MŰEGYETEM 1782

DIPLOMATERVEZÉSI FELADAT

Marussy Kristóf
mérnök informatikus hallgató részére

Tervezésitér-bejárás sztochasztikus metrikákkal

A kritikus rendszerek – biztonságkritikus, elosztott és felhő-alapú alkalmazások – helyességének biztosításához szükséges a funkcionális és nemfunkcionális követelmények matematikai igényességű ellenőrzése. Számos, szolgáltatásbiztonsággal és teljesítményvizsgálattal kapcsolatos tipikus kérdés jellemzően sztochasztikus analízis segítségével válaszolható meg, amely analízis elvégzésére változatos eszközök állnak a mérnökök rendelkezésére. Ezen megközelítések hiányossága azonban, hogy egyrészt az általuk támogatott formális nyelvek a mérnökök számára nehezen érthetőek, másrészt az esetleges hiányosságok kimutatásán túl nem képesek javaslatot tenni a rendszer kijavítására, azaz a megfelelő rendszerkonfiguráció megtalálására.

Előnyös lenne egy olyan modellezési környezet fejlesztése, amely támogatja a sztochasztikus metrikák alapján történő mérnöki modellfejlesztést, biztosítja a mérnöki modellek automatikus leképezését formális sztochasztikus modellekre, továbbá alkalmas az elkészült rendszertervek optimalizálására tervezésitér-bejárás segítségével. Mind sztochasztikus analízisre, mind pedig tervezésitér-bejárásra elérhető eszköztámogatás, azonban ezen megközelítések hatékony integrációja egy egységes keretrendszerben komplex feladat mind elméleti, mind gyakorlati szempontból.

A hallgató feladata megismerni a sztochasztikus analízis algoritmusokat és a tervezésitér-bejáró módszereket, majd a két megközelítés kombinálásával létrehozni egy keretrendszert a kvantitatív mérnöki tervezés támogatása érdekében.

A hallgató feladatának a következőkre kell kiterjednie:

1. Vizsgálja meg az irodalomban ismert technikákat a sztochasztikus modellek analízise és optimalizálása területén!
2. Tervezzen meg egy eszközt sztochasztikus metrika alapú tervezésitér-bejárás támogatására, ügyelve rá, hogy a megoldás a tervező mérnököktől ne igényeljen további különleges szaktudást!
3. Implementálja a megtervezett rendszert és egy esettanulmánnyal illusztrálja a megközelítés működését!
4. Értékelje a megoldást és vizsgálja meg a továbbfejlesztési lehetőségeket.

Tanszéki konzulens: Molnár Vince, doktorandusz

Budapest, 2017. március 9.

Dr. Dabóczi Tamás
egyetemi docens
tanszékvezető

M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Kristóf Marussy

Design Space Exploration with Stochastic Metrics

Master's Thesis

Supervisors:

Vince Molnár
András Vörös

Budapest, 2017

Typeset with the *XCharter* (by Michael Sharpe and others),
Fira Sans and *Fira Mono* (by the Mozilla Foundation) typefaces
and the *Libertine* (by the Open Fonts Projekt) typeface for mathematics
using the *pdfLaTeX* (by the TeX Users Group) typesetting engine
and the *Memoir* (by Peter Wilson, Lars Madsen and others) document class
on 86 A4 pages numbered i–x and 1–76
and 11 A4 pages of appendices numbered 77–87
on December 14, 2017.

Contents

Kivonat Ide kerül a kivonat.

Kulcsszavak diplomaterv, sablon, LaTeX

Abstract Here comes the abstract.

Keywords thesis, template, LaTeX

Hallgatói nyilatkozat

Chapter 1

Introduction

[TODO: General intro text comes here]

1.1 Related work: design-space exploration and stochastic analysis

1.2 Overview of the approach

Chapter 2

Background

The purpose of this chapter is to briefly recall some preliminaries related to our work, including concepts from model-driven engineering, as well as formal and stochastic modeling. As our work attempts to aid the calculation of stochastic metrics based on engineering models, we describe the stochastic analysis tasks that we are aiming to support.

Chapter 3 builds on generalized stochastic Petri nets, which are described in Section 2.2.4 on page 11, in order to define a formalism suitable for analysis model fragments. Chapter 4 uses incremental query execution of graph patterns, which are describen in Section 2.1.2 on page 5, to instantiate the analysis model fragments according to a changing domain model. Lastly, Chapter 5 discusses the application of the transformation toolchain to aid proposing and solving stochastic analysis task from Section 2.2.3 on page 10 in design-space exploration for the calculation of stochastic metrics on engineering models.

Furthermore, Running example 2.1 introduces the *dining philosophers* problem, which is used as a running example throughout this thesis.

2.1 Modeling and metamodeling

In model-driven engineering *models* provide abstractions of reality, including the structure and the behavior of systems we wish to analyze or design [Giese et al., 2007]. The models are described in some *formalism* or *modeling language*.

The syntax of modeling languages is traditionally partitioned into *abstract* and *concrete syntax*. Concrete syntax provides a textual or graphical representation of the modeling language. Abstract syntax is endowed with meaning by mapping into a *semantic domain*, which is another modeling language on a lower abstraction level.

2.1.1 Metamodels and instance models

Metamodels explicitly describe the abstract syntax of modeling languages. We will adopt the formal description of metamodels and first-order logical structures from the work of Varró et al. [2017] and extend them to support primitive-valued attributes. Metamodels are regarded as first-order signatures:

Definition 2.1 A *metamodel* is a first-order two-sorted logical signature

$$\Sigma = \{\mathsf{C}_1, \mathsf{C}_2, \dots, \mathsf{C}_n; \mathsf{R}_1, \mathsf{R}_2, \dots, \mathsf{R}_m; \mathsf{A}_1, \mathsf{A}_2, \dots, A_k\},$$

where

- *OBJ* and *PRIM* are the sorts of objects and primitives, respectively;
- $\mathsf{C}_1, \mathsf{C}_2, \ldots, \mathsf{C}_n :: \mathit{OBJ}$ are unary relation symbols called *classes*;
- $\mathsf{R}_1, \mathsf{R}_2, \ldots, \mathsf{R}_m :: \mathit{OBJ} \times \mathit{OBJ}$ are binary relation symbols called *references* (or *edges*);
- $\mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_k :: \mathit{OBJ} \times \mathit{PRIM}$ are binary relation symbols called *attributes*.

Instances of metamodels are regarded as first-order structures:

Definition 2.2 An *instance model* $M = \langle \mathcal{O}, \mathit{Prim}, \mathcal{I} \rangle$ of the metamodel Σ is a first-order two-sorted structure, where
- $\mathcal{O} = \{o_1, o_2, \ldots, o_N\}$ is a finite set of *objects* (or *individuals*);
- *Prim* is a set of possible primitive values of attributes, for example, $\mathit{Prim} \subseteq \mathbb{R} \cup \mathbb{B}$, where $\mathbb{R}$ is the set of real numbers and $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$;
- $\mathcal{I} : \Sigma \to \mathcal{O} \cup (\mathcal{O} \times \mathcal{O}) \cup (\mathcal{O} \times \mathit{Prim})$ is the *interpretation function*, such that
  - $\mathcal{I}(\mathsf{C}_i) = \mathsf{C}_i^M \subseteq \mathcal{O}$ for all classes $\mathsf{C}_i \in \Sigma$;
  - $\mathcal{I}(\mathsf{R}_i) = \mathsf{R}_i^M \subseteq \mathcal{O} \times \mathcal{O}$ for all references $\mathsf{R}_i \in \Sigma$;
  - $\mathcal{I}(\mathsf{A}_i) = \mathsf{A}_i^M \subseteq \mathcal{O} \times \mathit{Prim}$ for all attributes $\mathsf{A}_i \in \Sigma$.

Existing metamodeling technologies, such as the Ecore metamodeling langauge from the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009] often allow the designer of modeling languages to impose additional constraints on instances models. Some possible restrictions are listed below:

- *Class inheritance* may specify that instances of a class must be instances of another.
- *Abstract classes* and *interfaces* can have no direct instances.
- *Type constraints* on references and attributes restrict the class of the object originating the reference or attribute and the class of the object at the end of the reference, as well as the type of the primitive attribute value.
- *Multiplicity constraints* place lower or upper bounds on the number of references or attributes from an object.
- Objects may be organized into a *containment hierarchy*, in which and object is either a *containment root*, or is connected to its *container* in the hierarchy with exactly one *containment* reference. The containment hierarchy forms a forest of containment roots, which is traversed when the object graph is serialized.
- *Opposite constants* require some references to always go in a direction opposite to another, i.e. $\mathsf{R}(o_i, o_j)$ should hold if and only if we have $\mathsf{R}^{\mathrm{op}}(o_j, o_i)$.

Further *well-formedness constraints* may be specified with *constraint languages*, such as the object constraint language (OCL) [Object Management Group, 2014] or with graph patterns [Bergmann et al., 2011].

Running example 2.1 We now introduce the *dining philosophers* domain, which will be used as a running example throughout this thesis. A number of philosophers sit around a circular table, such that each philosopher has someone on her left and also on her right. Each philosopher has a single fork. The forks are places between philosophers so that adjacent philosophers must share a fork. A philosopher can only think if she is not hungry and can only eat if she is holding both forks from her left and right.

Philosophers have an *hungry rate*, which determines how often the get hungry and a *eating rate* which determines how fast they eat. We are interested in evaluating seating plans (arrangements) of philosophers to determine the total amount of philosophical

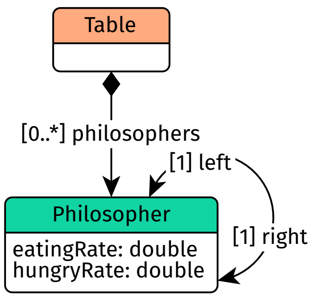


Figure 2.1 Class diagram for the dining philosophers metamodel.

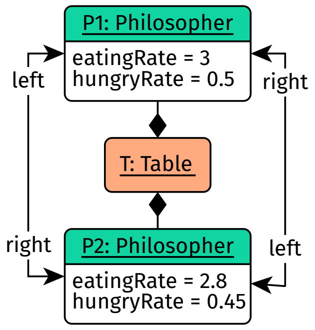


Figure 2.2 Instance model with two philosophers.

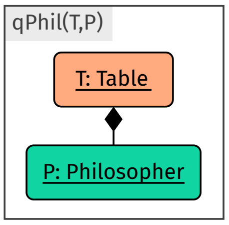


Figure 2.3 Example graph query matching a philosopher at a table.

knowledge produced, i.e. the time the philosophers spent thinking. By swapping philosophers around the table we can avoid seating two gluttonous philosophers or two slow eaters next to each other; thus we can minimize the contentions for forks that prevent philosophers from satiating their hunger before they can resume thinking.

Figure 2.1 shows a class diagram for the metamodel of the dining philosophers problem. It contains two classes, `Table` and `Philosopher`, three relations, `philosophers`, `left`, `right`, as well as two attributes, `eatingRate` and `hungryRate`.

Type constraints require that `philosophers` connect `Table` to `Philosopher` objects. It is also a containment reference, which makes `Table` the root of the containment hierarchy. Moreover, the multiplicity bound `[0..*]` allows any number of philosophers around the table, even zero.

The relations `left` and `right` connect `Philosopher` instances. The relation `right` is the opposite of `left`; therefore the philosopher on the left of someone sees that person on her right and vice verse. Due to the multiplicity bounds `[1]` each philosopher must have a single left and right neighbor.

Figure 2.2 shows an instance model with a single `Table T` and two `Philosopher` instances `P1` and `P2`. Because there are only two philosophers around the table both philosophers see the other on their left, as well as on their right.

2.1.2 Graph patterns

State-of-the-art modeling toolchains often rely of model queries to retrieve fragments of interest from model, specify model to model and model to text transformations, as well as to validate well-formedness constraints on models [Bergmann et al., 2011; Ujhelyi et al., 2015].

Formally, again following Varró et al. [2017], we define a graph query $\phi(x_1, x_2, \ldots, x_n)$ with n free variables as a first-order logic formula over the metamodel signature Σ. The formula may contain atomic propositions of the form $\mathsf{C}_i(x_j)$, $\mathsf{R}_i(x_j, x_\ell)$ and $\mathsf{A}_i(x_j, x_\ell)$, where $\mathsf{C}_i, \mathsf{R}_i, \mathsf{A}_i \in \Sigma$ are classes, references and attributes, respectively. Furthermore, logical connectives $\neg, \wedge, \vee, \Rightarrow$ and quantifiers $\exists, \forall$ may also appear in formulas.

Model query engine often allow transitive closure computation ϕ^+, which makes the query language strictly more expressive than first order logic. If ϕ is a formula with two free variables its transitive closue $\phi^+(x_i, x_j)$ holds if and only if there are some objects $x_i = y_1, y_2, \ldots, y_k = x_j$ $(k \geq 1)$ such that we have $\phi(y_1, y_2) \wedge \phi(y_2, y_3) \wedge \cdots \wedge \phi(y_{k-1}, y_k)$ [Bergmann et al., 2012].

For detailed semantics of first-order graph patterns extended with transitive closure we refer to the works of Semeráth and Varró [2017] and Varró et al. [2017].

The tuple $\langle q_1, q_2, \ldots, q_n \rangle \in (\mathcal{O} \cup \mathit{Prim})^n$ is a *match* of $\phi(x_1, x_2, \ldots, x_n)$ in the instance model M if ϕ holds with the *match arguments* $\langle q_1, q_2, \ldots, q_n \rangle$, i.e. $M \models \phi(q_1, q_2, \ldots, q_n)$. We will occasionally write the name of the graph pattern before the tuple to emphasize that it is a tuple of match arguments, e.g. $\phi\langle q_1, q_2, \ldots, q_n \rangle$.

The *match set* of the pattern ϕ in the instance model M contains all of its matches. It is a set, i.e. it contains each tuple at most once, regardless how many valid bindings are possible for quantified variables inside the pattern.

An *incremental* (or *live*) query engine, such as VIATRA Query [Ujhelyi et al., 2015] maintains the match sets of graph queries by subscribing to *change notifications* from the instance model. The changes are propagated to the match sets, and clients may get notifications about matches appearing and disappearing as the instance model is being modified. The Rete algorithm [Forgy, 1982] is often employed for change propagation.

Running example 2.2 Consider the instance model from Figure 2.2 along with the graph pattern

$$\mathtt{qPhil}(x_1, x_2) = \mathtt{Table}(x_1) \land \mathtt{Philosopher}(x_2) \land \mathtt{philosophers}(x_1, x_2)$$

depicted in Figure 2.3, which matches philosophers sitting at a table. Its match set contains two tuples, $\mathtt{qPhil}\langle \mathtt{T}, \mathtt{P1} \rangle$ and $\mathtt{qPhil}\langle \mathtt{T}, \mathtt{P2} \rangle$, which correspond to the two philosophers in the model.

2.2 Formal models for stochastic analysis

In this section we recall some of the basic formalisms involved in our work. Petri nets are introduced firsty, which serve as the basis in our framework to express stochastic models. Then we move to the background in stochastic modeling and analysis methods. Portions of this section were adapted from previous work by Klenik and Marussy [2015, Chapter 2].

Throughout this section and the rest of this work $\mathbb{N}$, $\mathbb{N}^+$, $\mathbb{R}$, $\mathbb{R}^+$ will refer to the sets of natural numbers including zero $\mathbb{N} = \{0, 1, 2, \ldots\}$, the set of positive natural numbers $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$, the set of real numbers and the set of positive real numbers, respectively.

2.2.1 Petri nets

Petri nets are a widely used graphical and mathematical modeling tool for systems which are concurrent, asynchronous, distributed, parallel or nondeterministic [Murata, 1989].

Definition 2.3 A *Petri net with inhibitor arcs and priorities* is a 7-tuple

$$PN = \langle P, T, m_0, \pi, \rightarrow, \leftarrow, \multimap \rangle,$$

where the sets P and T are disjoint and

- P is a finite set of *places*;
- T is a finite set of *transitions*;
- $m_0 \colon P \to \mathbb{N}$ is the *initial marking function*;
- $\pi \colon T \to \mathbb{N}$ is the *transition priority function*;
- $\leftarrow, \rightarrow, \multimap \subseteq P \times \mathbb{N}^+ \times T$ are the relations of *output, input* and *inhibitor arcs*, respectively, which are free of parallel arcs, i.e. $\langle p, n_1, t \rangle, \langle p, n_2, t \rangle \in \leftarrow$ implies $n_1 = n_2$ and this property holds also for $\rightarrow$ and $\multimap$.

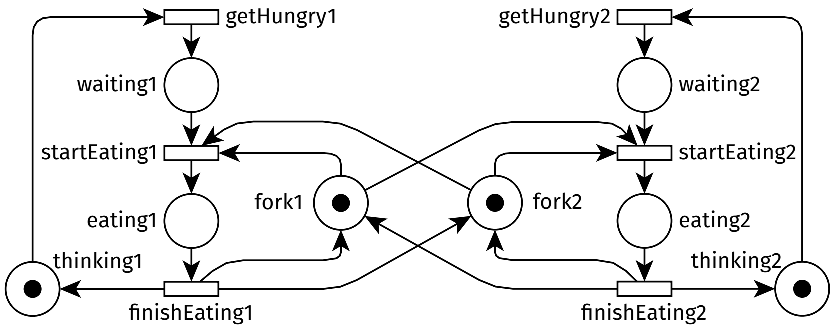


Figure 2.4 Example Petri net with two deadlock-free dining philosophers.

We will write $p \overset{n}{\leftarrow} t$, $p \overset{n}{\rightarrow} t$ and $p \overset{n}{\multimap} t$ for $\langle p, n, t\rangle \in \leftarrow$, $\langle p, n, t\rangle \in \rightarrow$ and $\langle p, n, t\rangle \in \multimap$, respectively. The arc *inscriptions* are omitted in the case $n = 1$.

Petri nets are graphically represented as edge weighted directed bipartite graphs. Places are drawn as circles, while transitions are drawn as bars or rectangles. The arc inscriptions are shown as edge weights.

A *marking* $m : P \rightarrow \mathbb{N}$ assigns a number of *tokens* to each place. The transition t is *enabled* in the marking m if $m(p) \geq n$ for all $p \overset{n}{\leftarrow} t$ and $m(p) < n$ for all $p \overset{n}{\multimap} t$.

An enabled transition is *fireable*, written as $m\boxed{t}\rangle$, if no enabled transition has higher priority, i.e. $\pi(t') \leq \pi(t)$ for all enabled transitions t'.

A fireable transition t can be *fired* to yield a marking m', written as $m\boxed{t}\rangle\, m'$, where $m'(p) = m(p) - n_{\text{in}} + n_{\text{out}}$ if $p \xrightarrow{n_{\text{in}}} t$ and $p \xleftarrow{n_{\text{out}}} t$. In only an input or output arc is present between t and p, the token count of p is only decreased or increased, respectively. Places not connected to t by an arc have their token count unchanged.

A marking m' is *reachable* from m, written as $m\boxed{*}\rangle\!\rangle\, m'$, if there is a sequence of markings and transitions such that $m = m_1 \boxed{t_1}\rangle\, m_2 \boxed{t_2}\rangle \cdots \boxed{t_{k-1}}\rangle\, m_k = m'$. The *reachable state space RS* of a Petri net is the set of markings reachable from its initial marking,

$$RS = \{m \colon P \rightarrow \mathbb{N} \mid m_0 \boxed{*}\rangle\!\rangle\, m\}.$$

The Petri net is *bounded* is there is an upper bound $K \in \mathbb{N}$ such that $m(p) \leq K$ for all $p \in P$ and $m \in RS$. The reachable state space RS is finite if and only if the net is bounded.

The state space of a bounded Petri net can be determined efficiently by the *saturation* algorithm [Ciardo et al., 2001, 2012]. Extensions have been proposed to the algorithm to handle transition priorities effectively [Miner, 2006; Marussy et al., 2017].

The arc inscriptions of Petri net may depend on the current marking by replacing the positive integers $\mathbb{N}^+$ with a set of algebraic expressions $Expr_P$ over the token counts of places. In the marking dependent setting $\leftarrow, \rightarrow, \multimap \subseteq P \times Expr_P \times T$ and the inscription expressions are evaluated according to the marking m when firing transitions $m\boxed{t}\rangle\, m'$. Such *Petri nets with marking-dependent arcs* can simplify formal modeling [Ciardo and Trivedi, 1993]; however, they may preclude the use of some analysis techniques.

Running example 2.3 Figure 2.4 shows an example Petri net modeling the dining philosophers problem with two philosophers. The philosophers $i = 1, 2$ are modeled by the places `thinking`*i*, `waiting`*i* and `eating`*i*. In the initial marking both philosophers are `thinking`. Upon firing `getHungry`*i* a philosopher may get hungry and gets to be `waiting` for forks to eat with. If both `fork1` and `fork2` is available, she may then `startEating`*i* by picking up both forks. The forks are picked up as a single atomic action, which avoids a possible case for deadlock when both philosophers pick up single forks but are unable to get the other fork to start eating. Therefore this model is the deadlock-free version of the dining philosophers problem. Upon finishing eating, the philosophers put the forks down by firing `finishEating`*i* and returning to the `thinking` state.

2.2.2 Continuous-time Markov chains

Continuous-time Markov chains (CTMCs) are mathematical tools for describing the behavior of systems in countinous time where the stochastic behavior of the system only depends on its current state [see e.g. Reibman et al., 1989]. This assumption is reasonable in a wide class of modeling tasks; hence CTMCs are commonly used in the reliability and performability prediction of critical systems.

Definition 2.4 A *continuous-time Markov chain* (CTMC) $X = \{X(t) \in S \mid t \geq 0\}$ over the finite state space $S = \{0, 1, \dots, n-1\}$ is a continuous-time random process with the *Markovian* or memoryless property:

$$\begin{aligned}\mathbb{P}(X(t_k) = x_k \mid X(t_{k-1}) = x_{k-1}, X(t_{k-2}) = x_{k-2}, \dots, X(t_0) = x_0)\\ = \mathbb{P}(X(t_k) = x_k \mid X(t_{k-1}) = x_{k-1}),\end{aligned}$$

where $t_0 \leq t_1 \leq \cdots \leq t_k$ and $X(t_k)$ is a random variable denoting the state of the CTMC at time t_k. A CTMC is said to be *time-homogenous* if it also satisfies

$$\mathbb{P}(X(t_k) = x_k \mid X(t_{k-1}) = x_{k-1}) = \mathbb{P}(X(t_k - t_{k-1}) = x_k \mid X(0) = x_{k-1}),$$

i.e. it is invariant to time shifting. In this work we will restrict our attention to time-homogenous CTMCs over finite state spaces.

The *state probabilities* at time t form a finite-dimensional vector $\boldsymbol{\pi}(t) \in \mathbb{R}^n$, where $\pi(t)[x] = \mathbb{P}(X(t) = x)$. Following the convention from CTMC literature all vectors considered will be row vectors, i.e. n element vectors are equivalent to matrices with a single row and n columns. Moreover, the ith element of the vector $\mathbf{v}$ will be written as $v[i]$, where indexing is zero-based ($i = 0, 1, \dots, n-1$).

The vectors $\boldsymbol{\pi}(t)$ satisfy the differential equation

$$\frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t)\, Q \tag{2.1}$$

for some square matrix Q. The matrix Q is called the *infinitesimal generator matrix* of the CTMC and satisfies $Q\mathbf{1}^T = \mathbf{0}^T$, where $\mathbf{1}$ and $\mathbf{0}$ are n-element vectors of ones and zeroes.

The diagonal elements $q[x,x] < 0$ of Q describe the holding times of the CTMC. If $X(t) = x$, the *holding time* $h_x = \inf\{h > 0 \mid X(t) = x, X(t+h) \neq x\}$ spent in state x is exponentially distributed with rate $\lambda_x = -q[x,x]$. If $q[x,x] = 0$ then no transitions are possible from the state x and it is said to be *absorbing*.

The off-diagonal elements $q[x,y] \geq 0$ of Q describe state transitions of the CTMC. The CTMC while being in state $X(t) = x$ will jump to state y at the next state transition with probability $-q[x,y]/q[x,x]$. Equivalently, there is an expontentially distributed countdown in the state x for each y that satisfies $q[x,y] > 0$ with *transition rate* $\lambda_{xy} = q[x,y]$. The first countdown to finish will trigger a state change to the corresponding state y. Therefore the CTMC is a transition system with exponentially distributed timed transitions.

$$\begin{tikzcd}
0 \arrow[r, bend left, "\lambda_1"] & 1 \arrow[l, "\mu_1"] \arrow[r, bend left, "\lambda_2"] & 2 \arrow[l, "\mu_2"] \arrow[ll, bend left, "\mu_3"]
\end{tikzcd}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} -\lambda_1 & \lambda_1 & 0 \\ \mu_1 & -\lambda_2 - \mu_1 & \lambda_2 \\ \mu_3 & \mu_2 & -\mu_2 - \mu_3 \end{pmatrix} \end{array}$$

Figure 2.5 Example CTMC with 3 states and its generator matrix.

Example 2.4 Figure 2.5 shows a CTMC with 3 states. The transitions from the state 0 to 1 and from 1 to 2 are associated with exponentially distributed countdowns with rates λ_1 and λ_2 respectively, while transitions in the reverse direction have rates μ_1 and μ_2. The transition form state 2 to 0 is also possible with rate μ_3.

The rows (corresponding to source states) and columns (destination states) of the infinitesimal generator matrix Q are labeled with the state numbers. The diagonal element $q[1, 1]$ is $-\lambda_2 - \mu_1$, hence the holding time in state 1 is exponentially distributed with rate $\lambda_2 + \mu_1$. The transition from state 1 to 0 is taken with probability $-q[1, 0]/q[1, 1] = \mu_1/(\lambda_2 + \mu_1)$, while the transition to 2 is taken with probability $\lambda_2/(\lambda_2 + \mu_1)$.

Steady-state probabilities

A state y is *reachable* from the state x, written as $x \boxed{*}\!\ggg y$, if there exists a sequence of states $x = z_1, z_2, z_3, \ldots, z_{k-1}, z_k = y$, such that $q[z_i, z_{i+1}] > 0$ for all $i = 1, 2, \ldots, k - 1$. If $x \boxed{*}\!\ggg y$ for all pairs of states $x, y \in S$ the Markov chain is *irreducible*.

The *steady-state probability distribution* $\boldsymbol{\pi} = \lim_{t\to\infty} \boldsymbol{\pi}(t)$ exists and is independent from the *initial distribution* $\boldsymbol{\pi}(0) = \boldsymbol{\pi}_0$ if and only if the CTMC is irreducible. The steady-state distribution satisfies the system of linear equations

$$\frac{\mathrm{d}\boldsymbol{\pi}}{\mathrm{d}t} = \boldsymbol{\pi} Q = \mathbf{0}, \quad \boldsymbol{\pi}\mathbf{1}^{\mathrm{T}} = 1. \tag{2.2}$$

The matrix Q is sparse and is often amenable to decomposed storage [Buchholz, 1999a]. However, solving the system of linear equations eq. (2.2) requires iterative linear equation solver algorithms, which can have varying convergence and running time characteristics [Buchholz, 1999b; Marussy et al., 2016b; Buchholz et al., 2017].

Selection of numerical solver backends and their parameters in the context of design-space exploration toolchains are discussed in Section 5.1.2 on page 55.

Parametric models

The infinitesimal generator matrix of *parametric CTMC* depends on a vector of *parameters* $\theta \in A \subseteq \mathbb{R}^k$, where A is the *feasible region* of parameter values. The parameters represent unknown or uncertain attributes of the system under study, while the feasible region describes realizable or plausible parameter values. *Parameter optimization* refers to selection of feasible parameter values $\theta \in A$ such that some *goal function* is maximized.

Analysis methods for parametric Markov chains include sensitivity analysis [Blake et al., 1988], parametric steady-state solution [Hahn et al., 2011; Vörös et al., 2017b] and parameter synthesis [Quatmann et al., 2016]. Some analysis methods only allow specific kinds of parameter-dependence in the infinitesimal matrix elements $\theta \mapsto q(\theta)[x, y]$, such as C^1 differentiable expressions [Blake et al., 1988] or rational functions [Hahn et al., 2011].

Markov reward models

Continuous-time Markov chains may be employed in the estimation of performance measures of models by defining *rewards* that associate *reward rates* with the states of a CTMC. The reward rate random variable $R(t)$ can describe performance measures defined at a single point of time, such as resource utilization or probability of failure, while the *accumulated reward* random variable $Y(t)$ may correspond to performance measures associated with intervals of time, such as total downtime.

Definition 2.5 A *continuous-time Markov reward process* over a finite state space $S = \{0, 1, \dots, n-1\}$ is a pair $\langle X, \mathbf{r} \rangle$, where X is a CTMC over S and $\mathbf{r} \in \mathbb{R}^n$ is a *reward rate vector*. The reward rate stochastic process $R = \{R(t) = r[X(t)] \mid t \geq 0\}$ describes the momentary *reward rate* associated with the active state of the CTMC.

The *accumulated reward* until time t is defined as the time integral of R,

$$Y = \left\{ Y(t) = \int_0^t R(\tau)\,\mathrm{d}\tau \;\middle|\; t \geq 0 \right\}.$$

Example 2.5 Let c_0, c_1 and c_2 denote operating costs per unit time associated with the states of the CTMC X in Figure 2.5. Consider the Markov reward process $\langle X, \mathbf{r} \rangle$ with the reward rate vector

$$\mathbf{r} = \begin{pmatrix} c_0 & c_1 & c_2 \end{pmatrix}.$$

The random variable $R(t)$ describes the momentary operating cost, while $Y(t)$ is the total operating expenditure until time t. The steady-state expectation $\lim_{t\to\infty} \mathbb{E}\, R(t)$ is the average maintenance cost per unit time of the long-running system.

In parameter-dependent reward models not only does the infinitesimal generator matrix $Q \colon A \longrightarrow \mathbb{R}^{n\times n}$ depend on the parameter vector $\boldsymbol{\theta} \in A$ but also can the reward rate vector $\mathbf{r} \colon A \longrightarrow \mathbb{R}^n$ be parameter-dependent.

2.2.3 Stochastic analysis tasks

Various analysis tasks concerning CTMCs and Markov reward models are performed to calculate stochastic metrics or determine whether the system satisfies a reliability or performability requirement. We will refer to such problems as *queries* concerning a stochastic model. Below we attempt to give a short summary of the most basic analyses and computational methods.

Steady-state analysis refers to the calculation of the steady-state expectation $\mathbb{E}\, R(\infty) = \lim_{t\to\infty} \mathbb{E}\, R(t)$ to characterize the values of reliability or performability metrics during long-term system operation. Because the steady state expectation is calculated according to the formula $\mathbb{E}\, R(\infty) = \boldsymbol{\pi}\, \mathbf{r}^T$, where $\boldsymbol{\pi}$ is the steady-state probability vector form eq. (2.2), this form of analysis is tantamount to the solution of eq. (2.2).

