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

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

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

Contents

Kivonat   Ide kerül a kivonat.

Kulcsszavak   diplomaterv, sablon, LaTeX

Abstract   Here comes the abstract.

Keywords   thesis, template, LaTeX

Hallgatói nyilatkozat

Alulírott **Marussy Kristóf** szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. december 5.

....
Marussy Kristóf

Chapter 1

Introduction

[TODO: General intro text comes here]

1.1 Related work: DSE for stochastic models

1.2 Overview of the approach

Chapter 2

Background

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 separate the input formal model and the *query* to be answered **[TODO: cite]**, which is usually 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 PetriDot-Net [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 to 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, contrasting 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 disctinction.]

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.

[TODO: Should we cite a review about QNs?]

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 Hejiao Huang et al. [2012].

   [TODO: Write about actual modular formalisms]

3.1.2   Query specifications

[TODO: Review modular query specification languages]

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

and the *inlining* of RGSPNs into GSPNs without references, which can be analyzed with existing tools.

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.

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 10, 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.

[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.]

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 3.4 on page 19, 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 15 we introduce *aggregation functions* into the syntax of query expressions. This lets the aggregate performance measure be written as $\mathsf{sum}(c)$ 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 **[TODO: Kind?]** introduced, which are *arcs*, *reference assingments* and *collection memberships*.

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 3.4.1 on page 19.

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.
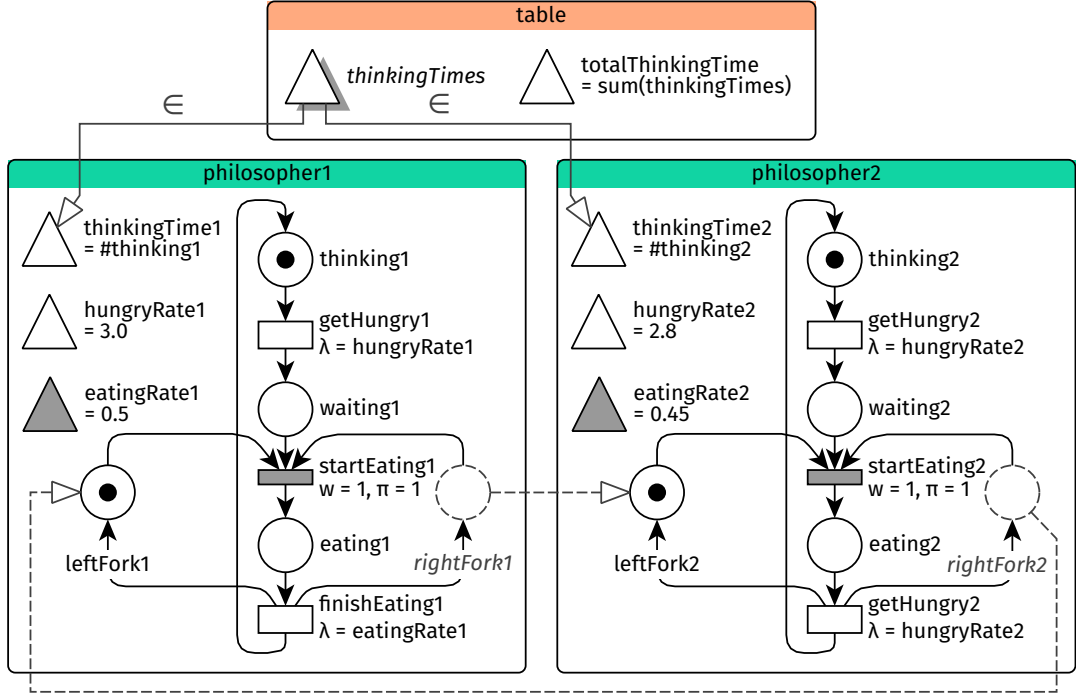
Figure 3.1   Example RGSPN model with an aggregate performance measure.

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.

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:

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

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

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

$$
\begin{pmatrix}
\texttt{path} \\
| \\
\texttt{prop} \\
| \\
\texttt{weight} \\
\diagup \quad \diagdown \\
\texttt{param} \quad \texttt{marking} \\
\diagdown \quad \diagup \\
\texttt{const}
\end{pmatrix}
\times
\begin{pmatrix}
\texttt{double} \\
| \\
\texttt{boolean} \quad \texttt{int}
\end{pmatrix}
\tag{3.2}
$$

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 17.

Collection types are covariant in their member types; therefore $\tau <: \tau'$ if and only if $\tau[] <: \tau'[]$. 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 $\tau \to \sigma$, i.e. operators that transform values of type τ into values of type σ, extending syntax of types ‹*Type*› ::= ... | ‹*Type*› → ‹*Type*›. 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 15 even though expression will serves as the properties of symbols in the definition of RGSPNs.

