argument `Simplified`. An assignment is implied by another one if the `merge_vars/3` predicate can be successfully unified with these assignments. The first rule of `del_implied_vars/2` treats a non-implied assignment. In this case the assignment is kept. The built-in `forall/2` predicate assures that all assignments in the tail of the list cannot fulfil the `merge_vars/3` predicate. This suffices since the merged assignments follow the assignments implied by them in the list created by `merge_assignments/2`. Otherwise the assignment is implied and the second rule applies which does not add it to `Simplified`. Cuts ensure correctness here. At an empty list, the third rule applies and finishes the recursion.

The computational effort needed for one `max_sat_minimized/5` query depends not only on the size of the SAT instance, but also on the number of possible assignments that satisfy the currently considered subset of clauses. There are pathway models for which the corresponding SAT instance has exponentially many satisfying assignments. The fixed states for these models are computed in worst-case exponential running time. The star shaped pathway model depicted on the right has at least 2^k fixed states for each configuration of the x_i when s is off, which is exponential in the number of actions. Recall that the output of the assignments is simplified with `dontcare` atoms when possible, but the backtracking algorithm of Prolog detects every assignment individually.



In this regard models with few fixed states can be processed faster. On the other hand the restrictiveness of models with few fixed states bases on many composite actions and many SAT clauses. As the free actions are always activated, it mainly depends on the number of non-free actions how much computational effort is necessary to find all fixed states. The attempt to verify a pathway model of more than 4000 non-free actions failed to terminate in acceptable time on a desktop computer.

Chapter 5

Finding Fixed States

With an implementation creating SAT instances for pathway models and a suitable Max-SAT solver available, determining a model's fixed states is straightforward. Given a pathway model as its encoded free actions, the `fixed_states/3` predicate returns all fixed states whereby the actions in the `Unsatisfied_Actions` list are unfulfilled in these fixed states. The Max-SAT solver assures that the fixed states contradicting the least number of actions are returned first.

```
fixed_states(Free_Actions, Fixed_States, Unsatisfied_Actions) :-
249     create_sat(Free_Actions, _Unfolded_Actions,
                   _Species_vars, Actions_vars, Vars, Clauses),
251     init_free_actions(Free_Actions, Actions_vars),
        max_sat_minimized(Clauses, Vars,
253                       _Satisfied_Clauses, Unsatisfied_Clauses, Fixed_States),
        maplist(label, Unsatisfied_Actions, _, Unsatisfied_Clauses).
```

After creating the SAT instance with `create_sat/6` for the model given by the list `Free_Actions`, the variables for the free actions are instantiated. Since free composite actions are always supposed to be active, their corresponding variables are set to `true` using the `init_free_actions/2` predicate.

```
init_free_actions(Free_Actions, A_vars) :- maplist(init_action(A_vars), Free_Actions).
257 init_action(_A_vars, Action) :- not(composite_action(Action)).
init_action(A_vars, Action) :- var_of(A_vars, Action, true).
```

This predicate calls `init_action/2` for every free action with the aid of the `maplist/2` predicate from the `apply` library. The first rule of `init_action/2` applies if the free action is not a composite action and hence a species[1]. In this case the action's variable is not instantiated yet. Otherwise, a composite action is present and the second rule

[1] As explained in Section 3.1, this case is futile but not forbidden.

instantiates the corresponding variable by calling the `var_of/3` predicate with the `true` atom as third argument.

Next in `fixed_states/3`, the `max_sat_minimized/5` predicate is called which finds all satisfying assignments for as many SAT clauses as possible. As described in Chapter 4, the last argument of this predicate is a list of all possible satisfying assignments in a minimized representation. Note that the variables of the free actions already instantiated before calling `max_sat_minimized/5` keep their values while solving the SAT instance. The unsatisfied clauses in the `Unsatisfied_Clauses` list belong to actions which could not be fulfilled. As all clauses correspond to model actions, the satisfying assignments correspond to the fixpoints of the model without the unfulfilled actions. For better readability the action labels corresponding to the unsatisfied clauses are extracted in the last line of the `fixed_states/3` predicate.

The rules for the $s \multimap T$, $s \rightarrowtail T$ and $s -? T$ actions are expressed with two SAT clauses each, whereas the rules for all other actions need only one clause each. Nevertheless, for any assignment no two unsatisfied clauses correspond to the same action. It is impossible to unsatisfy both such clauses at the same time since both clauses always share a literal with different polarity. This shows that n unsatisfied clauses correspond to exactly n unfulfilled actions.

A user, who wants to find out the fixed states for a pathway model, first encodes this model with a list of the free actions as described in Section 3.1. The `fixed_states/3` predicate is called with the list encoding the model as the first argument. Hence a query for the fixed states of the AND-pathway depicted in Figure 2.10 from Section 2.5 is

```
?- Model = [b -<> (a -<> a_and_b), a_and_b --> b],
   fixed_states(Model, Fixed_States, Unsatisfied_Actions).
```

The first answer to this query is the following:

```
Model = [b-<>a-<>a_and_b,a_and_b-->b],
Fixed_States =
     [[b:true,a:true,a_and_b:true,
       b-<>a-<>a_and_b:true,a-<>a_and_b:true,a_and_b-->b:true],
      [b:true,a:false,a_and_b:false,
       b-<>a-<>a_and_b:true,a-<>a_and_b:true,a_and_b-->b:true],
      [b:false,a:dontcare,a_and_b:false,
       b-<>a-<>a_and_b:true,a-<>a_and_b:false,a_and_b-->b:true]],
Unsatisfied_Actions = []
```