Transient and accumulated analysis is concerned with the transient behavior of the modeled system when it is started from an initial probability distribution $\boldsymbol{\pi}$. An initial problem with eq. (2.1) on page 8 is solved subject to the initial condition $\boldsymbol{\pi}(0) = \boldsymbol{\pi}_0$. Then the expected transient reward value $\mathbb{E}\, R(t) = \boldsymbol{\pi}(t)\, \mathbf{r}^T$ can be calculated.

Variations of the *uniformization* algorithm [see e.g. Morsel and Sanders, 1997; Dijk et al., 2017] can solve eq. (2.1) efficiently. Moreover, $\mathbf{L}(t) = \int_0^t \boldsymbol{\pi}(\tau)\,\mathrm{d}\tau$ can also be obtained by uniformization in order to calculate $\mathbb{E}\, Y(t) = \mathbf{L}(t)\, \mathbf{r}^T$ for the analysis of accumulated metrics.

Mean-time-to-state-partition analysis determines the expected time taken to reach a set of states $D \subsetneq S$ from an initial distribution $\boldsymbol{\pi}_0$. The calculation of the *mean time to first failure*, which is the mean time to reach the state partition D of failed states, has many applications in reliability engineering. Other tasks, such as the determination of the mean time between failures or the time taken to successfully complete a request can also be formalized as mean-time-to-state-partition problems.

These problems can be solved by the analysis of phase-type distributions [Neuts, 1975] derived from the CTMC and the state partitions D of interest by linear equations solvers, analogously to the calculation of steady-state expectations.

Sensitivity analysis concerns the rates of change in stochastic metrics due to changes in parameter values of a parametric CTMC or reward model. The model reacts to changes of parameters with high absolute sensitivity more prominently; therefore, they can be promising directions of system optimization. The partial derivatives of the expectation describe above can be computed with respect to the elements of the parameter vector [Blake et al., 1988; Ramesh and Trivedi, 1993].

Stochastic model checking consists of decision procedures to determine whether the system under consideration satisfies requirements formalized in stochastic logics. Often stochastic model checking involves the analysis tasks outlined above as subroutines. Logics suitable for continuous-time models include continuous stochastic logic (CSL) [Aziz et al., 1996] and continuous stochastic reward logic (CSRL) [Kwiatkowska et al., 2006]. Further approaches to specifying stochastic properties and queries are surveyed in Section 3.1.2 on page 18.

2.2.4 Generalized stochastic Petri nets

Although continuous-time Markov chains and reward processes base on CTMCs allow the study of dependability or reliability, the explicit specification of stochastic processes and rewards is often cumbersome. Generalized stochastic Petri Nets extend Petri nets by assigning exponentially distributed random delays to some transitions and instantaneous random firing to others [Marsan et al., 1984]. After the delay associated with an enabled transition is elapsed the transition fires *atomically* and transitions delays are reset.

Definition 2.6 A *generalized stochastic Petri net* (GSPN) is an 8-tuple

$$GSPN = \langle P, T, m_0, \pi, \lambda, \rightarrow, \leftarrow, \multimap \rangle,$$

where

- $\langle P, T, m_0, \pi, \rightarrow, \leftarrow, \multimap \rangle$ is a Petri net with priorities and inhibitor arcs,
- $\lambda \colon T \rightarrow \mathbb{R}^+$ is a *transition rate and weight function*.

Transitions $t \in T$ satisfying $\pi(t) = 0$ are called *timed transitions* and $\lambda(t)$ is the *rate* of such transitions. In contrast, transitions with higher priority are called *immediate* and $\lambda(t)$ is their *probability weight*. Timed transitions are usually depicted as rectangles, while immediate transitions are black bars.

Markings in which an immediate transitions is fireable are *vanishing*, while markings where only timed transitions are fireable are *tangible*. Reachable tangible markings form the *tangible state space*

$$TRS = \{m \in RS \mid \pi(t) = 0 \text{ for all } m\boxed{t}\rangle\}.$$

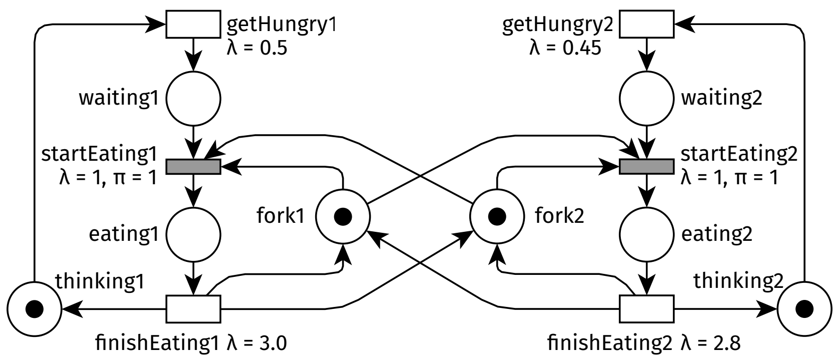


Figure 2.6 Example GSPN for the dining philosophers problem.

Timed transitions t have an associated exponentially distributed countdown with rate parameter $\lambda(t)$. The fireable timed transitions is fired when the countdown expires, which resets the countdown. In contrast, immediate transitions are fires as soon as they become fireable. If multiple immediate transitions are fireable a single transition is picked randomly to be fired. The probability of an immediate transition t being picked is proportional to its probability weight $\lambda(t)$. Immediate transitions are fired until a tangible marking is reached.

Running example 2.6 Figure 2.6 shows a GSPN version of the dining philosophers model form Running example 2.3 on page 7. The timed transitions `getHungry1`, `getHungry2`, `finishEating1` and `finishEating2` have rates 0.5, 0.45, 3.0 and 2.8, respectively.

The transitions `startEating1` and `startEating2` are immediate with equal priority and weight. Hence philosophers start eating as soon as they become hungry and there are forks available.

Stochastic Petri nets

A *stochastic Petri net* (SPN) is a GSPN with no immediate transitions, i.e. $\pi(t) \equiv 0$ for all $t \in T$. For such nets all reachable states are tangible, $TRS = RS$.

Bounded SPNs can be transformed into CTMCs in a straightforward way. The state space of the CTMC is the set $S = \{0, 1, \dots, |TRS| - 1\}$ and let $\iota \colon TRS \to S$ be a bijection. The off-diagonal elements of the infinitesimal generator matrix $Q \in \mathbb{R}^{|TRS| \times |TRS|}$ can be computed by summing the rates of transitions between states, while diagonal elements are set such that $Q\mathbf{1}^T = \mathbf{0}^T$ is satisfied. Formally, we have

$$q[\iota(m), \iota(m')] = \sum_{m \boxed{t\rangle} m'} \lambda(t) \quad \text{if } m \neq m', \qquad\qquad q[x, x] = -\sum_{y \neq x} q[x, y].$$

This simple translation makes SPNs especially amenable to analysis. For GSPNs with immediate transitions the reachable vanishing markings must be *eliminated* before a CTMC can be formed [Marsan et al., 1984].

Marking- and parameter-dependent models

The arc inscriptions of GSPNs can be made marking-dependent similarly to Petri nets, such that the arcs $\leftarrow, \rightarrow, \multimap \subseteq P \times Expr_P \times T$ contain algebraic expressions of the token counts of the places P. Moreover, the transition rates and weights can be made marking dependent, such that we have $\lambda \colon T \to Expr_P$.

Dependence on a set of parameters *Par* may also be introduced. In this case $\lambda \colon T \to \mathit{Expr}_{P,\mathit{Par}}$, where $\mathit{Expr}_{P,\mathit{Par}}$ is the set of algebraic expressions depending on token counts of P and the values of the parameters in *Par*. Parameter-dependent GSPNs are translated into parametric CTMCs, where the values of *Par* are encoded as the parameter vector $\theta \in A \subseteq \mathbb{R}^{|\mathit{Par}|}$ and the feasible region A of parameter values is determined according to domain requirements.

Reward nets

Generalized stochastic Petri nets can specify Markov reward models are well as CTMCs. The *reward expression* $r \in \mathit{Expr}_P$ is an algebraic expression which may refer to token counts of places. The reward expression r is translated into a reward vector $\mathbf{r}$ upon analysis in order to calculate expectations and answer queries regarding the reward value.

In the case of parameter-dependent GSPNs the reward expression may depend on the values of the parameters, such that we have $r \in \mathit{Expr}_{P,\mathit{Par}}$.

Running example 2.7 Consider the dining philosophers GSPN from Running example 2.6 along with the reward expression

$$r = \#\texttt{thinking1} + \#\texttt{thinking2},$$

which sums the token counts of the places `thinking1` and `thinking2`.

The expected transient reward rate $\mathbb{E}\, R(t)$ is the expected number of philosophers thinking at time t after starting from the initial marking. The expected steady-state reward rate $\mathbb{E}\, R(\infty)$ is the mean number of thinking philosophers during long-term operation. The expected accumulated reward $\mathbb{E}\, Y(t)$ is the mean total time spent thinking by philosophers until time t after starting.

Chapter 3

Modular formalism for stochastic models

In our current work we aim to propose an approach for the construction of stochastic models from engineering models without human intervention in order to evaluate automatically derived architecture proposals in design-space exploration by stochastic analysis.

The proposed transformation process should be flexible in the sense that—instead of basing our approach on a single engineering modeling language such as UML [Rumbaugh et al., 2004], SysML [Friedenthal et al., 2016], AADL [Feiler and Gluch, 2012] or Palladio [Becker et al., 2008]—the creation of transformations for new architectural domain-specific languages (DSLs) in new problem domains should be supported and should not demand additional specialized knowledge from the users. Therefore the formal models should be based on a stochastic formalism that has sufficient descriptive power to support engineering practice. In addition, compatibility of the derived models with existing stochastic verification tools should be ensured so that recent developments in formal methods may be leveraged for high-performance analysis. Hence reusing an existing formalism is dictated by both ① ease of use and ② portability.

Analysis tools usually separate the input formal model and the *query* to be answered [see e.g. Vörös et al., 2017a, Section 4.2], which is a performance metric to be calculated or a logical requirement to be verified. Therefore, when stochastic models are automatically derived for design-space exploration, ③ the appropriate queries must also be generated. The queries, which may depend on the structure of the engineering model in the same way as the derived stochastic model, serve as the objective functions and constraints of the exploration strategy.

To achieve these three objectives, in this chapter we turn to stochastic modeling approaches with modules to propose a formalism for the *modules* (or *fragments*) of the stochastic model corresponding to the analyzed aspects of the engineering model. The transformation, which is discussed in Chapter 4, will instantiate the modules specified by the user to automatically derive the analysis model.

After briefly reviewing related work, we describe our proposed formalism based on modular Petri nets, an extension of the ISO/IEC 15909-1:2004 standard on High-level Petri nets with a formally defined module concept [Kindler and Petrucci, 2009].

Petri nets and their extension to stochastic modeling, generalized stochastic Petri nets (GSPNs) are a widely used formalism for the analysis of software and hardware systems [Murata, 1989]. Various tools support GSPNs, such as SPNP [Hirel et al., 2000], SMART [Ciardo et al., 2006], Möbius [Courtney et al., 2009], GreatSPN [Babar et al., 2010] and PetriDotNet [Vörös et al., 2017b]. Hence we believe most of the target audience of our transformation design-space exploration approach are familiar with them. In addition, to aid finding bugs in the analysis models and to contribute to the ① ease of use, static typing, which was first proposed for modular high-level Petri nets by Kindler [2007], is supported for both the stochastic model and queries.

Models are serialized in the ISO/IEC 15909-2:2011 PNML format for ② compatibility with a wide variety of external tools.

In order to ③ generate queries for the stochastic models, we follow Kindler and Weber [2001] and extend modular Petri nets with symbols corresponding to the stochastic properties of interest. The addition of new symbols lets us encode the queries simultaneously with the structure of the analysis model.

3.1 Related work: modular stochastic modeling

In this section we briefly review some existing approaches for modular construction of logical and stochastic formal models, as well as for the specification of properties and metrics of interest over such models. For an overview on performance evaluation techniques for particular component-based software engineering languages, which contrasts with our present work that aims to be generic in the engineering DSL, we direct the interested reader to the survey by Koziolek [2010].

We are especially interested in *modular* formalisms that allow assembling structured models from modules (or fragments). While arbitrary combination of modules leads to high expressivity, it also restricts the opportunities for *compositional* verification. On the other hand, a formalism is compositional if the properties of model can be verified recursively by verifying simpler properties of its constituent components. These models are often constructed using *composition operators* that restrict arbitrary modularity in order to enforce property preservation.

[TODO: Cite!— Vöri said he has a good reference about this distinction.]

We opt for modularity instead of compositionality to avoid restricting the model transformations that will automatically assemble the stochastic models according to an architectural DSL instance. However, this means solution techniques will have to consider the assembled model in its entierty and cannot depend on preservation of the properties of the components.

3.1.1 Modeling formalisms

Continuous-time Markov chains (CTMCs) are common tools for the reliability and performability prediction of critical systems [see e.g. Reibman et al., 1989]. However, instead of modeling with CTMCs directly, usually higher-level formalisms are used to obtain more compact models. The semantics of these models are defined in terms of CTMCs or related stochastic processes, such as Markov regenerative processes [Logothetis et al., 1995; Telek and Pfening, 1996]. Usually the higher-level formalism belongs to one of these three classes:

Queuing networks (QNs) describe the routing of *customers* or *work items* between *queues*. The times spent in queues are described by random variables. The design and analysis of performance models based on hierarchical QNs were surveyed by Smith and Lladó [2011].

Stochastic Petri nets (SPNs) are Petri nets where transitions are equiped with expontentially distributed *firing delays*. Generalized stochastic Petri nets (GSPNs), may contain transitions with either exponentially distributed delays and *immediate* firing [Marsan et al., 1984]. Moreover, deterministic [Logothetis et al., 1995] and phase-type distributed [Longo and Scarpa, 2013] delays may also be incorporated; however, this makes verification significantly more complicated. Another generalization is the stochastic activity network formalism, where arbitrary input and output gates are allowed [Sanders and Meyer, 2001].

Stochastic process algebras incorporate random timings into the denotational semantics of process calculi [Hermanns et al., 2002] while allowing compositional verification. However, composition is syntactically restricted to set of allowed process operators, such as parallel and sequential composition of two subprocesses. An example formalism of this class is the Performance Enhanced Process Algebra (PEPA) defined by Hillston [1995].

Although all CTMCs can be expressed with any of these formalism classes, a significant advantage of higher-level models is the ability to expresses complicated behaviors of systems with small models. In this regard, GSPNs can express QNs without increasing model size [Vernon et al., 1986]. Comparison of Petri nets and process algebras is more difficult due to the vastly differing modeling styles [Donatelli et al., 1995]. The definable composition operators for Petri nets only conserve a limited set of properties; for a review we refer to Chapter 2 of the book by Huang et al. [2012].

Modularity in various Petri net formalisms was surveyed by Marechal and Buchs [2012]. Following their categorization we briefly review some formalisms below:

Syntactic (naïve) approaches to modularity construct analysis models from modules by structural sharing or merging of model elements. Marechal and Buchs [2012] refer to this approach as naïve because often the full analysis model must be assembled to determine its behavior. Thus compositional verification approaches based on the properties of modules are of limited use. However, this approach affords the greatest freedom to users when they define and assemble modules.

Stochastic automata networks[△] formalize the hierarchical structuring of CTMCs. Actions of the same *label* can be shared among automata for modeling communicating parallel systems. Labels are also a popular way of merging places (state variables) and transitions (actions) in Petri nets; among the 18 variants surveyed by Marechal and Buchs [2012] 11 used syntactic sharing of elements with equal labels. Bernardi and Donatelli [2003] proposed a methodology to construct stochastic Petri nets by merging places and transitions according to their labels as part of the DepAuDE project for dependability analysis. López-Grao et al. [2004] developed a transformation from UML activity diagram to stochastic Petri nets with the use of labeled Petri nets. Kühne et al. [2009] proposed a label-based model transformation framework that was also illustrated with Petri nets.

Syntactic merging is implemented in various stochastic analysis tools. GreatSPN provides support for labeled GSPNs [Bernardi et al., 2000]. Möbius lets users compose sub-models by sharing either state variables or actions [Courtney et al., 2009]; however, the two approaches cannot be mixed. The PRISM model checker allows both global state variables and synchronization on common actions [Kwiatkowska et al., 2011].

The downside of merging model elements is that *contradictions* may arise if modules prescribe different properties to a model element. For example, the firing rate of a stochastic action may be different across modules, or a state variable may have definitions with differing initial values. Contradictions may be signaled as errors when the models are analysed or may be resolved according to a resolution rule. For example, PRISM multiplies the rates of transitions when synchronization is used. More elaborate resolution rules were proposed for labeled GSPNs by Bernardi [2003, Section 2.2.3].

The leader-follower style composition in modular Petri nets proposed by Kindler and Petrucci [2009] avoid contradiction resolution by requiring each model element to have a single owner module that defines its properties. *References* in other modules may import

[△] Perhaps confusingly, stochastic *activity* networks and stochastic *automata* networks are both often abbreviated as SANs. We will refrain from abbreviating either.

foreign model elements to add adjacent Petri net arcs. Petri net modules may also define strongly typed interfaces separately from their implementation.

Synchronous or asynchronous message passing can serve as a means to assemble larger models from modules. The Behavior-Interactions-Priority (BIP) formalism presents a rigorous approach to the composition of state-based model using an algebra of *connectors* that transmit synchronous messages [Basu et al., 2006]. A stochastic extension of BIP was proposed by Nouri et al. [2015] for the composition of discrete-time probabilistic automata. On the other hand, hierarchical queuing networks constructed from simpler modules by model transformation [see e.g. Moreno et al., 2008] can be viewed as models on synchronously communicating components. Recently, a composition framework was proposed by Graics [2017] that supports mixing synchronous and asynchronous communication; however, it does not facilitate stochastic modeling.

Object-oriented approaches incorporate dynamic instantiation of modules into the operation semantics of the formalism. In Petri nets the net-within-net paradigm is employed most frequently, which allows higher level module operate on dynamically instantiated modules as tokens. An example of this class is the *reference net* formalism implemented by the Renew integrated development environment [Cabac et al., 2016].

3.1.2 Query specifications

Logical properties of complex asynchronous systems may be captured by temporal logics, such as the computational tree logic (CTL) [Clarke and Emerson, 1981] and linear temporal logic (LTL) [Vardi, 1996]. The logic CTL* contains both CTL and LTL fragments [Emerson and Halpern, 1986]; however, without restriction to either of these fragments model checking of properties becomes considerably more difficult in practice.

Markov reward models are the principal tools for defining performance metrics for CTMCs. They associate a stochastic process $\{R(t) : t \geq 0\}$ with the CTMC. A wide variety of analyses are possible, such as the computation of the expected steady-state reward $\mathbb{E}\, R(\infty)$, the expected transient reward $\mathbb{E}\, R(t)$ or the expected accumulated reward $\mathbb{E}\, Y(t) = \mathbb{E} \int_0^t R(\tau)\, \mathrm{d}\tau$ [see e.g. Reibman et al., 1989]. The results are interpreted as the mean values of the measures of interest. Sensitivity analysis [see e.g. Blake et al., 1988] can determine the rate of change in a metric caused by changing model parameters. Reward analysis can be generalized to Markov decision processes to study the effects of external control in stochastic systems [Baier et al., 2017a].

Several logics have been developed to study temporal and stochastic properties. Probabilistic computation tree logic (PCTL) was proposed by Hansson and Jonsson [1994] for discrete-time systems, while continuous stochastic logic (CSL) [Aziz et al., 1996] and CSL with timed automata (CSL^{TA}) [Donatelli et al., 2009] can describe properties of continuous-time systems. Cntinuous stochastic reward logic (CSRL) to incorporates reward analysis into CSL [Kwiatkowska et al., 2006]. A generalized version of CSRL (GCSRL) was developed by Kuntz and Haverkort [2007] for GSPNs and other systems with both timed and untimed stochastic behavior.

Queries for component-based models

An extension for UML class diagrams was proposed by Bernardi and Donatelli [2003] to construct stochastic metrics and property specifications for software models. Special derived features are added to represent model parameters, metrics and dependability requirements (prefixed with "/", "$" and "/$", respectively). Bernardi et al. [2004] also incorporated TRIO language for temporal property specification.

Measure Specification Language (MSL) uses first-order logic to specify reward structures for component-based stochastic models [Aldini and Bernardo, 2007]. Aldini et al. [2011] combined MSL with CSRL to extend its expressive power to temporal properties.

A temporal extension to Object Constraint Language (OCL) [Object Management Group, 2014] was proposed by Ziemann and Gogolla [2003] to formulate temporal constraints on UML models. Zalila et al. [2013] integrated temporal OCL and bidirectional model transformations for the formal verification of domain-specific modeling languages.

The ProMoBox framework was developed by Meyers et al. [2014] for the generation of domain-specific property specification languages. The input of ProMoBox is an engineering DSL based on which it specializes generic built-in sublanguages for 1. structural design, 2. run time state representation, 3. property specification, 4. input sequences and 5. output traces. The property specification sublanguage contains templates with expressive power equivalent to LTL. To our best knowledge, there was no attempt to incorporate stochastic properties into the framework.

3.2 Generalized stochastic Petri net modules

In this section we propose the specification of modules for GSPNs simultaneously with their reward measures and queries. When doing so, contradictions may arise in assembling the stochastic model from modules concerning the initial markings of places, the timings to transition firings and the definitions of the queries. In addition, care must be taken to avoid *circularity* in the merged models and queries, i.e. the structure of the model must not depend on the answers to the queries, as the state space and the CTMC derived from the model is used in producing the answer. Hence circular dependence between the model and queries makes analysis impossible.

To address these challenges, we base our approach on modular Petri nets [Kindler and Weber, 2001], which define modules as a collection of *symbols* (also referred to as *nodes*) and the *arcs* between them. Petri net places and transitions are represented as symbols. A symbol may either be *concrete* symbol or a *reference* to another symbol. *Imports* of a module are references that are pointed to *exports* of ther modules when the module is instantiated.

A module may only specify additional information about a concrete symbol, such as the initial marking of a concrete place or the rate of a timed transition. Thus there is a master-slave relationship between concrete and reference symbols, which avoids contradictions in assembled models. The specification of measures and queries is restricted analogously.

We incorporate three new symbol *kinds* into modular Petri nets to construct modular GSPNs. In addition, an *expression language* is proposed to specify the values of both the stochastic attributes of the model elements, such as transition firing rates, and the performance measures and queries of interest. Circularity in models is avoided by an adapting strict typing to mark invalid dependencies as type errors. This approach was inspired by the work of Kindler [2007] on strictly typed colored Petri net modules. We call the resulting formalism with extended symbols, expressions and typing *reference generalized stochastic Petri nets* (RGSPN).

To simplify presentation the separation of module interfaces and implementations, which enable information hiding for the design of modules, will be not considered. Moreover, the assembly of modules into a complete stochastic model is deferred to Chapter 4. The remainder of this chapter will focus on the structure and semantics of single RGSPN modules.

3.2.1 Symbols and edges

The RGSPN formalism consists of symbols, and *edges* between the symbols. The latter generalize Petri net arcs by also permitting reference assignments and collection memberships among the edges of the Petri net graph.

[TODO: We should find a better term than ***kind***, as in the current implementation, this term is used in another (slightly related) sense for determining the appearance of symbols in textual and graphical concrete syntaxes.]

Each symbol has a *kind*, which determines what information is needed to define the symbol, and a *type*, which determines the context where the symbol may be used. The type system, which is elaborated in Section 3.2.2 on page 22, contains type for places, transitions, and variables. However, the mapping between symbol kinds and types is not one-to-one, since the type of references can be set to determine the types of symbols they may point at.

Symbol kinds

The RGSPN formalism has six symbol kinds:

Places correspond to Petri net places. The token game of the net changes the markings of the places starting from their defined initial marking. The marking is a non-negative whole number, i.e. colored variants of GSPNS are not currently supported. When RGSPNS are shown as graphs places are displayed as circles.

Transitions correspond to Petri net transitions. They are equipped with a *firing policy*, which is either *timed* or *immediate*. Timed transitions have a *rate* parameter, which is the rate of the exponentially distributed firing delay. Immediate transitions have a probability *weight* and a *priority* consistently with the net-level specification of immediate transitions in GSPNS [Teruel et al., 2003]. Graphically, timed transitions are rectangles, while immediate transitions are filled.

Variables are expressions that may refer to the markings of transitions, other variables and parameters of the net. The *type* of the expression determines the context where a reference to a variable may appear in the net. Variables are shown as triangles.

Parameters are associated with constant real values and express the dependence of the model on continuous parameters. Parameter nodes are preserved during the inlining of the net into a GSPN as symbolic placeholders. Hence external tools may construct a parametric CTMC and apply sensitivity analysis [Blake et al., 1988] parametric solution [Hahn et al., 2011; Vörös et al., 2017b] or parameter synthesis [Quatmann et al., 2016]. The graphical notation for a parameter symbol is a filled triangle.

References can stand for other symbols from foreign RSGPN fragments. A reference has a *reference type*, which is the type of the symbol at which it may be *assigned* to point. A reference may only point at a single symbol at a time; however, references may be chain, as long as some concrete symbol can be resolved at the end of the chain. Graphical representation of references is derived from the pointed symbols but uses dashed lines.

References allow assembling different Petri net modules by merely adding reference assignments. As it will be shown in Section 4.5.1 on page 45, setting a single reference can correspond to redirecting many arcs in the net. Hence references help exploiting the modularity already present in the graph structure of Petri nets.

Collections, similarly to references, point to other symbols. A collection may point to multiple symbols at one is their type is consistent with the *member type* of the collection. The graphical notation is derived from the member type by adding a drop shadow.

Collections enable modular query specification in RGSPNs. While Petri nets are graphs, which can be easily extended by adding new arcs, performance measures are queries and described by algebraic expressions of a much stricter tree structure. Although variable references can serve as "holes" in the expression trees, they do not allow arbitrary aggregation of queries. For example, consider a performance measure which is defined as the sum of other measure corresponding to the components of the system. An expression of the form $v_1 + v_2$ can only serve as the aggregate measure of exactly two components, which must have their elementary performance measures assigned to the references v_1 and v_2.

In Section 3.3 on page 27 we introduce *aggregation functions* into the syntax of query expressions. This lets the aggregate performance measure be written as $\mathtt{sum}(c)$ analogously to the big operator expression $\sum_{v \in c} v$, where c is the collection of the constituent elementary measures. Collections may contain duplicate elements so that expressions like $v + v + v$ can be written in big operator form.

Edges

Any relation between two RGSPN symbol will be called an *edge*. Three kinds of edges are introduced, which are *arcs*, *reference assingments* and *collection memberships*. **[TODO: Kind?]**

Petri net arcs between transitions and places may be *output*, *input* or *inhibitor* arcs. Either end may be a reference to an appropriate place or transition instead of a concrete symbol. Arcs are equipped with possibly marking-dependent *inscription*, which is the number of tokens moved by the transition. If the inscription is the constant 1, we will omit it. Parallel arcs between the same symbols and with the same arc kind are forbidden.

Reference assignments connect references to the symbol at which they point. Indirect references, i.e. $r_1 := r_2$, $r_2 := s$ are possible and arbitrary chains of references may be built. In particular, an RGSPN may even contain reference cycles ($r_1 := r_2$, $r_2 := r_1$), or multiple, contradictory assignments ($r := s_1$, $r := s_2$, $s_1 \neq s_2$). However, *inconsistent* RGSPNs cannot be transformed into GSPNs for analysis. Inconsistency handling is discussed in detail in Section 4.5.3 on page 48.

Collection memberships connect collections to their member symbols. Either end of the membership edge may be a reference to a collection or an appropriate member symbol, respectively. In contrast with arcs, parallel membership edges are possible in order to express positive integer-weighted aggregations.

Running example 3.1 Figure 3.1 shows an RGSPN model of the dining philosophers problem with two philosophers sitting around a table.

While the immediate transitions *startEating1* and *startEating2* have constant weights and priorities, the timed transitions all refer to different symbols in their rate expressions. Node the difference between the variables *hungryRate1*, *hungryRate2* and the parameters *eatingRate1*, *eatingRate2*. Although these variables and symbols are all set to real number constants, the parameters are preserved as continuously changeable quantities when the model is passed to an external tool.

The self-contained subnets *philosohper1* and *philosopher2* contain reference places *rightFork1* and *rightFork2*. The subnets are connected by reference assignments. The reference places specify no initial marking at all—not even a zero marking—, because they are slaves of the pointed master symbols *leftFork2* and *leftFork1*, respectively.

The performance measures *thinkingTime1* and *thinkingTime2* are added to the collection *thinkingTimes*. Thus the aggregate performance measure *totalThinkingTime* can be formed by the aggregation operator `sum`.

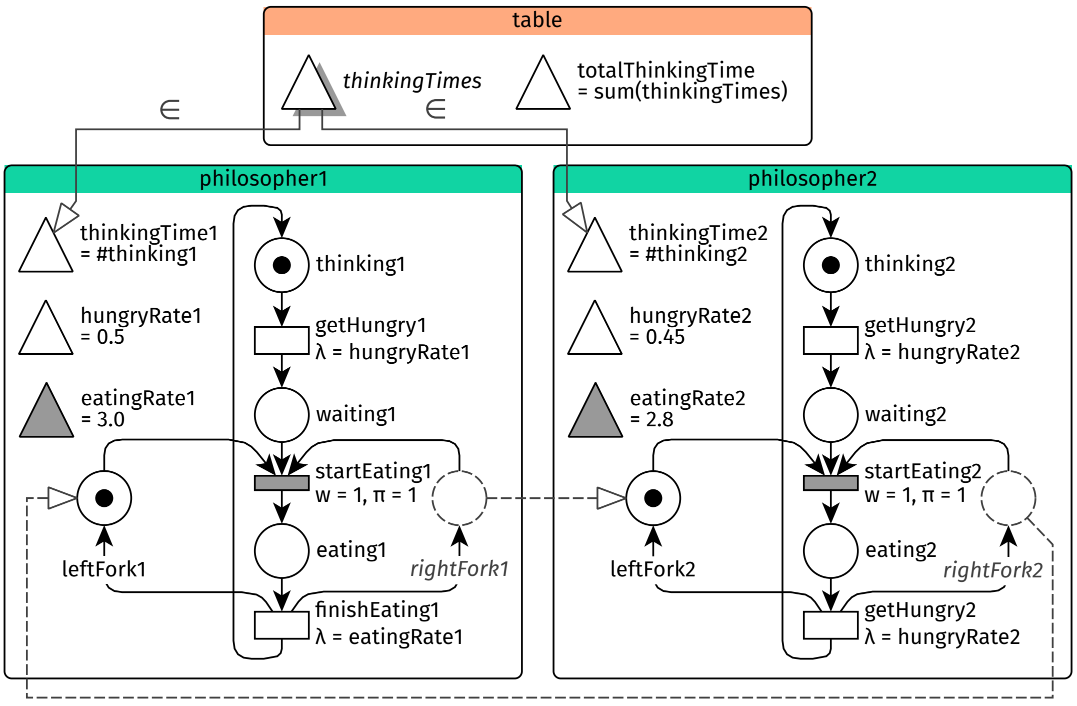


table
thinkingTimes
totalThinkingTime = sum(thinkingTimes)
∈
∈
philosopher1
thinkingTime1 = #thinking1
hungryRate1 = 0.5
eatingRate1 = 3.0
thinking1
getHungry1 λ = hungryRate1
waiting1
startEating1 w = 1, π = 1
eating1
rightFork1
leftFork1
finishEating1 λ = eatingRate1
philosopher2
thinkingTime2 = #thinking2
hungryRate2 = 0.45
eatingRate2 = 2.8
thinking2
getHungry2 λ = hungryRate2
waiting2
startEating2 w = 1, π = 1
eating2
rightFork2
leftFork2
getHungry2 λ = hungryRate2

Figure 3.1 Example RGSPN model with an aggregate performance measure.

