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.

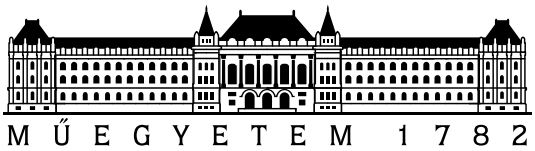
Tanszéki konzulens: Molnár Vince, doktorandusz

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

M.Sc. 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 szakdolgozatot 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. november 30.

.....
Marussy Kristóf

Chapter 1

Introduction

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 PetriDotNet (Vörös et al., 2017). 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 disct-inction.]

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.

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

Each symbol has a *kind*, which determines what information is needed to define the symbol, and a *type*, which determines the context where the symbol may be used. The type system, which is elaborated in Section 3.2.2 on page 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.

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., 2017) 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 18, 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 $\texttt{sum}(c)$ analogously to the big operator expression $\sum_{v \in c} v$, where c is the collection of the constituent elementary measures. Collections may contain duplicate elements so that expressions like $v + v + v$ can be written in big operator form.

Edges

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

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

Reference assignments connect references to the symbol at which they point. Indirect references, i.e. $r_1 := r_2$, $r_2 := s$ are possible and arbitrary chains of references may be built. In particular, an RGSPN may even contain reference cycles ($r_1 := r_2$, $r_2 := r_1$), or multiple, contradictory assignments ($r := s_1$, $r := s_2$, $s_1 \neq s_2$). However, *inconsistent* RGSPNs cannot be transformed into GSPNs for analysis. Inconsistency handling is discussed in detail in Section 3.4.1 on page 18.

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.

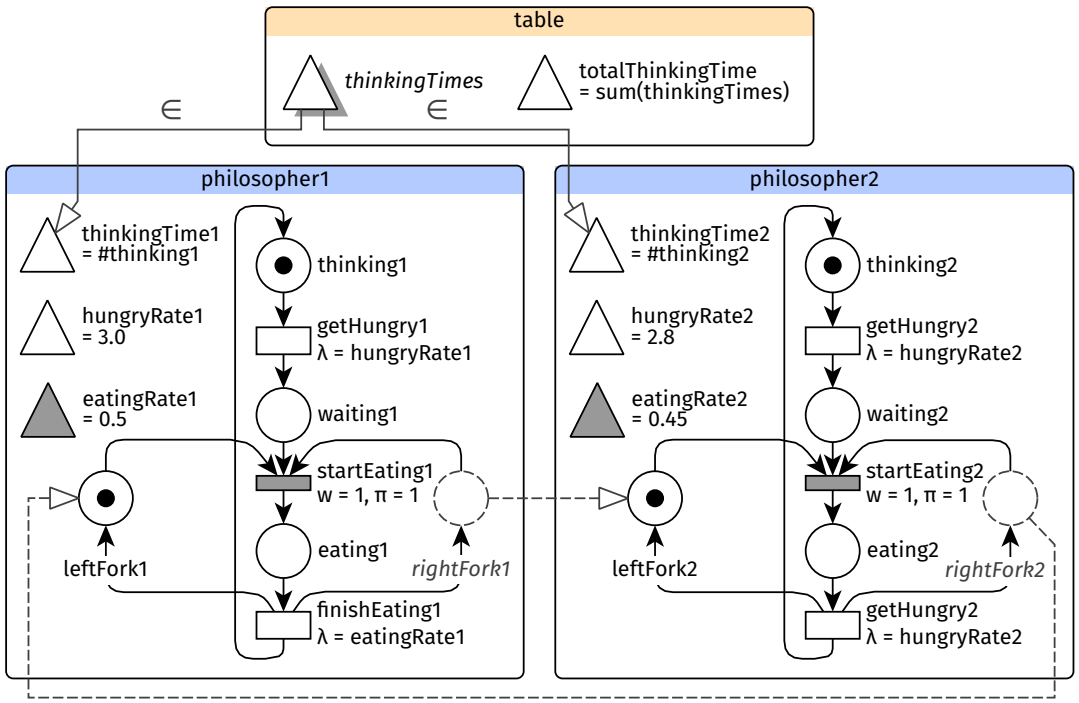


Figure 3.1 Example RGSPN model with an aggregate performance measure.