For improved clarity the output can be displayed as a table using the `print_assignments/1` predicate. It is called with the desired fixed states list as the argument. In the table the symbols `1`, `0` resp. `-` abbreviates `true`, `false` resp. `dontcare`. The AND-pathway query with the fixed states presented as a table is

```
?- Model = [b -<> (a -<> a_and_b), a_and_b --> b],
   fixed_states(Model, Fixed_States, Unsatisfied_Actions).
   print_assignments(Fixed_States).
```

Then the output also includes the following table:

```
 b | a | a_and_b | b-<>a-<>a_and_b | a-<>a_and_b | a_and_b-->b |
 1 | 1 |    1    |        1        |      1      |      1      |
 1 | 0 |    0    |        1        |      1      |      1      |
 0 | - |    0    |        1        |      0      |      1      |
```

Chapter 6

Dynamic Behaviour

6.1 Prediction of Dynamic Behaviour

So far, cells at their equilibrium were considered in order to verify a proposed pathway model. Looking at it the other way round and assuming the model to be correct, the model's fixed states reveal whether a measurement of the species' concentrations was undertaken at equilibrium or not. If a pathway model is in a fixed state, the appendant cell reached an equilibrium for all modelled processes. In a non-fixed state however, some processes in the cell have not taken effect. These processes will happen in future, provided that the model appropriately characterises the real processes inside the cell.

In pathway models discussed in this work, every action abstracts from a process which is assumed to occur in the cell. A composite action T' affects the action T on its right-hand side. This latter action T is dependent on T', such that either T' enhances T or inhibits T. If a model state conflicts the rule for T' stated at $(*)$ in Section 2.3, the model suggests that the process identified by T' is not at equilibrium and will activate resp. deactivate the process identified by T at some future point of time.

Considering a biological cell at discrete time steps, its dynamics can be reconstructed with a describing pathway model M and several state functions $f_t : M \to \{\texttt{true}, \texttt{false}\}$, one per time step t. If a rule in $(*)$ is unfulfilled at a given state f_t, the composite action corresponding to this rule suggests that the action on its right-hand side will be activated resp. deactivated by some process in the cell. With a state $f_{t'}$ where the latter action is toggled from *on* to *off* or vice versa, the status of the cell at time step $t' > t$ can be predicted. By repeatedly determining a succeding state for a non-fixed state, there may be opportunities to predict the cell's dynamic behaviour accurately with the pathway models. Consider for instance, the model containing only the enhancing action $a \to b$. The state f_1 where a is on, but b is off (i.e. $f_1(a) = \texttt{true}, f_1(b) = \texttt{false}$) is a non-fixed state, since the rule $f_1(a \to b) \;\Rightarrow\; \big[f_1(a) \Rightarrow f_1(b)\big]$ from $(*)$ does not hold. Since the rule for action $a \to b$ is conflicted in f_1, the right-hand side b is toggled in the

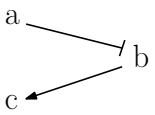


Figure 6.1: Different orders of action speeds

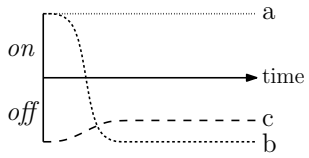


Figure 6.2: $a \dashv b$ faster than $b \to c$

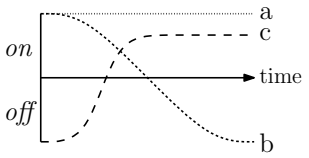


Figure 6.3: $a \dashv b$ slower than $b \to c$

succeeding state f_2. Hence, for f_2 holds $f_2(a) = f_2(b) = \texttt{true}$. This is a fixed state as all rules in $(*)$ are satisfied for f_2.

Let f_t be a state of the model M at the time step t and for an action $T' \in M$, the corresponding rule in $(*)$ shall be contradicted by f_t.[1] In the next state f_{t+1} the truth-value for the right-hand side of T' can be toggled, in order to fulfil the rule of T' in f_{t+1}. Hence, if T is the right-hand side of T', for the next state f_{t+1} holds

$$f_{t+1}(T) = \begin{cases} \texttt{true}, & f_t(T) = \texttt{false} \\ \texttt{false}, & f_t(T) = \texttt{true}. \end{cases}$$

In order to determine a succeeding state for a given state f_t, the truth-values of some actions in $alterable(f_t)$ are toggled.

$$alterable(f_t) := \left\{ T \in M \;\middle|\; \begin{array}{ll} \exists \text{ action in } M & \text{(i) whose rule is contradicted in } f_t \text{ and} \\ & \text{(ii) with } T \text{ as the right-hand side} \end{array} \right\}$$

Various processes inside cells may take effect with various speeds. Thus, whenever more than one action is not in equilibrium, i.e. its rule in $(*)$ is conflicting in the state, there are several possibilities for the order in which the dependent actions are toggled. The order influences the sequence of predicted model states and can even determine which states arise at all. For example, consider the model in Figure 6.1 and assume the state $f_1(a) = f_1(b) = \texttt{true}, f_1(c) = \texttt{false}$ which is not fixed. Both of the rules that are implied by the model $f_1(a \dashv b) \;\Rightarrow\; \big[f_1(a) \Rightarrow \neg f_1(b)\big]$ and $f_1(b \to c) \;\Rightarrow\; \big[f_1(b) \Rightarrow f_1(c)\big]$ are not fulfilled, so $alterable(f_1) = \{b, c\}$ and b or c can be toggled. If b is turned off first because of $a \dashv b$, the model reaches the fixed state $f_2(a) = \texttt{true}, f_2(b) = f_2(c) = \texttt{false}$. This series of states is illustrated in Figure 6.2. Since b is turned off fast, the species c is barely affected by $b \to c$ and thus not toggled. Otherwise, c is turned on first because of $b \to c$, after which $a \dashv b$ takes action. In this case the resulting fixed state is $\bar{f}_2(a) = \bar{f}_2(c) = \texttt{true}, \bar{f}_2(b) = \texttt{false}$ as illustrated in Figure 6.3. Whenever there is no certain or exact hierarchy of process speeds available, all possible sequences of state transitions can be enumerated. The modelling environment BIOCHAM [3] follows this approach using specific probabilities for the state transitions.

