CHAPTER 1

MODELING AND ANALYSIS OF BIOLOGICAL NETWORKS WITH MODEL CHECKING

Dragan Bošnački, Peter A.J. Hilbers, Ronny S. Mans, and Erik P. de Vink

1.1 INTRODUCTION

Over the last decades, biological networks, like signal transduction pathways, metabolic pathways, and genetic networks, have received increasing attention in biochemistry. In each living organism a growing plethora of such networks have been identified. Is has become clear that the understanding of the mechanisms and their functioning is crucial for elucidating the functioning of the cell and the organism as a whole.

Different formalisms and approaches exist for the modeling of biological networks. In this chapter we focus on model checking as a method that exploits executable models. Its main advantage is that they lend themselves to formal verification. Standard simulation on the model can only yield predictions regarding model properties with certain probability. The advantage of model checking over standard simulation is that it considers all possible behaviors of the systems, not just some subset of it and therefore yield conclusions with certainty.

After intrducing the basic concepts in the next section, in Section 1.3 we show how standard model checking can be used to model and analyze biological systems. To this end we use as the modeling language Promela, the specification language of the model checking tool SPIN. The SPIN tool can be used to check a broad range of properties. In particular, we show how SPIN can be used to detect steady states of the biological systems as well as periodic behavior. Some of the case studies that we discuss have also been modeled with other formalisms, like Petri nets or π-calculus. We discuss the advantages of model checking over those approaches.

Section 1.4 is devoted to modeling and analysis of biological systems which are inherently probabilistic. To this end we use a special kind of model checking – probabilistic model checking. We demonstrate the concepts of probabilistic model checking for biological systems using the probabilistic model checking tool Prism.

1.2 PRELIMINARIES

1.2.1 Model Checking

Roughly speaking, model checking [19, 2] is an automated technique that, given a model of the system and some property, checks whether the model satisfies the property. Compared to other automated or semi-automated formal techniques, such as deductive methods using theorem provers, model checking is relatively easy to use. The specification of the model is very similar to programming and as such it does not require much additional expertise from the user. The verification procedure is completely automated and often takes only seconds to several minutes. Another important advantage of the method is that, if the verification fails, i.e., the property that is checked does not hold, the erroneous behavior of the system can be reproduced. This significantly facilitates the location and correction of the errors.

Unlike simulation, model checking explores all possible states of the system. The model checker explores the complete system behavior, i.e., all possible executions of the system. Obviously, for model checking to be applicable, the state space of the system under study should be finite. Systems with infinite state spaces can be handled as well provided the state space can be reduced to a finite one. For this, various abstraction techniques are available. In general, the state space that reflects all system behavior is represented as a graph in which the states are nodes and the edges are transitions between states. A particular behavior of the system, which we also refer to as an execution sequence or a path, can be represented by a path in the graph consisting of states and the transitions between them. Of particular interest are states from which there are no outgoing transitions, which are called deadlock states, as well as cyclic paths in the state space graph. The deadlock states correspond to steady states in the real systems, whereas the cycles are associated with periodical behavior of the systems.

To illustrate the above notions related to the state spaces we consider the simple genetic network given in Figure 1.1 a, which consists of three genes. Gene A is an activator of gene B, whereas gene B inhibits gene C. Further, if B is not active, then

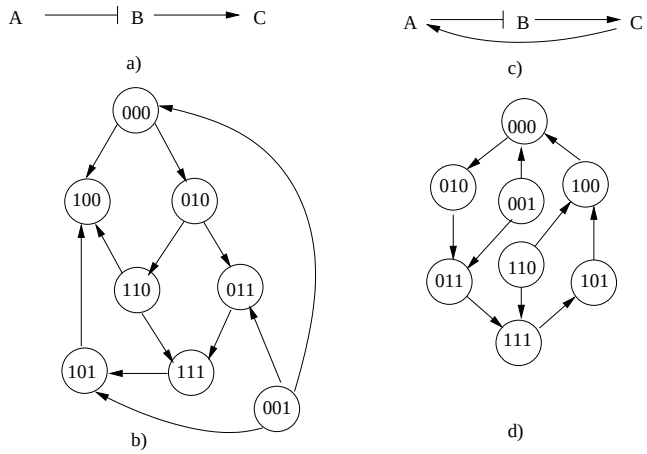

Figure 1.1 A simple genetic network with its state space.

gene C spontaneously deactivates, whereas if gene gene A is not active, then gene B spontaneously activates. Also gene A is a self-activating one. The state space of the network is given in Figure 1.1b. The states are represented by a state vector. For the sake of simplicity, in the figure the state vectors are simplified into three bit binary strings. (States are depicted as circles, transitions as arcs.) The components of the vector correspond to the genes A,B, and C, respectively. Each gene can take two values 0 (not activated) and 1 (activated).

We assume that the genes change their states asynchronously, one gene at a time. So, from the state 000 one can go non-deterministically either to state 100 or 010. The first transition happens when gene A activates spontaneously, whereas the second one corresponds to a spontaneous activation by gene B. The transition from state 101 to 100 happens by spontaneous deactivation of gene C and the transition from 110 to 100 because of the inhibiting influence of A to B. Similarly the other transitions can be coupled to activation/deactivation of a particular gene. The only deadlock or stable state is 100.

The state space does not contain cycles. However, cyclic behavior can be introduced, for instance, by adding a feedback activating influence between C and A and assuming that A deactivates spontaneously (see Figure 1.1c). In that case a spontaneously activated gene B activates gene C, which in its turn activates gene A. Gene A will deactivate gene B, which will result in spontaneous deactivation of gene C and as a consequence also gene A. Then the cycle can resume again with a new spontaneous activation of B. The state space of the new network is given in Figure 1.1d.

1.2.2 SPIN and Promela

Spin [18] is a software tool that supports the analysis and verification of concurrent systems. The system descriptions are modeled in a high-level language called

Promela. Its syntax is derived from the programming language C, and extended with constructs to model non-determinism, the so-called guarded commands due to Dijkstra, and with statements to model communication (sending and receiving messages) which are inspired from Hoare's CSP language.

In Promela, system components are specified as *processes* that can interact either by message passing, via *channels*, or memory sharing, via *global variables*. The message passing can either be buffered or unbuffered. Concurrency is asynchronous and modeled by interleaving, i.e. in every step exactly one *enabled* action is performed, if available at all. No assumptions are made on the relative speed of process executions.

Given a Promela description as input, SPIN generates a C program that performs the verification of a system property by generating the state space graph. Simultaneously, the check of the property is performed, i.e., each new state is checked if it is erroneous, e.g., a deadlock state, or if an erroneous cyclic path is closed. There are various ways to formally express the properties that we want to verify. Properties that boil down to the presence or absence of cycles in the state space can be formulated via special formal language called Linear Temporal Logic (LTL). Cf. [12]. We give more details about LTL in section 1.2.3. The most general way of expressing properties in SPIN is via so-called *never claims*, which are best seen as monitoring processes that run in lock step with the rest of the system.[1] SPIN provides an automatic translator from formulae in LTL to never claims. In case the system violates a property, the trace of actions leading to an invalid state, or a cycle, is reported. The erroneous trace can then be replayed, on the Promela source, by a guided simulation.

1.2.3 LTL

We give only an informal overview of the Linear temporal logic (LTL). For a formal definition, we refer the reader to [12]. Temporal logic is a formalism for specifying sequences of states. Temporal logic formulae are composed out of a small number of special temporal operators and state formulae. Linear temporal logic (LTL) is a specific branch of temporal logic which only contains future time temporal operators. This branch of logic is most relevant to the verification of concurrent systems.

For our purposes we only use two temporal operators. These are the operator *always* or *box*, which is represented by the symbol '$[\,]$' and the operator *eventually* or *diamond*, which is represented by the symbol '$\langle\,\rangle$'. Let us suppose that p is a formula expressing some property. Then formula $[\,]p$ captures the notion that the property specified by p remains invariantly *true* throughout an execution sequence, i.e., holds in each state of the sequence. The informal meaning of the formula $\langle\,\rangle p$ is that the property p is guaranteed to eventually become *true* at least once in an execution sequence. Besides the special temporal operators, LTL also provides the

[1]The never claims are, in fact, Büchi Automata [33], and thus can express what are called arbitrary omega-regular properties.

usual logical operators: '!' for negation, '||' for disjunction, '&&' for conjunction, and '$\rightarrow$' for logical implication. [2]

To illustrate the use of LTL formulae, some examples are given in Table 1.1.

Table 1.1 Examples of LTL formulae.