3.2.2 Type system

Type systems are tractable syntactic methods for proving the absence of certain unwanted behaviors by classifying terms according to the values they compute [Pierce, 2002, Chapter 1]. On the other hand, static type systems for symbols in a modular Petri net were introduced by Kindler [2007]. In RGSPNs, types are used in both senses for classifying expressions, which are terms describing the quantitative aspects of the stochastic model, as well as symbols, which carry structural information.

The main unwanted behavior is the dependence of some expression on contextual information that is not available when the expression is evaluated. For example, the inscription of a Petri net arc should not depend on the state space of the Petri net, as the inscriptions themselves determine the reachable states.

The possible types are described by the following EBNF-like grammar:

$$\begin{aligned} \langle \mathit{Type} \rangle &::= \texttt{place} \mid \texttt{tran} \mid \langle \mathit{VarType} \rangle \mid \langle \mathit{Type} \rangle\texttt{[]}, \\ \langle \mathit{VarType} \rangle &::= \langle \mathit{Dependence} \rangle\, \langle \mathit{Pretype} \rangle, \\ \mathit{Dependence} &::= \texttt{const} \mid \texttt{param} \mid \texttt{marking} \mid \texttt{weight} \mid \texttt{prop} \mid \texttt{path}, \\ \langle \mathit{Pretype} \rangle &::= \texttt{int} \mid \texttt{double} \mid \texttt{boolean}. \end{aligned} \tag{3.1}$$

The types `place` and `tran` correspond to places and transitions in the RGSPN and the references thereof. Types of collections are formed by appending the *collection qualifier* suffix `[]` to the type of the members.

The types of variables deviate from routine. Inspired by conventions from the presentation of substructural type systems [see e.g. Walker, 2005] the types of variables are split into a qualifier and a *pretype*. The pretype part expresses the domain of values, `boolean` for truth values $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$, `int` for integers and `double` for real numbers.

The *dependence qualifier* specifies the evaluation context of an expression as follows:

- A `const` expression yields a value without further input.
- A `param` expression refers to the values of continuous model parameters, which are embodied by parameter symbols.
- A `marking` expression refers to the token counts of places; therefore it yields a different value in different Petri net markings.
- A `weight` expression is both parameter- and marking-dependent.
- A `prop` expression is a performance measure or query that can be determined by model checking and stochastic analysis, but may also depend on the initial marking.
- A `path` expression is a path property defined along a trace of model execution. It may be a complete LTL query or appear as a path formula in a CTL* `prop` query.

Because symbol kinds are separated from types, the type system can be adapted for many different scenarios while leaving the Petri net structure intact. Some of these possible extension based on existing literature are explored in Remarks 3.2 and 3.3.

Remark 3.2 Some analysis methods only allow specific kinds of parameter-dependence, such as C^1 differentiable expressions [Blake et al., 1988] or rational functions [Hahn et al., 2011]. However, no attempt is make to track different classes of parameter-dependent functions in `param` expressions, because the restrictions on parametric expressions are highly specific to these analysis methods. If such validations is required, either the RGSPN can be inspected when being exported for analysis, or the type system can be modified for the needs of the particular analysis method.

Subtyping

The type system proposed in eq. (3.1) can be overly rigid, because otherwise valid usages of expressions are forbidden, e.g. a `const` literal is incompatible with a `marking` context. We introduce subtyping to our type system for flexibility by enabling coercions between different dependence contexts and pretypes.

Subtyping is a binary relation $<: \subseteq \mathit{Type} \times \mathit{Type}$, where $\tau <: \tau'$ signifies that terms of type τ are convertible to type τ. It is reflexive, i.e. $\tau <: \tau$ for all $\tau \in \mathit{Type}$.

Subtyping for variable types is the direct product of the partial orders

$$\left(\begin{tikzcd} & \texttt{path} \arrow[d, dash] & \\ & \texttt{prop} \arrow[d, dash] & \\ & \texttt{weight} \arrow[dl, dash] \arrow[dr, dash] & \\ \texttt{param} \arrow[dr, dash] & & \texttt{marking} \arrow[dl, dash] \\ & \texttt{const} & \end{tikzcd}\right) \times \left(\begin{tikzcd} & \texttt{double} \arrow[d, dash] \\ \texttt{boolean} & \texttt{int} \end{tikzcd}\right) \tag{3.2}$$

of the sets *Dependence* and *Pretype*, respectively, where comparable elements are connected with upward paths in the style of e.g. Walker [2005]. For example, `const int` $<:$ `marking double`, because `const` $\leq$ `marking` and `int` $\leq$ `double` in the partial orders. The semantics of variable type coercions are discussed in Section 3.3.2 on page 29.

Collection types are covariant in their member types; therefore $\tau <: \tau'$ if and only if $\tau\texttt{[]} <: \tau'\texttt{[]}$. Type coercion of collections is performed elementwise.

Remark 3.3 It would be possible to include more elaborate abstract syntax and subtyping rules for types, for example to describe colored Petri nets, where scalar token counts in markings are replaced by multisets over the elements of the *color class* or *sort* corresponding to each place. In the colored setting, instead of a single `place` type, types of places carry a sort parameter. Kindler [2007] studied modular colored Petri nets with sort and operator symbols. A sort symbol reference is a color class that can be imported into the module from outside and is thus left abstract inside the module. Types of places thus may depend on the sort symbols.

Modular colored nets may also contain *operator* symbols, which transform members of a color class into another. In our framework, these could be modeled by symbols of type $\tau \longrightarrow \sigma$, i.e. operators that transform values of type τ into values of type σ, extending syntax of types $\langle Type \rangle ::= \ldots \mid \langle Type \rangle \longrightarrow \langle Type \rangle$. The arising challenges seem to require more elaborate type theoretical machinery, such as typed lambda calculus with subtyping [see e.g. Pierce, 2002, Chapters 15 and 16].

3.2.3 Formal definition

[TODO: Change *kinds*.]

In this section we first define RGSPN signatures as set of symbols of various kinds. Then the definition of an RGSPN on a given signature is elaborated, which extends the signature with the properties of the symbols and the edges of the net. This separation allows deferring the details of the *expressions* of a signature to Section 3.3 on page 27 even though expression will serves as the properties of symbols in the definition of RGSPNs.

Definition 3.1 An RGSPN *signatrue* is an 11-tuple

$$\Sigma = \langle P, T_T, T_i, V, Par, R, C, dep, pretype, target, member \rangle,$$

where the sets $P, T_t, T_i, V, Par, R, C$, are disjoint and

- P is a set of *places*;
- T_T and T_i are a sets of *timed* and *immediate transitions*, respectively;
- V is a set of *variables*;
- Par is a set of *parameters*;
- R is a set of *references*;
- C is a set of *collections*;
- $dep\colon V \longrightarrow Dependence$ is the *variable dependence* function;
- $pretype\colon V \longrightarrow Pretype$ is the *variable pretype* function;
- $target\colon R \longrightarrow Type$ is the *reference target type* function;
- $member\colon C \longrightarrow Type$ is the *collection member type* function.

We will abuse notation such that Σ also stands for the set $P \cup T_t \cup T_i \cup V \cup Par \cup R \cup C$ of all symbols. Furthermore, $Expr_\Sigma$ will denote the set of all algebraic expressions that may mention symbols of Σ.

Definition 3.2 An RGSPN is an 11-tuple $N = \langle \Sigma, m_0, \lambda, w, \pi, value, \longleftarrow, \longrightarrow, \multimap, :=, += \rangle$, where

- $\Sigma = \langle P, T_T, T_i, V, Par, R, C, \ldots \rangle$ is an RGSPN signature;
- $m_0\colon P \longrightarrow Expr_\Sigma$ is the *initial marking* function;
- $\lambda\colon T_T \longrightarrow Expr_\Sigma$ is the *timed transition rate* function;
- $w\colon T_i \longrightarrow Expr_\Sigma$ is the *immediate transition weight* function;
- $\pi\colon T_i \longrightarrow Expr_\Sigma$ is the *immediate transition priority* function;
- $value\colon V \cup Par \longrightarrow Expr_\Sigma \cup \mathbb{R}$ is a function, such that $value(v) \in Expr_\Sigma$ for all $v \in V$ and $value(par) \in \mathbb{R}$ for all $par \in Par$;
- $\longleftarrow, \longrightarrow, \multimap \subseteq \Sigma \times Expr_\Sigma \times \Sigma$ are the relations of *output*, *input* and *inhibitor arcs*, respectively, which are free of parallel arcs, i.e. $\langle p, e_1, t \rangle, \langle p, e_2, t \rangle \in \longleftarrow$ implies $e_1 = e_2$ and this property holds also for $\longrightarrow$ and $\multimap$;
- $:= \subseteq R \times \Sigma$ is the relation of *reference assignments*;
- $+= \in Multiset(\Sigma \times \Sigma)$ is the multiset relation of *collection memberships*.

Note the separation between timed T_T and immediate transitions T_i. In GSPNs timed and immediate transitions are usually discriminated by setting $\pi(t) = 0$ for all $t \in T_T$ [Marsan et al., 1984]. However, in our setting the priority $\pi(t)$ may contain an algebraic expression; therefore determining whether $\pi(t) = 0$ would require nontrivial computations. By explicitly partitioning the set of transitions $T = T_T \sqcup T_i$ this computation is avoided.

All quantitative aspects of the net are described by expressions Expr_Σ with the exception of the values of the parameters, which must be real numbers. As any computation is forbidden inside parameter values, so that parameter synthesis tool may set new values of the parameters without needing to respect any constraints between parameter values implicit in the value computations. Explicit constraints, such as interval bounds for parameters may be added as an extension of RGSPNs; however, they are currently not supported. If multiple values depending on a shared set of parameters are needed, variable symbols with value expressions may be used instead. **[TODO: Should this go somewhere else?]**

Edges of the net are between pairs of arbitrary symbols, e.g. arcs are not restricted to go from place symbols to transition symbols, because any symbol may be replaced by a reference of compatible type. However, reference assignments must assign the symbol to be pointed at to a reference, as no other symbol kind can act as an assignable.

Although parallel arcs are forbidden, parallel collection membership edges are permitted by making $\mathrel{+}=$ a multiset relation, i.e. a *bag* of tuples, such as $\wr \langle c, s \rangle, \langle c, s \rangle, \ldots \wr$.

We write $p \overset{e}{\leftarrow} t$, $p \overset{e}{\rightarrow} t$, $p \overset{e}{\multimap} t$, $r \mathrel{:=} s$ and $c \mathrel{+}= s$ for $\langle p, e, t \rangle \in \leftarrow$, $\langle p, e, t \rangle \in \rightarrow$, $\langle p, e, t \rangle \in \multimap$, $\langle r, s \rangle \in \mathrel{:=}$ and $\langle c, s \rangle \in \mathrel{+}=$, respectively.

Type checking

Types for the symbols of the net are synthesized by the function $\mathit{type} \colon \Sigma \to \mathit{Type}$ defined as

$$\mathit{type}(s) = \begin{cases} \texttt{place}, & \text{if } s \in P, \qquad \texttt{tran}, \qquad\qquad \text{if } s \in T, \\ \mathit{dep}(s)\,\mathit{pretype}(s), & \text{if } s \in V, \qquad \texttt{param double}, \ \text{if } s \in \mathit{Par}, \\ \mathit{target}(s), & \text{if } s \in R, \qquad \mathit{member}(s)\texttt{[]}, \quad \text{if } s \in C, \end{cases}$$

The types of places and transitions match their kinds, while variables have a variable type according to their dependence and pretype. The types of parameters are fixed to `param double`, as they are continuous and parameter dependent by definition. References always have the type of the symbol they may point at; therefore they may stand for the pointed symbol. Collections append a collection type qualifier to the type of their members.

The *typing relation* $\_ \vdash \_ : \_$ assigns types to expressions $e \in \mathit{Expr}_\Sigma$. We write $\Sigma \vdash e : \tau$ if e is of type τ in the context of the RGSPN signature Σ. As it will be seen in Section 4.5.1 on page 45 the typing relation respects subtyping, i.e.

$$\frac{\Sigma \vdash e : \tau \quad \tau <: \tau'}{\Sigma \vdash e : \tau'} \,. \tag{T-Sub}$$

In well-typed RGSPNs, where expressions and edges respect strong typing to ensure context-appropriate use of symbols and expressions within both the structural part of the net and its queries. Below we propose some typing requirements that make analysis tractable without greatly restricting the modeler.

Definition 3.3 An RGSPN is well-typed if it has the following properties:

- For all $p \in P$ the initial marking is an integer constant, $\Sigma \vdash m_0(p) : \texttt{const int}$.
- For all timed transitions $t \in T_T$ the transition rate is a possibly marking- and parameter-dependent real number, $\Sigma \vdash \lambda(t) : \texttt{weight double}$.

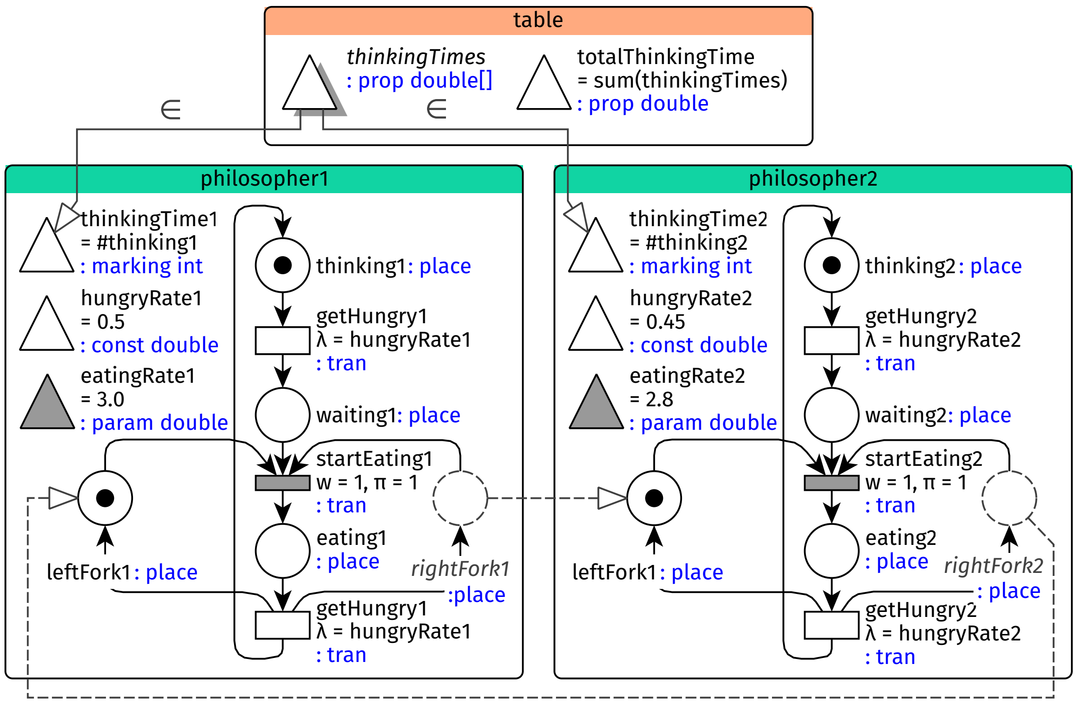


Figure 3.2 Example RGSPN with type annotations.

- For all immediate transitions $t \in T_i$ the probability weight is a possibly marking- and parameter-dependent real number, $\Sigma \vdash w(t)$: `weight double`. The transition priority is typed much more conservatively by requiring an integer constant, such that $\Sigma \vdash \pi(t)$: `const int` holds.
- For all variables $v \in V$ the value expression must match the type of the variable, $\Sigma \vdash \mathit{value}(v) : \mathit{type}(v)$.
- All arcs $p \overset{e}{\leftarrow} t$, $p \overset{e}{\rightarrow} t$ or $p \overset{e}{\multimap} t$ go between places and transitions such that $\mathit{type}(p)$ <: `place` and $\mathit{type}(t)$ <: `tran` holds. The inscription e may depend on the marking, $\Sigma \vdash e$: `marking int`,
- For all $r := s$, s is a compatible target of r, $\mathit{type}(s) <: \mathit{target}(r)$.
- For all $c \mathrel{+}= s$, s is a valid member of c, $\mathit{type}(s) <: \mathit{member}(s)$.

Remark 3.4 The requirements are based on the assumptions in GSPN and CTMC solution algorithms. While the inscriptions of arcs e are allowed to have dependence `marking`, because marking-dependent arcs may lead to simplifications of stochastic models [Ciardo and Trivedi, 1993]. However, as some external tools only support arcs with constant inscriptions, $\Sigma \vdash e$: `const int` may be enforced instead for compatibility.

Similarly, marking-dependent immediate transition weights can also pose a difficulty in solving the model [Teruel et al., 2003], which can be averted by requiring $\Sigma \vdash w(t)$: `param double` for all $t \in T_i$. In contrast, some state-space explorations methods, such as the decision diagram based algorithm proposed by Marussy et al. [2017], may permit marking-dependent priorities $\Sigma \vdash \pi(t)$: `marking int`.

From now on all discussed RGSPNs will be assumed to be well-typed.

Running example 3.5 Figure 3.2 shows the model from Running example 3.1 on page 21 extended with type annotations in blue. Places, transitions and parameters have their expected types `place`, `tran`, `param double`. Variables are annotated according to their *dep* and *pretype*, while collections bear the collection qualifier suffix `[]`.

There are several examples of subtyping in action: the symbols *thinkingTime1, thinkingTime2, eatingRate1, eatingRate2* are used as rates of timed transitions despite their types `const double` and `param double`. The collection *thinkingTimes* of `prop double` members contains the symbols *thinkingTime1* and *thinkingTime2* of type `marking int`.

3.3 Expressions

In this section we propose an abstract syntax for expressions that describe the quantitative aspects of RGSPN models, including arc inscriptions, initial markings and firing policies in Definition 3.1 on page 24, as well as the performance measures and queries of interest.

The expression language CTL* includes state and path operators in addition to references to net elements, basic arithmetic and logical operators. These additional operators enable defining queries concerning CTL, LTL or CTL* properties. Similarly to the flexibility of the type system, the syntax of expressions can be also extended if the definition of further properties, such as CSL formulas are desired. Validation and interpretation of the queries, such as checking whether a CTL* formula is in CTL when full CTL* is not supported, is the responsibility of the external model checking tool.

The valid expression on an RGSPN signature Σ form the set Expr_Σ described by the following EBNF-like grammar:

$$\begin{aligned}
\langle \mathit{Expr}_\Sigma \rangle &::= \langle \mathit{Literal} \rangle \mid \langle \Sigma \rangle \mid \texttt{\#}\langle \Sigma \rangle \mid \langle \mathit{Aggregate} \rangle \texttt{(}\langle \Sigma \rangle\texttt{)} \mid \langle \mathit{Unary} \rangle\ \langle \mathit{Expr}_\Sigma \rangle \\
&\quad \mid \langle \mathit{Expr}_\Sigma \rangle\ \langle \mathit{Binary} \rangle\ \langle \mathit{Expr}_\Sigma \rangle \mid \texttt{if (}\langle \mathit{Expr}_\Sigma \rangle\texttt{)}\ \langle \mathit{Expr}_\Sigma \rangle\ \texttt{else}\ \langle \mathit{Expr}_\Sigma \rangle, \\
\langle \mathit{Literal} \rangle &::= \langle \mathbb{N} \rangle \mid \langle \mathbb{R} \rangle \mid \langle \mathbb{B} \rangle, \\
\langle \mathit{Aggregate} \rangle &::= \texttt{sum} \mid \texttt{prod} \mid \texttt{all} \mid \texttt{any}, \\
\langle \mathit{Unary} \rangle &::= \texttt{+} \mid \texttt{-} \mid \texttt{!} \mid \texttt{A} \mid \texttt{E} \mid \texttt{X} \mid \texttt{F} \mid \texttt{G}, \\
\langle \mathit{Binary} \rangle &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{U}.
\end{aligned} \tag{3.3}$$

The expression language contains Boolean, integer and real literals, a standard set of unary and binary operators, a ternary conditional operator, as well as CTL* state operators `A`, `E` and path operators `X`, `F`, `G`, `U`. Variable symbols and references thereof from Σ may be mentioned as-is and are interpreted as the *value*s of the variables. Places can be also mentioned by prefixing them with `#` and correspond to `marking` dependent expressions referring to the number of tokens on the place. Collections must be paired with an *aggregation operator* to turn their multiset of member symbols into a single value.

Note that marking expressions and collection aggregations directly take a symbol from Σ instead of an expression Expr_Σ; therefore "`if (#`p_1 > `0) #`p_2 `else #`p_3" is a valid expression, but "`#(if (#`p_1 > `0)` p_2 `else` p_3`)`" is invalid. This restriction, while not constraining expressivity significantly, allow for more straightforward inlining and implification of expressions when the RGSPN is transformed into a GSPN.

3.3.1 Typing

A complete set of typing rules for Expr_Σ is presented in Table 3.1, which describes the relation $\_ \vdash \_ : \_$. The judgement $\Sigma \vdash e : \tau$ assigns a type τ to an expression $e \in \mathit{Expr}(\Sigma)$ in the context of an RGSPN signature Σ.

Table 3.1 Typing rules for expressions.

$\dfrac{\diamond \in \{\texttt{+},\texttt{-}\} \quad \rho \in \{\texttt{int},\texttt{double}\} \quad \Sigma \vdash e : \delta\,\rho}{\Sigma \vdash \diamond e : \delta\,\rho}$,	(T-UNARY±)
$\dfrac{\Sigma \vdash e : \delta\,\texttt{boolean}}{\Sigma \vdash \texttt{!}e : \delta\,\texttt{boolean}}$,	(T-UNARYNOT)
$\dfrac{\diamond \in \{\texttt{A},\texttt{U}\} \quad \Sigma \vdash e : \texttt{path boolean}}{\Sigma \vdash \diamond e : \texttt{prop boolean}}$,	(T-UNARYSTATE)
$\dfrac{\diamond \in \{\texttt{X},\texttt{F},\texttt{G}\} \quad \Sigma \vdash e : \texttt{path boolean}}{\Sigma \vdash \diamond e : \texttt{path boolean}}$,	(T-UNARYPATH)
$\dfrac{\diamond \in \{\texttt{+},\texttt{-},\texttt{*}\} \quad \rho \in \{\texttt{int},\texttt{double}\} \quad \Sigma \vdash e_1 : \delta\,\rho \quad \Sigma \vdash e_2 : \delta\,\rho}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\rho}$,	(T-BINNUMERIC)
$\dfrac{\Sigma \vdash e_1 : \delta\,\texttt{double} \quad \Sigma \vdash e_2 : \delta\,\texttt{double}}{\Sigma \vdash e_1 \,\texttt{/}\, e_2 : \delta\,\texttt{double}}$,	(T-BINDIV)
$\dfrac{\Sigma \vdash e_1 : \texttt{path boolean} \quad \Sigma \vdash e_2 : \texttt{path boolean}}{\Sigma \vdash e_1 \,\texttt{U}\, e_2 : \texttt{path boolean}}$,	(T-BINUNTIL)
$\dfrac{\diamond \in \{\texttt{==},\texttt{!=}\} \quad \Sigma \vdash e_1 : \delta\,\rho \quad \Sigma \vdash e_2 : \delta\,\rho}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\texttt{boolean}}$,	(T-BINEQ)
$\dfrac{\diamond \in \{\texttt{<},\texttt{<=},\texttt{>},\texttt{>=}\} \quad \Sigma \vdash e_1 : \delta\,\texttt{double} \quad \Sigma \vdash e_2 : \delta\,\texttt{double}}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\texttt{boolean}}$,	(T-BINCOMPARE)
$\dfrac{\diamond \in \{\texttt{\&\&},\texttt{\|\|}\} \quad \Sigma \vdash e_1 : \delta\,\texttt{boolean} \quad \Sigma \vdash e_2 : \delta\,\texttt{boolean}}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\texttt{boolean}}$,	(T-BINLOGICAL)
$\dfrac{\Sigma \vdash e_1 : \delta\,\texttt{boolean} \quad \Sigma \vdash e_2 : \delta\,\rho \quad \Sigma \vdash e_3 : \delta\,\rho}{\Sigma \vdash \texttt{if}\,(e_1)\,e_2\,\texttt{else}\,e_3 : \delta\,\rho}$,	(T-IF)
$\dfrac{\mathit{agg} \in \{\texttt{sum},\texttt{prod}\} \quad \rho \in \{\texttt{int},\texttt{double}\} \quad \mathit{type}(b) = \delta\,\rho\texttt{[]}}{\Sigma \vdash \mathit{agg}(b) : \delta\,\rho}$,	(T-AGGNUMERIC)
$\dfrac{\mathit{agg} \in \{\texttt{all},\texttt{any}\} \quad \mathit{type}(b) = \delta\,\texttt{boolean[]}}{\Sigma \vdash \mathit{agg}(b) : \delta\,\texttt{boolean}}$,	(T-AGGLOGICAL)
$v : \mathit{type}(v)$,	(T-VAR)
$\dfrac{\mathit{type}(p) <: \texttt{place}}{\#p : \texttt{marking int}}$,	(T-MARKING)
$\dfrac{\ell \in [\![\rho]\!]}{\Sigma \vdash \ell : \texttt{const}\,\rho}$,	(T-LITERAL)
$\dfrac{\Sigma \vdash e : \tau \quad \tau <: \tau'}{\Sigma \vdash e : \tau'}$,	(T-SUB)

where $[\![\texttt{int}]\!] = \mathbb{N}$, $[\![\texttt{double}]\!] = \mathbb{Z}$ and $[\![\texttt{boolean}]\!] = \mathbb{B}$.

The types of unary operators, binary operators, conditional and aggregate expressions are captured by the rules T-UNARY, T-BIN, T-IF and T-AGG. Instead of introducing types for operators and typing rules for operator application, typing rules for all operators are written out explicitly. While this approach increases the number of typing rules considerably, the lack of function types and polymorphic types allows the syntax of *Type* to remain simple. If more generality is desired, the type system may be extended to support user-defined operators and operator types as described in Remark 3.3 on page 23.

In spite of being handled only in the type derivation rules, several operators are polymorphic in the types of the arguments. However, T-BINARYDIV forces both arguments of the division operator be be real numbers, so that ambiguities concerning integer division are avoided. Most compound expressions are *dependency polymorphic*, that is, the types of their arguments may have any dependency qualifier δ, which will be inherited by the type of the whole expression. The exceptions are the CTL* operators, which operate on `path` formulas and produce `path` or `prop` state formulas.

Variable and marking references are handled by T-VAR and T-MARKING. Referring to markings of places always produces a `marking` dependent `int`. T-LITERAL assigns `const` types to literal constants. Lastly, T-SUB allows the use of subtyping in type derivations.

3.3.2 Semantics

In this section we sketch the semantics of $Expr_\Sigma$ both for structural expression of an RGSPN and for performance measures and queries. Most of the expression evaluation happens in external analysis tools when marking- and parameter-dependent expressions are interpreted to construct a CTMC from the Petri net and when queries are answered. Therefore, exporting RGSPNs for external tools must be performed with care to ensure that the tool interprets the provided input according to these semantics. This may require nontrivial transformation of the expressions to the input language of the tool and may even be impossible to fully achieve when the external tool is missing some analysis features. In the latter case, the user receives an error message during export.

Pretypes and dependence qualifiers

Values of pretypes `boolean`, `int` and `double` can be interpreted as members of the sets $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$ of truth values, $\mathbb{Z}$ of integers and $\mathbb{R}$ of real numbers, respectively.[△] Formally, pretypes have the interpretations specified in Table 3.1, i.e.

$$[\![\mathtt{boolean}]\!] = \mathbb{B}, \qquad [\![\mathtt{int}]\!] = \mathbb{Z}, \qquad [\![\mathtt{double}]\!] = \mathbb{R}.$$

Variable types $\delta\ \rho$ can be viewed as functions from some *context* determined by the dependence qualifier δ to the set $[\![\rho]\!]$. In the case $\delta = \mathtt{const}$, the context is empty, so $[\![\mathtt{const}\ \rho]\!]$ is isomorphic to $[\![\rho]\!]$. For other qualifiers, the context may be comprised of a vector $\theta \in \mathbb{R}^{|Par|}$ of parameter values and the current marking of the Petri net m. Queries with `prop` and `path` dependence may also require the entire CTMC that describes the logical and stochastic behavior of the RGSPN for evaluation. Finally, `path` properties are evaluated on an execution path $\Pi = m_1 \to m_2 \to \ldots$ of markings (or equivalently, CTMC states). The interpretations of variable types can be summarized as

$$\begin{aligned}
[\![\mathtt{const}\ \rho]\!] &: && p \in [\![\rho]\!], \\
[\![\mathtt{param}\ \rho]\!] &: \theta && \mapsto p \in [\![\rho]\!], \\
[\![\mathtt{marking}\ \rho]\!] &: \quad m && \mapsto p \in [\![\rho]\!], \\
[\![\mathtt{weight}\ \rho]\!] &: \theta, m && \mapsto p \in [\![\rho]\!], \\
[\![\mathtt{prop}\ \rho]\!] &: \theta, m, \textsc{ctmc} && \mapsto p \in [\![\rho]\!], \\
[\![\mathtt{path}\ \rho]\!] &: \theta, \quad \textsc{ctmc}, \Pi && \mapsto p \in [\![\rho]\!].
\end{aligned}$$

Type coercion from `int` to `double` act in the obvious way. Dependence coercion along the partial order from eq. (3.2) on page 23 introduces arguments to the interpretation functions that are ignored. For example, coercing `const` to `weight` results in a function that ignores

[△] In practice, representations on integers and floating-point numbers with a finite number of bits are used instead. However, this distinction only becomes important in the external analysis tools, where finite numerical precision necessitates careful design of algorithms to control approximation error [Baier et al., 2017b].

its θ and m arguments while returning a constant value. The only non-straightforward coercion is from `prop` to `path`. In order to be consistent with CTL* formulas this conversion is defined such that the first marking m_1 of the path $\Pi = m_1 \longrightarrow m_2 \longrightarrow \ldots$ serves as the current marking argument m of the `prop` computation when it is treated as a `path` property.

Operators and mentioned symbols

Now we will clarify the semantics $[\![\_]\!]$ of the expressions $Expr_\Sigma$ themselves. The intepretations of expressions conform with the types, i.e. if $\Sigma \vdash e : \tau$, then we have $[\![e]\!] \in [\![\tau]\!]$.