[1] Note that the action T' must be activated in f_t, i.e. $f_t(T') = \texttt{true}$, because otherwise the state f_t would always statisfy the rule for T'.

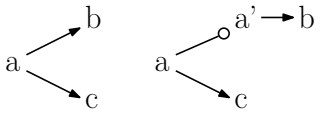


Figure 6.4: Expressing actions, acting with different speeds

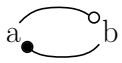


Figure 6.5: Negative-feedback loop (a simple oscillating model)

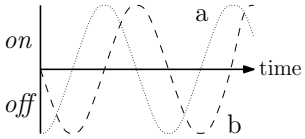


Figure 6.6: ODE-solution for the negative-feedback loop

For simplicity, all processes are assumed to act with the same speed in the following. When going over from a current state f_t to the next state f_{t+1}, all actions which are dependent in a disequilibrium, i.e. all actions in $alterable(f_t)$, are toggled:

$$\forall T \in alterable(f_t) \ : \ f_{t+1}(T) := \begin{cases} \texttt{true}, & f_t(T) = \texttt{false}, \\ \texttt{false}, & f_t(T) = \texttt{true}. \end{cases}$$

For all other actions the value does not change in the next state, in order to stay in equilibrium:

$$\forall T \in M \backslash alterable(f_t) \ : \ f_{t+1}(T) := f_t(T).$$

Although all actions in $alterable(f_t)$ are toggled simultaneously for the next state, different process speeds can be modelled using auxiliary species. Consider the model on the left of Figure 6.4. To express that $a \to b$ acts slower than $a \to c$, an additional species a' can be appended, which leads to the model depicted in the right of Figure 6.4. Starting in the state $f_1(a) = \texttt{true}, f_1(a') = f_1(b) = f_1(c) = \texttt{false}$, the next state will be $f_2(a) = f_2(a') = f_2(c) = \texttt{true}, f_2(b) = \texttt{false}$ whereupon the state $f_3(a) = f_3(a') = f_3(b) = f_3(c) = \texttt{true}$ follows. In this sequence of states c is indeed activated before b.

6.2 Oscillations

A fixed state will not always be reached by iteratively determining successive states. The model in Figure 6.5 for example has no fixed states at all. In a state where a is on, $a \multimap b$ will turn b on, after which a will be turned off according to $b \bullet\!\!\to a$. After that $a \multimap b$ turns b off. Then the rule for $b \bullet\!\!\to a$ in $(*)$ is unfulfilled and a is toggled back on again. The model oscillates in the four possible states forever. The set of states a model oscillates between is called a *limit cycle* of this model.[2]

In this case the predicted sequence of states is adequate. The model in Figure 6.5 represents a network motif[3] with two proteins, called negative-feedback loop, that can

[2]Starting at any state in a limit cycle, the model will come back to this state at some time.
[3]See Section 2.4.

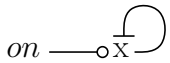


Figure 6.7: Negative autoregulation (mistakenly oscillating model)

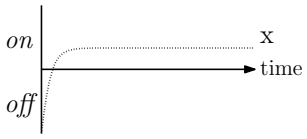


Figure 6.8: ODE-solution for the negative autoregulation

lead to oscillations [1]. A system of ordinary differential equations (ODE) describing the dynamics for this negative-feedback loop is the following:

$$\begin{aligned} a'(t) &= -b(t) + \lambda \\ b'(t) &= a(t) - \lambda \end{aligned} \qquad (\lambda > 0)$$

The functions $a(t)$ and $b(t)$ estimate the concentrations for both proteins a and b at a point of time $t \in \mathbb{N}$. Since a is inhibited by b, $b(t)$ appears with a minus-sign in the first derivative $a'(t)$. When the concentration of b is low, the concentration of a increases because the inhibiting action $b \multimap a$ is a controlling action. The positive constant λ in $a'(t)$ takes account of this fact. In the derivative $b'(t)$ the function $a(t)$ appears with a plus-sign since b is enhanced by a. As b is even controlled by a, the concentration of b decreases as well once the concentration of a is low. This circumstance is captured by the constant λ with a minus-sign in $b'(t)$. The differential equations have the following solution.

$$\begin{aligned} a(t) &= -k_2 \sin(t) + k_1 \cos(t) + \lambda \\ b(t) &= k_1 \sin(t) + k_2 \cos(t) + \lambda \end{aligned} \qquad (k_1, k_2 \in \mathbb{R})$$

The graphs of these functions are outlined in Figure 6.6.

Beginning with the state f_1, the following states are predicted by the model:

	$f_1(\cdot)$	$f_2(\cdot)$	$f_3(\cdot)$	$f_4(\cdot)$	$f_5(\cdot) = f_1(\cdot)$	$f_6(\cdot) = f_2(\cdot)$
a	`false`	`true`	`true`	`false`	`false`	...
b	`false`	`false`	`true`	`true`	`false`	

This sequence of states can be tracked in the outlined graph in Figure 6.6. In this case the state prediction of the transition function presented in Section 6.1 is adequately describing actual dynamics in the cell.