Formula	Informal meaning
$[]\langle\rangle p$	always eventually p, i.e., infinitely many p's
$\langle\rangle[]p$	eventually always p, i.e., p only from some point on
$p \rightarrow \langle\rangle q$	if initially p then eventually q
$[](p \rightarrow \langle\rangle q)$	every p is eventually followed by a q
$\langle\rangle p \rightarrow \langle\rangle q$	eventually p implies eventually q

1.3 ANALYZING GENETIC NETWORKS WITH MODEL CHECKING

We discuss how to use model checking to analyze genetic networks. In genetic networks, genes can activate or inhibit one another. Moreover, self-regulation (activation/inhibition) of a gene is possible too. We are interested in the qualitative behavior of genetic networks, i.e., for each gene we distinguish only two possible states: 'on' and '1' vs. 'off' and '0'. Consequently, we use boolean regulatory graphs [9] as formal models of the genetic networks.

1.3.1 Boolean Regulatory Networks

Let $G = \{g_1, \ldots, g_n\}$ be a set of genes. To each gene $g_i \in G$, we assign a a subset $I(i) \subseteq G$ and a boolean function K_i. Intuitively, $I(i)$ contains the source genes of all incoming interactions into g_i and is called *input of* g_i. The boolean function $K_i \colon 2^{I(i)} \rightarrow \{0, 1\}$ associates a parameter $K_i(X)$ to each subset X of $I(i)$. Intuitively, if all genes in the subset X are active then g_i is activated (if $K_i(X) = 1$) or inhibited (in case $K_i(X) = 0$). As a result, the output of function $K_i(X)$ produces the *new* value of gene g_i.[3] The corresponding regulatory graph is a (labeled) directed graph defined by the following three components:

- a set of nodes $G = \{g_1, \ldots, g_n\}$,
- a set of edges determined by the sets $I(i), i = 1, \ldots, n$, and
- a set of parameters $\mathcal{K} = \{K_i(X) \mid j = 1, \ldots, n, X \subseteq I(i)\}$.

[2] For the reader who is familiar with LTL and model checking: In most of the applications that we discuss in this chapter the usage of LTL formulae even for safety properties, i.e. properties that can be disproved with a finite counterexample sequence, is essential. This is because those safety properties are global and therefore cannot be expressed with assertions, that capture only local properties.

[3] Thus, the network can be regarded as an asynchronous sequential logical circuit.

1.3.2 A Case Study

We illustrate our approach in more detail on a case study – a genetic network of the plant *Arabidopsis thaliana* which is involved in the control of flower morphogenesis. Mendoza *et al.* [27] have proposed a Boolean regulatory model involving 10 genes cross-regulating each other. For proper parameter value sets, this model encompasses 6 stable states, four of them matching the qualitative gene expression patterns observed in the different flower organs, while the two last stable states correspond to non-flowering situations. All these stable states correspond to deadlocks in the state space of the system and as such can be detected by model checking.

Chaouiya *et al.* have introduced a simplified version of the network which focuses on a subset of six genes that play a crucial role in the selection of specific flowering differentiative pathways, leaving aside the genes which can be treated as simple inputs (EMF1, UFO, LUG and SUP). In [9] a parameter set has been chosen for which the system has four stable states, each corresponding to a gene expression pattern associated with a specific flower organ.

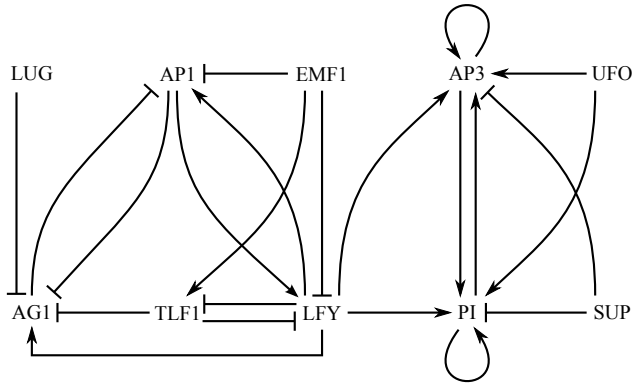

Figure 1.2 Genetic network for flowering in Arabidopsis.

Thus, only the following genes are considered: TLF1, LFY, AP1, AG1, AP3 and PI. The (gene network is depicted in Figure 1.2. Crossbar arrowheads indicate inhibition, while standard arrowheads indicate activation. The inhibition/activation interactions between genes in Figure 1.2 are rather informal. The precise definition of the interactions is given by the K_i parameters of the underlying regulatory graph which are given in Table 1.2. In the table g_i is associated with $g_i = 1$ ($g_i \in X$) and its complement $\bar{g_i}$ corresponds to $g_i = 0$ ($g_i \notin X$). For example, $K_L(A\bar{T}) = 0$ means that when AP1 is activated and TLF1 is inhibited, then gene LFY is inhibited.

1.3.3 Translating Boolean Regulatory Graphs into Promela

As a boolean function each K_i parameter has an unique value for all possible combinations of inputs. Therefore, it is relatively straightforward to model such functions in Promela. We model each gene g_i as a separate process (Promela proctype) which

Table 1.2 Parameters given in [9]

TLF1 (=T)	LFY (=L)	AP1 (=A)	AG1 (=G)	AP3 (=P)	PI (=I)
$K_T(\bar{L}) = 0$	$K_L(\bar{A}\bar{T}) = 0$	$K_A(\bar{L}\bar{G}) = 1$	$K_G(\bar{T}\bar{L}\bar{A}) = 1$	$K_P(\bar{P}\bar{I}\bar{L}) = 0$	$K_I(\bar{I}\bar{P}\bar{L}) = 0$
$K_T(L) = 0$	$K_L(\bar{A}T) = 0$	$K_A(\bar{L}G) = 0$	$K_G(\bar{T}\bar{L}A) = 0$	$K_P(\bar{P}\bar{I}L) = 1$	$K_I(\bar{I}\bar{P}L) = 1$
	$K_L(A\bar{T}) = 0$	$K_A(L\bar{G}) = 1$	$K_G(\bar{T}L\bar{A}) = 1$	$K_P(\bar{P}I\bar{L}) = 0$	$K_I(\bar{I}PL) = 0$
	$K_L(AT) = 0$	$K_A(LG) = 0$	$K_G(\bar{T}LA) = 0$	$K_P(\bar{P}IL) = 1$	$K_I(\bar{I}PL) = 1$
			$K_G(T\bar{L}\bar{A}) = 0$	$K_P(P\bar{I}\bar{L}) = 0$	$K_I(I\bar{P}\bar{L}) = 0$
			$K_G(T\bar{L}A) = 0$	$K_P(P\bar{I}L) = 0$	$K_I(I\bar{P}L) = 0$
			$K_G(TL\bar{A}) = 0$	$K_P(PI\bar{L}) = 1$	$K_I(IP\bar{L}) = 1$
			$K_G(TLA) = 0$	$K_P(PIL) = 1$	$K_I(IPL) = 1$

consists of an (infinite) loop given with the Promela `do` loop:

```
do
::  statement_1
::  statement_2
...
::  statement_n
od.
```

Each `statement` is of the form `condition->action` with the meaning that `action` is executed if the guard `condition` is fulfilled. Further, each `statement` corresponds to a row in definition of function K_i in Table 1.2. For instance, the code for the AG1 gene is given in Listing 1.3.1.

Listing 1.3.1 (`Promela code corresponding to gene AG1`**)**

```
1   proctype AG() {
2   do
3   :: atomic{!Active[G] && !Active[T] && !Active[L] && !Active[A] -> Active[G]=1}
4   :: atomic{!Active[G] && !Active[T] &&  Active[L] && !Active[A] -> Active[G]=1}
5
6   :: atomic{Active[G] && !Active[T] && !Active[L] &&  Active[A] -> Active[G]=0}
7   :: atomic{Active[G] && !Active[T] &&  Active[L] &&  Active[A] -> Active[G]=0}
8   :: atomic{Active[G] &&  Active[T] && !Active[L] && !Active[A] -> Active[G]=0}
9   :: atomic{Active[G] &&  Active[T] && !Active[L] &&  Active[A] -> Active[G]=0}
10  :: atomic{Active[G] &&  Active[T] &&  Active[L] && !Active[A] -> Active[G]=0}
11  :: atomic{Active[G] &&  Active[T] &&  Active[L] &&  Active[A] -> Active[G]=0}
12  od; }
```

Recall that the exclamation mark `!` denotes negation. The `atomic` clause is a technicality which denotes that the enclosed check of the guard and the corresponding action are executed in atomic fashion, i.e., they cannot be interrupted by some statement executed by another process (gene). The complete model is given in the appendix as Listing A.0.1.