Operators from eq. (3.3) on page 27 act pointwise on the interpretation functions, e.g. to calculate $e_1 \diamond e_2$, e_1 and e_2 are separately evaluated in the dependence context, then the operator $\diamond$ is applied to the resulting values.[△]

The CTL* operators, which explicitly require `prop` and `path` dependence contexts, are excepted from pointwise evaluation. A `prop` expression is treated as a state predicate over markings m after plugging the parameter binding θ and the CTMC into the interpretation function. Similarly, `path` expressions are treated as predicates over paths Π composed by the operators according to CTL* semantics [see e.g. Emerson and Halpern, 1986].

To interpret variables mentioned inside expressions, we introduce *reference resolution*. A reference symbol $r \in R$ may point at some concrete $s \in \Sigma \setminus R$ or at another references $r' \in R$. We say that *r resolves to* $s \in \Sigma \setminus R$, written as $r \rightsquigarrow s$, if s is the unique concrete symbol with a chain of reference assignments from r to s. In addition, every concrete symbol resolves to itself, $s \rightsquigarrow s$ for all $s \in \Sigma \setminus R$. This notion is formalized as follows:

Definition 3.4 Let $:=^* \subseteq \Sigma \times \Sigma$ be the reflexive transitive closure of the relation $:=$, i.e. $s_1 :=^* s_2$ if and only if

$$\exists\, k \geq 0,\, s_1 = r_0, r_1, \ldots, r_k = s_2 \in \Sigma \text{ such that } r_i := r_{i-1} \text{ for all } i = 1, \ldots, k.$$

The symbol s_1 *resolves to* s_2, written as $s_1 \rightsquigarrow s_2$, if $s_2 \in \Sigma \setminus R$ is the *unique* concrete symbol for which $s_1 :=^* s_2$ holds.

If a symbol v of variable type is mentioned in an expression, we simply substitute it with its *value*. However, if v refers to a parameter symbol—or is actually parameter symbol—it is instead interpreted to refer to the corresponding element of the parameter vector θ. Formally,

$$[\![v]\!] = \begin{cases} [\![value(v')]\!], & \text{if } v \rightsquigarrow v' \text{ and } v' \in V, \\ \theta \mapsto \theta[par], & \text{if } v \rightsquigarrow par \text{ and } par \in Par, \\ \bot, & \text{otherwise.} \end{cases}$$

Note that if the reference v cannot be resolved or it points to an invalid symbol the interpretation is not defined.

Mentioning the marking of place simply refers to the number of tokens of the place after resolving references,

$$[\![\#p]\!] = \begin{cases} m \mapsto m(p'), & \text{if } p \rightsquigarrow p' \text{ and } p' \in P, \\ \bot, & \text{otherwise.} \end{cases}$$

Collection aggregations are defined with “big operator” semantics. An aggregation operator is equipped with a monoid $\langle \diamond, n \rangle$, where $\diamond$ is an associative binary operator and n

[△] This makes variable types with a dependence qualifier other than `const` specializations of *Reader* (also known as *Environment*) applicative functors [McBride and Paterson, 2008, Section 8].

is a the neutral element of the operator. The $\diamond$ operator joins the elements of the collection, whereas for empty collections, n is returned instead. The monoid $\langle \texttt{+}, 0\rangle$ is associated with the aggregation operator `sum`, $\langle \texttt{*}, 1\rangle$ with `prod`, $\langle \texttt{\&\&}, \texttt{true}\rangle$ with `all` and $\langle \texttt{||}, 1\rangle$ with `any`.

Definition 3.5 The *resolved elements* of a collection $c \in C$ are

$$resolved(c) = \wr s \mid \exists\, r_1, r_2 \in \Sigma \text{ such that } r_1 \rightsquigarrow c, r_1 \mathrel{+}= r_2, r_2 \rightsquigarrow s \wr,$$

where the multiset-builder notation respects the multiplicities in the relation :=. Note that both ends of a collection membership edge $r_1 \mathrel{+}= r_2$ may be references, which resolve as $r_1 \rightsquigarrow c$ on the collection end and $r_2 \rightsquigarrow s$ on the member end, respectively.

The aggregation *agg* is interpreted as

$$[\![agg(c)]\!] = \begin{cases} \Diamond_{s \in resolved(c')} [\![s]\!], & \text{if } c \rightsquigarrow c',\ c' \in C \text{ and } |resolved(c')| \geq 1, \\ n, & \text{if } c \rightsquigarrow c',\ c' \in C \text{ and } |resolved(c')| = 0, \\ \bot, & \text{otherwise,} \end{cases}$$

where the operator $\diamond$ acts over the interpretation functions $[\![s]\!]$ as discussed above while respecting multiplicity and the constant n is type coerced as needed.

The interpretations given above for mentioned symbols, token counts and collection aggregation in expressions are exploited in Section 4.5.1 on page 45 for *inlining*. The inlining process transforms expressions into expressions with equivalent semantics that can be exported to analysis tools without special support for RGSPNs.

Chapter 4

Incremental view synchronization

Complex industrial toolchains used for the model-based design of safety-critical cyber-physical systems frequently depend on various models on different levels of abstraction where abstract models are derived by model transformations. The derived models are often *views*, which aim to focus attention from a given *viewpoint* such that details relevant to a specific group of stakeholders are retained [Brunelière et al., 2017]. The views contain information that is related to and coming from other models, which can also be themselves other views. Incremental small-step execution of model transformations aids in reducing the computations costs of view maintenance [Varró, 2015].

In this chapter we propose a means to assemble formal stochastic models from domain models by model transformation. The resulting analysis model is a *view* of the engineering model from a *reliability* or *performability* viewpoint. The transformation should be ① *parametric* in the sense that the source metamodel, the transformation rules and the analysis model fragments that are instantiated may be specified by the user. In addition, stochastic Petri nets produced by the transformation should be ② *compatible with external analysis tools*. As a key to interpret analysis results of the derived stochastic models automatically, the transformation should ensure ③ *end-to-end traceability* between source model elements and the quantitative aspects of the stochastic model. Lastly, to support efficient mapping of constantly changing design candidates in design-space exploration, the transformation should be ④ *executed incrementally* driven by change notifications of the source model.

Existing transformation languages, such as ATL [Jouault et al., 2008], QVTr [Object Management Group, 2016, Chapter 7] or VIATRA Views [Debreceni et al., 2014] can describe mappings between instances of arbitrary metamodels; therefore they satisfy the requirement of ① user configurability. These language require the specification of the results of the transformation at the low level of individual model objects and links. While creating single objects at once is satisfactory for views that aim to create *abstractions* of the source model, automatic derivation of stochastic models is closer to *compilation*. The result of mapping even just one source element may have a complicated result, such as a collection of Petri net places, transitions and expressions trees describing quantitative aspects of the model. Hence we propose a transformation specification language tightly integrated with RGSPNs introduced in Chapter 3 as an alternative to general-purpose transformation languages for stochastic model creation.

The left side of the transformation rules are graph patterns which select the parts of the source model to be mapped. On the right side, the transformation results are specified as *RGSPN modules*, which are RGSPN model fragments. The typing discipline from Definition 3.3 on page 25 is extended to transformation rules to aid in catching bugs.

In current analysis tools, there is little support for reference symbols, variables and collections introduced in RGSPNs. To ② ensure compatbility, a *inlining* step is also incorporated into the transformation chain. The inlining *concretizes* the *abstract* RGSPN constructed according to the user-provided view specification and yields a *concrete* RGSPN, that contains no references, collections or references to variables. Variable symbols are kept so that they can exported to the analysis tool as stochastic metrics to be computed or as queries to be answered. Matching of the queries to source model concepts is provided by ③ traceability relations that are maintained implicitly, i.e. without additional user intervention.

The ④ incremental execution of the transformation is ensured by the use of an incremental graph query engine [Ujhelyi et al., 2015] and a reactive model transformation platform [Bergmann et al., 2015]. If a step in the transformation chain cannot be executed due to a malformed input model the effects of the transformation are *delayed* until the issue is resolved. Upon delaying, an error marker is generated that is removed when the transformation can resume successfully.

After briefly reviewing related work we describe the proposed transformation chains, as well as its specification language and semantics. Then the instantiation of RGSPN modules is discussed, finally followed by the details of the concretization transformation and its handling of inconsistencies by the means of delayed execution.

4.1 Related work: view synchronization approaches

Now we briefly review some approaches for synchronizing view models for engineering DSLs with a focus on approaches supporting incremental synchronization and the creation of complex formal modes.

Model transformations are a pervasive concept in model-driven engineering (MDE) where they are employed to modify a model *in place* or convert between different representations and abstraction levels of models by either creating a *new target* model or *updating* an existing one [Czarnecki and Helsen, 2006]. For transformations with separate source and targets, *tracing* connects related source and target objects. The traceability model may be maintained *manually* (*explicitly*), or the transformation engine may provide *automatic* (*implicit*) traceability. *Batch* execution re-evaluates the whole transformation when the source model changes to create a new version of the target. In contrast, *target incrementality* (*change propagation*) only performs necessary changes on the target model. *Source incremental* transformations also attempt to minimize the re-examined portion of the source model, for example, by subscribing to change notifications.

Varró [2015] categorized the styles of change-driven model transformations as follows: 1. Transformations with *no incrementality* only execute in batch mode. 2. *Dirty incrementality* marks target models or elements to be recomputed as dirty upon source changes and re-runs the transformation for the dirty portions. 3. *Incrementality by traceability* relies of missing or dangling traceability links to determine the target elements that are created or removed. 4. *Reactive source incrementality* triggers transformation rules by change notifications without relying on traceability links.

View transformations are specific models transformations that aim to create target models that describe the source from a specific viewpoint. Brunelière et al. [2017] has surveyed view transformation tools, including incremental approaches.

The ATL transformation language is a domain-specific language for describing model-to-model transformations with implicit traceability [Jouault et al., 2008]. Algorithms for trace-based incremental ATL execution were proposed by Xiong et al. [2007], as well as by

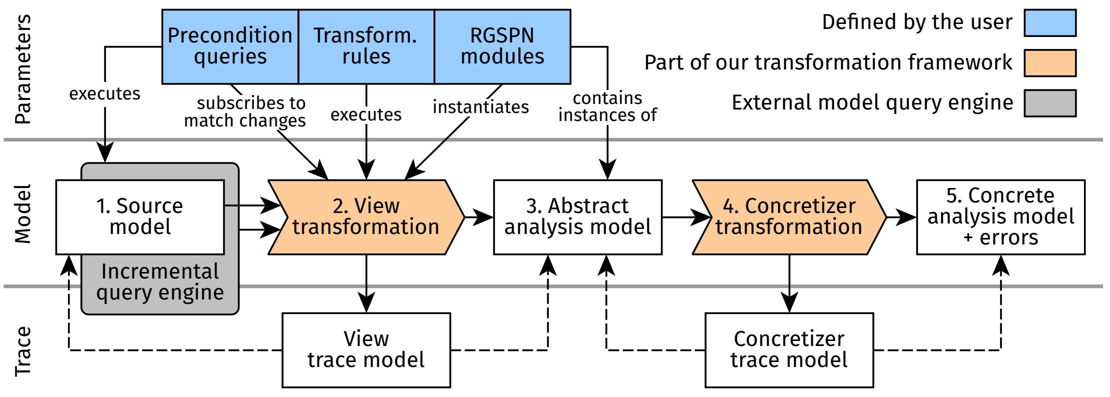


Figure 4.1 Overview of the transformation chain.

Jouault and Tisi [2010].

The Query-View-Transformation (QVT) family of languages were also developed for describing model-to-model transformations [Object Management Group, 2016]. The *relation* QVTr sublanguage is a declarative language for the correspondences between the source and target models with implicit traceability, which is compiled to the simple *core* QVTc sublanguage. The *operational* QVTo sublanguage provides imperative transformations. Song et al. [2011] proposed trace based incremental transformation for a subset of QVTr suitable for deriving runtime models.

Triple grammars grammars (TGG) can describe model transformations by employing and explicit trace model called the *correspondence* model [Schürr, 1994]. Restrictions of TGGs (view TGGs) have been proposed for incremental view synchronization [Jakob et al., 2006; Anjorin et al., 2014]. Greenyer and Rieke [2011] proposed extension for TGGs so that they could derive formal models from sequence diagram specifications. However, their approach does not support incremental execution.

The VIATRA reactive model transformation platform provides a framework for change-driven model transformations with optional manual traceability [Bergmann et al., 2015]. Debreceni et al. [2014] proposed VIATRA Views as a declarative, incremental view transformation engine built on incremental model queries [Ujhelyi et al., 2015] and VIATRA.

Mosteller et al. [2016] proposed an approach to define semantics of domain-specific modeling languages by Petri nets in the Renew meta-modeling and transformation framework. A specialized modeling tool is generated based on a mapping from DSL elements to Petri net fragments. The user can manipulate the concrete syntax of the DSL while a Petri net is assembled from the fragments by the tool for analysis and code generation.

The ability to reference target objects created by different transformation rules is especially important in incremental transformation languages, because the developer of the transformations cannot rely on the order of rule executions. Hence transformation rules must be able to share target objects regardless their execution order.

In ATL model elements generated by rules can be resolved at any time by the built-in `resolveTemp` operation. In QVTr relations between source and target elements can be joined by `when` clauses. In TGGs target object sharing is implicitly present in the correspondence parts of the rule triples. Greenyer and Rieke [2011] introduced *reusable patterns* to TGGs to further refine object sharing. VIATRA Views support referencing objects produces by other rules with the `@Lookup` annotation. In contrast, modular Petri nets [Kindler and Petrucci, 2009] and thus our RGSPN fragments model importing symbols explicitly by reference symbols and reference assignments.

4.2 Overview of the transformation engine

The transformation chain from engineering models to analyzable RGSPNs is shown in Figure 4.1. The architecture is divided into three parts: 1. the *parameters* of the transformation, which constitute the transformation specification provided by the user, 2. the *models* and model transformations participating in the chain and 3. the *trace* models providing end-to-end traceability.

4.2.1 Transformation specification

The transformation description contains the *precondition queries*, which are executed on an incremental model query engine. For each query match the *transformation rules* specify which *RGSPN* module should be instantiated.

In addition, the user is able to reuse quantitative aspects of the engineering model in the analysis model and define new quantitative aspects to be evaluated as stochastic queries. These *associated symbols*, along with their traceability information play roles similar to the parameters (prefixed with “$”), stochastic metrics (“/”) and queries (“/$”) introduced as an extension to UML diagrams by Bernardi and Donatelli [2003].

Transformation rules can govern the mapping of numeric attributes from the domain model to the variable symbols of the RGSPN. Attributes may be marked as parameters, which are retained as parameters symbols in RGSPN and when the analysis model is exported to external solvers. Therefore the parameter mapping relates domain attributes to sensitivity analysis [Blake et al., 1988], parametric solution of Markov chains [Hahn et al., 2011] and parameter synthesis [Quatmann et al., 2016; T. Molnár, 2017], letting users perform the aforementioned tasks directly on the domain model.

Moreover, *derived* features may also be specified that associate RGSPN symbols with domain model elements. In contrast with model query based approaches for the creation of derived features [Ráth et al., 2012] the domain model is not modified to incorporate the features. However, code generation and the *extension methods* feature of *Xtend*[1] are utilized in Section 5.2.1 on page 57 to emulate derived features syntactically in a general purpose programming language.

4.2.2 Transformation chain

As it is shown in Figure 4.1 the construction of RGSPN analysis models is realized as a *chain* of two model transformations.

The precondition queries of transformation rules are ran on the 1. *source model* by an *incremental query engine*. The 2. *view transformation* maintains a 3. *abstract analysis model* based on the query matches of the precondition queries and instantiates the RGSPN modules according to the transformation rules. In addition, the associated symbols relating to the quantitative aspects of the source model elements are instantiated. A *view trace model* links the elements and query matches of the source model to the symbols of the abstract RGSPN.

The abstract RGSPN contains reference symbols, variables and collections that are not directly exportable to analysis tools. Therefore the 4. *concretizer transformation* is needed

[1] https://www.eclipse.org/xtend/

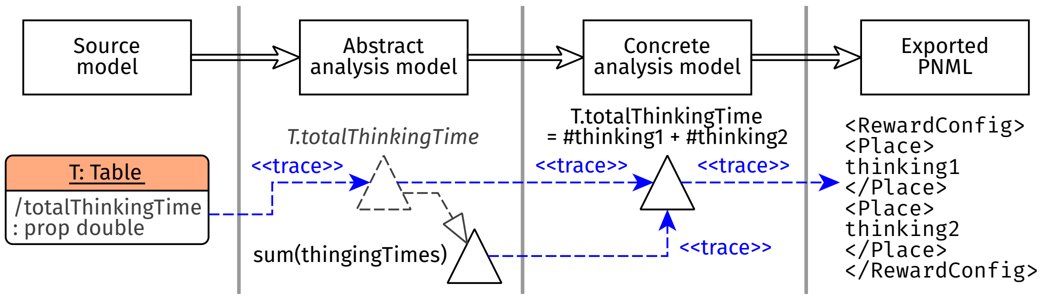


Figure 4.2 Traceability for associated symbols.

to *inline* these features and obtain a 5. *concrete analysis model*, which is an RGSPN without advanced features. The concrete model can be exported as a GSPN possibly parameter- and marking-dependent transition rates for analysis with external tools. Furthermore, the value expressions of the retained variable symbols, which refer to elements of the concrete model, can serve as stochastic metrics and queries to be analyzed.

If the abstract analysis model is inconsistent, e.g. it contains unassigned references or circular references, concretization is delayed and *error* markers are generated until the inconsistency is resolved. The *concretizer trace model* links the abstract analysis model to the concrete one; moreover, it also allows the interpretation of error markers.

Both the concrete and abstract RGSPNs are fully materialized as instance models so that they can be freely inspected and exported. It is also possible to subscribe to change notification of either of the models, for example, to incorporate our transformation chain into a larger chain.

4.2.3 End-to-end traceability

Fully traversing the view and concretizer trace models allows the association of concrete RGSPN symbols with source model elements. Thus when an external solver is interfaced with the transformation, it is sufficient to provide traceability between the concrete analysis model and the external solver so that the analysis results remain interpretable in the context of the source domain model.

Running example 4.1 Figure 4.2 shows the trace links for an RGSPN symbol derived feature `totalThinkingTime` of the domain class `Table`.

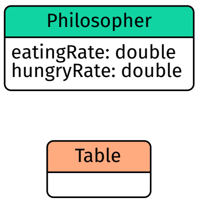
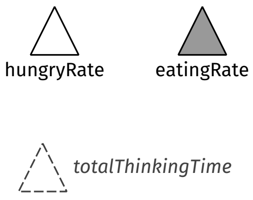
The first trace link is the view trace model associates the domain object `T` with the reference symbol `T.totalThinkingTime` in the abstract analysis model. A variable symbol is assigned to the reference.

The concretization transformation resolves and inlines all references, therefore both trace links from the reference symbol and the concrete symbol in the concretization trace model point at the same variable symbol in the concrete analysis model. As result of the inlining, the aggregation operator in value expression of the variable is replaced with the sum of the member variables. In the example, these members are token count expressions `#thinking1` and `#thinking2`.

If the concrete analysis model is exported to an external analysis tool, such as Petri-DotNet [Vörös et al., 2017b], the PNML serializer may also output traceability information. In the example, the value expression of `T.totalThinkingTime` is turned into a reward configuration for PetriDotNet. The end-to-end trace links associate the exported reward

Table 4.1 Feature rules for dining philosophers transformation specification.

Domain class	Transformation rule	Associated symbols

 Philosopher
eatingRate: double
hungryRate: double
Table
	`features {`
	`  Philosopher {`
	`    param eatingRate`
	`  }`
	`  Table {`
	`    derived prop double`
	`        totalThinkingTime`
	`  }`
	`}`
		

configuration with the derived feature, hence the results of stochastic analysis can be interpreted in the context of the domain model and its derived features.

4.3 Transformation specification language

We present the transformation specification language trough a running example. Tables 4.1 and 4.2 show an example transformation description for the dining philosophers domain. On the left graph patterns are displayed as subgraphs, while the RGSPN modules on the right also use graphical concrete syntax. In the middle column, the textual concrete syntax of transformation descriptions is shown.

4.3.1 Feature rules

The first section of the transformation specification contains *feature rules* that describe the associated symbols relating to the *features* (*attributes*) of the domain model elements. The feature rule section is introduced by the **`features`** keyword. Each domain class may have a sub-section describing the mapping of its features.

By default, each `int`, `double` and `boolean` attribute of a domain class is mapped to a `const` variable symbol in the abstract RGSPN. The value expression of the variable symbol is a literal that equals to the value of the domain attribute.

Users may override the attribute mapping of `double` features by specifying `param` mapping instead. Attributes marked as **`param`** are turned into parameter symbols instead.

Lastly, feature rules may specify **`derived`** features. An RGSPN reference symbol with the given type and name is created and is associated with the domain element.

Running example 4.2 The metamodel for the dining philosophers domain contains the classes `Philosopher` and `Table`. The two attributes of type `double` of `Philosopher` are `eatingRate` and `hungryRate`, while `Table` has no attributes.

For `Philosopher` the feature rule in Table 4.1 marks the attribute `eatingRate` as a parameter. Hence `eatingRate` is mapped to the RGSPN as a parameter symbol, while `const` variable symbol is created for `hungryRate`.

The `Table` feature rule prescribes a derived feature `totalThinkingTime` of type `prop double`. Hence a reference symbol with the same name and type is associated with `Table` objects.

4.3.2 Mapping rules

A *mapping* rule associates a *precondition* model query with a set of *lookup declarations*, assignments and collections membership declarations, as well as optionally a *postcondition* RGSPN module. Thus the abstract analysis modules is weaved from RGSPN module instances and the edges added between them by the mappings.

For every tuple of match arguments in the match set of the precondition query instances of the assignments, collections memberships and the RGSPN module are added to the abstract analysis model. The instantiation of modules is perfomed by copying their contents to the abstract analysis model after renaming their symbols to avoid collisions. The match argument tuple serves as the source of traceability links to the instantiated objects.

After the keyword **`mapping`** the precondition graph pattern is named, followed by its list of parameters. The associated symbols of match arguments are accessible in the body of the mapping rule by mentioning the name of the match argument, followed by the dot operator and the name of the associated symbol.

The name of the RGSPN module to instantiate and a *local name* for the module instance may be specified after the `=>` operator. If the module instantiation clause is present symbols inside the module instance can be referred to using the local name of the instance and the dot operator similarly to associated symbol references.

Lookup declarations

The view trace model can be traversed during the view transformation by *lookup declarations*, analogously to the `@Lookup` annotation introduced by Debreceni et al. [2014] for the traversal of traceability relations in view maintenance. Introduced by the keywords `lookup` they name a precondition pattern and provide a list of match arguments. The match arguments must be a subset of the parameters of the containing mapping rule and a pattern match of the specified pattern must exist.

After the operator **`as`** a local name may be given to the module instance created by the lookup up mapping rule. Hence it is possible to refer to symbols instantiated by other mapping rules in order to connect them with the rest of the analysis model. The execution of the view transformation, which is described in Section 4.4, ensures that the modules can be instantiated in any order and allows cyclic lookups between mapping rules.

Edge declarations

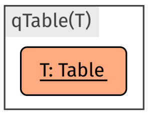
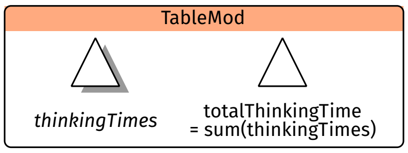
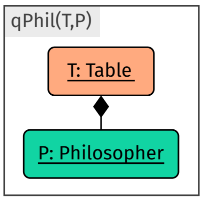
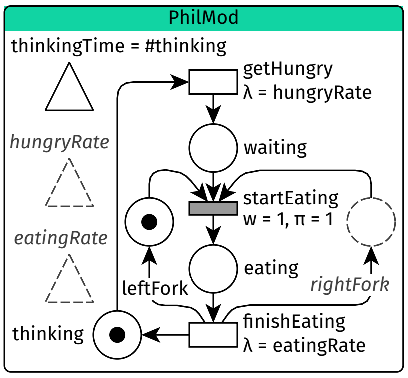
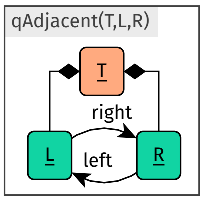
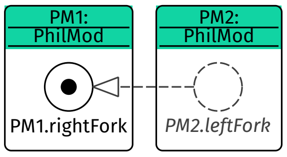
Edges between different RGSPN module instances and symbols associated with domain objects are also supported. The `:=` and `+=` operators may add reference assignments and collection membership edges, respectively. Typing rules in Definition 3.3 on page 25 are checked in mapping rules as well as in RGSPN modules. Thus the abstract RGSPN output by the view transformation is ensured to be well-typed.

Running example 4.3 In Table 4.2 three mapping rules are given for the dining philosophers domain. The feature rules in Table 4.1 are assumed for the mappings.

The pattern `qTable` matches for all instances `T` of the class `Table`. An instance of the module `TableMod` is created with the local name `TM`. The symbol `totalThinkingTime` of `TM` is assigned to the derived symbol with the same name of the domain object `T`.

The pattern `qPhil` matches each philosopher `P` sitting around a table `T`. The corresponding mapping instantiates `PhilMod` with the local name `PM`. The table `T` is included in the match parameter list such that the module `TM` created by the mapping `qTable` for

Table 4.2 Mapping rules for dining philosophers transformation specification.

Precondition	Transformation rule	RGSPN module
	`mapping qTable(T) => TableMod TM { T.totalThinkingTime := TM.totalThinkingTime }`	 TableMod thinkingTimes totalThinkingTime = sum(thinkingTimes)
	`mapping qPhil(T, P) => PhilMod PM { lookup qTable(T) as TM PM.hungryRate := P.hungryRate PM.eatingRate := P.eatingRate TM.thinkingTimes += PM.thinkingTime }`	 PhilMod thinkingTime = #thinking getHungry λ = hungryRate hungryRate waiting startEating w = 1, π = 1 eatingRate eating leftFork rightFork thinking finishEating λ = eatingRate
	`mapping qAdjacent(T, L, R) { lookup qPhil(T, L) as PM1 lookup qPhil(T, R) as PM2 PM2.leftFork := PM1.rightFork }`	

T can be looked up. The references hungryRate and eatingRate inside PM are assigned to the symbols constructed from the attributes of P. The variable symbol thinkingTime of PM is added to the collection thinkingTimes of PM.

The interaction between the instances of TableMod and PhilMod showcases the advantages of collection symbols in RGSPN. The individual performance measures thinkingTime of philosophers are added to a collection, such that the aggregate performance measure totalThinkingTime can be computed.

Lastly, qAdjacent find philosophers L and R sitting next to each other around the table T. While the corresponding mapping has no RGSPN module to instantiate, it looks up the modules instances PM1 and PM2, respectively. The reference to the left fork of the right philosopher is assigned to the right for of the left philosopher, which completes the dining philosophers model.

4.4 Generic view transformation to stochastic Petri nets

The first transformation in our transformation chain is the view transformation that derives abstract RGSPN analysis models from engineering model. Its four main objectives are

- the instantiation of associated symbols for domain objects,
- the instantiation of RGSPN modules for mapping rule precondition matches,
- the instantiation of additional assignment and collection membership edges according to lookup specifications and

- the synchronization of the value expressions of associated attribute symbols with the values of domain attributes.

The creation and removel of associated symbols, as well as RGSPN modules and edges follows the strategy for incremental view maintenance by graph queries proposed by Debreceni et al. [2014]. Analysis model elements are created for precondition pattern matches with missing traceability links, while analysis model elements with dangling traceability links are deleted. Hence the style of instantiation is small-step *incrementality by traceability* [Varró, 2015]. The trace model is *implicit*, i.e. users do not need to define a metamodel for traceability links themselves. The transformation engine maintains the view trace model automatically instead.

Edges defined inside mapping rules are only instantiated when there is a traceability link for the abstract RGSPN symbols on both ends of the edge and the precondition of the mapping rule matches. To this end a *connection* graph pattern is generated which incorporates the precondition pattern and also matches the traceability links for the looked up RGSPN modules and associated symbols of the mapping rule. By evaluating the generated pattern over the source model and the view trace model jointly the set of edges that can be added to the abstract analysis model are determined. Dedicated traceability links are also added between the matches of the generated patterns and the inserted edges; therefore the edges can be removed when their corresponding match of the connection pattern disappears and the traceability link becomes dangling.

Running example 4.4 The connection pattern generated from the mapping rule `qTable` in Table 4.2 is $\texttt{qTable}^*(x) = \texttt{qTable}(x) \land (\exists \ell_1 . \mathit{moduleInstanceTrace}(\texttt{qTable}, \langle x \rangle, \ell_1)) \land (\exists \ell_2 . \mathit{associatedSymbolTrace}(x, \ell_2))$, where $\mathit{moduleInstanceTrace}(\phi, t, \ell)$ indicates that ℓ is the traceability link for the RGSPN module instance created by the mapping rule with precondition ϕ for the pattern match tuple t and $\mathit{associatedSymbolTrace}(x, \ell)$ indicates that ℓ is the traceability link for the symbols associated with the source object x. Likewise we have $\texttt{qPhil}^*(x, y) = \texttt{qPhil}(x, y) \land (\exists \ell_1 . \mathit{moduleInstanceTrace}(\texttt{qPhil}, \langle x, y \rangle, \ell_1)) \land (\exists \ell_2 . \mathit{associatedSymbolTrace}(y, \ell_2)) \land (\exists \ell_3 . \mathit{moduleInstanceTrace}(\texttt{qTable}, \langle x \rangle, \ell_3))$ and $\texttt{qAdjacent}^*(x, y, z) = \texttt{qAdjacent}(x, y, z) \land (\exists \ell_1 . \mathit{moduleInstanceTrace}(\texttt{qPhil}, \langle x, y \rangle, \ell_1)) \land (\exists \ell_2 . \mathit{moduleInstanceTrace}(\texttt{qPhil}, \langle x, z \rangle, \ell_2))$.

Symbols associated with numerical attributes of domain objects are synchronized with the values of the attributes. The transformation engine subscribes to change notifications from the source model and updates values of the symbols associated with the changed object. Hence the attribute synchronization is *reactive source incremental*.

Running example 4.5 Figures 4.3 to 4.5 show an example transformation of a dining philosophers domain model according to the feature rules in Table 4.1 on page 38 and the mapping rules in Table 4.2. Symbols inside module instances with no adjacent edges between modules were suppressed for clarity. Trace links are indicated by writing the names of the linked pattern matches in the analysis model, as well as the coloring of pattern matches and model elements.

The initial model in Figure 4.3 contains a `Table T` and a `Philosopher P1`. The precondition query `qTable` has a single match ⟨`T`⟩ and `qPhil` has a single match ⟨`T`, `P1`⟩.