However, some predictions grounded upon the pathway models presented in this work are misleading. Figure 6.7 depicts another network motif with one protein, a so called negative autoregulation. The model for this pathway would suggest that the concentration of protein x will oscillate. When the species x is off, $on \multimap x$ will turn x on, whereupon x is turned off again because of $x \dashv x$. This predicted behaviour is not correctly describing how the concentration of a negatively autoregulated protein develops in biology. The concentration actually converges to a steady level near the threshold that seperates high (on) and low (off) concentrations [1].

Ordinary differential equations can help to model the species concentration over time more adequately in the case of a negative autoregulation. The dependencies of the autoregulated protein x are captured by the following equation.

$$x'(t) = \quad -x(t) + \lambda \qquad\qquad (\lambda > 0)$$

In the first derivative $x'(t)$ appears $x(t)$ with a minus-sign because of the inhibiting action $x \dashv x$. The fact that the concentration of x constantly increases is captured by the positive constant λ. The following function solves this differential equation.

$$x(t) = \quad ke^{-t} + \lambda \qquad\qquad (k \in \mathbb{R})$$

This exponential function with a negative exponent is not periodic and shows that the species x does not oscillate. The graph of $x(t)$ is outlined in Figure 6.8.

The misleading prediction for a negative autoregulation illuminates limitations of boolean modelling. Given a certain state f_t of a model, the next state f_{t+1} can easily be computed; for example by iterating over all actions and toggling all actions in $alterable(f_t)$. In [7] there is also an efficient algorithm presented that can be used to find fixed states and limit cycles for boolean models of this type. Nevertheless, a limit cycle found in the model does not necessarily mean that oscillations occur in the actual biological cell. Moreover, the boolean approach is insufficient, whenever the concentration of a species needs to be discretised in more than two niveaus in order to obtain an adequate model. The example of negative autoregulation shows both insufficiencies.

Chapter 7

Comparison

There are other boolean modelling approaches of signalling pathways, in which fixed states and limit cycles were determined using exhaustive search. Starting from any state, the next state is repeatedly computed until either a fixed state or a limit cycle is detected. By applying this strategy for every possible starting state, all fixed states of the examined model can be found as well. The results of this method for two different pathways of yeast cells are presented in [14] and [4].

Both papers take a state transition function as a basis which differs from the one described in this work. The pathway models in [14] and [4] only support $\rightarrow$ and $\dashv$ actions between two species, hence only simple and no nested composite actions are supported. This means that every composite action is free and thus always activated. So the model's state depends on the species configuration only. For the state transition function, the number of enhancing and the number of inhibiting actions is of concern. Given a model M, define the number of enhancing resp. inhibiting actions affecting a species $s \in M$ in a certain state $f : M \to \{\texttt{true}, \texttt{false}\}$ as

$$\begin{aligned} enh(f, s) &:= \left|\{s' \rightarrow s \in M \mid s' \in M, f(s') = \texttt{true}\}\right| \text{ resp.} \\ inh(f, s) &:= \left|\{s' \dashv s \in M \mid s' \in M, f(s') = \texttt{true}\}\right|. \end{aligned}$$

The transition from one state f_t to the next state f_{t+1} is given by

$$f_{t+1}(s) = \begin{cases} \texttt{true}, & enh(f_t, s) > inh(f_t, s), \\ \texttt{false}, & enh(f_t, s) < inh(f_t, s), \\ f_t(s), & enh(f_t, s) = inh(f_t, s). \end{cases}$$

Stable configurations according to this transition function shall be called *fixed points*, in contrast to fixed states which were discussed throughout this work. The fixed points of a model are the fixed points of the transition function above, so for a fixed point f_t holds $f_t = f_{t+1}$.

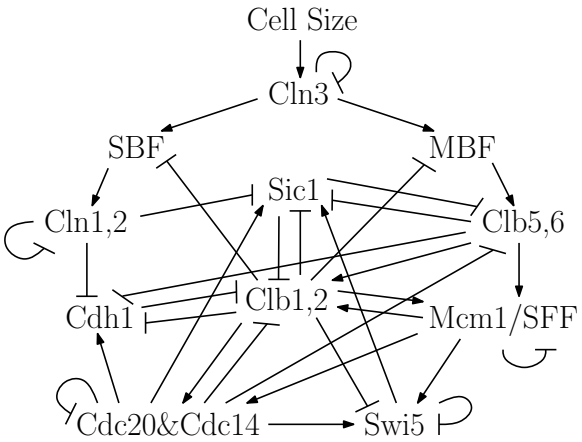


Figure 7.1: Simplified budding yeast cell cycle model from [14]

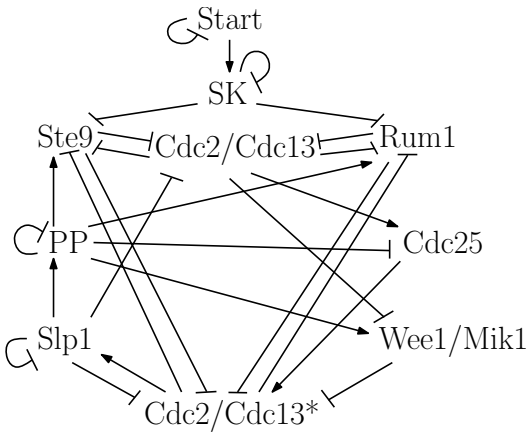


Figure 7.2: Fission yeast cell cycle regulation model from [4]

The pathway models of yeast cells depicted in Figure 7.1 resp. Figure 7.2 are taken from [14] resp. [4]. The fixed points of the models are stated already in these papers and are given in the tables below. The table entry 0, 1 resp. – represents `false`, `true` resp. `dontcare`, like in the output tables introduced in Chapter 5. The fixed points were computed using exhaustive search according to the transition function above, where the transition function in [4] was slightly adjusted to some particular species.