The `do` loop is executed as long as at least one of the options is executable. Otherwise the loop is blocked. Each branch (guarded command) of the do loop corresponds to a row in the table defining K_i. The condition (guard) encodes the fact that all genes in X are active. As a result the value of g_i is updated according to K_i from the table.

We simplify the model by the observation that to detect stable states, only the transition of a gene from active to inactive and inactive to active needs to be considered. (When we also model a transition of a gene from active to active and from inactive to inactive then the system can do a step, while the state of the system remains the same. As a result, a deadlock can never appear in the system.) Therefore, an extra variable x_0 is added to $K_i(X)$ which represents the gene i itself, and with which only transitions from inactive to active and from active to inactive can be considered.

Using fairly standard techniques from propositional logic one can also simplify the boolean functions. This often leads to a more compact code, which can be particularly useful for more complex networks. For example, the Promela code of the simplified logical expression of gene AG1 is presented in Listing 1.3.2.

Listing 1.3.2 (Simplified code for gene AG1)

```
1  proctype AG() {
2  do
3  :: atomic{!Active[G] && (!Active[T] && !Active[A]) -> Active[G]=1}
4  :: atomic{ Active[G] &&  (Active[T] ||  Active[A]) -> Active[G]=0}
5  od; }
```

1.3.4 Some Results

Finding Stable States. With the kind of models described above one can find stable states by checking for deadlocks which in SPIN are called *invalid end states*. Since a deadlock state is an error SPIN also always shows a scenario which leads to the found deadlock state. Finding deadlock states in SPIN is an option which is independent form the LTL verification. By default SPIN stops after the first deadlock state is found. This is not very convenient because in this way it is possible to detect only one stable state in the model. In principle, there is an option which instructs SPIN not to stop on the first error - in our case, the first found deadlock - and instead report all found deadlocks. However, for technical reasons that are beyond the scope of this chapter and that are related to SPIN's output, sometimes it could be more convenient to use the following trick: to each found stable state, we add a self-loop to that sate. In this way the latter is not found anymore by the deadlock detection algorithm. To achieve this we add a separate process with a `do`-loop in it that contains the following line: `stable_state -> skip` (where `stable_state` is a correct representation of the stable state in SPIN, and `skip` is a dummy statement). In this way, there is always a transition from `stable_state` to itself. Obviously, one could repeat this procedure until no more stable states are found.

As it was already mentioned above, in order to simplify the model and make the verification more efficient, we exploited the fact that in our setting the stable states correspond to deadlock states. However, with model checking techniques one can detect also a more general type of stable states which are not necessarily deadlocks. They correspond to a partially stable states in which only subset of the genes in the network remains stable.

Finding such states can be achieved in two steps. First, we have to verify that the state itself can be reached. This can be done with the following formula (that, by

default, is checked for all executions): !⟨⟩`state` meaning 'not eventually `state`', i.e., '`state` is never reachable'. When the model checker gives an error trail, then the state exists. Second, we have to verify that the state is indeed a stable state. In other words, when the system reaches the state, it has to remain in that state forever. This can be done with the following formula which has to hold for all executions of the system: `[](state -> []state)`. When the model checker indicates that the verification result is valid, than `state` is indeed stable.

Using deadlock detection techniques, we obtain some interesting results with the Arabidopsis model. In [9] it is claimed that, for the set of parameters given in Table 1.2 and with an initial state in which both LFY and AP1 are active and the others being not active, the system has four stable states. These stable states are shown in Figure 1.3. (The first stable state in the figure indicates that gene A (AP1) is active and the other genes are not. The others can be interpreted analogously.) However, SPIN reports that the last two stable states in Figure 1.3 cannot be reached at all. This can be also analytically proven.

$$\begin{bmatrix} T=0 \\ L=0 \\ A=1 \\ G=0 \\ P=0 \\ I=0 \end{bmatrix} \begin{bmatrix} T=0 \\ L=0 \\ A=1 \\ G=0 \\ P=1 \\ I=1 \end{bmatrix} \begin{bmatrix} T=0 \\ L=0 \\ A=0 \\ G=1 \\ P=0 \\ I=0 \end{bmatrix} \begin{bmatrix} T=0 \\ L=0 \\ A=0 \\ G=1 \\ P=1 \\ I=1 \end{bmatrix}$$

Figure 1.3 The four stable states.

Guided by the counter-examples produced by SPIN, we define an alternative set of parameter values given in Table 1.3 for which all four stable states exist. When choosing these parameter values we have tried to respect as much as possible the activatory and inhibitory relationships among the genes, defined in Figure 1.2. Sometimes this was impossible though. So, in the cases in which the pictorial representation is ambiguous, i.e., for the genes for which there are both activating and inhibiting incoming edges, we have used the predetermined values of the parameters, given in [27]. That those predetermined values are not always in accord with Figure 1.2 can be seen from the following example. In [27] it has been stated that in order to obtain stable states $K_L(AT)$ needs to be set to zero. Since we have that in two of the stable states AP1 is activated and LFY is inhibited, we conclude that $K_L(A)$ has to be set to zero too. This is contradictory with the relationships among the genes in Figure 1.2 which imply that gene AP1 activates gene LFY. By applying similar reasoning we were able to establish the parameter set given in in Table 1.3. With these values for all states in Figure 1.3 we could check with SPIN that they can be reached and that they are indeed stable states. It should be emphasized that despite some discrepancies, the values from Table 1.3 are much closer to the experimental data (see e.g. [27]) than the parameters in Table 1.2 proposed in [9]. Thus, in this we are able to substantially improve the model based on the results obtained with the model checker.

Checking Cycles. Besides stable states, one can also detect cycles in the state space of the network. Suppose that we want to check if there is a cycle consisting

Table 1.3 Set of parameters which respect as much the activatory and repressory relationships and the predetermined values

TLF1 (=T)	LFY (=L)	AP1 (=A)	AG1 (=G)	AP3 (=P)	PI (=I)
$K_T(\bar{L}) = 0$	$K_L(\bar{A}\bar{T}) = 0$	$K_A(\bar{L}\bar{G}) = 1$	$K_G(\bar{T}\bar{L}\bar{A}) = 1$	$K_P(\bar{P}\bar{I}\bar{L}) = 0$	$K_I(\bar{I}\bar{P}\bar{L}) = 0$
$K_T(L) = 0$	$K_L(\bar{A}_T) = 0$	$K_A(\bar{L}_G) = 0$	$K_G(\bar{T}\bar{L}_A) = 0$	$K_P(\bar{P}\bar{I}_L) = 1$	$K_I(\bar{I}\bar{P}_L) = 1$
	$K_L(_A\bar{T}) = 0$	$K_A(_L\bar{G}) = 1$	$K_G(\bar{T}_L\bar{A}) = 1$	$K_P(\bar{P}_I\bar{L}) = 1$	$K_I(\bar{I}_P\bar{L}) = 1$
	$K_L(_AT) = 0$	$K_A(_LG) = 1$	$K_G(\bar{T}_{LA}) = 1$	$K_P(\bar{P}_{IL}) = 1$	$K_I(\bar{I}_{PL}) = 1$
			$K_G(_T\bar{L}\bar{A}) = 0$	$K_P(_P\bar{I}\bar{L}) = 1$	$K_I(_I\bar{P}\bar{L}) = 1$
			$K_G(_T\bar{L}_A) = 0$	$K_P(_P\bar{I}_L) = 1$	$K_I(_I\bar{P}_L) = 1$
			$K_G(_TL\bar{A}) = 1$	$K_P(_PI\bar{L}) = 1$	$K_I(_IP\bar{L}) = 1$
			$K_G(_TLA) = 1$	$K_P(_PIL) = 1$	$K_I(_IPL) = 1$

of (a subset of) the states s_0 to s_n. To find the cycle we have to check the negation of the formula: `!<>[](`s_0`||`s_1`||...||`s_n`)`, i.e., not eventually always one of the states s_0 to s_n. In other words, there is no execution sequence of the model such that some of the states s_0 to s_n is reached, and the system remains in that state forever. If the cycle exists, the previous property does not hold. Strictly speaking, one has to check first that none of the states s_i, $i = 0 \ldots n$ is a deadlock state. If one of the states turns out to be a deadlock, this would automatically mean that a cycle through composed of those states does not exist.

It is worth emphasizing that it takes SPIN just couple of seconds to produce all results described above.