> **Definition 3.1**   An *RGSPN signatrue* is a 12-tuple
>
> $$\Sigma = \langle P, T_T, T_i, V, Par, R, C, dep, pretype, value, target, member \rangle,$$
>
> 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*;
> - $dep \colon V \to Dependence$ is the *variable dependence* function;
> - $pretype \colon V \to Pretype$ is the *variable pretype* function;
> - $value \colon V \cup Par \to Expr_\Sigma \cup \mathbb{R}$ is a function, such that $value(v) \in Expr_\Sigma$ for all $v \in V$ and $value(\theta) \in \mathbb{R}$ for all $\theta \in Par$;
> - $target \colon R \to Type$ is the *reference target type* function;
> - $member \colon C \to Type$ 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 a 10-tuple $N = \langle \Sigma, m_0, \lambda, w, \pi, \leftarrow, \rightarrow, \multimap, :=, += \rangle$, where
> - $\Sigma = \langle P, T_T, T_i, V, Par, R, C, \ldots \rangle$ is an RGSPN signature;
> - $m_0 \colon P \to Expr_\Sigma$ is the *initial marking* function;
> - $\lambda \colon T_T \to Expr_\Sigma$ is the *timed transition rate* function;
> - $w \colon T_i \to Expr_\Sigma$ is the *immediate transition weight* function;
> - $\pi \colon T_i \to Expr_\Sigma$ is the *immediate transition priority* function;
> - $\leftarrow, \rightarrow, \multimap \subseteq \Sigma \times Expr_\Sigma \times \Sigma$ are the relations of *output, input* and *inhibitor arcs*, respectively, which are free of parallel arcs, i.e. $\langle p, e_1, t \rangle, \langle p, e_2, t \rangle \in \leftarrow$ implies $e_1 = e_2$ and this property holds also for $\rightarrow$ 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 $Expr_\Sigma$ with the exception of the values of the parameters, which must be real numbers. As any computation is **[TODO: Should** forbidden inside parameter values, so that parameter synthesis tool may set new values of **this go some-** the parameters without needing to respect any constraints between parameter values implicit **where else?]** 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.

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 $+=$ a multiset relation, i.e. a *bag* of tuples, such as $\wr \langle c,s \rangle, \langle c,s \rangle, \ldots \wr$.

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

Type checking

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

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

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 $e \in Expr_\Sigma$. 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 3.3 on page 15 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.
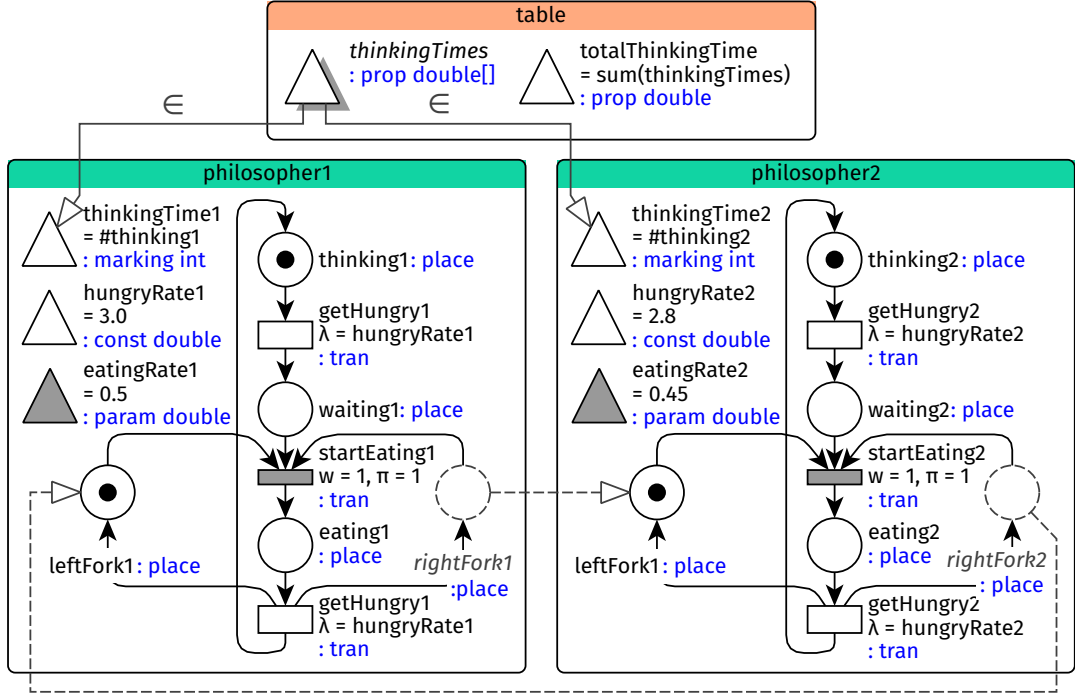
Figure 3.2   Example RGSPN with type annotations.