The fixed points of the budding yeast model depicted in Figure 7.1 according to [14]:

Cln3	MBF	SBF	Cln1,2	Cdh1	Swi5	Cdc20	Clb5,6	Sic1	Clb1,2	Mcm1
0	0	1	1	0	0	0	0	0	0	0
0	-	0	0	-	0	0	0	1	0	0
0	0	0	0	-	0	0	0	-	0	0

The fixed points of the fission yeast model depicted in Figure 7.2 according to [4]:

SK	Cdc2/Cdc13	Ste9	Rum1	Slp1	Cdc2/Cdc13*	Wee1/Mik1	Cdc25	PP
0	0	-	1	0	0	-	-	0
0	0	1	-	0	0	-	-	0

The fixed states of these models interpreted as models with the semantics presented in this work are given below. Because of different semantics of the model actions, the resulting stable configurations differ from each other as well. The different modelling approaches cannot be converted into one another in a simple and obvious way. Pathway models must be transformed manually into the other model semantics.

The fixed states of the budding yeast model depicted in Figure 7.1:

Cln3	MBF	SBF	Cln1,2	Cdh1	Swi5	Cdc20	Clb5,6	Sic1	Clb1,2	Mcm1
0	0	0	0	-	0	0	0	-	0	0

The fixed states of the fission yeast model depicted in Figure 7.2:

SK	Cdc2/Cdc13	Ste9	Rum1	Slp1	Cdc2/Cdc13*	Wee1/Mik1	Cdc25	PP
0	0	-	-	0	0	-	0	0

Nevertheless, the two yeast pathways indicate that fixed states of such small models can be found efficiently with the modelling approach from in this work. The fixed points for both models were determined by the Prolog implementation presented in the Chapters 3-5 in just a few minutes each on a desktop computer.

Chapter 8

Conclusion and Outlook

This work presented a boolean modelling approach for pathways of biological cells. The foundation of the models is the intuition that every object of research in a cell can be regarded as a process, which is occuring or not occuring at a certain point of time. The processes of research interest are abstracted by the actions of the model, where dependencies between processes are expressed in actions as well.

The specified semantics of a model's actions are immediately expressed in boolean formulae. These formulae are considered as a SAT instance. Every satisfying assignment of the SAT instance corresponds to a fixed state of the model which represents an equilibrium situation of the modelled cell. In order to verify a model, measured configurations of the cell at equilibrium are compared against the fixed states of the model. It has been shown how models can be encoded and transformed into SAT instances in Prolog and how fixed states can be determined.

When a measurement is not compatible with a model, a Max-SAT solver can find submodels with the least number of contradicted actions. For that purpose a Prolog Max-SAT solver was presented. This approach could be improved further by specifying a number indicating the certainty for every stated action in the model. Dependencies between proteins which were already asserted in several studies would get a high number, whereas new claims recieve a low number. A suitable solver can treat the resulting Weighted-Max-SAT instance.

Prolog proved to be useful and easy to use for SAT solving when all possible satisfying assignments are required. The Prolog SAT solver from [9] can be enhanced to a Max-SAT solver by additional predicates, but without changing the original solver. The presented implementation for finding fixed states of pathway models benefits from the use of higher-order predicates that are already built-in in SWI-Prolog. More about higher-order programs in logic programming can be found in [17].

Some possiblities, but also limitations were outlined for predicting the cell's dynamic behaviour with the presented approach. The models are not suitable for a reliable prediction of dynamic behaviour.

The comparison with another pathway model showed that the semantics of other modelling approaches cannot be simulated easily with the approach presented in this work.

Bibliography

[1] U. Alon (2007). *Network motifs: theory and experimental approaches*. Nature Review Genetics, Vol. 8, pp. 450-461, doi:10.1038/nrg2102.

[2] G. Arellano (2011). *"Antelope": a hybrid-logic model checker for branching-time Boolean GRN analysis*. BMC Bioinformatics, Vol. 12, Issue 1, pp. 490-504, doi:10.1186/1471-2105-12-490.

[3] L. Calzone, F. Fages and S. Soliman (2006). *BIOCHAM: an environment for modeling biological systems and formalizing experimental knowledge*. Bioinformatics, Vol. 22, Issue 14, pp. 1805-1807, doi:10.1093/bioinformatics/btl172.

[4] M. I. Davidich and S. Bornholdt (2008). *Boolean Network Model Predicts Cell Cycle Sequence of Fission Yeast*. PLoS ONE, Vol. 3, Issue 2, doi:10.1371/journal.pone.0001672.

[5] M. Davis, G. Logemann and D. Loveland (1962). *A Machine Program for Theorem-Proving*. Communications of the ACM, Vol. 5, Issue 7, pp. 394-397, doi:10.1145/368273.368557.

[6] W. F. Dowling and J. H. Gallier (1984). *Linear-time algorithms for testing the satisfiability of propositional Horn formulae*. Journal of Logic Programming, Vol. 1, Issue 3, pp. 267–284, doi:10.1016/0743-1066(84)90014-1.

[7] E. Dubrova and M. Teslenko (2011). *A SAT-Based Algorithm for Computing Attractors in Synchronous Boolean Networks*. IEEE/ACM Transactions on Computational Biology and Bioinformatics, Vol. 8, Issue 5, pp. 1393-1399, doi:10.1109/TCBB.2010.20.

[8] A. Horn (1951). *On Sentences Which are True of Direct Unions of Algebras*. The Journal of Symbolic Logic, Vol. 16, No. 1, pp. 14-21, doi:10.2307/2268661.

[9] J. M. Howe and A. King (2010). *A Pearl on SAT Solving in Prolog*. 10th International Symposium on Functional and Logic Programming, Vol. 6009, pp. 165-174, doi:10.1007/978-3-642-12251-4˙13.