1.3.5 Concluding Remarks

In the previous sections we considered genetic networks which are defined by a boolean function. We presented a method for which a genetic network, which is defined by a boolean function can be translated into Promela. With a small extension to this method, it is possible to detect stable states in the network. Also, we showed some LTL-formulae with which it is possible to verify stable states and cycles.

The use of standard model checking techniques, exploiting Promela and SPIN in particular, are not limited to genetic networks. They can be applied also to other types of biological networks, like signaling and metabolic pathways. More specifically, since SPIN originally has been developed for modeling of communication protocols, it can be used in quite a natural way for modeling signaling pathways.

1.3.6 Related Work and Bibliographic Notes

There are several papers on model checking or closely related formal techniques applied to biological networks (e.g. [20, 6, 30, 25]). Here we briefly discuss in more detail the works that are the most relevant with regard to the subjects treated in this chapter.

In [8] symbolic model checking techniques are applied to the querying and validation of both quantitative and qualitative models of biomolecular systems. The main

difference with our approach is that [8] uses the symbolic model checker NuSMV and the constraint-based model checker DMC, that both accept the temporal logic CTL as a language to formulate the queries (properties). It is well known that for some applications the explicit model checkers, like SPIN, are more efficient and more intuitive to use. For instance, it would be easier to translate the π-calculus into SPIN than into NuSMV. Also the property language of SPIN, Linear Temporal Logic (LTL), and CTL are not comparable in the sense that there are LTL formulae that cannot be expressed in CTL and vice versa.

In [3] an approach for model checking genetic regulatory networks has been proposed which consists in connecting GNA to the CADP verification toolbox. GNA is a quantitative simulation tool well-adapted to the available information on genetic regulatory networks. Also, it is capable of analyzing large and complex genetic regulatory networks. The μ-calculus has been used as a property language. Although the μ-calculus is more general than LTL, in the sense that each LTL formula there is an equivalent μ-calculus formula, the latter are usually much more cryptic and difficult to grasp than their LTL counterparts.

In [1] two tools are described, Simpathica and XSSYS, which involves an automaton-based semantics of the temporal evolution of complex biochemical reactions starting from the representation given as a set of differential equations. Also, the ability is provided to qualitatively reason about the systems using a propositional temporal logic. However, those tools are essentially simulation tools which deal with systems which are deterministic with nature. The Promela/SPIN models are able to capture also the non-deterministic feature of the biological systems.

SPIN is a open source software which is available from `http://spinroot.com`. Other model checking tools can be applied also to biological systems, for instance, NuSMV available from `http://nusmv.irst.itc.it`, and DiVinE, available from `http://divine.fi.muni.cz`.

1.4 PROBABILISTIC MODEL CHECKING FOR BIOLOGICAL SYSTEMS

Many biological systems have inherently a probabilistic/stochastic nature. A probabilistic interpretation, rather than a deterministic one underlying the continuous view based on ordinary differential equations (ODEs), is necessary when the number of molecules in the system is small or the time interval considered is short. A standard examples from the literature are genetic switches, in particular the λ-phage [26]. In this section we consider another class of biological systems in which probabilities play an indispensable role. We show how a specific type of model checking, so-called probabilistic model checking, can be used for such systems. The probabilistic approach we present has been developed in the last several years and constitutes an alternative to traditional methods such as Gillespie-type simulations. Also here, the advantage of probabilistic model checking over simulation is that model checking considers all possible behaviors of the systems, i.e., all simulation runs. Thus, model checking, when feasible, is more reliable and in the case considered also faster

than simulation. To demonstrate the basic approach we use the probabilistic model checker Prism.

1.4.1 Motivation and Background

The transfer of genetic information from DNA to mRNA to protein happens with very high precision. This is because each a single error potentially can have dramatic consequences for the organism as a whole. Here we analyze the stage of this information pathway which corresponds to the translation from mRNA to protein, i.e., the protein biosynthesis. In particular, we are interested in translation errors and the factors of potential influence.

An mRNA molecule can be considered as a string of codons, each of which encodes for a specific amino acid. The codons of an mRNA molecule are sequentially read by a ribosome and each codon is translated into an amino acid. As a result we obtain a chain of amino acids, i.e. a protein. The amino acids are carried to the ribosome by a specific transfer-RNA (aa-tRNA). Each aa-tRNA contains a so-called anticodon and carries a specific amino acid. Arriving by Brownian motion, an aa-tRNA, docks into the ribosome and may succeed in adding its amino acid to the chain under construction. Alternatively, the aa-tRNA dissociates in some stage of the translation. This depends on the pairing of the codon under translation with the anticodon of the aa-tRNA, as well as on the stochastic influences such as the changes in the conformation of the ribosome.

Thanks to the vast amount of research during the last thirty years, the overall process of translation is reasonably well understood from a qualitative perspective. The process can be divided into around twenty small steps/reactions, a number of them being reversible. Relatively little is known exactly about the kinetics of the translation. Over the past several years, Rodnina and collaborators have measured kinetic rates for various steps in the translation process for a small number of specific codons and anticodons [28, 31, 32, 15]. They were able to experimentally show that in several of those steps the rates strongly depend on the degree of matching between the codon and the anticodon. Additionally, in [11] the average concentrations (amounts) of aa-tRNAs per cell have been collected for the model organism *Escherichia coli*. Viljoen and co-workers [13] proposed a model which is based on those results. One of their basic assumption is that the rates found by Rodnina *et al*. can be used in general, for all codon-anticodon pairs. Thus, the model in [13] covers all 64 codons and all 48 aa-tRNA classes for *E. coli*. The model is used to perform extensive Monte Carlo simulations and to establish codon insertion times and frequencies of erroneous elongations. The results show a strong correlation of the translation error and the ratio of the concentrations of the so-called near-cognate and cognate aa-tRNA species. Consequently, one can argue that the competition of aa-tRNAs, rather than their availability, decides both speed and fidelity of codon translation.

In this text, we model the translation kinetics via the modelchecking of continuous-time Markov chains (CTMCs) using the tool Prism [22, 16]. The tool provides built-in performance analysis algorithms and a formalism (Computational Stochastic Logic, CSL) to reason about various properties of the CTMCs, removing the burden of

extensive mathematical calculations from the user. Additionally, in our case, the Prism tool provides much shorter response times compared to Gillespie simulation.

1.4.2 A Kinetic Model of mRNA Translation

There exists a fixed correspondence between codons and amino acids given by the well known genetic code. With exception of the three so called stop codons, which denote the end of the genetic message, each codon codes for exactly one amino acid. As a consequence an mRNA encodes for a unique protein. This ideal picture is disturbed by the fact that, in theory, each codon can bind with each anti-codon. However, the binding intensity can significantly differ from pair to pair. This influences the speed of the actual translation and also the chances for errors. Thus, the translation is quite accurate, but not perfect. The biological model of the translation mechanism that we adopt here is based on [31, 21]. Two main phases can be distinguished: peptidyl transfer and translocation. Here we focus on the peptidyl transfer since it this part that determines the error probabilities. This phase can be divided in several steps which are represented in Figure 1.4 and which we briefly describe in the sequel. The

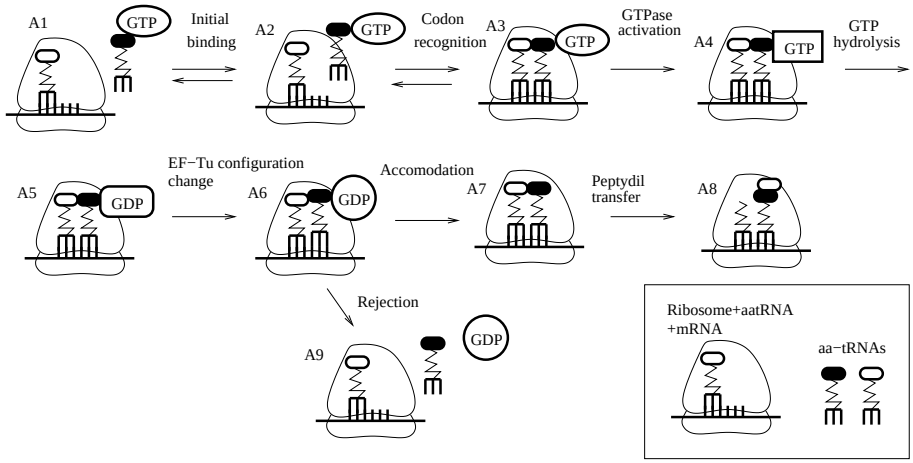

Figure 1.4 Kinetic scheme of peptidyl transfer [13].

transfer begins with aa-tRNA arriving at the A-site of the ribosome-mRNA complex by diffusion (state A1 in Figure 1.4). The initial binding leads to state A2. Since the binding is relatively weak the reverse process, i.e., unbinding of the aa-tRNA is also possible and this brings us to the initial state. Codon recognition comprises (i) establishing contact between the anticodon of the aa-tRNA and the current codon in the ribosome-mRNA complex, and (ii) subsequent conformational changes of the ribosome. *GTP*ase-activation of the elongation factor *EF-Tu* is largely favoured in case of a strong complementary matching of the codon and anticodon. After *GTP*-hydrolysis, producing inorganic phosphate P_i and *GDP*, the affinity of the ribosome for the aa-tRNA reduces. The subsequent accommodation step also depends on the

fit of the aa-tRNA. This step happens rapidly for cognate *aa-tRNA*, whereas for near-cognate *aa-tRNA* this proceeds slower and the *aa-tRNA* is likely to be rejected. These different speeds are expressed via the reaction rates from A6 to A7 and A6 to A9. For a cognate *aa-tRNA* the rate A6–A7 is much bigger than the rate A6–A9, whereas for a near cognate the situation is the other way around.