> 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:

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

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

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

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

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

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

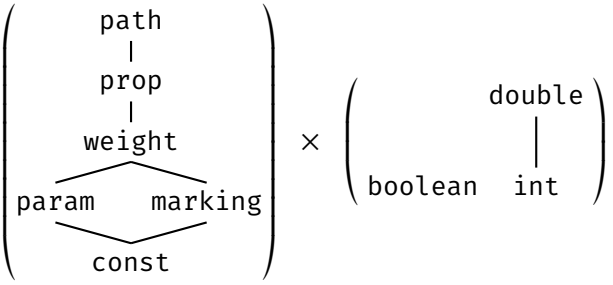
Remark 3.2 Some analysis methods only allow specific kinds of parameter-dependence, such as C^1 differentiable expressions (Blake et al., 1988) or rational functions (Hahn et al., 2011). However, no attempt is make to track different classes of parameter-dependent functions in `param` expressions, because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

Subtyping

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

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

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



(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` $\preceq$ `marking` and `int` $\preceq$ `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 $\tau\texttt{[]} <: \tau'\texttt{[]}$. Type coercion of collections is performed elementwise.

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

Modular colored nets may also contain *operator* symbols, which transform members of a color class into another. In our framework, these could be modeled by symbols of type $\tau \to \sigma$, 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. Chapters 15 and 16 of Pierce, 2002).

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*;
- $dep : V \to Dependence$ is the *variable dependence* function;
- $pretype : V \to Pretype$ is the *variable pretype* function;
- $value : V \cup Par \to Expr_\Sigma \cup \mathbb{R}$ is a function, such that $value(v) \in Expr_\Sigma$ for all $v \in V$ and $value(\theta) \in \mathbb{R}$ for all $\theta \in Par$;
- $target : R \to Type$ is the *reference target type* function;
- $member : C \to Type$ 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;
- $m_0 : P \to Expr_\Sigma$ is the *initial marking* function;
- $\lambda : T_T \to Expr_\Sigma$ is the *timed transition rate* function;
- $w : T_i \to Expr_\Sigma$ is the *immediate transition weight* function;
- $\pi : T_i \to Expr_\Sigma$ 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*;
- $+= \in \mathit{Multiset}(\Sigma \times \Sigma)$ is the multiset relation of *collection memberships*.

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

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

[TODO: Should this go somewhere else?]

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

Although parallel arcs are forbidden, parallel collection membership edges are permitted by making $+=$ 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 $p \overset{e}{\longleftarrow} t$, $p \overset{e}{\longrightarrow} t$, $p \overset{e}{\multimap} t$, $r := s$ and $c \mathrel{+}= s$ for $\langle p, e, t\rangle \in \longleftarrow$, $\langle p, e, t\rangle \in \longrightarrow$, $\langle p, e, t\rangle \in \multimap$, $\langle r, s\rangle \in :=$ and $\langle c, s\rangle \in +=$, respectively.

Type checking

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

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

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 $e \in \mathit{Expr}_\Sigma$. We write $\Sigma \vdash e : \tau$ if e is of type τ in the context of the RGSPN signature Σ. As it will be seen in Section 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.

[TODO: Should we split the formal stuff and the discussion?]

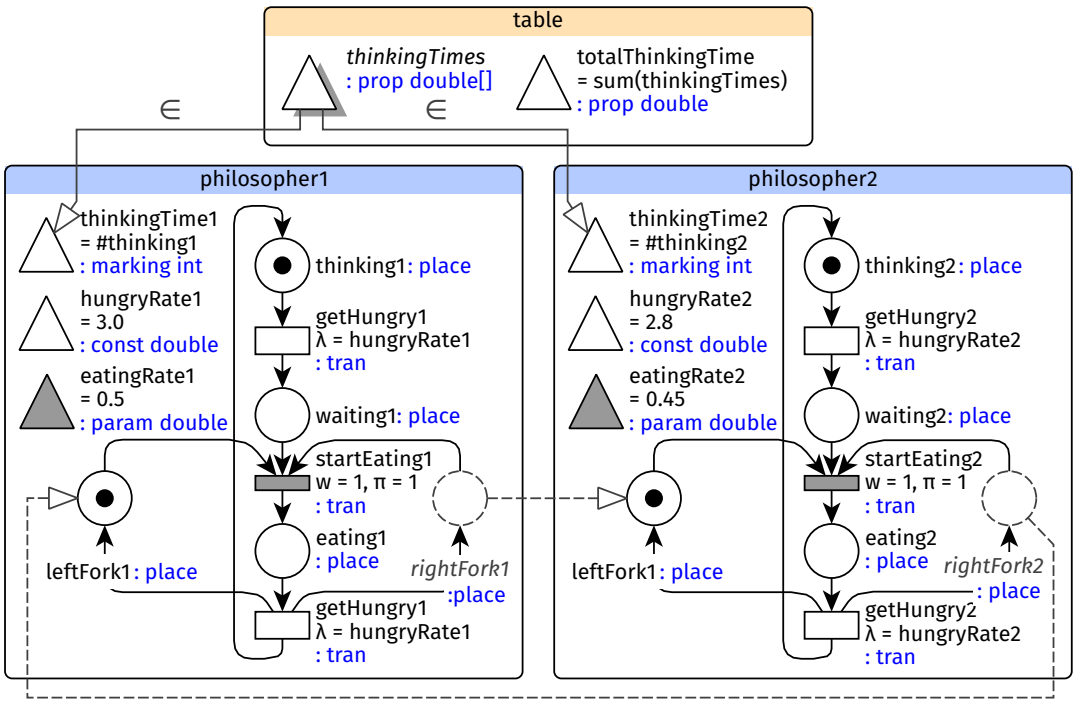


Figure 3.2 Example RGSPN with type annotations.