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, $\Sigma \vdash m_0(p) :$ const int.
- For all timed transitions $t \in T_T$ the transition rate is a possibly marking- and parameter-dependent real number, $\Sigma \vdash \lambda(t) :$ weight double.
- 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, $\Sigma \vdash value(v) : type(v)$.
- All arcs $p \xleftarrow{e} t$, $p \xrightarrow{e} t$ or $p \overset{e}{\multimap} t$ go between places and transitions such that $type(p) <:$ place and $type(t) <:$ 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, $type(s) <: target(r)$.
- For all $c \mathrel{+}= s$, s is a valid member of c, $type(s) <: member(s)$.

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 weight 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 9 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 12, 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*$_\Sigma$ described by the following EBNF-like grammar:

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

(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 *values* 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*$_\Sigma$; therefore "`if ` $(\#p_1 > 0) \#p_2$ `else` $\#p_3$" is a valid expression, but "$\#(\texttt{if } (\#p_1 > 0) \, p_2 \texttt{ else } p_3)$" 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*$_\Sigma$ is presented in Table 3.1, which describes the relation _ ⊢ _ : _. The judgement $\Sigma \vdash e : \tau$ assigns a type τ to an expression $e \in Expr(\Sigma)$ in the

Table 3.1   Typing rules for expressions.

$$\frac{\diamond \in \{\texttt{+},\texttt{-}\} \quad \rho \in \{\texttt{int},\texttt{double}\} \quad \Sigma \vdash e : \delta\,\rho}{\Sigma \vdash \diamond e : \delta\,\rho}, \qquad \text{(T-Unary\pm)}$$

$$\frac{\Sigma \vdash e : \delta\,\texttt{boolean}}{\Sigma \vdash \,!e : \delta\,\texttt{boolean}}, \qquad \text{(T-UnaryNot)}$$

$$\frac{\diamond \in \{\texttt{A},\texttt{U}\} \quad \Sigma \vdash e : \texttt{path boolean}}{\Sigma \vdash \diamond e : \texttt{prop boolean}}, \qquad \text{(T-UnaryState)}$$

$$\frac{\diamond \in \{\texttt{X},\texttt{F},\texttt{G}\} \quad \Sigma \vdash e : \texttt{path boolean}}{\Sigma \vdash \diamond e : \texttt{path boolean}}, \qquad \text{(T-UnaryPath)}$$

$$\frac{\diamond \in \{\texttt{+},\texttt{-},\texttt{*}\} \quad \rho \in \{\texttt{int},\texttt{double}\} \quad \Sigma \vdash e_1 : \delta\,\rho \quad \Sigma \vdash e_2 : \delta\,\rho}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\rho}, \qquad \text{(T-BinNumeric)}$$

$$\frac{\Sigma \vdash e_1 : \delta\,\texttt{double} \quad \Sigma \vdash e_2 : \delta\,\texttt{double}}{\Sigma \vdash e_1 \,/\, e_2 : \delta\,\texttt{double}}, \qquad \text{(T-BinDiv)}$$

$$\frac{\Sigma \vdash e_1 : \texttt{path boolean} \quad \Sigma \vdash e_2 : \texttt{path boolean}}{\Sigma \vdash e_1 \,\texttt{U}\, e_2 : \texttt{path boolean}}, \qquad \text{(T-BinUntil)}$$

$$\frac{\diamond \in \{\texttt{==},\texttt{!=}\} \quad \Sigma \vdash e_1 : \delta\,\rho \quad \Sigma \vdash e_2 : \delta\,\rho}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\texttt{boolean}}, \qquad \text{(T-BinEq)}$$

$$\frac{\diamond \in \{\texttt{<},\texttt{<=},\texttt{>},\texttt{>=}\} \quad \Sigma \vdash e_1 : \delta\,\texttt{double} \quad \Sigma \vdash e_2 : \delta\,\texttt{double}}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\texttt{boolean}}, \qquad \text{(T-BinCompare)}$$

$$\frac{\diamond \in \{\texttt{\&\&},\texttt{||}\} \quad \Sigma \vdash e_1 : \delta\,\texttt{boolean} \quad \Sigma \vdash e_2 : \delta\,\texttt{boolean}}{\Sigma \vdash e_1 \diamond e_2 : \delta\,\texttt{boolean}}, \qquad \text{(T-BinLogical)}$$