Next, the translocation phase follows. Another GTP-hydrolysis involving elongation factor *EF-G*, produces *GDP* and P_i and results in unlocking and movement of the aa-tRNA to the P-site of the ribosome. The latter step is preceded or followed by P_i-release. Reconform ation of the ribosome and release of *EF-G* moves the tRNA, that has transferred its amino acid to the polypeptide chain, into the so called E-site of the ribosome. Further rotation eventually leads to dissociation of the used tRNA. As mentioned above, we assume that this phase does not further influence the probability of incorporating the amino acid in the chain. More precisely, we assume that once the final state (A8) of the peptidyl transfer is reached the amino acid will be for sure added to the chain. Because of that in our formal model that we present later we deal only with the peptidyl transfer.

There is not much quantitative information regarding the translation mechanism. For *E. coli*, a number of specific rates have been collected [31, 15], whereas some steps are known to be relatively rapid. Here we adopt the fundamental assumption of [13] that the experimental data found by Rodnina *et al*. for the *UUU* and *CUC* codons, extrapolate to other codons as well. Also, accurate rates for the translocation phase are largely missing. Again following [13], we have chosen to assign, if necessary, high rates to steps for which data is lacking. This way these steps will not be rate limiting.

1.4.3 Probabilistic Model Checking

Traditional model checking, which we presented in the previous sections of this chapter, deals with the notion of absolute correctness or failure of a given property. On the other hand, probabilistic[4] model checking is motivated by the fact that probabilities are often an unavoidable ingredient of the systems we analyze. Therefore, the satisfaction of properties is quantified by a probability. This makes probabilistic model checking a powerful framework for modeling various systems ranging from randomized algorithms via performance analysis to biological networks.

From an algorithmic point of view, probabilistic model checking overlaps with the conventional technique, since it too requires computing reachability of the underlying state space graphs. Still, there are also important differences because numerical methods are used to compute the transition probabilities. However, these details

[4]In the literature probabilistic and stochastic model checking often are used interchangeably. A more clear distinction is made by relating the adjectives probabilistic and stochastic to the underlying model, viz. discrete-time and continuous-time Markov chain, respectively. For the sake of simplicity, in this chapter our focus is on discrete-time Markov chains, so we opted for consistently using the qualification 'probabilistic'. Nevertheless, as we also emphasize in the sequel, the concepts and algorithms that we present here can be applied as well to continuous-time Markov chains.

are beyond the scope of this chapter. They will play no role in the application to biological systems that we consider in the below.

1.4.4 The Prism Model

To obtain our formal Prism model we apply a twofold abstraction to the above informally sketched biological model: (i) Instead of dealing with 48 classes of aa-tRNA, that are identified by the their anticodons, we use four types of aa-tRNA distinguished by their matching strength with the codon under translation. (ii) We combine various detailed steps into one transition. The first reduction greatly simplifies the model, more clearly eliciting the essentials of the underlying process. The second abstraction is more a matter of convenience, though it helps in compactly presenting the model.

For each codon we distinguish four types of aa-tRNA: *cognate, pseudo-cognate, near-cognate, non-cognate*. Cognate aa-tRNAs carry an amino acid which is the correct one for the according to the genetic code and their anticodon strongly couples with the codon. The binding of the anticodon of a pseudo-cognate aa-tRNA or a near-cognate aa-tRNA is weaker, but sufficiently strong to occasionally result in the addition of the amino acid to the nascent protein. In case the amino acid of the aa-tRNA is, accidentally, the right one for the codon, we call the aa-tRNA of the pseudo-cognate type. If the amino acid does not coincide with the amino acid the codon codes for, we speak in such a case of a near-cognate aa-tRNA.[5] The match of the codon and the anticodon can be very poor too. We refer to such aa-tRNA as being non-cognate for the codon. This type of aa-tRNA does not initiate a translation step at the ribosome.

Here we focus on the computation of insertion errors. As a result the model can be even further simplified by assuming that the non-cognates do not play any role in the process. In our model, the main difference of cognates vs. pseudo-cognates and near-cognates is in the kinetics. At various stages of the peptidyl transfer the rates for true cognates differ from the others up to three orders of magnitude.

Figure 1.5 depicts the relevant abstract automaton, derived from the Prism model discussed above. In case a transition is labeled with two rates, the left hand number concerns the processing of a cognate aa-tRNA, the right hand number that of a pseudo-cognate or near-cognate. In three states a probabilistic choice has to be made. The probabilistic choice in state 2 is the same for cognates, pseudo-cognates and near-cognates alike, the ones in state 3 and in state 4 differs for cognates and pseudo-cognates or near-cognates.

For example, after recognition in state 3, a cognate aa-tRNA will go through the hydrolysis phase leading to state 4 for a fraction 0.999 of the cases (computed as $260/(0.23 + 260)$), a fraction being close to 1. In contrast, for a pseudo-cognate or near-cognate aa-tRNA this is 0.005 only. Cognates will accommodate and continue to state 6 with probability 0.736, while pseudo-cognates and near-cognates will do so with the small probability 0.044, the constant FAST being set to 1000 in our

[5]The notion of a pseudo-cognate comes natural in our modeling. However, the distinction between a pseudo-cognate and a near-cognate is non-standard. Usually, a near-cognate refers to both type of tRNA.

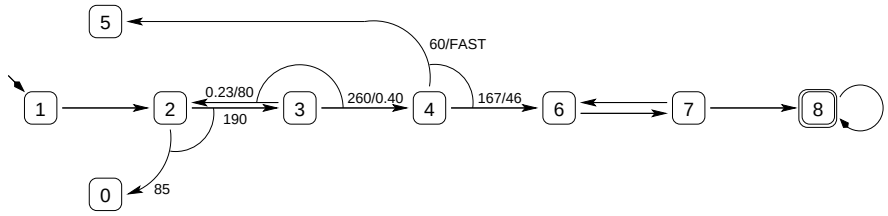

Figure 1.5 Abstract automaton for error insertion.

experiments. As the transition from state 4 to state 6 is irreversible, the rates of the remaining transitions are not of importance here.

One can see the Prism model as a superposition of three stochastic automata, each encoding the interaction of one of the types of aa-tRNA, except the non-cognate type. Each automaton is obtained from the automaton in Figure 1.5 by applying the corresponding rates.

We can further simplify our model by taking into account that we deal with average transition times and probabilities based on exponential distributions. Under this assumption it is a common practice in performance analysis to merge two subsequent sequential transitions with given rates λ and μ into a combined transition of rate $\lambda\mu/(\lambda+\mu)$. However, it should be noted that in general such a simplification is not compositional and should be taken with care.

In the models that we are considering, which are based on continuous-time Markov chains, Prism commands have the form $[\texttt{label}]\ \texttt{guard} \rightarrow \texttt{rate} : \texttt{update}$;. From the commands whose guards are satisfied in the current state, one command is selected with a probability proportional to its relative rate. Thus, a probabilistic choice is made. Executing the selected command results in a progress of time according to the exponential distribution for the particular rate. Also an update is performed on the state variables. More information about the Prism model checker can be found in [22, 16].