[10] B. Jaumard and B. Simeone (1987). *On the complexity of the maximum satisfiability problem for Horn formulas*. Information Processing Letters, Vol. 26, Issue 1, pp. 1-4, doi:10.1016/0020-0190(87)90028-7.

[11] H. de Jong (2002). *Modeling and simulation of genetic regulatory systems: A literature review*. Journal of Computational Biology, Vol. 9, Issue 1, pp. 67-103, doi:10.1089/10665270252833208.

[12] E. Klipp et. al. (2009). *Systems Biology*. Wiley-VCH.

[13] M. Kwiatkowska, G. Norman and D. Parker (2010). *Advances and Challenges of Probabilistic Model Checking*. 48th Annual Allerton Conference on Communication, Control, and Computing, pp. 1691-1698, doi:10.1109/ALLERTON.2010.5707120.

[14] F. Li (2004). *The yeast cell-cycle network is robustly designed*. PNAS, Vol. 101, No. 14, pp. 4781-4786, doi:10.1073/pnas.0305937101.

[15] E. J. McCluskey (1956). *Minimization of Boolean functions*. The Bell System Technical Journal, Vol. 35, No. 5, pp. 1417-1444.

[16] M. W. Moskewicz (2001). *Chaff: Engineering an Efficient SAT Solver*. Proceedings of the 38th annual Design Automation Conference (DAC 2001), pp. 530-535, doi:10.1145/378239.379017.

[17] L. Naish (1996). *Higher-order logic programming in Prolog*. Available at `http://ww2.cs.mu.oz.au/~lee/papers/ho/`, last visited 10/05/2013.

[18] National Human Genome Institute (2012). *Biological Pathways* (Fact Sheet). Available at `http://www.genome.gov/27530687`, last visited 10/05/2013.

[19] A. Regev and E. Shapiro (2002). *Cellular abstractions: Cells as Computation*. Nature, Vol. 419, Issue 6905. pp. 343.

[20] O. Shoval and U. Alon (2010). *SnapShot: Network Motifs*. Cell, Vol. 143, Issue 2, pp. 326-326.e1, doi:10.1016/j.cell.2010.09.050.

[21] P. Smolen, D. A. Baxter and J. H. Byrne (2000). *Mathematical Modeling of Gene Networks*. Neuron, Vol. 26, pp. 567-580, doi:10.1016/S0896-6273(00)81194-0.

[22] B. Snel et. al. (2000). *STRING: a web-server to retrieve and display the repeatedly occurring neighbourhood of a gene*. Nucleic Acids Research, Vol. 28, No. 18, pp. 3442-3444, doi:10.1093/nar/28.18.3442.

[23] R. Thomas and R. D'Ari (1990). *Biological Feedback*. CRC Press.

[24] A. Tiwari et. al. (2007). *Analyzing Pathways using SAT-based Approaches*. Proceedings of the 2nd international conference on Algebraic biology, Vol. 4545, pp. 155-169, doi:10.1007/978-3-540-73433-8˙12.