The associated symbols `P1.eatingRate` and `P1.hungryRate` were created for `P1`. The derived feature symbol `T.totalThinkingTime` is associated with `T`. Module instances `TM` of `TableMod` and `PM1` of `PhilMod` were also added to the abstract analysis model for the precondition matches `qTable`⟨`T`⟩ and `qPhil`⟨`T`, `P1`⟩, respectively. The

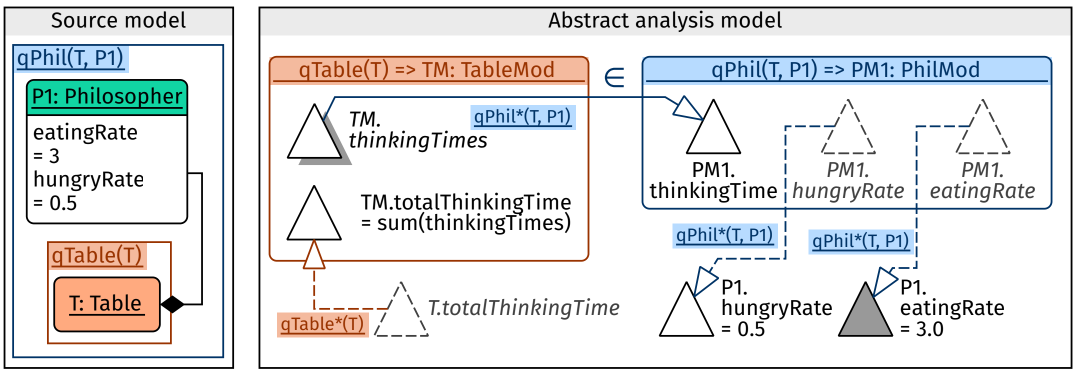


Figure 4.3 Initial setup for the example view transformation.

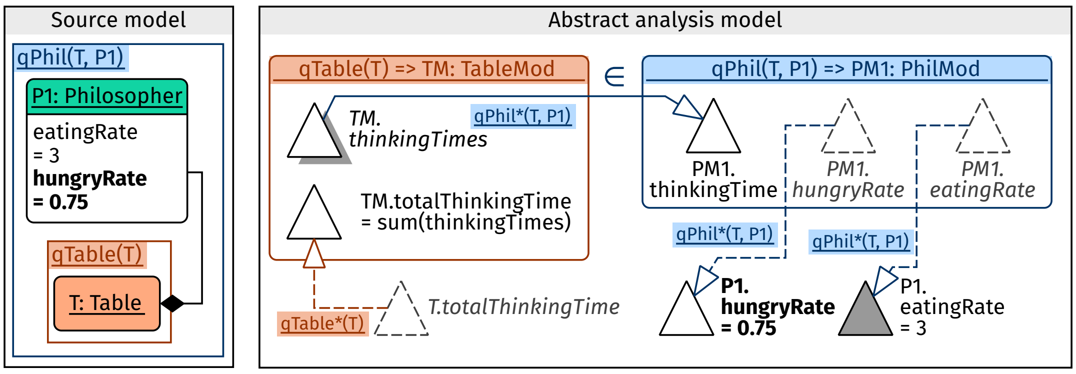


Figure 4.4 State of the example view transformation after modifying `P1.eatingRate`.

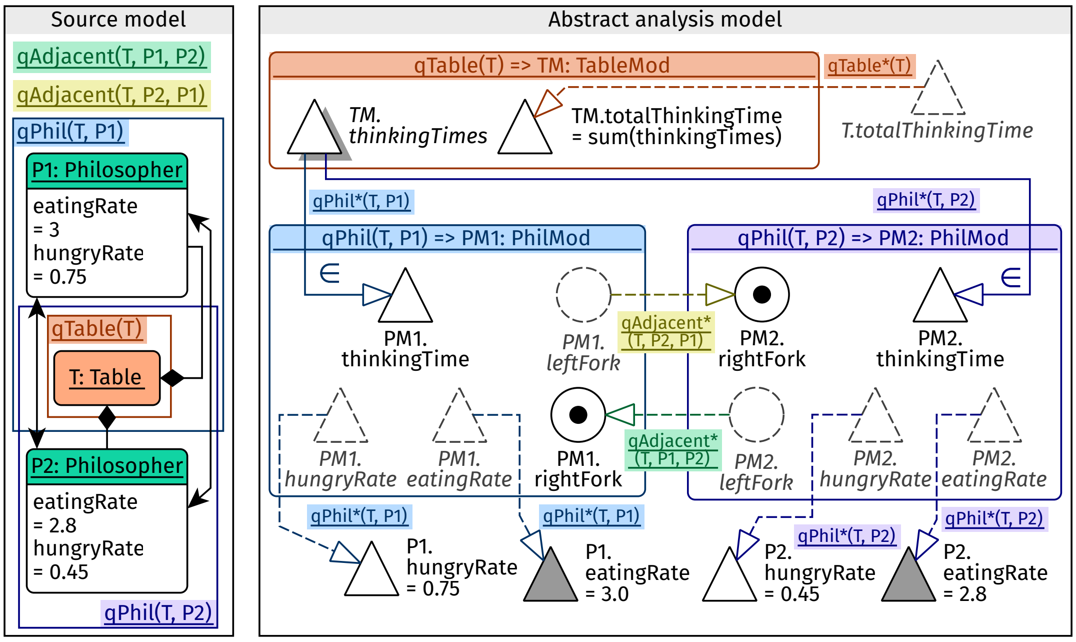


Figure 4.5 State of the example view transformation after adding a new philosopher P2.

connection patterns `qTable`* generated from the `qTable` mapping rule and `qPhil`* generated from the `qPhil` mapping rule govern the insertion of RGSPN edges between modules. The connection match `qTable`*⟨`T`⟩ assigns the variable `TM.totalThinkingTime` to the derived reference `T.totalThinkingTime`. In addition, `qPhil`*⟨`T`, `P1`⟩ adds `PM1.thinkingTime` to the collection `TM.thinkingTimes` and assigns the features symbols associated with `T` to the respective reference symbols in the module `PM1` such that they can be mentioned in the expressions inside the implementation of `PhilMod`.

In Figure 4.4 the attribute `hungryRate` of `P1` was changed. Therefore the transformation synchronized the literal in the value expression of the feature symbol `P1.hungryRate` in the abstract analysis model.

In Figure 4.5 a new `Philosopher P2` was created, which sits both on the left and right of `P1` around the circular table `T`. New precondition matches `qPhil`⟨`T`, `P`⟩, `qAdjacent`⟨`T`, `P1`, `P2`⟩ and `qAdjacent`⟨`T`, `P2`, `P1`⟩ appeared, which lead to the instantiation of a new `PhilMod PM2`. The symbols inside the `PM2` are connected to the rest of the analysis model with edges due to the connection match `qPhil`*⟨`T`, `P`⟩. Furthermore, matches `qAdjacent`*⟨`T`, `P1`, `P2`⟩ and `qAdjacent`*⟨`T`, `P2`, `P1`⟩ of the connection query generated from the mapping rule `qAdjacent` caused the assignments of `PM1.leftFork` to `PM2.rightFork` and `PM2.leftFork` to `PM2.rightFork`.

Because analysis model elements with dangling trace links are removed the deletion of `P2` from the source model would cause the view transformation to restore the analysis model to the state shown in Figure 4.4.

4.5 Stochastic Petri net concretization

The second step in our transformation chain is the concretization with derives a RGSPN containing only concrete symbols from the abstract analysis model. The resulting concrete analysis model can be readily exported as a parametric GSPN to external solvers. In addition, variable symbols are preserved in the concrete analysis model so that they can serve as stochastic metrics and queries to be evaluated.

The three main responsibilities of the concretization transformation are

- the copying of concrete place, transition, variable and parameter symbols from the abstract analysis model to the concrete analysis model,
- the resolution of reference symbols and
- the inlining of the variables and collection aggregations into RGSPN expressions.

The execution of the the aforementioned transformations may be prevented by errors and inconsistencies of the abstract analysis model. Inconsistency may be caused by a reference symbol having no resolution to a concrete symbol, reference resolution leading to parallel arcs or cyclic dependencies of expression. Robust inconsistency handling is especially important in change-driven incremental transformation chains, as a sequence of modifications of the source model may induce inconsistency in the abstract analysis model during execution even if the abstract analysis model becomes consistent at the end of the sequence. Our handling delays parts of the concretization until the inconsistency is resolved, while an error marker is generated to alert the user. The list of error markers can be checked at the end of source model modification sequences to ensure that the transformation chain fully synchronized the analysis models without hampering the execution of individual modification operations in the sequence.

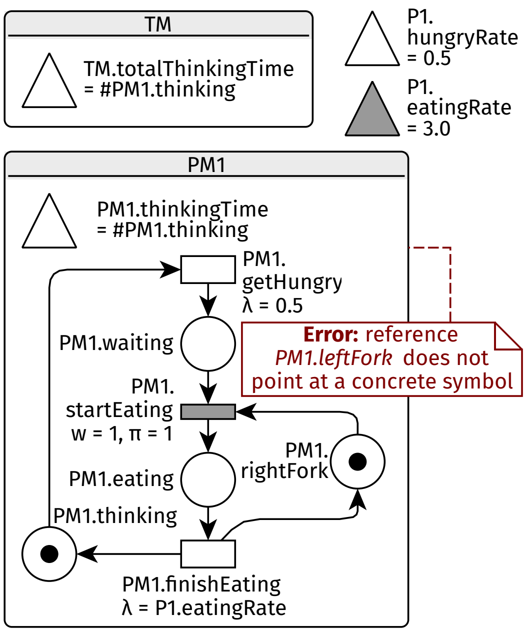

TM
TM.totalThinkingTime = #PM1.thinking
P1. hungryRate = 0.5
P1. eatingRate = 3.0
PM1
PM1.thinkingTime = #PM1.thinking
PM1. getHungry λ = 0.5
PM1.waiting
Error: reference PM1.leftFork does not point at a concrete symbol
PM1. startEating w = 1, π = 1
PM1. rightFork
PM1.eating
PM1.thinking
PM1.finishEating λ = P1.eatingRate

Figure 4.6 Concretization of the initial RGSPN from Figure 4.3 on page 42.

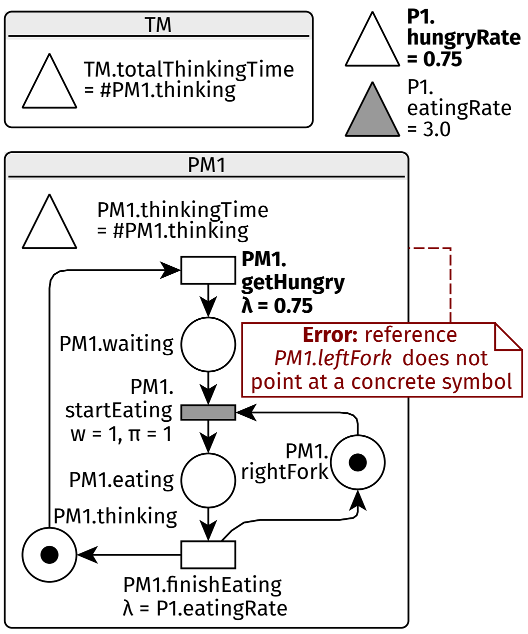

TM
TM.totalThinkingTime = #PM1.thinking
P1. hungryRate = 0.75
P1. eatingRate = 3.0
PM1
PM1.thinkingTime = #PM1.thinking
PM1. getHungry λ = 0.75
PM1.waiting
Error: reference PM1.leftFork does not point at a concrete symbol
PM1. startEating w = 1, π = 1
PM1. rightFork
PM1.eating
PM1.thinking
PM1.finishEating λ = P1.eatingRate

Figure 4.7 Concretization of the RGSPN from Figure 4.4 on page 42 after changing `P1.eatingRate`.

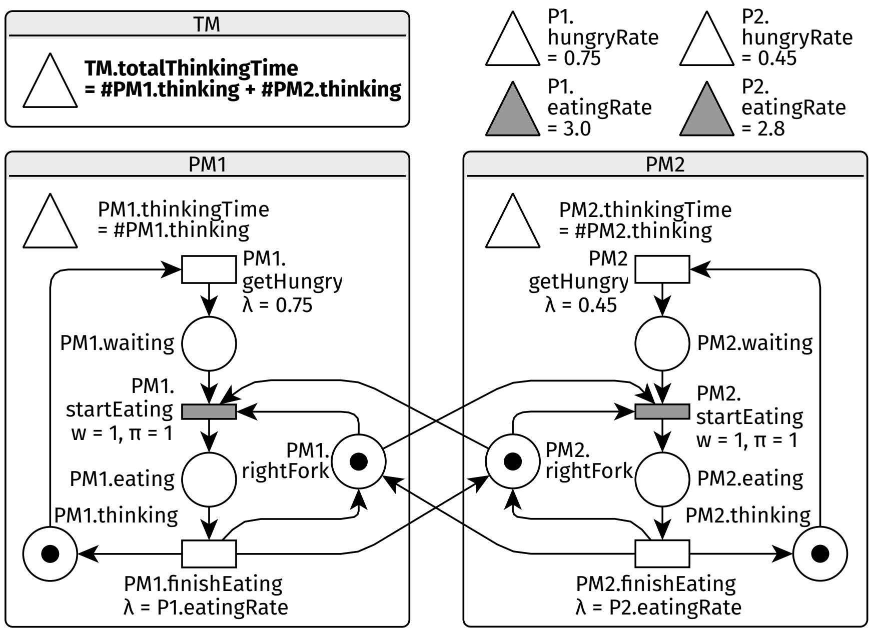

TM
TM.totalThinkingTime = #PM1.thinking + #PM2.thinking
P1. hungryRate = 0.75
P2. hungryRate = 0.45
P1. eatingRate = 3.0
P2. eatingRate = 2.8
PM1
PM1.thinkingTime = #PM1.thinking
PM1. getHungry λ = 0.75
PM1.waiting
PM1. startEating w = 1, π = 1
PM1. rightFork
PM1.eating
PM1.thinking
PM1.finishEating λ = P1.eatingRate
PM2
PM2.thinkingTime = #PM2.thinking
PM2 getHungry λ = 0.45
PM2.waiting
PM2. startEating w = 1, π = 1
PM2. rightFork
PM2.eating
PM2.thinking
PM2.finishEating λ = P2.eatingRate

Figure 4.8 Concretization of the RGSPN from Figure 4.5 on page 42.

Running example 4.6 Figures 4.6 to 4.8 show the concretizations of the abstract analysis models from Figures 4.3 to 4.5 on page 42. Traceability links are indicated by identical names of symbols in the abstract and concrete analysis models.

Figure 4.6 shows the initial concrete analysis model. Reference and collections symbols have been eliminated from the model. The mentioned reference symbol `PM1.eatingRate` in λ(`PM1.finishEating`) was replaced with the parameter symbol `P1.eatingRate` by reference resolution. The value of the variable symbol `P1.hugryRate` was inlined into λ(`PM1.getHungry`). The aggregation expression in `TM.totalThinkingTime` was expanded into `#PM1.thinking`.

All expressions in the conrete model are “flat”, i.e. they contain no mentions of variables or collection aggregations. The only non-constant expressions are direct parameter and marking dependencies. While variable symbols are not mentioned in expressions of the Petri net, they remain in the model so that they can be exported as metrics and queries to external analysis tools.

In Figure 4.3 on page 42 the referenced `PM1.leftFork` in the abstract analysis model did not point at any (concrete) symbol. Therefore the reference could not be resolved in Figure 4.6. Creation of the arcs between `PM1.startEating`, `PM1.finishEating` and `PM1.leftFork` in the concrete analysis model is *delayed*. An error marker is generated indicating that the concretization was not fully completed.

In Figure 4.4 on page 42 the modification of the `eatingRate` of `P1` is propagated to the concrete RGSPN. In contrast with Figure 4.4 on page 42 not only the value expression of the variable symbol `P1.eatingRate` is synchronized but also λ(`PM1.finishEating`) is updated. Because the reference `PM1.leftFork` is still unresolved the error marker is preserved; however, the rest of the transformation could be executed.

The addition of the `Philosopher` `P2` lead to a new `PhilMod` instance `PM2` in Figure 4.5 on page 42, which was copied into Figure 4.8. Due to the reference resolutions (see Definition 3.4 on page 30) `PM1.leftFork` $\rightsquigarrow$ `PM2.rightFork` and `PM2.leftFork` $\rightsquigarrow$ `PM1.rightFork` all symbols and arcs could be concretized successfully. In the concrete RGSPN the `rightFork` symbols stand for the `leftFork` symbols of the abstract RGSPN. Moreover, the value expression of `TM.totalThinkingTime` was update to accommodate the new element `PM2.thinkingTime = #PM2.thinking` of the aggregated collection `TM.thinkingTimes` in the abstract analysis model.

4.5.1 Transformation execution

The copying of place, transition, variable and parameter symbols is performed in a *trace incremental* style, similarly to the instantiation of the abstract analysis model. The symbol in the abstract analysis model is connected to its *image* in the concrete analysis model with a traceability link.

Resolution of references during copying is also trace incremental. To copy Petri net arcs $s_1 \xrightarrow{e} s_2$ (or $s_1 \xleftarrow{e} s_2$, $s_1 \overset{e}{\multimap} s_2$, respectively) the symbols at both ends of the arc are resolved to concrete places and transitions according to Definition 3.4 on page 30 first, such that we have $s_1 \rightsquigarrow p$ and $s_2 \rightsquigarrow t$ for some transition t and place p. Since p and t are located in the abstract analysis model, traceability links must be traversed to locate their images p' and t' in the concrete analysis model. Then the arc of the form $p' \xrightarrow{e} t'$ can be added to the concrete analysis model and connected to $s_1 \xrightarrow{e} s_2$ with a traceability link. The concretized arc is removed when any of its traceability links become dangling.

In contrast, the inlining of expressions is not trace incremental, as the value of an expression may change even if no objects are added or removed in the abstract analysis model. A purely notification driven approach is also unsuitable, as an abstract analysis model change requires updating not only the image of the affected symbol in the concrete analysis model, but also the images of any expressions referring to the symbol, possibly through a chain of references or collection aggregations. A *dirty incrementality* is employed instead, where symbols depending on changed source model elements are marked as dirty when the change notification is received. After the rest of the transformation rules have been processed the *cleanup* rule is fired, which re-evaluates any expressions connected to dirty symbols without unnecessary duplication of computations.

4.5.2 Expression dependencies

In this section we describe the algorithm used for dirty marking and re-evaluation of expressions. The algorithm works on the level of symbols and arcs and re-evaluates any expressions connected to a symbol when cleaning up dirty marks. Although expression-level granularity could prevent event more re-evaluations compared to symbol-level dirty marking, in our evaluation Section 5.3 on page 59 we found the scalability of the current approach acceptable.

Re-evaluation of the image of an expression may be required

- when the value of a variable mentioned directly in the expression or indirectly through variable, references and collection aggregations changes,
- when a reference assignment $r := s$ of a directly or indirectly mentioned reference symbol r is created or removed,
- when a collection membership edge $c \mathrel{+}= s$ of a directly or indirectly mentioned collection symbol c is created or removed.

We formalize these notion as follows by mutually recursively defining the *support*: $\Sigma \cup \mathit{Expr}_\Sigma \cup (\rightarrow) \cup (\leftarrow) \cup (\multimap) \rightarrow 2^\Sigma$ of expressions, symbols and arcs of an RGSPN over the signature signature Σ, as well as *touching* of symbols:

Definition 4.1 The *support* of an expression is the union of the supports of the symbols mentioned directly,

$$\mathit{support}(e) = \bigcup_{s \text{ is mentioned in } e} \mathit{support}(e).$$

The *support* of a symbol contains itself, the supports of any expressions associated with the symbol, its pointed symbols and its collection members,

$$\begin{aligned}\mathit{support}(s) = \{s\} &\cup \mathit{support}(m_0(s)) \cup \mathit{support}(\lambda(s)) \cup \mathit{support}(w(s)) \\ &\cup \mathit{support}(\pi(s)) \cup \mathit{support}(\mathit{value}(s)) \cup \bigcup_{s := s'} \mathit{support}(s') \cup \bigcup_{s \mathrel{+}= s'} \mathit{support}(s')\end{aligned}$$

where $P, T_T, T_i, V, \mathit{Par}, R, C$ are the sets of places, timed transitions, immediate transitions, variables, parameters, references as collections of Σ according to Definition 3.1 on page 24.

The *support* of an arc $\langle p, e, t\rangle \in (\rightarrow) \cup (\leftarrow) \cup (\multimap)$ is the support of its inscription,

$$\mathit{support}(\langle p, e, t\rangle) = \mathit{support}(e).$$

The *cone*: $\Sigma \rightarrow 2^{\Sigma \cup (\rightarrow) \cup (\leftarrow) \cup (\multimap)}$ of a symbol is the set of symbols and arcs that contain it in their supports, which is the set of RGSPN elements the symbol can affect,

$$\mathit{cone}(s) = \{x \in \Sigma \cup (\rightarrow) \cup (\leftarrow) \cup (\multimap) \mid s \in \mathit{support}(x)\}.$$

Example 4.7 In Figure 4.5 on page 42 the *support* of `T.totalThinkingTime` is {`T.totalThinkingTime`, `TM.totalThinkingTime`, `TM.thinkingTimes`, `PM1.thinkingTime`, `PM1.thinking`, `PM2.thinkingTime`, `PM2.thinking`}. The *cone* of `P1.hungryRate` is {`P1.hungryRate, PM1.hungryRate, PM1.getHungry`}.

Note that $r :=^* s$ (and thus $r \rightsquigarrow s$) implies $s \in \mathit{support}(r)$ and $r \in \mathit{cone}(s)$.

Definition 4.2 A change of the abstract analysis model *touches* a symbol s if

- it changes the value of $m_0(s)$, $\lambda(s)$, $w(s)$, $\pi(s)$ of $\mathit{value}(s)$,
- it creates or removes an assignment edge of the form $s := s'$,
- it creates or removes a collection membership edge of the form $s \mathrel{+}= s'$.

Example 4.8 Changing `P1.hungryRate` in Figure 4.4 on page 42 touched `P1.hungryRate`, because `value(P1.hungryRate)` was modified. Adding a new philosopher `P2` and its RGSPN module `PM2` in Figure 4.5 on page 42 touched `TM.thinkingTimes`, because the membership edge `TM.thinkingTimes += PM2.thinkingTime` was added.

When a change is to be propagated from the abstract analysis model to the conrete RGSPN, the cones of any touched symbols are inspected. The images of symbols and arcs in the touched cones (if any) are marked as dirty in the concrete RGSPN. Moreover, any symbol copied from the abstract analysis model starts as dirty. Dirtyness is tracked in the concretizer trace model.

After synchronizing any other change, such as the copying of concrete symbols and the removal of symbols with dangling traceability edges, the cleanup of dirty symbols and arcs proceeds. Symbols are cleaned up in the order of their dependencies. Lastly, dirty arcs are cleaned up by re-evaluating their inscriptions.

Definition 4.3 A symbol s_1' of the concrete analysis model *precedes* s_2', written as $s_1' \preceq s_2'$ if s_1' and s_2' are the images of s_1 and s_2 from the abstract analysis model, respectively, and $s_1 \in \mathit{support}(s_2)$.

Symbols s_1 and s_2 such that $s_1 \neq s_2$, but both $s_1 \in \mathit{support}(s_2)$ and $s_2 \in \mathit{support}(s_1)$ constitute a *circular dependency*, which will be identified as a source of inconsistency of the abstract analysis model in Section 4.5.3. If there are no cicrular dependencies $\preceq$ forms a partial order over the symbols of the concrete RGSPN.

Symbol cleanup processes dirty symbols in nondecreasing order according to $\preceq$. Each expression associated with a symbol is re-evaluated by inlining the values of mentioned variables, unfolding collection aggregations, as well as replacing mentioned places and parameter symbols by their images. These operations produce expressions equivalent to the originals according to the semantics described in Section 3.3.2 on page 29.

As the symbols are ordered by their dependencies upon traversal, there is no need to recursively process the value expressions of mentioned variable symbols. The variable symbol was already processed, because any mentioned variable is in the support of the expression; hence the image of the variable already has a re-evaluated value expression that was simplified as much as possible. Moreover, basic constant propagation arithmetic is performed if the result of evaluation is a Boolean or numerical constant.

After processing all the dirty symbols the inscriptions of dirty arcs are similarly re-evaluated. Because variables cannot depend on arc inscriptions the dependency order is not violated and variables can be inlined as above.

The concretizer transformation is aided by incremental model queries over the abstract RGSPN and the concretizer trace model. The reflexive transitive closure of the assignment relation $:=^*$, the reference resolution relation $\rightsquigarrow$ and supports of symbols in the abstract analysis model, as well as the dependency order $\preceq$ on the concrete analysis model is maintained by incremental transitive closure computation [Bergmann et al., 2012].

Remark 4.9 Petri net *slicing* [see e.g. Llorens et al., 2017] uses tools closely related to our dependency tracking mechanism to extracts parts of a Petri net influencing the satisfaction of a property. Slicing could be incorporated in the future into the concretization transformation to avoid copying symbols to the concrete analysis model that are irrelevant to the (stochastic) properties of interest. Alternatively, the external analysis tools can slice their input Petri nets to reduce computational burden.

4.5.3 Handling of inconsistencies

Inconsistencies in the abstract RGSPN analysis model refer to constraint violations that prevent the model from being concretized. While type checking for RGSPNs as defined in Definition 3.3 on page 25 can prevent some problems at design time, other violations arise from the mapping rules the structure of the dsource model of the view transformation.

View transformation specification developers should strive for creating concretizable analysis models from valid source models. However, a sequence of source model modifications may produce inconsistent analysis models event if the source model becomes valid at the end of the sequence. For example, if a source model object is replaced with another, there may an instant when an RGSPN reference has no assignments or there are multiple contradictory assignments, depending on whether the replacement first removes the old object or inserts the new one. Hence the concretization must be robust in face of inconsistent abstract RGSPNs and can resume transformation when the consistency is restored.

Inconsistency may appear in the abstract analysis model due to three main reasons:

- Reference assignments may be missing or contradictory. If a reference r has no assignments, or any assigned symbol s (i.e. $r :=^* s$) is itself a reference r cannot be concretized. Moreover, if there are multiple concrete symbols s_1, s_2 such that $s_1 \neq s_2$ but $r :=^* s_1$ and $r :=^* s_2$, there is no unambiguous concretization of r.
- There may be circular dependencies among symbols which prevent expression inlining. Possible circular dependencies include
  - simple circularly mentioned variables, $\mathit{value}(v_1) = v_2$, $\mathit{value}(v_2) = v_1$,
  - circular mentioning through a reference, $\mathit{value}(v_1) = r$, $r := v_2$, $\mathit{value}(v_2) = v_1$,
  - circular mentioning through a collection, $\mathit{value}(v) = \texttt{sum}(c)$, $c := v$

  and many variations thereof. We decided to disallow any form of circular dependency to avoid confusion. Only one of $s_1 \in \mathit{support}(s_2)$ and $s_2 \in \mathit{support}(s_1)$ may hold if $s_1 \neq s_2$. This always makes $\preceq$ a preorder on the concrete analysis model.
- Concretization may lead to disallowed parallel arcs. For example, if $p \xrightarrow{e_1} t$,$p \xrightarrow{e_2} r$ and $r := t$ hold in the abstract analysis model there would be two parallel arcs $p' \xrightarrow{e_1, e_2} t'$ in the concrete RGSPN. This is forbidden by Definition 3.2 on page 24.

Inconsistencies are tracked by the incremental model query engine during the concretization transformation in the same way as reference resolutions and the dependency order. The cleanup transformation is prevented from traversing symbols and arcs depending on inconsistent parts of the model even if they were marked dirty; thus the concretization is *delayed* until consistency is restored. Changes restoring consistency touch the delayed symbols, hence the cleanup transformation can handle them properly again.

The match sets model queries tracking inconsistencies can be retrieved from the transformation engine. Therefore if the analysis model is presumed to be consistent after a valid sequence of source model changes this fact can be verified. In addition, the error markers generated by inconsistencies can serve as a tool for transformation specification debugging.

Chapter 5

Application for design-space exploration

To achieve our goal of supporting design-space exploration with stochastic metrics, a formalism for the convenient modular construction of stochastic models was presented in the previous chapters along with a technique for transforming engineering (architectural) models into stochastic models. Now the application of these tools in design-space exploration (DSE) toolchains is discussed.

Users may configure the model transformation framework proposed in Chapter 4 by providing a transformation description, which determines the source DSL and the RGSPN fragments instantiated by transformation according to source model. Integrating the transformation engine into a DSE pipeline enables running any such transformation description to provide analysis models.

Queries associated with the analysis models are represented as variables in the RGSPNs derived by our transformation. The answers to the queries, which can calculated by external stochastic analysis tools, guide the DSE process as constraints to satisfy and goal functions to optimize. To carry out the computation the design space explorer must interface with the analysis tools. Serialization in ISO/IEC 15909-2:2011 PNML format was provided for interoperability with externals tools. However, the toochain integrator must provide means to run the external solver, to serialize stochastic queries in its input format and to read the analysis results.

As the literature pertaining the optimization of stochastic models was already reviewed in Section 1.1 on page 1, in this chapter we start by describing the tasks related to the integration of our analysis model transformation framework with a DSE toolchain. In addition, we describe the implementation of the framework along with the interfaces provided to users and our empirical evaluation of its scalability.

5.1 Integration with design-space exploration toolchains

Kang et al. [2010] have identified cornerstones of an effective DSE framework as 1. a suitable *representation* of the design space, 2. *analysis* capabilities to check discovered potential candidates against design constraints and 3. an *exploration method* for navigating interesting solutions. The approaches and representations used for DSE in the context of model-driven engineering were further classified by Vanherpen et al. [2014]. They have identified the following *DSE patterns* of exploration methods:

- The *Model Generation Pattern* synthesizes design candidates that satisfy a set of constraints, which are imposed based on the metamodel and in addition by the designer. During the exploration, design candidates are represented as solutions of a

constraint satisfaction problem. Tools based on this pattern include FORMULA [Kang et al., 2010] and Alloy Analyzer [Jackson, 2011].
- The *Model Adaptation Pattern* constructs an exploration representation, such as a string of genes in genetic algorithms [see e.g. Deb et al., 2002] from an initial model provided by the designer. Based on the guidance of a goal function further design candidates are devised in this intermediate form using (meta-)heuristic search. For example, the DSE tool PerOpteryx [Martens et al., 2010] uses this pattern.
- The *Model Transformation Pattern* directly represents the design candidates as an instance model. Model transformation rules that yield alternative models are scheduled using (meta-)heuristics to optimize a goal function. An example of this approach is VIATRA-DSE [Hegedűs et al., 2013; Abdeen et al., 2014].
- The *Exploration Chaining Pattern* adds multiple abstraction layers to DSE to prune the space of alternative solutions. At each abstraction layer, an exploration pattern is used to prune non-feasible solutions while selecting feasible solutions to be refined in the next layer. Domain knowledge is used to define abstraction layers. Costly evaluation of design candidates is usually deferred to the lower layers.