Initially, control is in state `s=1` of the Prism model with four boolean variables `cogn`, `pseu`, `near` and `nonc` set to false. The initial binding of aa-tRNA is modeled by selecting one of the boolean variables that is to be set to true. There is a race between the three types of aa-tRNA: cognate, pseudo-cognate, or near-cognate. The outcome of the race depends on the concentrations `c_cogn`, `c_pseu`, `c_near` and `c_nonc` of the three types of aa-tRNA and a kinetic constant `k1f`. According to the Markovian semantics, the probability that `cogn` is set to true (the others remaining false) is the relative concentration $\texttt{c\_cogn}/(\texttt{c\_cogn} + \texttt{c\_pseu} + \texttt{c\_near})$. Analogously the probabilities for the other two types of aa-tRNA are computed. This amounts to the following code:

```
// initial binding
[ ] (s=1) -> k1f * c_cogn : (s'=2) & (cogn'=true) ;
[ ] (s=1) -> k1f * c_pseu : (s'=2) & (pseu'=true) ;
[ ] (s=1) -> k1f * c_near : (s'=2) & (near'=true) ;
```

The aa-tRNA that has just attached can also dissociate. We model this below by returning the control to the state s=0. Although it might seem more natural to return to the initial state, as we will see later, we need this state for model checking purposes. The boolean that was set to true is reset. We assume the same dissociation rate for all aa-tRNA types. rate k2b.

```
// dissociation
[ ] (s=2) -> k2b :
        (s'=0) & (cogn'=false) & (pseu'=false) & (near'=false) ;
```

Regardless of the type, aa-tRNA can continue from state s=2 in the codon recognition phase, leading to state s=3. This step can also be reversed, hence we include transitions from state s=3 back to state s=2. The fidelity of the translation mechanism is ensured by the fact that the rates for cognates vs. pseudo- and near-cognates, viz. k3bc, k3bp and k3bn, differ significantly (see Table 1.4). The boolean variables remain unchanged since aa-tRNA is not released.

```
// codon recognition
[ ] (s=2)        -> k2f :  (s'=3) ;
[ ] (s=3) & cogn -> k3bc : (s'=2) ;
[ ] (s=3) & pseu -> k3bp : (s'=2) ;
[ ] (s=3) & near -> k3bn : (s'=2) ;
```

The next step, from state s=3 to state s=4, is one-direcitonal. It corresponds to a combination of detailed steps in the biological model which involves modification of GTP. We assume that rates k3fp and k3fn, resp. for pseudo-cognate and near-cognate aa-tRNA, are equal. The progress of the translation in the right direction is again ensured by a significant difference between these rates and rate k3fc for a cognate aa-tRNA.

```
// GTPase activation, GTP hydrolysis, EF-Tu conformation change
[ ] (s=3) & cogn -> k3fc : (s'=4) ;
[ ] (s=3) & pseu -> k3fp : (s'=4) ;
[ ] (s=3) & near -> k3fn : (s'=4) ;
```

State s=4 is an important crossroad in the process. The aa-tRNA can either be rejected, after which control moves to the state s=5, or it can be accepted. This corresponds to the various accommodation steps in the biological model, i.e. the ribosome reconforms such that the aa-tRNA can hand over the amino acid it carries, so-called peptidyl transfer. In our model the accepting step means moving to to state s=6. In this step too the rates for cognates and those for pseudo-cognates and near-cognates differ significantly.

```
// rejection
[ ] (s=4) & cogn -> k4rc : (s'=5) & (cogn'=false) ;
[ ] (s=4) & pseu -> k4rp : (s'=5) & (pseu'=false) ;
[ ] (s=4) & near -> k4rn : (s'=5) & (near'=false) ;
// accommodation, peptidyl transfer
[ ] (s=4) & cogn -> k4fc : (s'=6) ;
[ ] (s=4) & pseu -> k4fp : (s'=6) ;
[ ] (s=4) & near -> k4fn : (s'=6) ;
```

The step from state s=6 to state s=7 models the binding of the EF-G complex. This step is also reversible, but eventually the binding becomes permanent. The transition to the final state s=8 subsumes many different steps of the translation mechanism which start with GTP hydrolysis and ends with elongation of the polypeptide chain with the amino acid carried by the aa-tRNA. Non-cognates never pass beyond state s=2, but the outcome of the translation can still be an error if aa-tRNA is near-cognate, i.e., if boolean near is true. In this case an amino acid is inserted that does not correspond to the codon in the genetic code.

```
// EF-G binding
[ ] (s=6) -> k6f : (s'=7) ;
[ ] (s=7) -> k7b : (s'=6) ;
// GTP hydrolysis, unlocking, tRNA movement and Pi release,
// rearrangements of ribosome and EF-G, dissociation of GDP
[ ] (s=7) -> k7f : (s'=8) ;
```

The model is completed by transitions from the dissociation state s=0 and the rejection state s=5 back to the start state s=1. After a aa-tRNA is rejected the race of the four aa-tRNA types resumes. Also, for technical reasons, a self-loop at the final state s=8 is added.

```
// no entrance, re-entrance at state 1
[ ] (s=0) -> FAST : (s'=1) ;
// rejection, re-entrance at state 1
[ ] (s=5) -> FAST : (s'=1) ;
// elongation
[ ] (s=8) -> FAST : (s'=8) ;
```

The rates that are used in our model are given in Table 1.4. they are collected from the biological literature [31, 13].

Table 1.4 Rates of the Prism model

k1f	140	k3fc	260	k4rc	60	k6f	150
k2f	190	k3fp, k3fn	0.40	k4rp, k4rn	FAST	k7f	145.8
k2b	85	k3bc	0.23	k4fc	166.7	k7b	140
k2bx	2000	k3bp, k3bn	80	k4fp, k4fn	46.1		

The complete Prism model can be found in the appendix as Listing A.0.2. In the sequel we use the Prism model described above for the analysis of the probability for insertion errors, i.e. the chance that the peptidyl chain is extended with an amino acid that differs from the one encoded by the codon which is translated.

1.4.5 Insertion Errors

Once having the model we can use the model checking capabilities of the Prism tool to predict the misreading frequencies for individual codons. To this end we need to give Prism the exact property that corresponds to our question about the probability. In other words, we need the right formula with the above mentioned logic CSL. The

formula should state that we want to compute the probability that an erroneous state is reached in which a wrong amino acid is added.

For a codon under translation, a pseudo-cognate anticodon carries precisely the amino acid that the codon codes for. Therefore, successful matching of a pseudo-cognate does not lead to an insertion error.

Taking into account the above we come up with the following CSL-formula:

```
P=?  [ (true) U ((s=8) and not near) ]
```

The formula is of the form `P=?[Φ]`, which is a basic formula template for CSL. The part `P=?` means that we want a numerical result, i.e., the cumulative probability of all paths that satisfy formula Φ. Like the formulae for the LTL logic in standard model checking, CSL formulae are also interpreted on sequences of states, i.e., paths, of the model. So, the inner formula Φ states that we are interested only in paths that end in state `s=8` – in which the amino acid is added to the chain – and moreover, that the added amino acid is the wrong one, i.e., the tRNA is not cognate or pseudo-cognate, but near-cognate. This last fact is expressed as `near`, where `not` is the negation operator. The paths start by default in the initial state `s=0`. The formula Φ itself is of the form Φ_1 U Φ_2, where U is the so called until operator. [6] The meaning of this kind of formulae is that along the path formula Φ_1 must hold until a state is reached in which formula Φ_2 holds. If Φ_2 holds in the initial state Φ_1 does not need to hold in that state, since in this case the formulae is trivially true. In our case we have set Φ_1 to `true`. Since `true` holds trivially in all states, this means that we do not care about the intermediate states of the path and that it is only important that a state is reached in which Φ_2 holds, i.e., a wrong amino acid is added to the chain.