$$\frac{\Sigma \vdash e_1 : \delta\,\texttt{boolean} \quad \Sigma \vdash e_2 : \delta\,\rho \quad \Sigma \vdash e_3 : \delta\,\rho}{\Sigma \vdash \texttt{if } (e_1)\ e_2 \texttt{ else } e_3 : \delta\,\rho}, \qquad \text{(T-If)}$$

$$\frac{agg \in \{\texttt{sum},\texttt{prod}\} \quad \rho \in \{\texttt{int},\texttt{double}\} \quad type(b) = \delta\,\rho\texttt{[]}}{\Sigma \vdash agg(b) : \delta\,\rho}, \qquad \text{(T-AggNumeric)}$$

$$\frac{agg \in \{\texttt{all},\texttt{any}\} \quad type(b) = \delta\,\texttt{boolean[]}}{\Sigma \vdash agg(b) : \delta\,\texttt{boolean}}, \qquad \text{(T-AggLogical)}$$

$$v : type(v), \qquad \text{(T-Var)} \qquad\qquad \frac{type(p) \,\texttt{<:}\, \texttt{place}}{\texttt{\#}p : \texttt{marking int}}, \qquad \text{(T-Marking)}$$

$$\frac{\ell \in [\![\rho]\!]}{\Sigma \vdash \ell : \texttt{const } \rho}, \qquad \text{(T-Literal)} \qquad\qquad \frac{\Sigma \vdash e : \tau \quad \tau \,\texttt{<:}\, \tau'}{\Sigma \vdash e : \tau'}, \qquad \text{(T-Sub)}$$

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

context of an RGSPN signature Σ.

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 12.

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 exception 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 $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ of truth values, $\mathbb{Z}$ of integers and $\mathbb{R}$ of real numbers, respectively.$^{\triangle}$ Formally, pretypes have the interpretations specified in Table 3.1, i.e.

$$\llbracket \texttt{boolean} \rrbracket = \mathbb{B}, \qquad \llbracket \texttt{int} \rrbracket = \mathbb{Z}, \qquad \llbracket \texttt{double} \rrbracket = \mathbb{R}.$$

Variable types $\delta\ \rho$ can be viewed as functions from some *context* determined by the dependence qualifier δ to the set $\llbracket \rho \rrbracket$. In the case $\delta = \texttt{const}$, the context is empty, so $\llbracket \texttt{const}\ \rho \rrbracket$ is isomorphic to $\llbracket \rho \rrbracket$. 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 $\Pi = m_1 \rightarrow m_2 \rightarrow \ldots$ of markings (or equivalently, CTMC states). The

---

$^{\triangle}$ 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 **[TODO: Cite Bayer MDP paper]**.

interpretations of variable types can be summarized as

$$
\begin{aligned}
\llbracket \texttt{const } \rho \rrbracket: & & & p \in \llbracket \rho \rrbracket, \\
\llbracket \texttt{param } \rho \rrbracket: \theta & & & \mapsto p \in \llbracket \rho \rrbracket, \\
\llbracket \texttt{marking } \rho \rrbracket: \quad m & & & \mapsto p \in \llbracket \rho \rrbracket, \\
\llbracket \texttt{weight } \rho \rrbracket: \theta, m & & & \mapsto p \in \llbracket \rho \rrbracket, \\
\llbracket \texttt{prop } \rho \rrbracket: \theta, m, \textsc{ctmc} & & & \mapsto p \in \llbracket \rho \rrbracket, \\
\llbracket \texttt{path } \rho \rrbracket: \theta, \quad \textsc{ctmc}, \Pi & & & \mapsto p \in \llbracket \rho \rrbracket.
\end{aligned}
$$

Type coercion from `int` to `double` act in the obvious way. Dependence coercion along the partial order from eq. (3.2) on page 11 introduces arguments to the interpretation functions that are ignored. For example, coercing `const` to `weight` results in a function that ignores 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 $\textsc{ctl}^*$ formulas this conversion is defined such that the first marking m_1 of the path $\Pi = m_1 \rightarrow m_2 \rightarrow \ldots$ 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 $\llbracket \_ \rrbracket$ of the expressions $Expr_\Sigma$ themselves. The intepretations of expressions conform with the types, i.e. if $\Sigma \vdash e : \tau$, then we have $\llbracket e \rrbracket \in \llbracket \tau \rrbracket$.