Vanherpen et al. [2014] also classified the representations employed by DSE patterns:

1. The starting point for exploration is expressed in a *model* formalism.
2. Constraints to be satisfied by the design alternatives and objective function to be optimized are captured by *constraint* and *goal* formalisms.
3. Design candidates are stored in an *exploration formalism* during the exploration. In the *Model Transformation Pattern*, this coincides with the *model* formalism.
4. The exploration formalism may be transformed into an *analysis* formalism to check feasibility with respect to the constraints.
5. A second transformation may target a *performance* formalism to check optimality with respect to the goal functions.
6. Execution traces yielding the design alternatives are stored in a *trace* formalism.
7. Finally, the solution is output in a *solution* formalism, which may coincide with either the model or the trace formalism.

The RGSPN formalism proposed in Chapter 3 may serve as both an *analysis* formalism when constraints are formulated in terms of stochastic analysis queries and as a *performance* formalism when the optimized goal function is a stochastic metric. Hence in DSE the transformation proposed in Chapter 4 should be employed as a means of transforming models in the *exploration* formalism to the *analysis* formalism. In more elaborate transformation chains, where a separate analysis formalism is employed and RGSPNs are only used as *performance* formalism, the *analysis* formalism may serve as a source instead. The traceability links produced by the transformation ensure that the results of the analysis can be interpreted as information about the satisfaction of constraints and the values of goal functions defined over the engineering formalisms.

The proposed approach based on incremental model transformation is especially suited for the *Model Transformation* DSE pattern, where the change-driver mapping to RGSPNs can be performed directly from the *model* formalism. Hence the same mapping is applicable for both stand-alone engineering models and in DSE, while the change-driver transformation may react to source model changes caused by the exploration rules. The transformation description, which specifies the creation of RGSPNs from the *model* formalism, can serve as the *constraint* or *goal* formalism, since it encodes which stochastic queries are constructed and evaluated for the instance models.

For application in the context of *Model Generation* and *Model Adaptation* the transformation description for our analysis transformation engine must be formulated with the *exploration* or an intermediate *analysis* formalism as the source. The resulting transformation will be only suitable for DSE and not for standalone model mapping. Moreover, in *Model Generation* change-driven incrementality may have diminished utility, because constraint solvers often generate solution in the *exploration* formalism from scratch instead of applying change operations. Adaptation of constraint solvers to the incremental setting is challenging due to scalability issues [Semeráth et al., 2016b], especially in the case of graph generation with complex structural constraints [Semeráth et al., 2016a].

Remark 5.1 A recent approach in *Model Generation* combines partial interpretations from mathematical logic and techniques from Boolean satisfiability (SAT) solvers to formulate the problem in terms of *Model Adaptation* [Varró et al., 2017]. The *exploration* formalism in this approach is a partial interpretation of the original *model* formalism. It is possible to evaluate model queries on the partial interpretation by constraint rewriting of queries over the original *model* formalism [Semeráth and Varró, 2017]. Therefore our transformation engine could be adapted to construct RGSPNs from the partially interpreted *exploration* formalism based on a transformation description developed for the original *model* language by rewriting ("lifting") the involved model queries, which would enable incremental execution in all three major DSE paradigms.

Remark 5.2 Retaining parameter symbols in RGSPNs for use with external solvers provides an opportunity for *Exploration Chaining*. The elements of the CTMC parameter vector $\theta \in \mathbb{R}^{|Par|}$ correspond to primitive attributes of engineering model elements after transformation. Hence the vector is a concise *exploration* representation of a design alternative once its structure is fixed and only attributes need to be filled in. As a nested exploration method, algorithms based on sensitivity analysis and numerical optimization [T. Molnár, 2017] or parametric abstractions [Quatmann et al., 2016] may be employed so that the higher-level exploration method can be reserved to propose candidate structures for the design.

5.1.1 Model transformation based design-space explorers

Incremental transformation to RGSPN analysis models was considered above in the contexts of various DSE patterns. We now describe the operation of our transformation engine with the *Model Transformation Pattern*, which is perhaps the most amenable to change-driven synchronization of analysis models.

The Formalism Transformation Graph and Process Model (FTG+PM) notation was proposed by Lúcio et al. [2012] as a guide to carry out model transformations in multi-paradigm modeling. An extended version of the *Model Transformation Pattern* FTG+PM of Vanherpen et al. [2014, Figure 4] is shown in Figure 5.1, which illustrates model transformation based DSE with incrementally synchronized RGSPN analysis models. The Formalism Transformation Graph (FTG) on the left shows the modeling languages as rectangles and the involved model transformations as circles. Arrows indicate the direction of transformations, such that bidirectional arrows correspond to in-place model modification. The Process Model (PM) contains the transformation activities, which are displayed as rounded rectangles, their control flow (solid arrows) and data flows (dashed arrows). Languages and transformations provided by our framework are emphasized in bold.

Model transformation based DSE works directly on the *model* formalism. Heuristics or meta-heuristics provided by the DSE toolchain in the *Create Candidate Solutions* activity apply model transformations according to some goal functions. In order to support change-driven synchronization of the RGSPN view for analysis, the transformations should be in-place so that change notifications can be propagated. In the PM, the in-place model modification is indicated by data flows of pieces of input and output data of the same type.

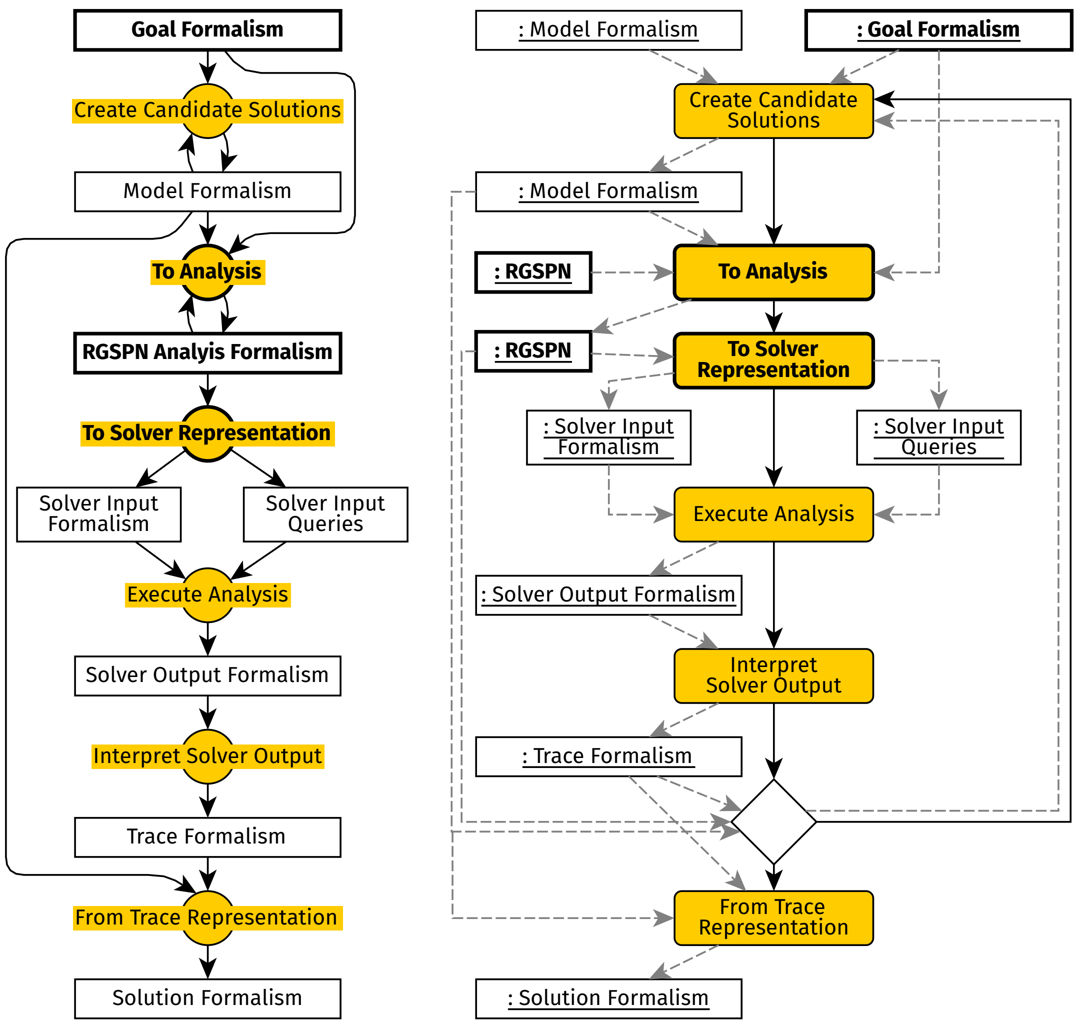


Figure 5.1 FTG+PM of the *Model Transformation* DSE pattern with RGSPN-based analysis. The components in **bold** were implemented in our work, while the rest of the components should be supplied by the DSE framework and stochastic analysis tool.

The *To Analysis* activity derives the RGSPN analysis model by interpreting the transformation description in the *goal* formalism with our transformation engine, which was described in Chapter 4. The analysis models are derived incrementally by modifying the RGSPNs in place according to changes in the candidate solution. The resulting RGSPN contains both the stochastic analysis model and the variable symbols that correspond to the goal functions.

The queries pertaining goal functions and queries can be answered on the analysis model by executing the stochastic analysis in an external tool. The RGSPN model is transferred to the external tool by serializing it in a standardized interchange format, which is the *Solver Input Formalism*. Moreover, the queries themselves must be serialized in the appropriate *Solver Input Query* formalism. The *To Solver Representation* activity performs this task. We provide an implementation of this activity as part of our framework that targets them ISO/IEC 15909-2:2011 PNML format as the *Solver Input Formalsm* while using extensions defined by the PetriDotNet tool [Vörös et al., 2017b] to convey timings of Petri net transitions and stochastic queries.

After the *Execute Analysis* activity, which usually involves calling an external program, the answers to the stochastic queries are obtained in the *Solver Output Formalism*. This representation must be parsed in the *Interpret Solver Output* activity so that the values of the goal functions are available to the DSE toolchain. The DSE toolchain incorporates the results of the analysis into the *trace* representation; thus the candidate designs in the solution store can be compared according to their fitness.

The *Model Transformation* DSE pattern is iterative. The traces, which are enriched with the values of the goal functions, are incorporated by the *Create Candidate Solutions* heuristics to produce new design candidates. The in-place modification of the candidate design and the RGSPN is signified in the PM by the data flow going into the decision node at the end of the loop and the data flow back to the start of the loop.

Finally, if required, the optimal solution or a set of solutions can be transformed from the trace representation to the *solution* formalism by the *From Trace Representation* activity.

5.1.2 Stochastic analysis tools

To answer the queries posed as variable symbols in the RGSPN analysis models external stochastic analysis tools must be invoked. As discussed in the previous section, this requires a modification of the DSE toolchain to produce input for the external tool, invoke it and parse its output. However, additional support for this workflow must be incorporated into the analysis tool, too.

Firstly, the analysis tool needs to have an interface for unattended execution. Various ways to provide this interface include command-line applications and web services. For example, the PetriDotNet analysis tool contains a separate binary executable for running stochastic analyses in the command line.△ While initiatives such as the Model Checking Contesti (MCC) [Kordon et al., 2017] and the Petri Nets Repository [Hillah and Kordon, 2017] strive for common interfaces for Petri net analysis tools, to our best knowledge, no generic interface is widely supported. Hence even though models serialized in the PNML format are portable between solvers, each of them must be called in a specific way.

Secondly, any parameters required by the analysis in addition to the stochastic model and the queries must be supplied automatically. For stochastic Petri net analysis, these parameters include the ordering of state variable in symbolic analysis methods when the model is converted into a CTMC and the choice of numeric algorithm to solve the arising systems of linear or differential equations.

Variable ordering

Symbolic computations methods such as *saturation* [Ciardo et al., 2001, 2012] are often used in the state-space exploration of Petri nets, which is required for model checking logical properties and the construction of CTMCs from stochastic Petri nets [Miner, 2004]. Symbolic algorithms represent the reachable state space of the formal model as a *decision diagram*, such as a multi-valued decision diagram (MDD) [Kam et al., 1998]. The decision diagram is a directed acyclic graph where each node belongs to a given *level*. Each state variable of the model, which may be the marking of single place or a collection of places in Petri nets, is assigned to a different level. The assignment is referred to as the variable *ordering*. Outgoing edges from nodes are labeled with the possible values of the state variable, such that each path in the graph is a reachable state, i.e. a reachable Petri net marking.

The transitions in the formal model induce a next-state relation over the states in the diagram. The reachable state space can be determined by fixed-point iteration of the

△ This tool, similarly to the rest of PetriDotNet 1.5b2 is available from https://inf.mit.bme.hu/en/research/tools/petridotnet upon request. More information can be found in the user manual by Vörös et al. [2017a].

next-state relation. By selecting appropriate representation of the next-state relation, even complex models, such as stochastic Petri nets with immediate transition priorities can be handled [Miner, 2006; Marussy et al., 2017]. However, the variable ordering has dramatic effects on the run time of the fixed point computation [Amparore et al., 2017]. Stochastic analysis is further made difficult due to the decompositions employed in the numerical solution of CTMCs often requiring variable assignments that differ from those suitable for symbolic analysis [Marussy et al., 2016a].

As the structure of the derived Petri net model may constantly change during exploration, the variable ordering cannot be provided to the solver manually. Either the solver itself or some other component of the DSE pipeline must generate an acceptable variable ordering. Based on the abstraction level at which the generation is done, we suggest three possible solutions as follows:

- The stochastic analysis tool itself may generate a variable order by some heuristic, such as those surveyed by Amparore et al. [2017].
- The RGSPN transformation engine may communicate the groupings of places induced by the instantiation of Petri net modules to the analysis tool. The nested-unit Petri net (NUPN) format was proposed by Garavel [2015] to encode such grouping and was employed in the 2017 edition of the MCC to aid variable ordering heuristics of the participating tools [Kordon et al., 2017].△
- It would be also possible to extend the transformation specifications such that our transformation engine could generate variable orderings along with RGSPNs.

Numerical algorithm selection

Another setting which may dramatically impact the solution time and accuracy of stochastic models is the choice of the numerical algorithms.

In stead-state and mean time to state partition analysis, solving the CTMC reduces to a system of linear equations, where the number of variables and equations equal to the size of the reachable state space of the model. The matrix of this system of linear equations is the *infinitesimal generator matrix* of the CTMC, which is sparse and often amenable to decomposed storage [Buchholz, 1999a].

Due to the size of the systems direct solution methods are infeasible and iterative numerical methods are employed instead. However, the choice the iterative linear equations solver method and its parameters determines the run time and convergence of the solution; moreover, no numerical method was found to be suitable for all classes of models [Buchholz, 1999b; Marussy et al., 2016b; Buchholz et al., 2017].

In transient analysis, transitions with orders of magnitude timing difference cause *stiffness* the system of differential equations associated with the CTMC. Stiff Markov chains may be handled by numerical differential equation solver algorithms especially tailored to such situations [Reibman et al., 1989] or by adaptive variants of the *uniformization* algorithm [Morsel and Sanders, 1997; Dijk et al., 2017].

To our best knowledge, no method was proposed in the literature to automatically select a suitable numeric algorithm for stochastic analyses. An analysis tool may offer a default selection; however, for ill-conditioned problems, the user should override it before starting design-space exploration. Alternatively, a portfolio of algorithms may be specified that are tried sequentially or in parallel until one of them converges successfully.

△ The specification for embedding NUPN data in PNML files is available at https://mcc.lip6.fr/nupn.php.

Remark 5.3 Deeper, change-driven integration between external analysis tools and model transformation toolchains has been suggested recently by V. Molnár et al. [2016] and Meyers [2016, Section 2.8] inspired by incremental approaches in the evaluation of expensive model queries [Ujhelyi et al., 2015]. Such integration might allow solvers to receive model changes and compute the analysis result incrementally by reusing parts of the previous solution.

Since our RGSPN transformation is engine is fully change-driven it is able to translate engineering model changes to analysis model changes, which could be sent directly to the solver. Moreover, in the numeric analysis of CTMCS it is sometimes possible to reuse the previous solution vector as an initial approximation. However, no existing analysis tool is in our knowledge that is able to take advantage of model change information; therefore extending change-driven execution throughout the analysis remains in the scope of future work.

5.2 Software implementation

A software tool for the development of transformation specifications and their execution was implemented as a plug-in for the Eclipse Oxygen.1 Integrated Development Environment[2] (IDE). The plug-in is based on open source technologies from the Eclipse Modeling Project: the *Eclipse Modeling Foundation* (EMF) [Steinberg et al., 2009], the *XText*[3] framework for language engineering and *VIATRA* scalable reactive model queries and transformations.

The software consists of two major components. Both RGSPN modules and model transformations from arbitrary EMF-based DSLS to RGSPNS can be developed in the transformation specification environment. The transformation can be executed either inside the IDE for testing or inside a DSE program after Java code generation. Together with a runtime library implementing the transformation engine, the generated Java code provides incremental transformation to stochastic Petri nets from DSLS defined with *Ecore* metamodels, the metamodeling core of EMF.

5.2.1 Specification environment

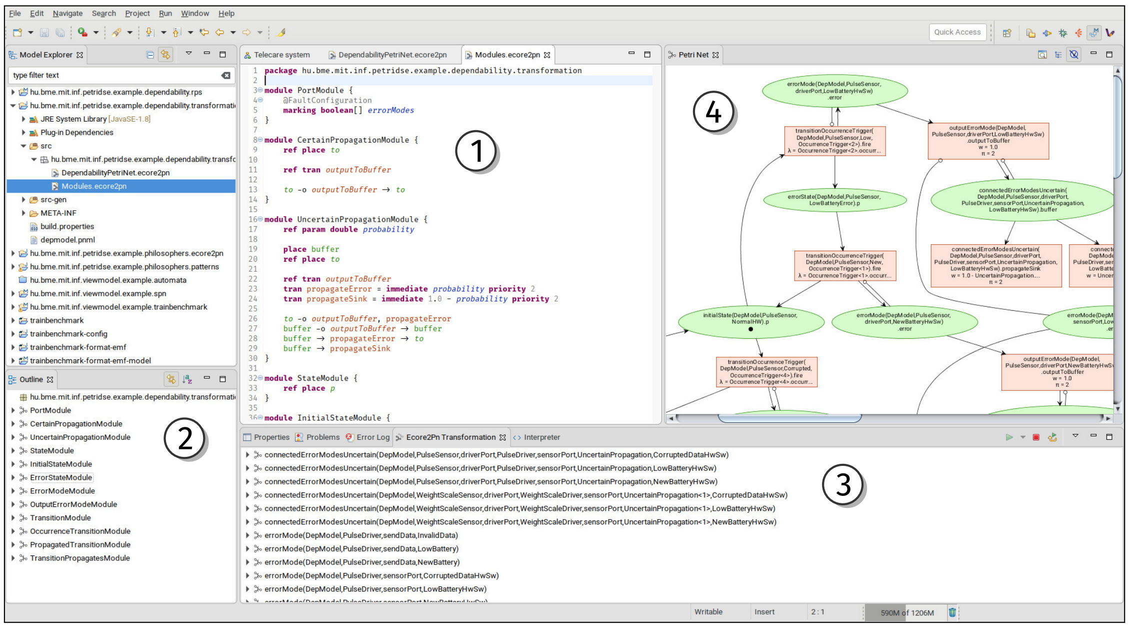
A specification development environment named *Ecore2Pn* (Ecore to Petri net transformation) was implemented for RGSPN-based transformation development.[△] A screenshot of the tool is shown in Figure 5.2.

The plug-in suite contains concrete textual syntaxes for RGSPN modules and transformation descriptions based on the Xtext language engineering framework. Integrated devlopment environment (IDE) features, such as semantics-aware syntax highlighting, content assist for code completion, jump-to-definition and outline view are available. The definition of model queries as preconditions for the RGSPN transformation rules is possible with the VIATRA Query EMF query definition editor [Ujhelyi et al., 2015].

Similarly to the VIATRA query editor, integration is offered with the modeling services of the Eclipse IDE to try out and debug transformation descriptions. The transformation may be executed live on models loaded as XMI [Object Management Group, 2015] files or with graphical concrete syntax as Sirius[4] diagrams. Modification of the model triggers change-based synchronization of the RGSPN. A listing of instantiated RGSPN modules symbols along with traceability links is displayed in the *Ecore2Pn Transformation* view. Moreover, a graphical view of the RGSPN is available in the *Petri Net* view.

[△] The specification development environment was implemented by the author during his summer internship at *ThyssenKrupp Presta Hungary Kft.*

[2] http://www.eclipse.org/downloads/packages/release/Oxygen/1 [3] https://www.eclipse.org/Xtext/
[4] http://www.eclipse.org/sirius/



```
type filter text
hu.bme.mit.inf.petridse.example.dependability.rps
hu.bme.mit.inf.petridse.example.dependability.transformati
    JRE System Library [JavaSE-1.8]
    Plug-in Dependencies
    src
        hu.bme.mit.inf.petridse.example.dependability.transfc
            DependabilityPetriNet.ecore2pn
            Modules.ecore2pn
    src-gen
    META-INF
    build.properties
    depmodel.pnml
hu.bme.mit.inf.petridse.example.philosophers.ecore2pn
hu.bme.mit.inf.petridse.example.philosophers.patterns
hu.bme.mit.inf.viewmodel.example.automata
hu.bme.mit.inf.viewmodel.example.spn
hu.bme.mit.inf.viewmodel.example.trainbenchmark
trainbenchmark
trainbenchmark-config
trainbenchmark-format-emf
trainbenchmark-format-emf-model
```

```
hu.bme.mit.inf.petridse.example.dependability.transformati
PortModule
CertainPropagationModule
UncertainPropagationModule
StateModule
InitialStateModule
ErrorStateModule
ErrorModeModule
OutputErrorModeModule
TransitionModule
OccurrenceTransitionModule
PropagatedTransitionModule
TransitionPropagatesModule
```

```
package hu.bme.mit.inf.petridse.example.dependability.transformation

module PortModule {
    @FaultConfiguration
    marking boolean[] errorModes
}

module CertainPropagationModule {
    ref place to

    ref tran outputToBuffer

    to -o outputToBuffer → to
}

module UncertainPropagationModule {
    ref param double probability

    place buffer
    ref place to

    ref tran outputToBuffer
    tran propagateError = immediate probability priority 2
    tran propagateSink = immediate 1.0 - probability priority 2

    to -o outputToBuffer, propagateError
    buffer -o outputToBuffer → buffer
    buffer → propagateError → to
    buffer → propagateSink
}

module StateModule {
    ref place p
}

module InitialStateModule {
```

```
connectedErrorModesUncertain(DepModel,PulseSensor,driverPort,PulseDriver,sensorPort,UncertainPropagation,CorruptedDataHwSw)
connectedErrorModesUncertain(DepModel,PulseSensor,driverPort,PulseDriver,sensorPort,UncertainPropagation,LowBatteryHwSw)
connectedErrorModesUncertain(DepModel,PulseSensor,driverPort,PulseDriver,sensorPort,UncertainPropagation,NewBatteryHwSw)
connectedErrorModesUncertain(DepModel,WeightScaleSensor,driverPort,WeightScaleDriver,sensorPort,UncertainPropagation<1>,CorruptedDataHwSw)
connectedErrorModesUncertain(DepModel,WeightScaleSensor,driverPort,WeightScaleDriver,sensorPort,UncertainPropagation<1>,LowBatteryHwSw)
connectedErrorModesUncertain(DepModel,WeightScaleSensor,driverPort,WeightScaleDriver,sensorPort,UncertainPropagation<1>,NewBatteryHwSw)
errorMode(DepModel,PulseDriver,sendData,InvalidData)
errorMode(DepModel,PulseDriver,sendData,LowBattery)
errorMode(DepModel,PulseDriver,sendData,NewBattery)
errorMode(DepModel,PulseDriver,sensorPort,CorruptedDataHwSw)
errorMode(DepModel,PulseDriver,sensorPort,LowBatteryHwSw)
```

Figure 5.2 Sreenshot of the transformation specification environment. The showcased features include ① the transformation description editor with syntax highlighting, ② the outline view for transformations, ③ the *Ecore2Pn Transformation* execution and traceability viewer and ④ the RGSPN graph *Petri Net* visualizer.

Model export

In addition to transformation development and execution, *Ecore2Pn* offers export facilities for model interchange. These features are part of the transformation engine runtime; therefore they are also available for developers who wish to integrate RGSPNs into DSE toolchains. *Ecore2Pn* merely provides a convenient user interface for exporting single models.

Serialization in ISO/IEC 15909-2:2011 PNML format allows model interchange with external analysis tools. The exporter also supports the *state reward configuration* and *fault configuration* facilities of PetriDotNet [Vörös et al., 2017a, Section 4.2] for Markovian steady-state, transient and mean time to state partition analysis. Symbols marked with the `@RewardConfiguration` and `@FaultConfiguration` annotations in the RGSPN textual editor get translated into reward and fault configurations, respectively, and are available for analysis once the exported PNML is opened with PetriDotNet.

An additional export facility is available targeting the `dot` format compatible with the *Graphviz*[5] graph visualization software. The `dot` utility provides automatic layouting and drawing for directed graphs, which allows visual inspection of RGSPN models. This exporter is also employed along with a Java port[6] of Graphviz by the *Petri Net* view of the specification environment to display the results of the currently running RGSPN transformation.

Code generation

Code generation is used throughout the specification development environment to ensure that the transformation specification can be ran in a wide variety of environments, such as within and Eclipse plugin-in or as a standalone Java application.

[5] https://www.graphviz.org/ [6] https://github.com/nidi3/graphviz-java

The RGSPN modules and the transformation description defined by the user are turned into Java code for compilation. In this way the transformation description can be passed to the execution engine by just instantiating a class, just like how VIATRA Query generates pattern-specific matcher code from graph patterns for type-safe consumption [Ujhelyi et al., 2015, Section 2.3]. Moreover, as model queries, RGSPN modules and transformations become Java classes, their dependencies can be managed by the Java CLASSPATH mechanism. For example, and RGSPN module can be seamlessly upgraded by replacing its containing archive on the CLASSPATH without breaking compatibility with transformation definitions in other archives that refer to the module.

Additional helper code is generated for derived features defined in transformations. The helper enables simulates derived features in code written in the *Xtend*[7] programming language. The *extension methods* feature allows traversal of the traceability relations created by the RGSPN transformation engine to obtain derived feature symbols as if they were true properties of the domain model elements.

5.2.2 Transformation execution

The transformation execution engine is a Java library than can be used either as an Eclipse plug-in or in standalone applications. The transformation engine can be instantiated with an existing VIATRA Incremental Query engine over an *EMF scope* which contains the intended source model. The other argument required for the transformation is the generated transformation specification object, which refers to the VIATRA model queries and RGSPN modules involved in the transformation. Once instantiated, the engine executes in an incremental fashion and reacts to changes in the source model.

The transformation rules are scheduled and fired by the VIATRA Event-driven Virtual Machine (EVM) [Bergmann et al., 2015]. Hence the transformation engine can be easily integrated with other EMF-related technologies, such as VIATRA Query [Ujhelyi et al., 2015] and VIATRA-DSE [Abdeen et al., 2014].

It is possible to only execute the view transformation, which yields an abstract RGSPN with collections and references, or both the view and the concretizer transformation, which also yields a concrete RGSPN that can be exported to external analysis tools. The engine can be customized by overriding *Google Guice*[8] dependency injections.

Traceability relations can be traversed either by explicitly reading them, or by the derived features helper classes generated for Xtend programming. In addition, extra *annotations* specified in the RGSPN modules and the transformation description are also propagate through the transformation chain, which may influence the behavior of RGSPN exporters.

5.3 Evaluation of incremental transformations

We carried out preliminary scalability evaluation of our transformation runtime in order to study the overhead the imposed on transformation imposes on design-space exploration. Both *batch execution*—where the transformation engine is initially instantiated and the intermediate and target RGSPN models are materialized according to the engineering models—and *incremental execution*—where each source model change is immediately translated into intermediate and target model changes—were studied. More specifically, we carried out the evaluation in the *dining philosophers* domain to address the following three research questions:

[7] https://www.eclipse.org/xtend/ [8] https://github.com/google/guice

Table 5.1 Source model, abstract net and concrete net sizes for the philosophers models.

N	#Source	#Abstract net	#Concrete net
8	9	644	532
16	17	1268	1060
32	33	2516	2116
64	65	5012	4228
128	129	10 004	8452

RQ1 How does the initial batch transformation from the engineering DSL to the formal stochastic model (GSPN) scale with respect to size of the input model?

RQ2 How does the incremental transformation scale with respect to the size and the change operations of the input model?

RQ3 What is the overhead associated with the serialization of models to the ISO/IEC PNML interchange format?

Answering these questions may help identifying strengths and weaknesses of the proposed approach to the stochastic evaluation of engineering models. Moreover, the answers to **RQ1** and **RQ2** aid in determining whether incremental or batch model transformation should be used according to the usual size of source changes. This choice arises when there is no need to construct the target model change as a sequence of operations for each source change; therefore incremental execution is not necessitated and the system integrator can chose between either execution schemes. Lastly, the answer to **RQ3** tells whether the overhead of serialization into a portable format is acceptable or more direct integration and communication with the external solver is needed.

5.3.1 Measurement setup

Measurements were performed on instances of the *dining philosophers* domain model, which was used throughout this work as a running example. The number of philosophers and thus the size of the source model was set to $N = 8$, 16, 32, 64 and 128. Table 5.1 shows the sizes of the source models, as well as the sizes of the derived intermediate abstract RGSPNs and target concrete RGSPNs, including any symbol, edge and expression objects.

To evaluate incremental execution, various *change operations* were defined as follows:

- **Swap** rotates the seating order two philosophers adjacent around the table. This change only modifies references in the source model; hence is simulates a DSE rule with no object creation and deletion.
- **New** creates a new philosopher and inserts it between two existing philosophers.
- **Delete** removes a philosopher from the table and deletes it from the model.
- **FixedMix** simulates a compound model change of fixed size by a randomly ordered mixture of 8 **swap**, 4 **new** and 4 **delete** operations.
- **ScaledMix** simulates a compound model change of model-dependent size by a randomly ordered mixture of N **swap**, $\frac{N}{2}$ **new** and $\frac{N}{2}$ **delete** operations.