Our results obtained with Prism are given in Table 1.5. Prism produces these

Table 1.5 Probabilities per codon for erroneous elongation

UUU	27.4e-4	CUU	46.7e-4	GUU	1.12e-10	AUU	14.4e-4
UUC	91.2e-4	CUC	13.6e-4	GUC	55.0e-4	AUC	35.0e-4
UUG	7.59e-4	CUG	4.49e-4	GUG	2.68e-4	AUG	58.3e-4
UUA	23.5e-4	CUA	18.9e-4	GUA	22.3e-4	AUA	34.4e-4
UCU	2.81e-10	CCU	34.1e-4	GCU	1.77e-10	ACU	2.73e-10
UCC	56.1e-4	CCC	10.4e-4	GCC	12.5e-4	ACC	34.2e-4
UCG	20.3e-4	CCG	37.6e-4	GCG	3.187e-4	ACG	31.7e-4
UCA	9.09e-4	CCA	22.8e-4	GCA	28.2e-4	ACA	29.1e-4
UGU	6.97e-4	CGU	1.21e-10	GGU	1.32e-10	AGU	8.70e-4
UGC	30.4e-4	CGC	4.59e-4	GGC	9.40e-4	AGC	37.2e-4
UGG	39.8e-4	CGG	88.7e-4	GGG	2.72e-10	AGG	140.7e-4
UGA	7.50e-4	CGA	3.98e-4	GGA	100.3e-4	AGA	48.1e-4
UAU	2.81e-10	CAU	91.1e-4	GAU	18.6e-4	AAU	15.2e-4
UAC	15.7e-4	CAC	47.5e-4	GAC	43.2e-4	AAC	49.3e-4
UAG	41.3e-4	CAG	69.4e-4	GAG	7.09e-4	AAG	32.1e-4
UAA	6.04e-4	CAA	22.7e-4	GAA	21.4e-4	AAA	14.6e-4

[6] Actually, this operator exists also in LTL, but we 'hid' it in the temporal operators $[\,]$ and $\langle\,\rangle$. The latter are just syntactic sugar and they can be expressed using the until operator U.

results within a couple of minutes. Checking for an individual codon takes just a few seconds.

As reported in [13], the probability for an erroneous insertion, is strongly correlated with the quotient of the number of near-cognate anticodons and the number of cognate anticodons.This can be seen also in Figure 1.6. On the y-axis is the quotient of the

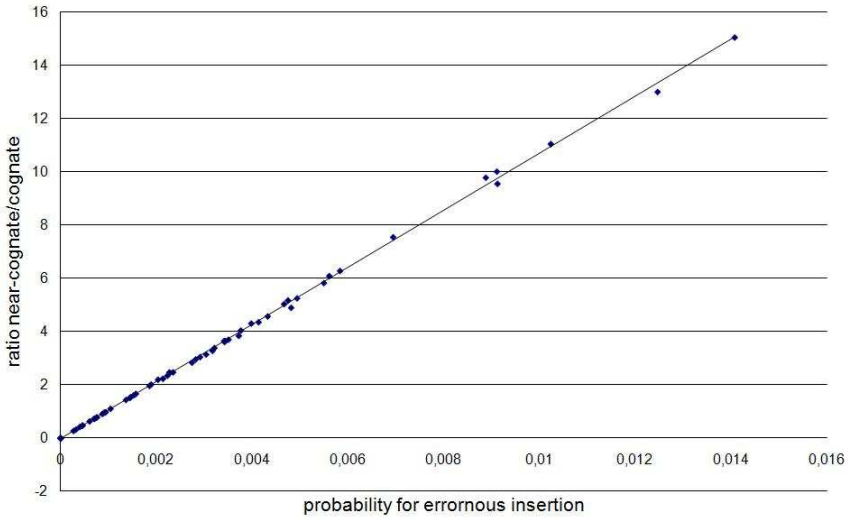

Figure 1.6 Correlation of ratio near-cognate vs. cognates aa-tRNAs and error probabilities.

concentrations (number of molecules) of near-cognate and cognate tRNAs, whereas on the x-axis are the probabilities for erroneous insertion.

1.4.6 Concluding Remarks

We showed how probabilistic model checking can be used to analyze biological networks as an alternative for Gilliespie-like simulation. As an example we discussed a stochastic model of the translation process based on realistic data of ribosome kinetics. We used the probabilistic model checker Prism and in particular its capabilities to deal with continuous time Markov chains. Compared to simulation, our approach is computationally more reliable as it is independent on the number of simulations. Also, it this case, it has faster response times, taking seconds rather then minutes or hours.

The kind of probabilistic/stochastic models, as we presented here, has opened new avenues for future work on biological systems that possess intrinsically probabilistic properties. E.g., current research using the model checking based method is targeted at biological processes that require high precision, like DNA translation, DNA repair, charging of the tRNAs with amino acids, etc. In [4] we show how with our model one

could check if amino acids with similar biochemical properties substitute erroneously for one another with greater probabilities than dissimilar ones.

1.4.7 Related Work and Bibliographic Notes

The model that is used in this chapter builds upon [5], which was inspired by the simulation experiments of mRNA translation reported in [13]. A similar model, based on ordinary differential equations, was developed in [17]. Although probabilistic, it is used to compute insertion times, but no translation errors. The model of mRNA translation in [14] assumes insertion rates that are directly proportional to the mRNA concentrations, but assigns the same probability of translation error to all codons.

Applications of probabilistic model checking and in particular Prism can be found in [24]. More about probabilistic model checking and the underlying algorithms can be found in [23].

There exist numerous applications of formal methods to biological systems. A selection of recent papers from model checking and process algebra includes [29, 8, 10]. More specifically pertaining to this chapter, [7] applies the Prism model checker to analyze stochastic models of signaling pathways. Their methodology is presented as a more efficient alternative to ordinary differential equations models, including properties that are not of probabilistic nature. Also, [16] employs Prism on various types of biological pathways, showing how the advanced features of the tool can be exploited to tackle large models.

Prism is an free available software and can be downloaded from its web page `http://www.prismmodelchecker.org`. Of course, any model checking tool that supports CTMCs can be used too for analyzing biological systems. One such a tool is MRMC which is also in the public domain, see `http://www.mrmc-tool.org`.

REFERENCES

1. M. Antoniotti, A. Policriti, N. Ugel, and B. Mishra. Model building and model checking for biochemical processes. *Cell Biochemistry and Biophysics*, 38:271–286, 2003.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
3. G. Batt, D. Bergamini, H. de Jong, H. Garavel, and R. Mateescu. Model checking genetic regulatory networks using gna and cadp. In *Proceedings of the 11th SPIN Workshop Barcelone, Spain*, volume 2989 of *Lecture Notes in Computer Science*, 2004.
4. D. Bošnački, H.M.M. ten Eikelder, M.N. Steijaert, and E.P. de Vink. Stochastic analysis of amino acid substitution in protein synthesis. In M. Heiner and A.M. Uhrmacher, editors, *Proc. CMSB 2008*, pages 367–386. LNCS 5307, 2008.
5. D. Bošnački, T.E. Pronk, and E.P. Vink. In silico modelling and analysis of ribosome kinetics and aa-trna competition. *Transactions on Computational Systems Biology*, IX:69–89, 2009. CompMod 2008 special issue, R.-J. Back and I. Petre, guest editors.
6. M. Calder, V. Vyshemirsky, D. Gilbert, and R. Orton. Analysis of signalling pathways using the prism model checker. In *CMSB*, pages 179–190, 2005.

7. M. Calder, V. Vyshemirsky, D. Gilbert, and R. Orton. Analysis of signalling pathways using continuous time Markov chains. In *Transactions on Computational Systems Biology VI*, pages 44–67. LNBI 4220, 2006.

8. N. Chabrier and F. Fages. Symbolic model checking of biochemical networks. In *Proc. CMSB 2003*, pages 149–162. LNCS 2602, 2003.

9. C. Chaouiya, E. Remy, P. Ruet, and D. Thieffry. Qualitative modelling of genetic networks: From logical regulatory graphs to standard petri nets. In J. Cortadella and W. Reisig, editors, *ICATPN*, volume 3099 of *Lecture Notes in Computer Science*. Springer, 2004.

10. V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling. In *Proc. CONCUR*, pages 17–41. LNCS 4703, 2007.

11. H. Dong, L. Nilsson, and C.G. Kurland. Co-variation of tRNA abundance and codon usage in *Escherichia coli* at different growth rates. *Journal of Molecular Biology*, 260:649–663, 1996.

12. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, pages 995–1072. Elsevier, 1990.

13. A. Fluitt, E. Pienaar, and H. Viljoen. Ribosome kinetics and aa-tRNA competition determine rate and fidelity of peptide synthesis. *Computational Biology and Chemistry*, 31:335–346, 2007.

14. M.A. Gilchrist and A. Wagner. A model of protein translation including codon bias, nonsense errors, and ribosome recycling. *Journal of Theoretical Biology*, 239:417–434, 2006.

15. K.B. Gromadski and M.V. Rodnina. Kinetic determinants of high-fidelity tRNA discrimination on the ribosome. *Molecular Cell*, 13(2):191–200, 2004.

16. J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. In *Proc. CMSB 2006*, pages 32–47. LNBI 4210, 2006.

17. A.W. Heyd and D.A. Drew. A mathematical model for elongation of a peptide chain. *Bulletin of Mathematical Biology*, 65:1095–1109, 2003.