Operators from eq. (3.3) on page 15 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.$^\triangle$

The $\textsc{ctl}^*$ 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 $\textsc{ctmc}$ into the interpretation function. Similarly, `path` expressions are treated as predicates over paths Π and are composed by the operators according to $\textsc{ctl}^*$ semantics [see e.g. **[TODO: Cite]**].

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 there is a unique, non-cyclic chain of reference assignments from r to s. In addition, every concrete symbol resolves to itself, $s \rightsquigarrow s$ for all $s \in \Sigma \setminus R$. Formally, let $:=^* \subseteq \Sigma \times \Sigma$ be the reflexive transitive closure of the relation $:=$, i.e.

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

and write $s_1 \rightsquigarrow s_2$ if $s_2 \in \Sigma \setminus R$ is the unique 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,

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

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

$^\triangle$ This makes variable types with a dependence qualifier other than `const` specializations of *Reader* applicative functors **[TODO: cite]**.

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

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

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 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 $\langle +, 0 \rangle$ is associated with the aggregation operator `sum`, $\langle *, 1 \rangle$ with `prod`, $\langle \&\&, \text{true} \rangle$ with `all` and $\langle ||, 1 \rangle$ with `any`.

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

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

where the multiset-builder notation respects the multiplicities in the relation $\mathrel{:=}$. 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

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

where the operatior $\diamond$ acts over the interpretation functions $\llbracket s \rrbracket$ as discussed above while respecting multiplicity and the constant n is type coerced as needed.

Stochastic queries

[TODO: The sections below should go to the transformation chapter.]

3.4   Reference inlining

3.4.1   Handling inconsistent models

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 [Bruneliere 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 14 is extended to transformation rules to aid in catching bugs.
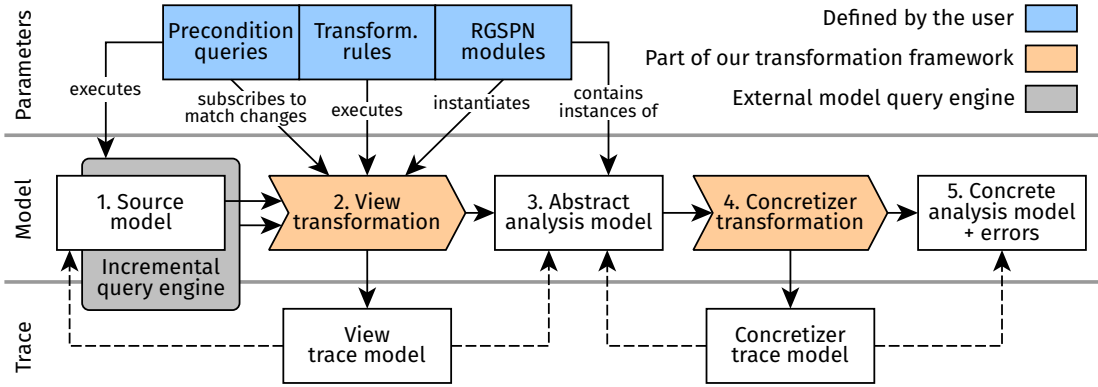
Figure 4.1 Overview of the transformation chain.

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 for formal models

4.1.1 Incremental transformation languages

[TODO:]

4.1.2 Transformation languages for stochastic models

[TODO:]

Table 4.1 Transformation specification for dining philosophers domain models.

Precondition	Transformation rule	RGSPN module

Transformation rule column:

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

mapping qTable(T)
    => TableMod TM {
  T.totalThinkingTime
      := TM.totalThinkingTime
}

mapping qPhil(T, P)
    => PhilMod PM {
  lookup qTable(T) => TM
  PM.hungryRate := P.hungryRate
  PM.eatingRate := P.eatingRate
  TM.thinkingTimes
      += PM.thinkingTime
}