The compound model change **scaledMix** was devised such that half of the philosophers is replaced around the table, while **fixedMix** is obtained from **scaledMix** by setting $N = 8$

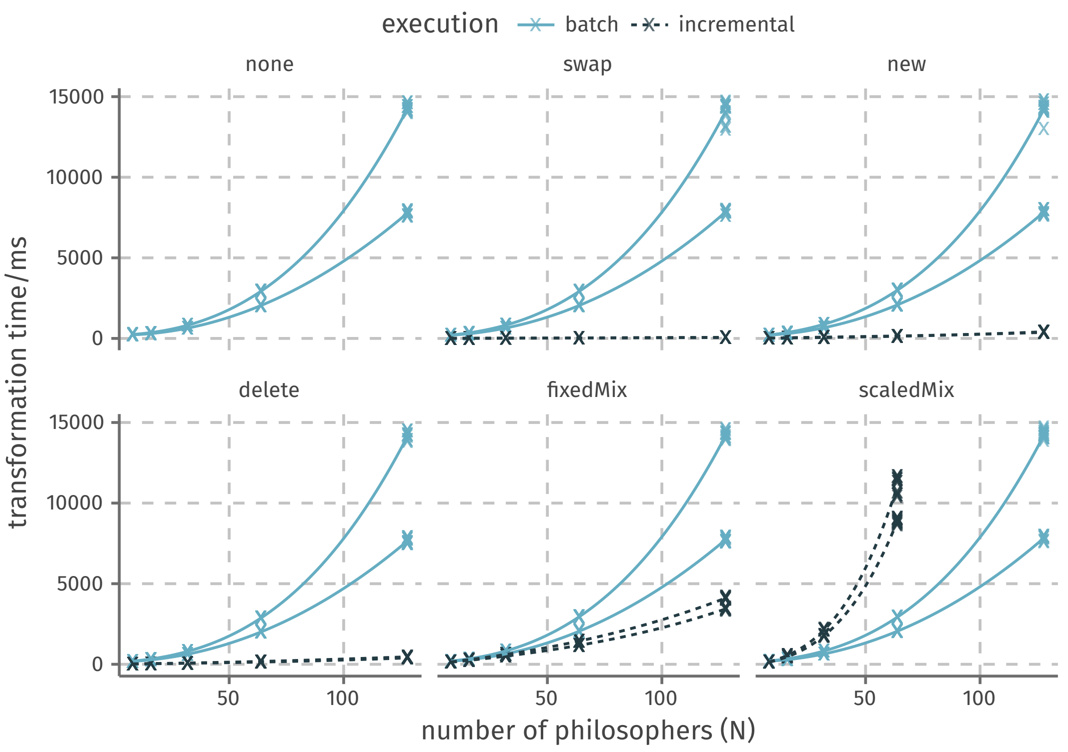


Figure 5.3 Execution times of transformations.

to the size of the smallest input model. The model elements involved in the simple and compound model changes were randomized similarly to the order of simple operations without compound ones. However, the random seed was fixed for each measurements, i.e. the model changes are always deterministic given the input model size.

Measurements of a given execution scheme and change type comprise a *scenario*. Batch transformation of the initial models was studied in an additional scenario without any model change. Every scenario was executed for each model size $N \in \{8, 16, 32, 64, 128\}$ multiple times. A single execution of the transformation is an *iteration*. After 10 warm-up iterations, the run times of 30 iterations were measured for each scenario and model size.

To avoid measuring the latency of the hard disk, the target GSPN models were serialized in the PNML format to an in-memory output stream. However, for externals tools that can only read Petri nets from a disk, an in-memory file system may be needed instead.

Measurements were performed on a workstation with two dual-core Intel Xeon 5160 3.00 GHz processors and 16 GB memory. The heap size of the Java 1.8u144 virtual machine was limited to 8 GB with a 30 s wall clock time limit for each iteration.

5.3.2 Results

The execution time of the transformations on the various model sizes and change operations is shown in the scatter plot in Figure 5.3. It is apparent that the distribution of run times is extremely bimodal, especially for larger source models.

Therefore instead of fitting a single curve for each scenario, data points were split into two clusters for each scenario and model size. First the threshold *thresh* = $\frac{max - min}{2}$ was determined, where *max* and *min* were the smallest and largest execution times, respectively. Due to the heavy bimodality, no data points were adjacent to this threshold. The upper and

Table 5.2 Minimum and maximum execution times of transformations/ms.

		Incremental				
N	Batch	Swap	New	Delete	FixedMix	ScaledMix
8	209 – 294	6 ↕ 9	15 ↕ 26	17 – 33	126 ↕ 166	124 ↕ 163
16	281 ↕ 344	8 ↕ 15	23 ↕ 41	27 ↕ 45	224 ↕ 291	456 ↕ 575
32	631 ↕ 852	13 ↕ 18	46 ↕ 91	51 ↕ 86	505 ↕ 631	1714 ↕ 2221
64	2006 ↕ 2975	26 ↕ 34	119 – 164	129 ↕ 181	1148 ↕ 1473	8644 ↕ 11 681
128	7568 ↕ 14 659	52 ↕ 68	357 ↕ 427	383 ↕ 478	3342 ↕ 4211	Timed out

lower clusters were then formed by data points above and below *thresh*. The upper and lower curves of degree up 3, which are shown in Figure 5.3, were fit to data points from the upper and lower clusters of each scenario. It is apparent that execution times of the batch scenarios in both the upper are lower clusters scale superlinearly, and the same phenomemon also occurs with incremental view synchronization of mixes of change operations. There was no correlation between the iteration numbers and the clusters, i.e. the bimodality was not found to be a warm-up transient artifact.

The minimum and maximum execution times of each scenario and model size, which are representative of the execution times in two clusters, are shown in Table 5.2. Because the considered model changes did not affect the run times of batch transformations, we only report the run time of the batch transformation of the initial model. The symbol ↕ indicates significant ($p < 0.05$) bimodality of the execution time distributions according to Hartigan's dip test [Maechler, 2016], while – denotes unimodal distributions.

In order to study the source of bimodality in the execution times, a further experiment was conducted. The batch transformations, which had the most striking bimodality, was executed with further instrumentation on the source model containing $N = 128$ philosophers. Four stages of the transformation were distinguished:

1. The *view query* phase prepares the model queries that are the preconditions of the view transformation. In VIATRA Query, this corresponds to query optimization, as well as the traversal of the source model to populate the various base relations and caches for incremental query evaluation.
2. The *view transformation* phase fires the transformation rules on the VIATRA Event-driven Virtual Machine (EVM) to construct the abstract RGSPN model with references.
3. The *concretizer query* phase traverses the abstract net to prepare the precondition queries of the concretizer transformation.
4. The *concretizer transformation* phase is ran on the EVM to resolve references and inline expression in the abstract net to construct the concrete RGSPN target model.

In ordinary transformation execution, the query phases are ran simultaneously to avoid spurious model traversal. Moreover, the transformation phases share an EVM execution schema that provides sequential execution by prioritized firing of transformation rules. However, in our experiment, we separated the phases to observe their run times individually.

The histogram of the transformation phases with 30 iterations is shown in Figure 5.4. The concretizer transformation phase, which is running an order of magnitude slower than other phases, is revealed as the source of the heavy bimodality.

Lastly, the time taken by serialization of the target models in ISO/IEC PNML format to an in-memory output stream is show in Table 5.3 and the accompanying figure. Both the

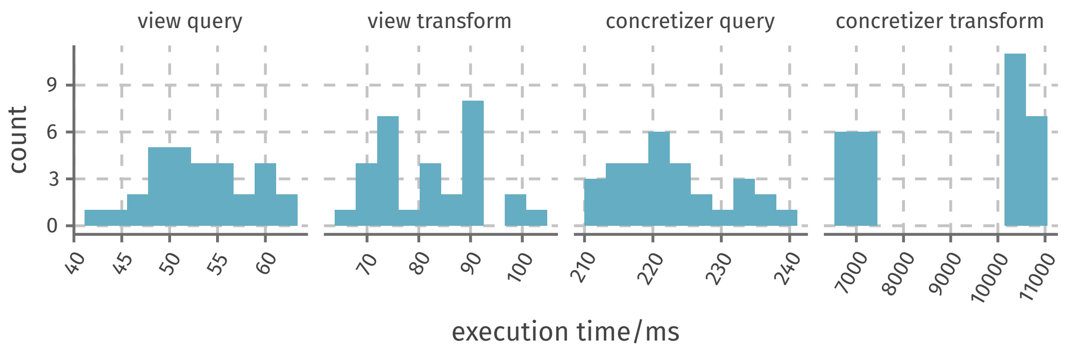
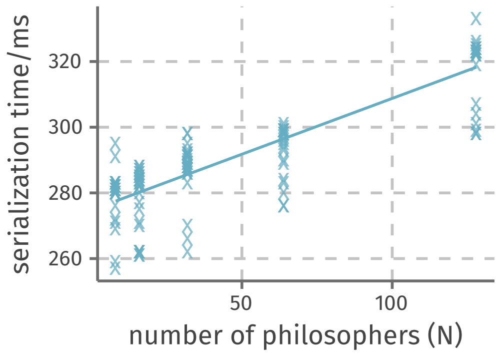


Figure 5.4 Execution times of batch transformation phases with $N = 128$ philosophers.

Table 5.3 Execution times of PNML serializations.



N	Time/ms	PNML size/bytes
8	257 – 295	50 700
16	261 ↕ 288	100 441
32	262 – 298	200 572
64	276 – 301	400 736
128	298 – 333	802 454

serialization time and the size of the resulting PNML descriptions scale linearly with the model size. Significant bimodality was detected by the dip test on in the case of $N = 16$ with $p = 0.004$. However, it is possible that the latter observation is only due to randomness.

5.3.3 Observations

The research questions **RQ1–3** may be answered based on the presented measurement results as follows:

RQ1 Batch transformations scaled superlinearly in the size of the input model. Transformation of the largest studied source model, which had 129 elements, took up to 15 s to produce a 8452-element output model along with traceability information which affords incremental synchronization of the target RGSPN according to future source model changes.

The run time exhibited significant bimodality, apparent to both visual examination an Hartigan's dip test of bimodality. In the most extreme case of $N = 128$ philosophers, iterations in the upper cluster of run times took nearly twice as long a those in the lower cluster, while for smaller input models, the difference was up to 50%.

RQ2 Incremental synchronization of the **swap** change operation was found to take linear time as the function of the source model time. Therefore the synchronization time depends on not only the changes to be synchronized but also on the size of the input model. Synchronization time for **create** and **delete** changes was found to be superlinear similarly to the batch transformation. This indicates the creation and removal of objects has larger overhead than the modification of references in the source model and the RGSPN.

Synchronization time was below that of batch transformation in the **fixedMix** compound change operation. However, for the change operation **scaledMix** of model-dependent size, batch transformation was found to be faster that incremental synchronization in all cases except $N = 8$. Therefore we can conclude that if change operations affect large portions of the input model batch transformation may be more economical than incremental synchronization; although for smaller input changes, synchronization won by a margin of at least 14%, the smallest difference being achieved on the **fixedMix** change with $N = 16$.

RQ3 The PNML serialization routine, which traverses the concrete RGSPN model to produce its PNML equivalent, scaled linearly in the size of the input model. However, the cause of this phenomemon is probably that the size of concrete GSPN itself is only a constant multiple of the input model size. The size of the generated PNML was also a multiple of the input model size. In all measured cases PNML serialization took no more than 1/3 of a second, much less than the time taken by analysis tools to analyze stochastic Petri net models similar to the ones considered. Therefore PNML serialization is not a significant overhead compared to stochastic analysis. It is also generally smaller than the time taken by batch transformation.

Bimodality of transformation run time distributions was found to be caused by the execution of the RGSPN concretizer transformation on the VIATRA Event-driven Virtual Machine. We hypothesize that the large differences is execution time are caused by the nondeterministic scheduling in EVM.

While conflicting transformation rules of differing priorities are fired in the order of their priorities, the ordering between rules of the same priority are not defined. The firing of a low-priority rule may activate a higher priority one. In the implementation of our concretizer transformation, the work performed by some high-priority rules may be occasionally undone by a low-priority rule when RGSPN references are resolved and expressions are inlined due to the dependency tracking required for expression inlining. Thus if low-priority rules are fired in an unsuitable order, some work must be redone by high-priority rules after the correct dependencies are taken into account. Although taking dependencies between RGSPN symbols and expressions at the level of EVM conflict resolution may alleviate this issue, performing such tracking efficiently remains in the scope of future work.

Due to the hashing employed by the conflict resolver, the firing order of equal priority rules is determined at runtime by `hashCode` of the rule activation objects, which is not overridden from its default implementation. In the Java runtime environment, the default `hashCode` is connected with the allocation of objects and forcing it to be deterministic for the sake of consistent measurements is difficult. Hence the apparently random switching between fast and slow execution of the concretizer transformation.

5.3.4 Threats to validity

An internal threat to validity was the possibility of an incorrect implementation of the transformation engine or the incorrect description of the transformation from the dining philosophers domain model to Petri nets. To ensure correctness the transformation outputs were manually inspected for the small source models for consistency with the source models and the transformation description.

Moreover, interferences may have occurred in the measurement environment. To reduce interferences, the measurements were ran on a physical machine on which no other task was executed at the time. Each scenario and input model was measured 30 times after 10 warm-up iterations to reduce random noise and the interferences caused by ongoing just-in-time compilation. Garbage collection within the runtime environment was also controlled manually to ensure that subsequent iterations did not interfere.

Despite these attempts, run time distributions were found to be bimodal having two clusters with small variance instead of a single cluster with small variance. We conducted further measurements to break down the transformation into phases and hypothesize that this phenomenon is intrinsic to the current implementation of the transformation instead of being caused by interferences.

As we conducted our experiments on in single domain with a single transformation description, several external threats to validity impede generalization. Firstly, further studies are needed to observe the behavior of the transformation on different domain models and transformation descriptions. Secondly, as the size of the target RGSPN models was a constant multiple of the size of the source models, behaviors depending on the sizes of either of these models could not be distinguished from each other.

Chapter 6

Conclusions and future work

Acknowledgements

References

Abdeen, Hani, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedűs, and Ákos Horváth [2014].
Multi-objective optimization in rule-based design space exploration.
In: *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.* ACM, pp. 289–300.
DOI: 10.1145/2642937.2643005.

Aldini, Alessandro and Marco Bernardo [2007].
Mixing logics and rewards for the component-oriented specification of performance measures.
J. Theor. Comput. Sci., **382** (1): 3–23. DOI: 10.1016/j.tcs.2007.05.006.

Aldini, Alessandro, Marco Bernardo, and Jeremy Sproston [2011].
Performability Measure Specification: Combining CSRL and MSL. In: *FMICS 2011*. LNCS 6959.
Springer, pp. 165–179. DOI: 10.1007/978-3-642-24431-5_13.

Amparore, Elvio Gilberto, Susanna Donatelli, Marco Beccuti, Giulio Garbi, and Andrew S. Miner [2017]. Decision Diagrams for Petri Nets: which Variable Ordering?
In: *Proc. Int. Workshop on Petri Nets and Softw. Eng.* CEUR Workshop Proceedings 1846.
CEUR-WS, pp. 31–50. URL: hhttp://ceur-ws.org/Vol-1846/paper3.pdf.

Anjorin, Anthony, Sebastian Rose, Frederik Deckwerth, and Andy Schürr [2014].
Efficient Model Synchronization with View Triple Graph Grammars. In: *ECMFA 2014*. LNCS 8569.
Springer, pp. 1–17. DOI: 10.1007/978-3-319-09195-2_1.

Aziz, Adnan, Kumud Sanwal, Vigyan Singhal, and Robert Brayton [1996].
Verifying continuous time Markov chains. In: *CAV 1996*. LNCS 1102. Springer, pp. 269–276.
DOI: 10.1007/3-540-61474-5_75.

Babar, Junaid, Marco Beccuti, Susanna Donatelli, and Andrew S. Miner [2010].
GreatSPN Enhanced with Decision Diagram Data Structures. In: *PETRI NETS 2010*. LNCS 6128.
Springer, pp. 308–317. DOI: 10.1007/978-3-642-13675-7_19.

Baier, Christel, Joachim Klein, Sascha Klüppelholz, and Sascha Wunderlich [2017a].
Maximizing the Conditional Expected Reward for Reaching the Goal. In: *TACAS 2017*.
LNCS 10206. Springer, pp. 269–285. DOI: 10.1007/978-3-662-54580-5_16.

Baier, Christel, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich [2017b].
Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes.
In: *CAV 2017*. LNCS 10426. An extended version of the paper with implementation is available at https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/CAV17/. Springer, pp. 160–180.
DOI: 10.1007/978-3-319-63387-9_8.

Basu, Ananda, Marius Bozga, and Joseph Sifakis [2006].
Modeling Heterogeneous Real-time Components in BIP.
In: *4th IEEE Int. Conf. Softw. Eng. and Formal Methods*. IEEE. DOI: 10.1109/SEFM.2006.27.

Becker, Steffen, Heiko Koziolek, and Ralf Reussner [2008].
The Palladio component model for model-driven performance prediction.
J. Sys. Softw., **82** (1): 3–22. DOI: 10.1016/j.jss.2008.03.066.

Bergmann, Gábor, István Dávid, Ábel Hegedűs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró [2015]. VIATRA 3: A Reactive Model Transformation Platform. In: *ICMT 2015*.
LNCS 9152. Springer, pp. 101–110. DOI: 10.1007/978-3-319-21155-8_8.

Bergmann, Gábor, Ábel Hegedűs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varroó [2011]. Implementing Efficient Model Validation in EMF Tools.
In: *Proc. 26th ACM/IEEE Int. Conf. Automated Softw. Eng.* IEEE, pp. 580–583.
DOI: 10.1109/ASE.2011.6100130.

Bergmann, Gábor, István Ráth, Tamás Szabó, Paolo Torrini, and Dániel Varró [2012].
Incremental Pattern Matching for the Efficient Computation of Transitive Closure. In: *ICGT 2012*. LNCS 7562. Springer, pp. 386–400. DOI: 10.1007/978-3-642-33654-6_26.

Bernardi, Simona [2003].
Building Stochastic Petri Net models for the verification of complex software systems. PhD thesis. Department of Informatics, University of Turin.

Bernardi, Simona and Susanna Donatelli [2003].
Building Petri net scenarios for dependable automation systems.
In: *IEEE Proc. 7th Int. Workshop on Petri Nets and Performance Models*. IEEE, pp. 72–81. DOI: 10.1109/PNPM.2003.1231544.

Bernardi, Simona, Susanna Donatelli, and Giovanna Dondossola [2004]. Towards a Methodological Approach to Specification and Analysis of Dependable Automation Systems. In: LNCS 3253. Springer, pp. 36–51. DOI: 10.1007/978-3-540-30206-3_5.

Bernardi, Simona, Susanna Donatelli, and Andrá Horváth [2000]. Compositionality in the GreatSPN tool and its application to the modelling of industrial applications.
In: *Proc. Workshop on the practical use of High Level Petri Nets*.
URL: http://www.di.unito.it/~horvath/publications/papers/BeDoHo00.ps.

Blake, James T., Andrew L. Reibman, and Kishor S. Trivedi [1988].
Sensitivity analysis of reliability and performability measures for multiprocessor systems. *ACM SIGMETRICS Perf. Eval. Review*, **16** (1): 177–186. DOI: 10.1145/1007771.55616.

Brunelière, Hugo, Erik Burger a6nd Jordi Cabot, and Manuel Wimmer [2017].
A Feature-based Survey of Model View Approaches. *Softw. Sys. Mod.* DOI: 10.1007/s10270-017-0622-9.

Buchholz, Peter [1999a]. Hierarchical structuring of superposed GSPNs.
IEEE Tran. Softw. Eng., **25** (2): 166–181. DOI: 10.1109/32.761443.

Buchholz, Peter [1999b]. Structured analysis approaches for large Markov chains.
Appl. Numer. Math., **31** (4): 375–404. DOI: 10.1016/S0168-9274(99)00005-7.

Buchholz, Peter, Tuğrul Dayar, Jan Kriege, and Mushin Can Orhan [2017].
On compact solution vectors in Kronecker-based Markovian analysis. *J. Perf. Eval.*, **115**: 132–149. DOI: 10.1016/j.peva.2017.08.002.

Cabac, Lawrence, Michael Haustermann, and David Mosteller [2016]. Renew 2.5 – Towards a Comprehensive Integrated Development Environment for Retri Net-Based Applications.
In: *PETRI NETS 2016*. LNCS 9698. Springer, pp. 101–112. DOI: 10.1007/978-3-319-39086-4_7.

Ciardo, Gianfranco, Robert L. Jones, Andrew S. Miner, and Radu Siminiceanu [2006].
Logic and stochastic modeling with SMART. *J. Perf. Eval.*, **63** (6): 578–608. DOI: 10.1016/j.peva.2005.06.001.

Ciardo, Gianfranco, Gerald Lüttgen, and Radu Siminiceanu [2001].
Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation. In: *TACAS 2001*. LNCS 2031. Springer, pp. 328–342. DOI: 10.1007/3-540-45319-9_23.

Ciardo, Gianfranco and Kishor S. Trivedi [1993].
A decomposition approach for stochastic reward net models. *J. Perf. Eval.*, **38** (1): 37–59. DOI: 10.1016/0166-5316(93)90026-Q.

Ciardo, Gianfranco, Yang Zhao, and Xiaoqing Jin [2012].
Ten Years of Saturation: A Petri Net Perspective. In: *ToPNoC V*. LNCS 6900. Springer, pp. 51–95. DOI: 10.1007/978-3-642-29072-5_3.

Clarke, Edmund M. and E. Allen Emerson [1981].
Design and synthesis of synchronization skeletons using branching time temporal logic.
In: *Logics of Programs 1981*. LNCS 131. Springer, pp. 52–71. DOI: 10.1007/BFb0025774.

Courtney, Tod, Shravan Gaonkar, Ken Keefe, Eric W. D. Rozier, and William H. Sanders [2009].
Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In: *IEEE/IFIP Int. Conf. Dependable Systems & Networks, 2009*. IEEE. DOI: 10.1109/DSN.2009.5270318.

Czarnecki, Krzysztof and Simon Helsen [2006].
Feature-based survey of model transformation approaches.
IBM Systems J. – Model-driven software development, **45** (3): 621–645. DOI: 10.1147/sj.453.0621.

Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. Meyarivan [2002].
A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Tran. Evolutionary Comp.*, **6** (2).
DOI: 10.1109/4235.996017.

Debreceni, Csaba, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró [2014].
Query-driven incremental synchronization of view models.
In: *Proc. 2nd Workshop View-Based, Aspect-Oriented and Orthographic Software*. ACM, pp. 31–38.
DOI: 10.1145/2631675.2631677.

Dijk, Nicolaas M. van, Sem P. J. van Brummelen, and Richard J. Boucherie [2017].
Uniformization: Basics, extensions and applications. *J. Perf. Eval.*
DOI: 10.1016/j.peva.2017.09.008. In press.

Donatelli, Susanna, Serge Haddad, and Jeremy Sproston [2009].
Model Checking Timed and Stochastic Properties with CSL$^{\text{TA}}$.
IEEE Tran. Softw. Eng., **35** (2): 224–240. DOI: 10.1109/TSE.2008.108.

Donatelli, Susanna, Marina Ribaudo, and Jane Hillston [1995].
A comparison of performance evaluation process algebra and generalized stochastic Petri nets.
In: *Proc. of the 6th Int. Workshop on Petri Nets and Performance Models*. IEEE.
DOI: 10.1109/PNPM.1995.524326.

Emerson, E. Allen and Joseph Y. Halpern [1986].
"Sometimes" and "not never" revisited: on branching versus linear time temporal logic.
J. ACM, **33** (1): 151–178. DOI: 10.1145/4904.4999.

Feiler, Peter H. and David P. Gluch [2012]. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional.
ISBN: 978-0-32-188894-5.

Forgy, Charles L. [1982].
Rete: A fast algorithm for the many pattern/many object pattern match problem.
J. Artificial Intelligence, **19** (1): 17–37. DOI: 10.1016/0004-3702(82)90020-0.

Friedenthal, Sanford, Alan Moore, and Rick Steiner [2016].
A Practical Guide to SysML: The Systems Modeling Language. 3rd ed. Morgan Kaufmann.
ISBN: 978-0-12-800202-5.

Garavel, Hubert [2015]. Nested-Unit Petri Nets: A Structural Means to Increase Efficiency and Scalability of Verification on Elementary Nets. In: *PETRI NETS 2015*. LNCS 9115. Springer, pp. 179–199. DOI: 10.1007/978-3-319-19488-2_9.

Giese, Holger, Tihamér Levendovszky, and Hans Vangheluwe [2007].
Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools.
In: *Workshops and symphosia at MODELS 2006*. LNCS 4364. Springer, pp. 252–262.
DOI: 10.1007/978-3-540-69489-2_31.

Graics, Bence [2017]. *Model-Driven Development of Reactive Systems with Mixed Synchronous and Asynchronous Hierarchical Composition*. Scientific Students' Association Report. Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics.
URL: http://tdk.bme.hu/VIK/DownloadPaper/Vegyes-szemantika-szerinti-hierarchikus.

Greenyer, Joel and Jan Rieke [2011]. Applying Advanced TDD Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In: *AGTIVE 2011*. LNCS 7233. Springer, pp. 222–237. DOI: 10.1007/978-3-642-34176-2_19.

Hahn, Ernst Moritz, Holger Hermanns, and Lijun Zhang [2011].
Probabilistic reachability for parametric Markov models.
Int. J. Softw. Tools Technol. Transf., **13** (1): 3–19. DOI: 10.1007/s10009-010-0146-x.

Hansson, Hans and Bengt Jonsson [1994]. A logic for reasoning about time and reliability.
Formal Aspects of Computing, **6** (5): 512–535. DOI: 10.1007/BF01211866.

Hegedűs, Ábel, Ákos Horváth, and Dániel Varró [2013].
A model-driven framework for guided design space exploration.
Automated Softw. Eng., **22** (3): 339–436. DOI: 10.1007/s10515-014-0163-1.

Hermanns, Holger, Ulrich Herzog, and Joost-Pieter Katoen [2002].
Process algebra for performance evaluation. *J. Theor. Comput. Sci.*, **274** (1–2): 43–87.
DOI: 10.1016/S0304-3975(00)00305-4.

Hillah, Lom-Messan and Fabrice Kordon [2017].
Petri Nets Repository: A Tool to Benchmark and Debug Petri Net Tools. In: *PETRI NETS 2017*. LNCS 10258. Springer, pp. 125–135. DOI: 10.1007/978-3-319-57861-3_9.

Hillston, Jane [1995]. Compositional Markovian Modelling Using a Process Algebra. In: *Computations with Markov Chains*. Springer, pp. 177–196. DOI: 10.1007/978-1-4615-2241-6_12.

Hirel, Christophe, Bruno Tuffin, and Kishor S. Trivedi [2000].
SPNP Stochastic Petri Nets. Version 6.0. In: *TOOLS 2000*. LNCS 1786. Springer, pp. 354–357. DOI: 10.1007/3-540-46429-8_30.

Huang, Hejiao, Li Jiao, To-Yat Cheung, and Wai Ming Mak [2012].
Property-Preserving Petri Net Process Algebra in Software Engineering. World Scientific. ISBN: 978-981-4324-28-1.

International Organization for Standardization [2004]. *Systems and software engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation*. Standard ISO/IEC 15909-1:2004.

International Organization for Standardization [2011].
Systems and software engineering – High-level Petri nets – Part 2: Transfer format. Standard ISO/IEC 15909-2:2012.

Jackson, Daniel [2011]. *Software Abstractions*. Revised edition. The MIT Press. ISBN: 978-0-262-01715-2.

Jakob, Johannes, Alexander Königs, and Andy Schürr [2006].
Non-materialized Model View Specification with Triple Graph Grammars. In: *ICGT 2006*. LNCS 4178. Springer, pp. 321–335. DOI: 10.1007/11841883_23.

Jouault, Frédéric, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev [2008].
ATL: A Model Transformation Tool. *J. Sci. Comp. Prog.*, **72** (1-2): 31–39. DOI: 10.1016/j.scico.2007.08.002.

Jouault, Frédéric and Massimo Tisi [2010]. Towards Incremental Execution of ATL Transformations. In: *ICMT 2010*. LNCS 6142. Springer, pp. 123–137. DOI: 10.1007/978-3-642-13688-7_9.

Kam, Timothy, Tiziano Villa, Robert Brayton, and Alberto Sangiovanni-Vincentelli [1998].
Multi-valued decision diagrams: theory and applications. *Int. J. Multiple-Valued Logic*, 4 (1–2): 9–62.

Kang, Eunsuk, Ethan Jackson, and Wolfram Schulte [2010].
An Approach for Effective Design Space Exploration. In: *Monterey Workshop 2010*. LNCS 6662. Springer, pp. 33–54. DOI: 10.1007/978-3-642-21292-5_3.

Kindler, Ekkart [2007]. Modular PNML revisited: Some ideas for strict typing. In: *Proc. 14th Workshop Algorithmen und Werkzeuge für Petrinetze*. Universität Koblenz-Landau, pp. 20–25. URL: http://www2.cs.uni-paderborn.de/cs/kindler/Publikationen/copies/AWPN07-PNMLmodules.pdf.

Kindler, Ekkart and Laure Petrucci [2009].
Towards a Standard for Modular Petri Nets: A Formalisation. In: *PETRI NETS 2009*. LNCS 5606, pp. 43–62. DOI: 10.1007/978-3-642-02424-5_5.

Kindler, Ekkart and Michael Weber [2001].
A Universal Module Concept for Petri Nets – an implementation-oriented approach. Informatik-Bericht 150.
URL: https://www2.informatik.hu-berlin.de/top/pnml/download/about/modPNML_TB.ps.

Klenik, Attila and Kristóf Marussy [2015].
Configurable Stochastic Analysis Framework for Asynchronous Systems.
Scientific Students' Association Report. Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics.
URL: http://tdk.bme.hu/VIK/ViewPaper/Aszinkron-rendszerek-konfigurarhato.

Kordon, Fabrice, Hubert Garavel, Lom-Messan Hillah, Francis Hulin-Hubard, Bernard Berthomieu, Gianfranco Ciardo, Maximilien Colange, Silvano Dal Zilio, Elvio Gilberto Amparore, Marco Beccuti, Torsten Liebke, Jeroen J. G. Meijer, Andrew S. Miner, Christian Rohr, Jiri Srba, Yann Thierry-Mieg, Jaco van der Pol, and Karsten Wolf [2017].
Complete Results for the 2017 Edition of the Model Checking Contest.
URL: http://mcc.lip6.fr/2017/results.php.

Koziolek, Heiko [2010]. Performance evaluation of component-based software systems: A survey. *J. Perf. Eval.*, **67** (8): 634–658. DOI: 10.1016/j.peva.2009.07.007.

Kühne, Thomas, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer [2009]. Explicit Transformation Modeling. In: *MODELS 2009*. LNCS 6002. Springer, pp. 240–255. DOI: 10.1007/978-3-642-12261-3_23.

Kuntz, G. W. Matthias and Boudewijn R. H. M. Haverkort [2007]. GCSRL – A Logic for Stochastic Reward Models with Timed and Untimed Behaviour. In: *Proc. 8th Workshop on Performability of Modeling Computer and Communication Systems*. CTIT Workshop Proceedings LNCS4549. Centre for Telematics and Information Technology, University of Twente, pp. 50–56. URL: https://research.utwente.nl/files/5312765/cam_ready_GCSRL.pdf.

Kwiatkowska, Marta, Gethin Norman, and David Parker [2011]. PRISM 4.0: Verification of Probabilistic Real-time Systems. In: *CAV 2011*. LNCS 6806. Springer, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

Kwiatkowska, Martha, Gethin Norman, and António Pacheco [2006]. Model checking expected time and expected reward formulae with random time bounds. *Comp. & Math. with Appl.*, **51** (2): 305–316. DOI: 10.1016/j.camwa.2005.11.016.