Appendix: Prolog source code

```
%------[ SAT-Solver by J. Howe and A. King ]------
% J. M. Howe and A. King (2010). A Pearl on SAT Solving in Prolog.
% 10th International Symposium on Functional and Logic Programming,
% Vol. 6009, p. 165-174, doi:10.1007/978-3-642-12251-4 13.
% available at http://www.soi.city.ac.uk/~jacob/solver/index.html

initialise(_).

search(Clauses, Vars, Sat, _) :-
    sat(Clauses, Vars),
    !,
    Sat = true.
search(_Clauses, _Vars, false, _).

sat(Clauses, Vars) :-
    problem_setup(Clauses), elim_var(Vars).

elim_var([]).
elim_var([_Label:Var | Vars]) :-
    elim_var(Vars), (Var = true; Var = false).

problem_setup([]).
problem_setup([_Label:Clause | Clauses]) :-
    clause_setup(Clause),
    problem_setup(Clauses).

clause_setup([Pol-Var | Pairs]) :- set_watch(Pairs, Var, Pol).

set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1) :-
    when((nonvar(Var1);nonvar(Var2)),watch(Var1, Pol1, Var2, Pol2, Pairs)).

watch(Var1, Pol1, Var2, Pol2, Pairs) :-
    nonvar(Var1) ->
      update_watch(Var1, Pol1, Var2, Pol2, Pairs);
      update_watch(Var2, Pol2, Var1, Pol1, Pairs).
```

```
38 update_watch(Var1, Pol1, Var2, Pol2, Pairs) :-
       Var1 == Pol1 -> true; set_watch(Pairs, Var2, Pol2).
40
42
44
   %------[ Max-SAT-Solver ]------
46 max_sat(Clauses, Vars, Satisfied, Unsatisfied) :-
        gen_subset(Clauses, Satisfied, Unsatisfied),
48      sat_unsat(Vars, Satisfied, Unsatisfied).

50 max_sat_minimized(Clauses, Vars, Satisfied, Unsatisfied, Sat_Assignments) :-
        gen_subset(Clauses, Satisfied, Unsatisfied),
52      findall(Vars,
                sat_unsat(Vars, Satisfied, Unsatisfied),
54              Sat_Assignments_),
        Sat_Assignments_ \== [],
56      simplify_assignments(Sat_Assignments_, Sat_Assignments).

58
   range(X, _Y, X).
60 range(X, Y, Z) :- X>Y, X1 is X-1, range(X1, Y, Z).
   range(X, Y, Z) :- X<Y, X1 is X+1, range(X1, Y, Z).
62
   subset([],[],[]).
64 subset([E|R], [E|Xs], Ys) :- subset(R, Xs, Ys).
   subset([E|R], Xs, [E|Ys]) :- subset(R, Xs, Ys).
66
   gen_subset(R, X, Y) :-
68      length(R, K),
        range(K, 0, N),
70      M is K - N,
        length(X, N),
72      length(Y, M),
        subset(R, X, Y).
74

76 unsat(true-Var) :- Var = false.
   unsat(false-Var) :- Var = true.
78 unsatisfied(_Label:Clause) :- maplist(unsat, Clause).

80 sat_unsat(Vars, Satisfied, Unsatisfied) :-
        maplist(unsatisfied, Unsatisfied),
82      sat(Satisfied, Vars).

84
   simplify_assignments(Assignments, Simplified) :-
86      merge_assignments(Assignments, Merged),
```

```
     del_implied_vars(Merged, Simplified_),
88   list_to_set(Simplified_, Simplified).

90 merge_assignments([], []) :- !.
   merge_assignments(Assignments, Merged) :-
92   merge_assignments_(Assignments, Merged_Step),
     merge_assignments(Merged_Step, Merged_Rest),
94   append(Assignments, Merged_Rest, Merged).
   merge_assignments_([], []) :- !.
96 merge_assignments_([X|Assignments], Merged) :-
     findall(X_Y,
98           (member(Y, Assignments), merge_vars(X, Y, X_Y)),
             Merged_Current),
100  merge_assignments_(Assignments, Merged_Rest),
     append(Merged_Current, Merged_Rest, Merged).
102
   merge_vars([Label:Var|Assignments1], [Label:Var|Assignments2], [Label:Var|Merged]) :-
104  !, merge_vars(Assignments1, Assignments2, Merged).
   merge_vars([Label:_|Merged], [Label:_|Merged], [Label:dontcare|Merged]) :- !.
106
   del_implied_vars([M|Merged], [M|Simplified]) :-
108  forall(member(M0, Merged), not(merge_vars(M, _, M0))),
     !, del_implied_vars(Merged, Simplified).
110 del_implied_vars([_|Merged], Simplified) :-
     !, del_implied_vars(Merged, Simplified).
112 del_implied_vars([], []).

114



116

118 %------[ Pathway Model encoding ]------
   :- op(500, xfy, '-->').
120 :- op(500, xfy, '--/').
   :- op(500, xfy, '-<>').
122 :- op(500, xfy, '->').
   :- op(500, xfy, '--?').
124
   action(X-->(Y), X, Y).
126 action(X--/(Y), X, Y).
   action(X-<>(Y), X, Y).
128 action(X->(Y), X, Y).
   action(X--?(Y), X, Y).
130
   composite_action(A) :- action(A, _, _).
132

134 valid_model(_Species, []) :- !.
   valid_model(Species, [Term|Actions]) :-
```

```
136       valid_term(Species, Term), !,
          valid_model(Species, Actions).
138
    valid_term(Species, Action) :-
140       action(Action, X, Y),
          member(X, Species),
142       (member(Y, Species);
           valid_term(Species, Y)), !.
144


146


148
    %------[ Model-to-SAT transformation ]------
150 create_sat(Free_Actions, Unfolded_Actions, Species_vars, Actions_vars, Vars, Clauses) :-
          extract_species(Free_Actions, Species),
152       create_vars(Species, Species_vars),
          unfold_actions(Free_Actions, Unfolded_Actions),
154       create_vars(Unfolded_Actions, Actions_vars),
          append(Species_vars, Actions_vars, Vars),
156       maplist(create_clauses(Species_vars, Actions_vars),
                  Unfolded_Actions, Clauses0),
158       flatten(Clauses0, Clauses), !.

160
    extract_species(Actions, Species) :-
162       extract_species_(Actions, Species0),
          list_to_set(Species0, Species).
164 extract_species_([on-<>(T)|Actions], Species) :-
          extract_species_([T|Actions], Species), !.
166 extract_species_([off-<>(T)|Actions], Species) :-
          extract_species_([T|Actions], Species), !.
168 extract_species_([Act|Actions], [S|Species]) :-
          action(Act, S, T), extract_species_([T|Actions], Species), !.
170 extract_species_([S|Actions], [S|Species]) :-
          extract_species_(Actions, Species), !.
172 extract_species_([], []).

174
    unfold_actions(Actions, Unfolded) :-
176       unfold_actions_(Actions, Unfolded0),
          list_to_set(Unfolded0, Unfolded), !.
178 unfold_actions_([S--?(T)|Actions], [S--?(T),S-->(T),S--/(T)|Unfolded]) :-
          unfold_actions_([T|Actions], Unfolded), !.