mapping qAdjacent(T, L, R) {
  lookup qPhil(T, L) => PM1
  lookup qPhil(T, R) => PM2
  PM2.leftFork := PM1.rightFork
}
```

Precondition column labels:

Philosopher
eatingRate: double
hungryRate: double

Table

qTable(T) — T: Table

qPhil(T,P) — T: Table, P: Philosopher

qAdjacent(T,L,R) — T, right, L, left, R

RGSPN module column labels:

hungryRate   eatingRate

totalThinkingTime

TableMod
thinkingTimes   totalThinkingTime = sum(thinkingTimes)

PhilMod
thinkingTime = #thinking
hungryRate
eatingRate
leftFork
thinking
getHungry λ = hungryRate
waiting
startEating w = 1, π = 1
eating
rightFork
finishEating λ = eatingRate

PM1: PhilMod — rightFork
PM2: PhilMod — leftFork

4.2 Overview of the transformation engine

4.2.1 Transformation specifications

4.3 Generic view transformation to stochastic Petri nets

4.4 Stochastic Petri net concretization

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 [Abdeen et al., 2014; Hegedűs et al., 2013].
- 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
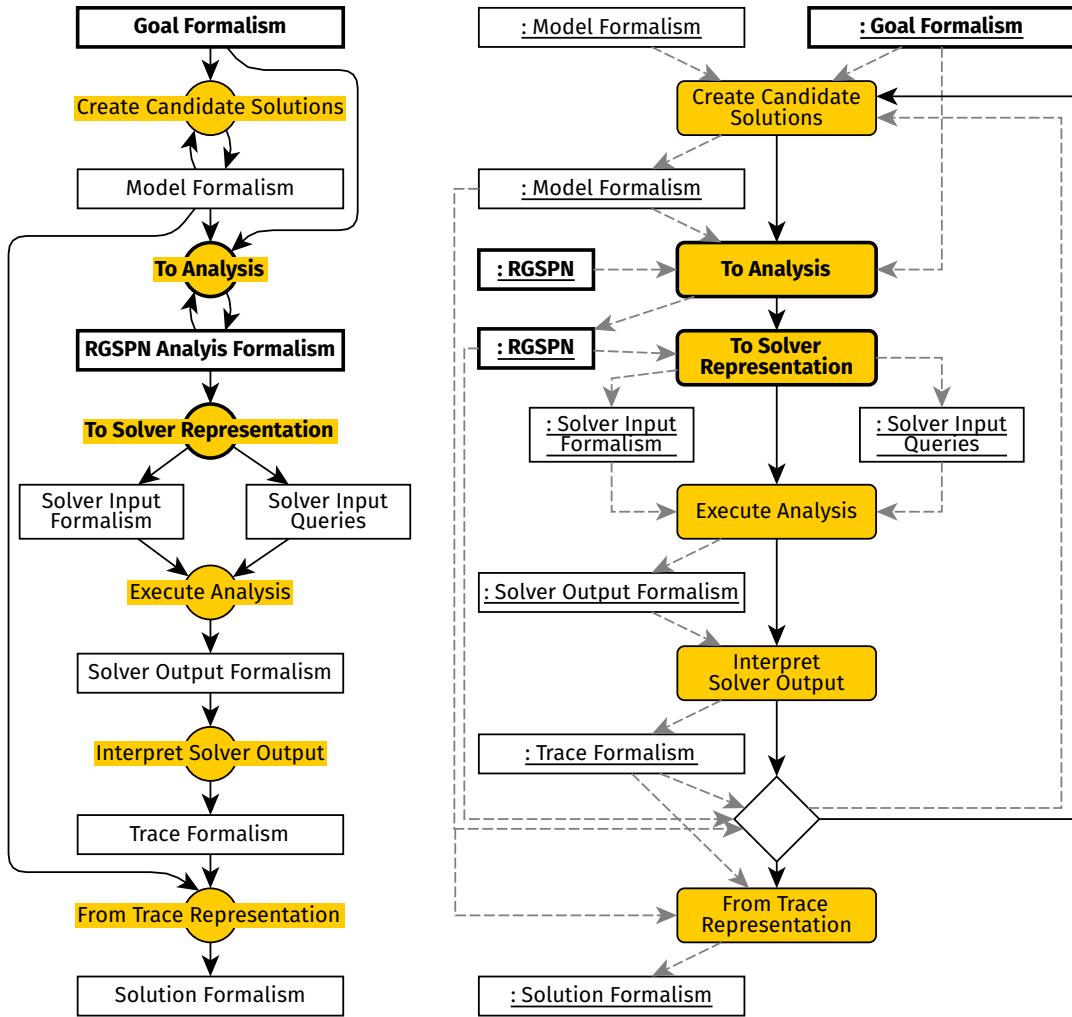
Figure 5.1   FTG+PM of the *Model Transformation* DSE pattern with RGSPN-based analysis. The components in **bold** were implemented in our work, while the rest of the components should be supplied by the DSE framework and stochastic analysis tool.

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.

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*.

[△] 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].

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 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 [Marussy et al., 2017; Miner, 2006]. 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; Buchholz et al., 2017; Marussy et al., 2016b].

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 [van Dijk et al., 2017; van Morsel and Sanders, 1997].

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

---

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

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.

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[1] (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*[2] 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.
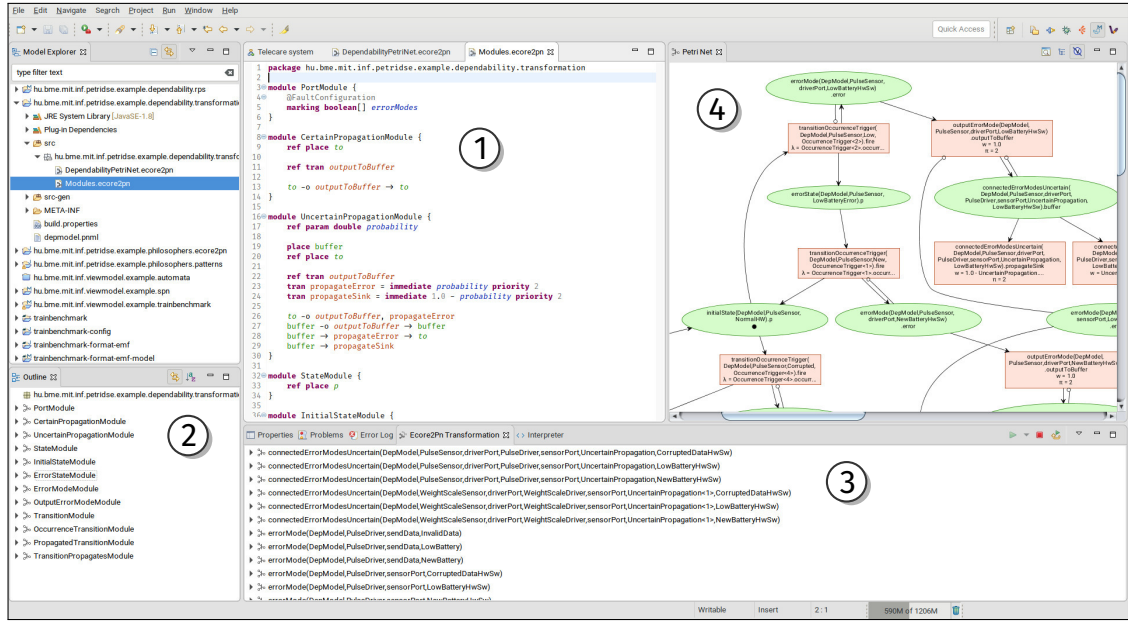
5.2.1   Specification environment

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

---

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

---

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

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.

graphical concrete syntax as Sirius[3] 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.

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*[4] 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[5] of Graphviz by the *Petri Net* view of the specification environment to display the results of the currently running RGSPN transformation.

---

[3] http://www.eclipse.org/sirius/　　[4] https://www.graphviz.org/　　[5] https://github.com/nidi3/graphviz-java