Llorens, Marisa, Javier Oliver, Josep Silva, and Salvador Tamarit [2017]. An Integrated Environment for Petri Net Slicing. In: *PETRI NETS 2017*. LNCS 10258. Springer, pp. 112–124. DOI: 10.1007/978-3-319-57861-3_8.

Logothetis, Dimitris, Kishor S. Trivedi, and Antonio Puliafito [1995]. Markov regenerative models. In: *Proc. of the 1995 IEEE Int. Comput. Perf. and Dependability Symp.* IEEE. DOI: 10.1109/IPDS.1995.395809.

Longo, Francesco and Marco Scarpa [2013]. Two-layer symbolic representation for stochastic models with phase-type distributed events. *Int. J. Syst. Sci.*, **46** (9): 1540–1571. DOI: 10.1080/00207721.2013.822940.

López-Grao, Juan Pablo, José Merseguer, and Javier Campos [2004]. From UML activity diagrams to Stochastic Petri nets: application to software performance engineering. In: *Proc. 4th Int. Workshop Softw. Perf.* ACM, pp. 25–36. DOI: 10.1145/974044.974048.

Lúcio, Levi, Joachim Denil, Hans Vangheluwe, Sadaf Mustafiz, and Bart Meyers [2012]. *The Formalism Transformation Graph as a Guide to Model Driven Engineering*. Tech. rep. CS-TR-2012.1. School of Computer Science, McGill University. URL: https://www.cs.mcgill.ca/media/tech_reports/10_The_Formalism_Transformation_Graph_as_a_Guide_to_Model_Driven_Engineering.pdf.

Maechler, Martin [2016]. *diptest: Hartigan's Dip Test Statistic for Unimodality – Corrected*. R package version 0.75-7. URL: https://CRAN.R-project.org/package=diptest.

Marechal, Alexis and Didier Buchs [2012]. *Modular extensions of Petri Nets: a survey*. Tech. rep. 218. Faculty of Sciences, University of Geneva. URL: https://smv.unige.ch//technical-reports/pdfs/ModularitySurvey.pdf.

Marsan, Marco Ajmone, Gianni Conte, and Gianfranco Balbo [1984]. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Trans. Comput. Syst.*, **2** (2): 93–122. DOI: 10.1145/190.191.

Martens, Anne, Heiko Koziolek, Steffen Becker, and Ralf Reussner [2010]. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: *Proc. 1st joint WOSP/SIPEW int. conf. Perf. eng.* ACM, pp. 105–116. DOI: 10.1145/1712605.1712624.

Marussy, Kristóf, Attila Klenik, Vince Molnár, András Vörös, István Majzik, and Miklós Telek [2016a]. Efficient Decomposition Algorithm for Stationary Analysis of Complex Stochastic Petri Net Models. In: *PETRI NETS 2016*. LNCS 9698. Springer, pp. 281–300. DOI: 10.1007/978-3-319-39086-4_17.

Marussy, Kristóf, Attila Klenik, Vince Molnár, András Vörös, Miklós Telek, and István Majzik [2016b]. Configurable numerical analysis for stochastic systems. In: *2016 Int. Workshop on Symbolic and Numerical Methods for Reachability Analysis*. IEEE. DOI: 10.1109/SNR.2016.7479383.

Marussy, Kristóf, Vince Molnár, András Vörös, and István Majzik [2017].
Getting the Priorities Right: Saturation for Prioritised Petri Nets. In: *PETRI NETS 2017*. LNCS 10258. Springer, pp. 223–242. DOI: 10.1007/978-3-319-57861-3_14.

McBride, Connor and Ross Paterson [2008]. Applicative programming with effects.
J. Functional Prog., **18** (1): 1–13. DOI: 10.1017/S0956796807006326.

Meyers, Bart [2016]. *A Multi-Paradigm Modeling Approach to Design and Evolution of Domain-Specific Modeling Languages*.
PhD thesis. Department of Mathematics and Computer Science, University of Antwerp.
URL: http://msdl.cs.mcgill.ca/people/bart/publ/thesis.pdf.

Meyers, Bart, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer [2014].
ProMoBox: A Framework for Generating Domain-Specific Property Languages. In: *SLE 2014*. LNCS 8706. Springer, pp. 1–20. DOI: 10.1007/978-3-319-11245-9_1.

Miner, Andrew S. [2004]. Implicit GSPN reachability set generation using decision diagrams.
J. Perf. Eval., **56** (1–4): 145–165. DOI: 10.1016/j.peva.2003.07.005.

Miner, Andrew S. [2006]. Saturation for a General Class of Models.
IEEE Trans. Softw. Eng., **32** (8): 559–570. DOI: 10.1109/TSE.2006.81.

Molnár, Tímea [2017].
Sztochasztikus modellek paramétereinek optimalizációja: eszközök és kihívások. In Hungarian.
Bachleor's thesis. Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics.
URL: https://diplomaterv.vik.bme.hu/hu/Theses/Sztochasztikus-modellek-parametereinek.

Molnár, Vince, András Vörös, Dániel Darvas, Tamás Bartha, and István Majzik [2016].
Component-wise incremental LTL model checking. *Formal Aspects of Comp.*, **28** (3): 345–379.
DOI: 10.1007/s00165-015-0347-x.

Moreno, Gabriel A., Connie U. Smith, and Lloyd G. Williams [2008].
Performance analysis of real-time component architectures: a model interchange approach.
In: *Proc. 8th Workshop Softw. Perf.* ACM, pp. 115–126. DOI: 10.1145/1383559.1383574.

Morsel, Aad P. A. van and William H. Sanders [1997].
Transient solution of Markov models by combining adaptive and standard uniformization.
IEEE Tran. Reliability, **46** (3): 430–440. DOI: 10.1109/24.664016.

Mosteller, David, Lawrence Cabac, and Michael Haustermann [2016].
Integrating Petri Net Semantics in a Model-Driven Approach: The Renew Meta-Modeling and Transformation Framework. In: *ToPNoC XI*. LNCS 9930. Springer, pp. 92–113.
DOI: 10.1007/978-3-662-53401-4_5.

Murata, Tadao [1989]. Petri nets: Properties, analysis and applications. *Proc. IEEE*, **77** (4): 541–580.
DOI: 10.1109/5.24143.

Neuts, Marcel F. [1975]. Probability distributions of phase type.
In: *Liber Amicorum Prof. Emeritus H. Florin*. University of Louvain, pp. 173–206.

Nouri, Ayoub, Saddek Bensalem, Marius Bozga, Benoit Delahaye, Cyrille Jegourel, and Axel Legay [2015]. Statistical model checking QoS properties of systems with SBIP.
Int. J. Softw. Tools Technol. Transf., **17** (2): 171–185. DOI: 10.1007/s10009-014-0313-6.

Object Management Group [2014]. *Object Constraint Language Specification*. Version 2.4.
URL: http://www.omg.org/spec/OCL/2.4/.

Object Management Group [2015]. *XML Metadata Interchange (XMI) Specification*. Version 2.5.1.
URL: http://www.omg.org/spec/XMI/2.5.1/.

Object Management Group [2016]. *MOF Query/View/Transformation Specification*. Version 1.3.
URL: http://www.omg.org/spec/QVT/1.3/.

Pierce, Benjamin C. [2002]. *Types and programming languages*. The MIT Press.
ISBN: 978-0-262-16209-8.

Quatmann, Tim, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen [2016].
Parameter Synthesis for Markov Models: Faster Than Ever. In: *ATVA 2016*. LNCS 9938. Springer, pp. 50–67. DOI: 10.1007/978-3-319-46520-3_4.

Ramesh, A. V. and Kishor S. Trivedi [1993].
On the Sensitivity of Transient Solutions of Markov Models.

In: *Proc. of the 1993 ACM SIGMETRICS conf. on Measurement and modeling of comput. syst.* ACM, pp. 122–134. DOI: 10.1145/166955.166998.

Ráth, István, Ábel Hegedüs, and Dániel Varró [2012].
Derived Features for EMF by Integrating Advanced Model Queries. In: *ECMFA 2012*. LNCS 7349. Springer, pp. 102–117. DOI: 10.1007/978-3-642-31491-9_10.

Reibman, Andrew L., Roger Smith, and Kishor S. Trivedi [1989].
Markov and Markov reward model transient analysis: An overview of numerical approaches. *Eur. J. Oper. Res.*, **4** (2): 257–267. DOI: 10.1016/0377-2217(89)90335-4.

Rumbaugh, James, Ivar Jacobson, and Grady Booch [2004].
The Unified Modeling Language Reference Manual. 2nd ed. Pearson Higher Education. ISBN: 0321245628.

Sanders, William H. and John F. Meyer [2001].
Stochastic Activity Networks: Formal Definitions and Concepts. In: *EEF School 2000*. LNCS 2090. Springer, pp. 315–343. DOI: 10.1007/3-540-44667-2_9.

Schürr, Andy [1994]. Specification of graph translators with triple graph grammars. In: *WG 1994*. LNCS 903. Springer, pp. 151–163. DOI: 10.1007/3-540-59071-4_45.

Semeráth, Oszkár, Csaba Debreceni, Ákos Horváth, and Dániel Varró [2016a].
Incremental backward change propagation of view models by logic solvers.
In: *Proc. ACM/IEEE 19th Int. Conf. Model Driven Eng. Lang. Syst.* ACM, pp. 306–316.
DOI: 10.1145/2976767.2976788.

Semeráth, Oszkár and Dániel Varró [2017].
Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In: *ICMT 2017*. LNCS 10374. Springer, pp. 138–154. DOI: 10.1007/978-3-319-61473-1_10.

Semeráth, Oszkár, András Vörös, and Dániel Varró [2016b].
Iterative and Incremental Model Generation by Logic Solvers. In: *FASE 2016*. LNCS 9633. Springer, pp. 87–103. DOI: 10.1007/978-3-662-49665-7_6.

Smith, Connie U. and Catalina M. Lladó [2011].
Model Interoperability for Performance Engineering: Survey of Milestones and Evolution. In: *PERFORM 2010*. LNCS 6821. Springer, pp. 10–23. DOI: 10.1007/978-3-642-25575-5_2.

Song, Hui, Gang Huang, Franck Chauvel, Wei Zhang, Yanchun Sun, Weizhong Shao, and Hong Mei [2011]. Instant and Incremental QVT Transformation for Runtime Models. In: *MODELS 2011*. LNCS 6981. Springer, pp. 273–288. DOI: 10.1007/978-3-642-24485-8_20.

Steinberg, Dave, Frank Budinsky, Marcelo Paternostro, and Ed Merks [2009].
EMF: Eclipse Modeling Framework. 2nd ed. Addison-Wesley Professional. ISBN: 978-0-321-33188-5.

Telek, Miklós and András Pfening [1996].
Performance analysis of Markov regenerative reward models. *J. Perf. Eval.*, **27–28**: 1–18. DOI: 10.1016/S0166-5316(96)90017-6.

Teruel, Enrique, Giuliana Franceschinis, and Massimiliano De Pierro [2003].
Well-defined generalized stochastic Petri nets: a net-level method to specify priorities. *IEEE Tran. Softw. Eng.*, **29** (11): 962–973. DOI: 10.1109/TSE.2003.1245298.

Ujhelyi, Zoltán, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró [2015].
EMF-IncQuery: An integrated development environment for live model queries. *J. Sci. Comp. Prog.*, **98** (1): 80–99. DOI: 10.1016/j.scico.2014.01.004.

Vanherpen, Ken, Joachim Denil, Paul De Meulenaere, and Hans Vangheluwe [2014].
Design-Space Exploration in MDE: An Initial Pattern Catalogue. In: *Proc. of the 1st Int. Workshop on Combining Modelling with Search- and Example-Based Approaches*. CEUR Workshop Proceedings 1340. CEUR-WS, pp. 42–51.
URL: http://ceur-ws.org/Vol-1340/paper6.pdf.

Vardi, Moshe Y. [1996]. An automata-theoretic approach to linear temporal logic.
In: *Logics for Concurrency*. LNCS 1043. Springer, pp. 238–266. DOI: 10.1007/3-540-60915-6_6.

Varró, Dániel [2015]. Patterns and Styles for Incremental Model Transformations.
In: *Proc. 1st Workshop Patterns in Model Eng.* CEUR Workshop Proceedings 1657. CEUR-WS, pp. 41–43. URL: http://ceur-ws.org/Vol-1657/paper8.pdf.

Varró, Dániel, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth [2017]. Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models. In: *Festschrift in Memory of Hartmut Ehrig*. LNCS. Springer. URL: https://inf.mit.bme.hu/sites/default/files/publications/fmhe2017-model-generation.pdf. Forthcoming.

Vernon, Mary, John Zahorjan, and Edward D. Lazowska [1986]. *A Comparison of Performance Petri Nets and Queueing Network Models*. Computer Sciences Techninal Report 669. URL: http://ftp.cs.wisc.edu/pub/techreports/1986/TR669.pdf.

Vörös, András, Dániel Darvas, Ákos Hajdu, Attila Jámbor, Attila Klenik, Kristóf Marussy, Vince Molnár, Tamás Bartha, and István Majzik [2017a]. *PetriDotNet 1.5 User Manual*. URL: http://petridotnet.inf.mit.bme.hu/releases/pdn1_manual.pdf.

Vörös, András, Dániel Darvas, Ákos Hajdu, Attila Klenik, Kristóf Marussy, Vince Molnár, Tamás Bartha, and István Majzik [2017b]. Industrial applications of the PetriDotNet modelling and analysis tool. *J. Sci. Comp. Prog.* DOI: 10.1016/j.scico.2017.09.003. In press.

Walker, David [2005]. Substructural Type Systems. In: *Advanced Topics in Types and Programming Languages*. The MIT Press, pp. 3–43. ISBN: 0-262-16228-8.

Xiong, Yingfei, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei [2007]. Towards automatic model synchronization from model transformations. In: *Proc. 22nd IEEE/ACM int. conf. on Automated Softw. Eng.* ACM, pp. 164–173. DOI: 10.1145/1321631.1321657.

Zalila, Faiez, Xavier Crégut, and Marc Pantel [2013]. Formal Verification Integration Approach for DSML. In: *MODELS 2013*. LNCS 8107. Springer, pp. 336–351. DOI: 10.1007/978-3-642-41533-3_21.

Ziemann, Paul and Martin Gogolla [2003]. OCL Extended with Temporal Logic. In: *PSI 2003*. LNCS 2890. Springer, pp. 351–357. DOI: 10.1007/978-3-540-39866-0_35.

Appendix A

Case study: dining philosophers

A.1 Graph queries

Listing A.1 Philosophers.vql

```
package hu.bme.mit.inf.petridse.example.philosophers.patterns

import "http://example.org/philosophers"

/** Find the table around which the philosophers sit. */
pattern table(Table : Table) {
   Table(Table);
}

/** Find each Phil around the Table. */
pattern philosopher(Table : Table, Phil : Philosopher) {
   Philosopher(Phil);
   Table.philosophers(Table, Phil);
}

/** Left and Right sit next to each other around the Table. */
pattern adjacentPhilosophers(Table : Table, Left : Philosopher,
      Right : Philosopher) {
   Philosopher.right(Left, Right);
   Table.philosophers(Table, Left);
   Table.philosophers(Table, Right);
}
```

A.2 Petri net modules

Listing A.2 PhilModule.ecore2pn

```
package hu.bme.mit.inf.petridse.example.philosophers.ecore2pn

module PhilModule {
   ref param double hungryRate
   ref param double eatingRate

```

```
    prop double thinkingTime = if (#thinking >= 1) 1 else 0
    prop boolean isHungry = #waiting >= 1

    place thinking = 1
    place waiting
    place eating
    place rightFork = 1
    ref place leftFork

    // Timed transition with rate hungryRate.
    tran getHungry = exp hungryRate
    // Immediate transition with probability weight 1.0, priority is 1 by default.
    tran startEating = immediate 1.0
    tran finishEating = exp eatingRate

    // Chains of arcs can be written with chains of arrows.
    // Arc weights default to 0, but can be specified as e.g. -2->.
    thinking -> getHungry -> waiting -> startEating -> eating
        -> finishEating -> thinking
    rightFork, leftFork -> startEating
    finishEating -> rightFork, leftFork
}
```

Listing A.3 TableModule.ecore2pn

```
package hu.bme.mit.inf.petridse.example.philosophers.ecore2pn

module TableModule {
   prop double[] thinkingTimes
   prop double totalThinkingTime = sum(thinkingTimes)
   prop boolean[] hungryFaultModes
}
```

A.3 Transformation specification

Listing A.4 DiningPhilosophers.ecore2pn

```
package hu.bme.mit.inf.petridse.example.philosophers.ecore2pn

import "http://example.org/philosophers"
import hu.bme.mit.inf.petridse.example.philosophers.patterns.*

transformation DiningPhilosophers {
   features {
      Philosopher {
         @Description(text="Likes rice 'this' much.")
         param eatingRate

         @RewardConfiguration derived prop double thinkingTime
         @FaultConfiguration derived prop boolean isHungry
      }

      Table {
         @RewardConfiguration
         derived prop double totalThinkingTime
         @FaultConfiguration
         derived prop boolean[] hungryFaultModes
      }
   }

   /** Create the Petri net mapping for the table. */
   mapping table(Table) => TableModule TableM {
      // Bind the metric totalThinkingTime and the collection of fault modes
      // hungryFaultModes to the domain object Table as derived features.
      Table.totalThinkingTime := TableM.totalThinkingTime
      Table.hungryFaultModes := TableM.hungryFaultModes
   }

   /** Create the Petri net mapping for each philosopher. */
   mapping philosopher(Table, Phil) => PhilModule PhilM {
      lookup table(Table) as TableM
      PhilM.hungryRate := Phil.hungryRate
      PhilM.eatingRate := Phil.eatingRate
      // First PhilM.totalThinkingTime is assigned to the derived feature
      // Phil.totalThinkingTime...
      Phil.thinkingTime := PhilM.thinkingTime
      // ...then Phil.totalThinkingTime is added to the collection
      // TableM.thinkingTimes.
      TableM.thinkingTimes += Phil.thinkingTime
      Phil.isHungry := PhilM.isHungry
      TableM.hungryFaultModes += Phil.isHungry
   }

   /** Adjacent philosophers must share forks. */
   mapping adjacentPhilosophers(Table, Left, Right) {
      lookup philosopher(Table, Left) as LeftM
      lookup philosopher(Table, Right) as RightM
      RightM.leftFork := LeftM.rightFork
   }
}
```

Appendix B

Case study: architectural modeling language

B.1 Architectural modeling language metamodel

B.2 Graph queries

Listing B.1 Dependability.vql

```
package hu.bme.mit.inf.petridse.example.dependability.queries

import "http://example.org/dependability"
```

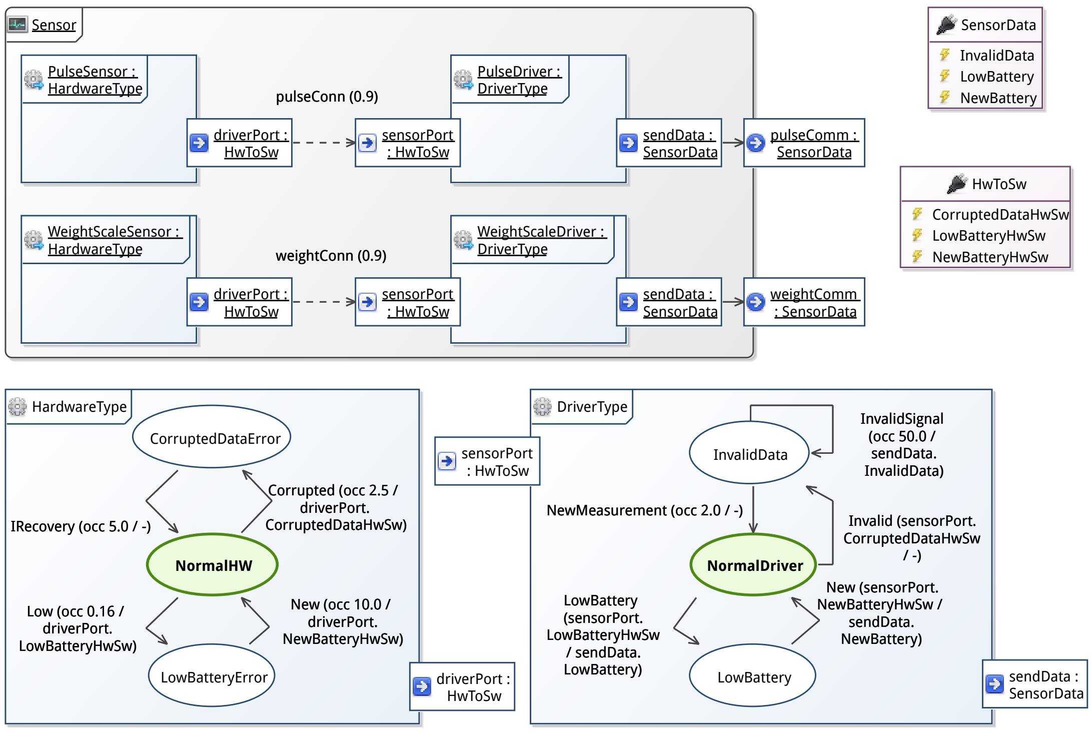


Figure B.1 Telecare system architecture example.

```

pattern state(DepModel : DepModel, Component : Component, State : State)
    {
   find component(DepModel, Component);
   Component.componentType.errorModel.states(Component, State);
}

pattern initialState(DepModel : DepModel, Component : Component, State :
    State) {
   find component(DepModel, Component);
   Component.componentType.errorModel.initialState(Component, State);
}

pattern errorState(DepModel : DepModel, Component : Component, State :
    State) {
   find state(DepModel, Component, State);
   neg find initialState(DepModel, Component, State);
}

pattern port(DepModel : DepModel, Component : Component, Port : Port) {
   find component(DepModel, Component);
   find componentPort(Component, Port);
}

pattern errorMode(DepModel : DepModel, Component : Component, Port :
    Port, ErrorMode : ErrorMode) {
   find port(DepModel, Component, Port);
   Port.portType.errorModes(Port, ErrorMode);
}

pattern outputErrorMode(DepModel : DepModel, Component : Component, Port
    : Port, ErrorMode : ErrorMode) {
   find component(DepModel, Component);
   Component.componentType.outputPorts(Component, Port);
   Port.portType.errorModes(Port, ErrorMode);
}

pattern transition(DepModel : DepModel, Component : Component,
    Transition : Transition) {
   find component(DepModel, Component);
   Component.componentType.errorModel.transitions(Component, Transition);
}

pattern transitionFromTo(DepModel : DepModel, Component : Component,
    Transition : Transition, From : State, To : State) {
   find transition(DepModel, Component, Transition);
   Transition.sourceState(Transition, From);
   Transition.targetState(Transition, To);
}

pattern transitionOccurrenceTrigger(DepModel : DepModel, Component :
    Component, Transition : Transition, Trigger : OccurrenceTrigger) {
   find transition(DepModel, Component, Transition);
   Transition.trigger(Transition, Trigger);
   OccurrenceTrigger(Trigger);
```

```
}

pattern transitionPropagatedTrigger(DepModel : DepModel, Component :
    Component, Transition : Transition, Port : Port, ErrorMode :
    ErrorMode) {
   find transition(DepModel, Component, Transition);
   Transition.trigger(Transition, Trigger);
   PropagatedErrorTrigger.inputError.port(Trigger, Port);
   PropagatedErrorTrigger.inputError.errorMode(Trigger, ErrorMode);
}

pattern transitionPropagates(DepModel : DepModel, Component : Component,
    Transition : Transition, Port : Port, ErrorMode : ErrorMode) {
   find transition(DepModel, Component, Transition);
   Transition.outputError.port(Transition, Port);
   Transition.outputError.errorMode(Transition, ErrorMode);
}

pattern connectedErrorModesUncertain(DepModel : DepModel, From :
    Component, FromPort : Port, To : Component, ToPort : Port,
    Propagation : UncertainPropagation, ErrorMode : ErrorMode) {
   find connectedErrorModes(DepModel, From, FromPort, To, ToPort,
       Propagation, ErrorMode);
   UncertainPropagation(Propagation);
}

pattern connectedErrorModesCertain(DepModel : DepModel, From :
    Component, FromPort : Port, To : Component, ToPort : Port, ErrorMode
    : ErrorMode) {
   find connectedErrorModes(DepModel, From, FromPort, To, ToPort,
       Propagation, ErrorMode);
   CertainPropagation(Propagation);
}

pattern component(DepModel : DepModel, Component : Component) {
   DepModel.systems.components(DepModel, Component);
}

private pattern componentTypePort(ComponentType : ComponentType, Port :
    Port) {
   ComponentType.inputPorts(ComponentType, Port);
} or {
   ComponentType.outputPorts(ComponentType, Port);
}

private pattern componentPort(Component : Component, Port : Port) {
   Component.componentType(Component, ComponentType);
   find componentTypePort(ComponentType, Port);
}

private pattern connectedPorts(From : Component, FromPort : Port, To :
    Component, ToPort : Port, Connection : Connection) {
   ComponentConnection.sourcePort.component(Connection, From);
   ComponentConnection.sourcePort.port(Connection, FromPort);
   ComponentConnection.targetPort.component(Connection, To);
```

```
    ComponentConnection.targetPort.port(Connection, ToPort);
} or {
    SystemConnection.sourcePort.componentPort.component(Connection, From);
    SystemConnection.sourcePort.componentPort.port(Connection, FromPort);
    SystemConnection.targetPort.componentPort.component(Connection, To);
    SystemConnection.targetPort.componentPort.port(Connection, ToPort);
}

pattern connectedErrorModes(DepModel : DepModel, From : Component,
     FromPort : Port, To : Component, ToPort : Port, Propagation :
     Propagation, ErrorMode : ErrorMode) {
    find component(DepModel, From);
    find component(DepModel, To);
    find connectedPorts(From, FromPort, To, ToPort, Connection);
    Connection.propagation(Connection, Propagation);
    Port.portType.errorModes(FromPort, ErrorMode);
}
```

B.3 Petri net modules

Listing B.2 DependabilityModules.ecore2pn

```
package hu.bme.mit.inf.petridse.example.dependability.ecore2pn

module StateModule {
    ref place p
}

module InitialStateModule {
    place p = 1
}

module ErrorStateModule {
    place p
}

module ErrorModeModule {
    marking boolean errorModeOccurred = #error >= 1

    place error
}

module OutputErrorModeModule {
    ref place error
    tran outputToBuffer = immediate 1.0 priority 2

    error -> outputToBuffer
}

module TransitionModule {
    ref place from
    ref place to
    ref tran fire
```

```

   from -> fire -> to
}

module OccurrenceTransitionModule {
   ref param double occurrenceRate

   tran fire = exp occurrenceRate
}

module PropagatedTransitionModule {
   ref place triggerError
   tran fire = immediate 1.0 priority 1

   triggerError -> fire
}

module PortModule {
   @FaultConfiguration
   marking boolean[] errorModes
}

module CertainPropagationModule {
   ref place error
   ref tran fire

   error -o fire -> error
}

module UncertainPropagationModule {
   ref param double probability

   place buffer
   ref place to

   ref tran outputToBuffer
   tran propagateError = immediate probability priority 1
   tran propagateSink = immediate 1.0 - probability priority 1

   to -o outputToBuffer, propagateError
   buffer -o outputToBuffer -> buffer
   buffer -> propagateError -> to
   buffer -> propagateSink
}
```

B.4 Transformation specification

Listing B.3 DependabilityPetriNet.ecore2pn

```
package hu.bme.mit.inf.petridse.example.dependability.ecore2pn

import "http://example.org/dependability" as dependability
import hu.bme.mit.inf.petridse.example.dependability.queries.*
```

```

transformation DependabilityPetriNet {
   features {
      dependability::OccurrenceTrigger {
         @Description(text="Occurrence rate of the fault.")
         param occurrenceRate
      }

      dependability::UncertainPropagation {
         @Description(text="Probability of fault propagation.")
         param probability
      }
   }

   mapping state(DepModel, Component, State) => StateModule

   mapping initialState(DepModel, Component, State) =>
       InitialStateModule IS {
      lookup state(DepModel, Component, State) as S
      S.p := IS.p
   }

   mapping errorState(DepModel, Component, State) => ErrorStateModule ES
       {
      lookup state(DepModel, Component, State) as S
      S.p := ES.p
   }

   mapping port(DepModel, Component, Port) => PortModule

   mapping errorMode(DepModel, Component, Port, ErrorMode) =>
       ErrorModeModule EM {
      lookup port(DepModel, Component, Port) as P
      P.errorModes += EM.errorModeOccurred
   }

   mapping outputErrorMode(DepModel, Component, Port, ErrorMode) =>
       OutputErrorModeModule OutEM {
      lookup errorMode(DepModel, Component, Port, ErrorMode) as EM
      OutEM.error := EM.error
   }

   mapping transition(DepModel, Component, Transition) =>
       TransitionModule

   mapping transitionFromTo(DepModel, Component, Transition, From, To) {
      lookup transition(DepModel, Component, Transition) as Tran
      lookup state(DepModel, Component, From) as SF
      lookup state(DepModel, Component, To) as ST
      Tran.from := SF.p
      Tran.to := ST.p
   }

   mapping transitionOccurrenceTrigger(DepModel, Component, Transition,
       Trigger) => OccurrenceTransitionModule OT {
        lookup transition(DepModel, Component, Transition) as Tran
        OT.occurrenceRate := Trigger.occurrenceRate
        Tran.fire := OT.fire
    }

    mapping transitionPropagatedTrigger(DepModel, Component, Transition, Port, ErrorMode) => PropagatedTransitionModule PT {
        lookup transition(DepModel, Component, Transition) as Tran
        lookup errorMode(DepModel, Component, Port, ErrorMode) as EM
        PT.triggerError := EM.error
        Tran.fire := PT.fire
    }

    mapping transitionPropagates(DepModel, Component, Transition, Port, ErrorMode) => CertainPropagationModule TP {
        lookup transition(DepModel, Component, Transition) as Tran
        lookup errorMode(DepModel, Component, Port, ErrorMode) as EM
        TP.fire := Tran.fire
        TP.error := EM.error
    }

    mapping connectedErrorModesCertain(DepModel, From, FromPort, To, ToPort, ErrorMode) => CertainPropagationModule P {
        lookup outputErrorMode(DepModel, From, FromPort, ErrorMode) as EMFrom
        lookup errorMode(DepModel, To, ToPort, ErrorMode) as EMTo
        P.fire := EMFrom.outputToBuffer
        P.error := EMTo.error
    }

    mapping connectedErrorModesUncertain(DepModel, From, FromPort, To, ToPort, Propagation, ErrorMode) => UncertainPropagationModule P {
        lookup outputErrorMode(DepModel, From, FromPort, ErrorMode) as EMFrom
        lookup errorMode(DepModel, To, ToPort, ErrorMode) as EMTo
        P.outputToBuffer% := EMFrom.ToutputToBuffer^
        P.to := EMTo.error
        P.probability := Propagation.probability
    }
}
```