18. G.J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

19. E.M. Clarke Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 2000.

20. N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E.J. Albert Hubbard, and M.J. Stern. Formal modeling of c. elegans development: A scenario-based approach. In C. Priami, editor, *CMSB*, volume 2602 of *Lecture Notes in Computer Science*, pages 4–20. Springer, 2003.

21. G. Karp. *Cell and Molecular Biology, 5th ed.* Wiley, 2008.

22. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model cheking with Prism: a hybrid approach. *Journal on Software Tools for Technology Transfer*, 6:128–142, 2004. See also `http://www.prismmodelchecker.org/`.

23. M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

24. M. Kwiatkowska, G. Norman, and D. Parker. *Symbolic Systems Biology*, chapter Probabilistic Model Checking for Systems Biology. Jones and Bartlett, 2009. To appear.

25. P. Lecca and C. Priami. Cell cycle control in eukaryotes: A biospi model. In *Proc. Workshop on Concurrent Models in Molecular Biology (BioConcur'03)*, ENTCS, 2003.

26. H.H. McAdams and A. Arkin. Stochastic mechanisms in gene expression. *PNAS*, 94:814–819, 1997.

27. L. Mendoza, D. Thieffry, and E.R. Alvarez-Buylla. Genetic control of flower morphogenesis in arabidopsis thaliana: a logical analysis. *Bioinformatics*, 15(7/8):593–606, February 1999.

28. T. Pape, W. Wintermeyer, and M. Rodnina. Complete kinetic mechanism of elongation factor Tu-dependent binding of aa-tRNA to the A-site of *E. coli*. *EMBO Journal*, 17:7490–7497, 1998.

29. C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a stochastic name-passing calculus to represent ation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.

30. A. Regev. *Computational Systems Biology: A Calculus for Biomolecular Knowledge*. PhD thesis, Tel Aviv: Tel Aviv University, December 2002.

31. M.V. Rodnina and W. Wintermeyer. Ribosome fidelity: tRNA discrimination, proofreading and induced fit. *TRENDS in Biochemical Sciences*, 26(2):124–130, 2001.

32. A. Savelsbergh, V.I. Katunin, D. Mohr, F. Peske, M.V. Rodnina, and W. Wintermeyer. An elongation factor G-induced ribosome rearrangement precedes tRNA–mRNA translocation. *Molecular Cell*, 11:1517–1523, 2003.

33. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Formal Models and Semantics*, pages 995–1072. Elsevier, 1990.

Appendix

Listing A.0.1 (Promela description of the Arabidopsis network)

```
1  #define T       0
2  #define L       1
3  #define A       2
4  #define G       3
5  #define P       4
6  #define I       5
7
8  bool Active[6];
9
10 proctype TLF1() {
11 do
12 :: atomic{Active[T] && !Active[L] -> Active[T]=0}
13 :: atomic{Active[T] && Active[L] -> Active[T]=0}
14 od; }
15
16 proctype LFY() {
17 do
18 :: atomic{Active[L] && !Active[A] && !Active[T] -> Active[L]=0}
19 :: atomic{Active[L] && !Active[A] && Active[T] -> Active[L]=0}
20 :: atomic{Active[L] && Active[A] && !Active[T] -> Active[L]=0}
21 :: atomic{Active[L] && Active[A] && Active[T] -> Active[L]=0}
```

```
od; }

proctype AP1() {
do
:: atomic{!Active[A] && !Active[L] && !Active[G] -> Active[A]=1}
:: atomic{!Active[A] && Active[L] && !Active[G] -> Active[A]=1}
:: atomic{Active[A] && !Active[L] && Active[G] -> Active[A]=0}
:: atomic{Active[A] && Active[L] && Active[G] -> Active[A]=0}
od; }

proctype AG() {
do
:: atomic{!Active[G] && !Active[T] && !Active[L] && !Active[A] -> Active[G]=1}
:: atomic{!Active[G] && !Active[T] && Active[L] && !Active[A] -> Active[G]=1}
:: atomic{Active[G] && !Active[T] && !Active[L] && Active[A] -> Active[G]=0}
:: atomic{Active[G] && !Active[T] && Active[L] && Active[A] -> Active[G]=0}
:: atomic{Active[G] && Active[T] && !Active[L] && !Active[A] -> Active[G]=0}
:: atomic{Active[G] && Active[T] && !Active[L] && Active[A] -> Active[G]=0}
:: atomic{Active[G] && Active[T] && Active[L] && !Active[A] -> Active[G]=0}
:: atomic{Active[G] && Active[T] && Active[L] && Active[A] -> Active[G]=0}
od; }

proctype AP3() {
do
:: atomic{!Active[P] && !Active[I] && Active[L] -> Active[P]=1}
:: atomic{!Active[P] && Active[I] && Active[L] -> Active[P]=1}
:: atomic{Active[P] && !Active[I] && !Active[L] -> Active[P]=0}
:: atomic{Active[P] && !Active[I] && Active[L] -> Active[P]=0}
od; }

proctype PI() {
do
:: atomic{!Active[I] && !Active[P] && Active[L] -> Active[I]=1}
:: atomic{!Active[I] && Active[P] && Active[L] -> Active[I]=1}
:: atomic{Active[I] && !Active[P] && !Active[L] -> Active[I]=0}
:: atomic{Active[I] && !Active[P] && Active[L] -> Active[I]=0}
od; }

init {
atomic{
  Active[L]=1;
  Active[A]=1;
  run TLF1();
  run LFY();
  run AP1();
  run AG();
  run AP3();
  run PI();
  }
}
```

Listing A.0.2 (Prism model of mRNA translation)

```
stochastic

// constants
const double ONE=1;
const double FAST=1000;

// tRNA rates
const double c_cogn ;
const double c_pseu ;
const double c_near ;
const double c_nonc ;

const double k1f = 140;
```

```
const double k2b =  85;
const double k2bx=2000;
const double k2f = 190;
const double k3bc=   0.23;
const double k3bp=  80;
const double k3bn=  80;
const double k3fc= 260;
const double k3fp=   0.40;
const double k3fn=   0.40;
const double k4rc=  60;
const double k4rp=FAST;
const double k4rn=FAST;
const double k4fc= 166.7;
const double k4fp=  46.1;
const double k4fn=  46.1;
const double k6f = 150;
const double k7b = 140;
const double k7f = 145.8;

module ribosome

s : [0..8] init 1 ;
cogn : bool init false ;
pseu : bool init false ;
near : bool init false ;
nonc : bool init false ;

// initial binding
[ ] (s=1) -> k1f * c_cogn : (s'=2) & (cogn'=true) ;
[ ] (s=1) -> k1f * c_pseu : (s'=2) & (pseu'=true) ;
[ ] (s=1) -> k1f * c_near : (s'=2) & (near'=true) ;

[ ] (s=2) -> k2b : (s'=0) &
      (cogn'=false) & (pseu'=false) & (near'=false) ;

// codon recognition
[ ] (s=2) &      -> k2f : (s'=3) ;
[ ] (s=3) & cogn -> k3bc : (s'=2) ;
[ ] (s=3) & pseu -> k3bp : (s'=2) ;
[ ] (s=3) & near -> k3bn : (s'=2) ;

// GTPase activation, GTP hydrolysis, reconformation
[ ] (s=3) & cogn -> k3fc : (s'=4) ;
[ ] (s=3) & pseu -> k3fp : (s'=4) ;
[ ] (s=3) & near -> k3fn : (s'=4) ;

// rejection
[ ] (s=4) & cogn -> k4rc : (s'=5) & (cogn'=false) ;
[ ] (s=4) & pseu -> k4rp : (s'=5) & (pseu'=false) ;
[ ] (s=4) & near -> k4rn : (s'=5) & (near'=false) ;


// accommodation, peptidyl transfer
[ ] (s=4) & cogn -> k4fc : (s'=6) ;
[ ] (s=4) & pseu -> k4fp : (s'=6) ;
[ ] (s=4) & near -> k4fn : (s'=6) ;

// EF-G binding
[ ] (s=6) -> k6f : (s'=7) ;
[ ] (s=7) -> k7b : (s'=6) ;

// GTP hydrolysis, unlocking,
// tRNA movement and Pi release,
// rearrangements of ribosome and EF-G,
// dissociation of GDP
[ ] (s=7) -> k7f : (s'=8) ;

```

```
81  // no entrance, re-entrance at state 1
82  [ ] (s=0) -> FAST : (s'=1) ;
83  // rejection, re-entrance at state 1
84  [ ] (s=5) -> FAST : (s'=1) ;
85  // elongation
86  [ ] (s=8) -> FAST : (s'=8) ;
87
88  endmodule
```