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.

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*[6] 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*[7] 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

---

[6] https://www.eclipse.org/xtend/   [7] 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

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:

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.
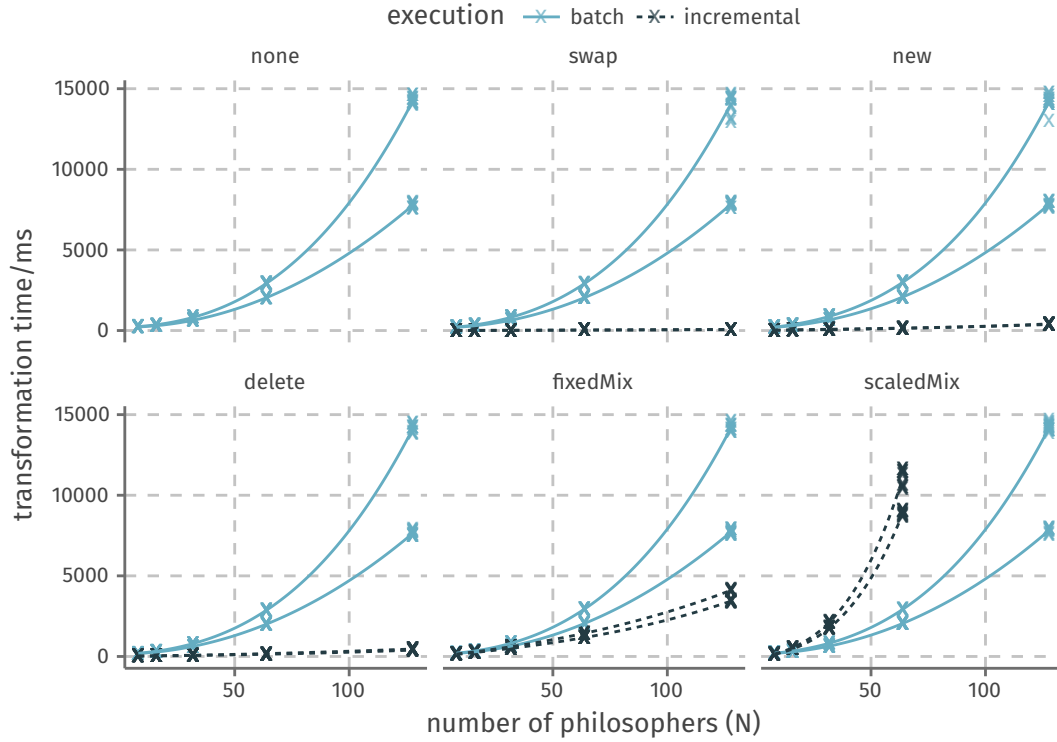
Figure 5.3 Execution times of transformations.

- **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 $N = 8$ 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.

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

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 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.

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

Table 5.3   Execution times of PNML serializations.
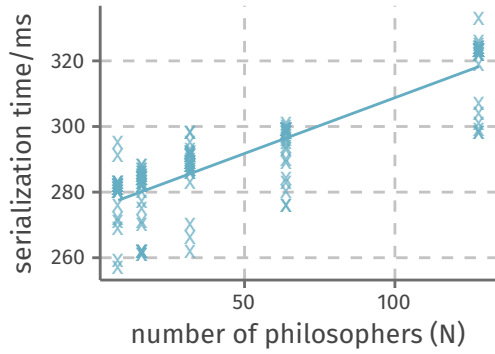
N	Time/ms	PNML size/bytes
8	257 – 295	50 700
16	261 ↕ 288	100 441
32	262 – 298	200 572
64	276 – 301	400 736
128	298 – 333	802 454

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 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 the latter observation is only due to pure 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.

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.

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.

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.

Becker, Steffen, Heiko Koziolek, and Ralf Reussner [2008]. "The Palladio component model for model-driven performance prediction". In: *J. Sys. Softw.* 82(1), pp. 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.

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

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

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

Buchholz, Peter [1999b]. "Structured analysis approaches for large Markov chains". In: *Appl. Numer. Math.* 31(4), pp. 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". In: *J. Perf. Eval.* 115, pp. 132–149. DOI: 10.1016/j.peva.2017.08.002.

Ciardo, Gianfranco, Robert L. Jones, Andrew S. Miner, and Radu Siminiceanu [2006]. "Logic and stochastic modeling with SMART". In: *J. Perf. Eval.* 63(6), pp. 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". In: *J. Perf. Eval.* 38(1), pp. 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.

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.

Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. Meyarivan [2002]. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *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.

Donatelli, Susanna, Marina Ribaudo, and Jane Hillston [1995]. "A comparison of performance evaluation process algebra and generalized stochastic Petri nets". In: DOI: 10.1109/PNPM.1995.524326.

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.