180 unfold_actions_([A|Actions], [A|Unfolded]) :-
          action(A, _S, T), unfold_actions_([T|Actions], Unfolded), !.
182 unfold_actions_([_|Actions], Unfolded) :-
          unfold_actions_(Actions, Unfolded), !.
184 unfold_actions_([], []).
```

```
186
    create_var(Label, Label:V) :- nonvar(Label), var(V).
188 create_vars(As, Vars) :- maplist(create_var, As, Vars).

190
    % s-->X (s enhances X)
192 create_clauses(Species_vars, Actions_vars, S-->(X), Clauses) :-
        var_of(Species_vars, S, Var_S),
194     var_of(Actions_vars, S-->(X), Var_A),
        var_of(Species_vars, Actions_vars, X, Var_X),
196     Clauses = [S-->(X):[false-Var_S,false-Var_A,true-Var_X]].
    % a--/X (a inhibits X)
198 create_clauses(Species_vars, Actions_vars, S--/(X), Clauses) :-
        var_of(Species_vars, S, Var_S),
200     var_of(Actions_vars, S--/(X), Var_A),
        var_of(Species_vars, Actions_vars, X, Var_X),
202     Clauses = [S--/(X):[false-Var_S,false-Var_A,false-Var_X]].
    % on-<>X (X is on)
204 create_clauses(Species_vars, Actions_vars, on-<>(X), Clauses) :-
        var_of(Actions_vars, on-<>(X), Var_A),
206     var_of(Species_vars, Actions_vars, X, Var_X),
        Clauses = [on-<>(X):[false-Var_A, true-Var_X]].
208 % off-<>X (X is off)
    create_clauses(Species_vars, Actions_vars, off-<>(X), Clauses) :-
210     var_of(Actions_vars, off-<>(X), Var_A),
        var_of(Species_vars, Actions_vars, X, Var_X),
212     Clauses = [off-<>(X):[false-Var_A, false-Var_X]].
    % a-<>X (a controls X enhancing)
214 create_clauses(Species_vars, Actions_vars, S-<>(X), Clauses) :-
        var_of(Species_vars, S, Var_S),
216     var_of(Actions_vars, S-<>(X), Var_A),
        var_of(Species_vars, Actions_vars, X, Var_X),
218     Clauses = [S-<>(X):[false-Var_A,false-Var_S, true-Var_X],
                   S-<>(X):[false-Var_A, true-Var_S,false-Var_X]].
220 % a-><X (a uncontrols X inhibiting)
    create_clauses(Species_vars, Actions_vars, S-><(X), Clauses) :-
222     var_of(Species_vars, S, Var_S),
        var_of(Actions_vars, S-><(X), Var_A),
224     var_of(Species_vars, Actions_vars, X, Var_X),
        Clauses = [S-><(X):[false-Var_A,true-Var_S, true-Var_X],
226                S-><(X):[false-Var_A, false-Var_S,false-Var_X]].
    % a--?X (a enhances or inhibits X)
228 create_clauses(_Species_vars, Actions_vars, S--?(X), Clauses) :-
        var_of(Actions_vars, S--?(X), Var_Aq),
230     var_of(Actions_vars, S-->(X), Var_Ae),
        var_of(Actions_vars, S--/(X), Var_Ai),
232     Clauses = [S--?(X):[false-Var_Aq,true-Var_Ae,true-Var_Ai],
                   S--?(X):[false-Var_Aq,false-Var_Ae,false-Var_Ai]].
```

```
234
236 var_of([X:V|_Vars], X, V) :- !.
    var_of([_|Vars], X, V) :- var_of(Vars, X, V), !.
238
    var_of(_Species_vars, Actions_vars, X, V) :- composite_action(X),
240                                              var_of(Actions_vars, X, V), !.
    var_of(Species_vars, _Actions_vars, X, V) :- var_of(Species_vars, X, V).
242


244


246
    %------[ Finding a model's fixed states ]------
248 fixed_states(Free_Actions, Fixed_States, Unsatisfied_Actions) :-
        create_sat(Free_Actions, _Unfolded_Actions,
250                 _Species_vars, Actions_vars, Vars, Clauses),
        init_free_actions(Free_Actions, Actions_vars),
252     max_sat_minimized(Clauses, Vars,
                          _Satisfied_Clauses, Unsatisfied_Clauses, Fixed_States),
254     maplist(label, Unsatisfied_Actions, _, Unsatisfied_Clauses).

256 init_free_actions(Free_Actions, A_vars) :- maplist(init_action(A_vars), Free_Actions).
    init_action(_A_vars, Action) :- not(composite_action(Action)).
258 init_action(A_vars, Action) :- var_of(A_vars, Action, true).

260
    label(Label, Var, Labeled_Var) :- Labeled_Var = Label:Var.
262


264


266
    %------[ Output in table ]------
268 space_string(0, '') :- !.
    space_string(Length, String) :-
270     Length1 is Length - 1,
        space_string(Length1, String1),
272     string_concat(' ', String1, String).

274 var_specification(Label, Specification) :-
        atom_length(Label, LabelLength),
276     PaddingLength is div(LabelLength - 1, 2) + 1,
        space_string(PaddingLength, Padding),
278     B is mod(LabelLength + 1, 2),
        space_string(B, Padding1),
280     string_concat(Padding, Padding1, Specification1),
        string_concat(Specification1, '~s', Specification2),
282     string_concat(Specification2, Padding, Specification3),
```

```
    string_concat(Specification3, '|', Specification).

state_specification(Species, Specification) :-
    state_specification_(Species, Specification1),
    string_concat(Specification1, '~n', Specification).
state_specification_(Species, Specification) :-
    maplist(term_to_atom, Species, AtomLine),
    maplist(var_specification, AtomLine, Specification1),
    foldr(string_concat, Specification1, '', Specification).

foldr(_F, [], V0, V0).
foldr(F, [X|Xs], V0, V) :- foldr(F, Xs, V0, V1), call(F, X, V1, V), !.

head_specification([], '') :- !.
head_specification([_|Species], Specification) :-
    head_specification(Species, Specification1),
    string_concat(' ~w |', Specification1, Specification).

label_of(Label:_Var, Label).
var_of(_Label:Var, Var).

var_to_symbol(true, '1').
var_to_symbol(false, '0').
var_to_symbol(dontcare, '-').

print_assignments(Assignments) :-
    Assignments = [FirstAssignment|_],
    maplist(label_of, FirstAssignment, Species),
    head_specification(Species, HeadSpecification),
    state_specification(Species, StateSpecification),
    format(HeadSpecification, Species),
    format('~n'),
    maplist(print_assignment(StateSpecification), Assignments).

print_assignment(StateSpecification, Assignment) :-
    maplist(var_of, Assignment, Vars1),
    maplist(var_to_symbol, Vars1, Vars),
    format(StateSpecification, Vars).
```