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.

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

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

Hejiao Huang, 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.

Hermanns, Holger, Ulrich Herzog, and Joost-Pieter Katoen [2002]. "Process algebra for performance evaluation". In: *Theor. Comput. Sci.* 274(1–2), pp. 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.

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.

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

Kam, Timothy, Tiziano Villa, Robert Brayton, and Alberto Sangiovanni-Vincentelli [1998]. "Multi-valued decision diagrams: theory and applications". In: *Int. J. Multiple-Valued Logic* 4(1–2). **[TODO: Find DOI or some other way to access this article]**, pp. 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.

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". In: *J. Perf. Eval.* 67(8), pp. 634–658. DOI: 10.1016/j.peva.2009.07.007.

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". In: *Int. J. Syst. Sci.* 46(9), pp. 1540–1571. DOI: 10.1080/00207721.2013.822940.

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.

Marsan, Marco Ajmone, Gianni Conte, and Gianfranco Balbo [1984]. "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems". In: *ACM Trans. Comput. Syst.* 2(2), pp. 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.

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.

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

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

Molnár, Tímea [2017]. "Sztochasztikus modellek paramétereinek optimalizációja: eszközök és kihívások". In Hungarian. **[TODO: Add URL once it is available.]** Bachleor's thesis. Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics.

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

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

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.

Reibman, Andrew L., Roger Smith, and Kishor S. Trivedi [1989]. "Markov and Markov reward model transient analysis: An overview of numerical approaches". In: *Eur. J. Oper. Res.* 4(2), pp. 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.

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.

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". In: *J. Perf. Eval.* 27-28, pp. 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". In: *IEEE Tran. Softw. Eng.* 29(11), pp. 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". In: *J. Sci. Comp. Prog.* 98(1), pp. 80–99. DOI: 10.1016/j.scico.2014.01.004.

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

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

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.

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 Technical 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". In: *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.