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, $\Sigma \vdash m_0(p) : \texttt{const int}$.
- For all timed transitions $t \in T_T$ the transition rate is a possibly marking- and parameter-dependent real number, $\Sigma \vdash \lambda(t) : \texttt{weight double}$.
- For all immediate transitions $t \in T_i$ the probability weight is a possibly marking- and parameter-dependent real number, $\Sigma \vdash w(t) : \texttt{weight double}$. The transition priority is typed much more conservatively by requiring an integer constant, such that $\Sigma \vdash \pi(t) : \texttt{const int}$ holds.
- For all variables $v \in V$ the value expression must match the type of the variable, $\Sigma \vdash \mathit{value}(v) : \mathit{type}(v)$.
- All arcs $p \overset{e}{\leftarrow} t$, $p \overset{e}{\rightarrow} t$ or $p \overset{e}{\multimap} t$ go between places and transitions such that $\mathit{type}(p) <: \texttt{place}$ and $\mathit{type}(t) <: \texttt{tran}$ holds. The inscription e may depend on the marking, $\Sigma \vdash e : \texttt{marking int}$,
- For all $r := s$, s is a compatible target of r, $\mathit{type}(s) <: \mathit{target}(r)$.
- For all $c \mathrel{+}= s$, s is a valid member of c, $\mathit{type}(s) <: \mathit{member}(s)$.

Remark 3.4 The requirements are based on the assumptions in GSPN and CTMC solution algorithms. While the inscriptions of arcs e are allowed to have dependence `marking`, because marking-dependent arcs may lead to simplifications of stochastic models (Ciardo and Trivedi, 1993). However, as some external tools only support arcs with constant inscriptions, $\Sigma \vdash e : \texttt{const int}$ 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 $\Sigma \vdash w(t) : \texttt{param double}$ 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 $\Sigma \vdash \pi(t) : \texttt{marking int}$.

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

Running example 3.5 Figure 3.2 on page 14 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 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:

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

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

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

3.3.1 Typing

A complete set of typing rules for $Expr_\Sigma$ is presented in Table 3.1, which described the relation $\_ \vdash \_ : \_$. The judgement $\Sigma \vdash e : \tau$ assigns a type τ to an expression $e \in Expr(\Sigma)$ in

Table 3.1 Typing rules for expressions.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

the 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.

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.[1] Formally, pretypes have the interpretations

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

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

[1] 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

	parameters	marking	CTMC	path	value
$[\![\texttt{const}\ \rho]\!]$:					$p \in [\![\rho]\!]$,
$[\![\texttt{param}\ \rho]\!]$:	θ				$\mapsto p \in [\![\rho]\!]$,
$[\![\texttt{marking}\ \rho]\!]$:		m			$\mapsto p \in [\![\rho]\!]$,
$[\![\texttt{weight}\ \rho]\!]$:	θ,	m			$\mapsto p \in [\![\rho]\!]$,
$[\![\texttt{prop}\ \rho]\!]$:	θ,	m,	CTMC		$\mapsto p \in [\![\rho]\!]$,
$[\![\texttt{path}\ \rho]\!]$:	θ,		CTMC,	Π	$\mapsto p \in [\![\rho]\!]$.

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 CTL* formulas this conversion is defined such that the first marking m_1 of the path $\Pi = m_1 \to m_2 \to \ldots$ serves as the current marking argument m of the `prop` computation when it is treated as a `path` property.

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.[2]

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

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

3.4 Reference inlining

3.4.1 Handling inconsistent models

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

Chapter 4

Incremental view synchronization

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.

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 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.
- The *Model Adaptation Pattern* constructs an exploration representation, such as a string of genes in genetic algorithms **[TODO: cite]**, 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.
- 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.
- 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 feasability 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 transformin 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 intepreted as information about the satisfaction of constraints and the values of goal functions defined over the engineering formalisms.

(Varró et al., 2017)

[TODO:]

5.1 Integration with design-space exploration toolchains

5.1.1 Design-space explorers

5.1.2 Formal analysis tools

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 (IDE). The plug-in is based on open source technologies from the Eclipse Modeling Project: the *Eclipse Modeling Foundation* (EMF), the *XText* framework for language engineering and *VIATRA* scalable reactive model queries and transformations.

[TODO: Cite the relevant technologies, or use URL footnotes?]

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 from the engineering DSL to stochastic Petri nets.

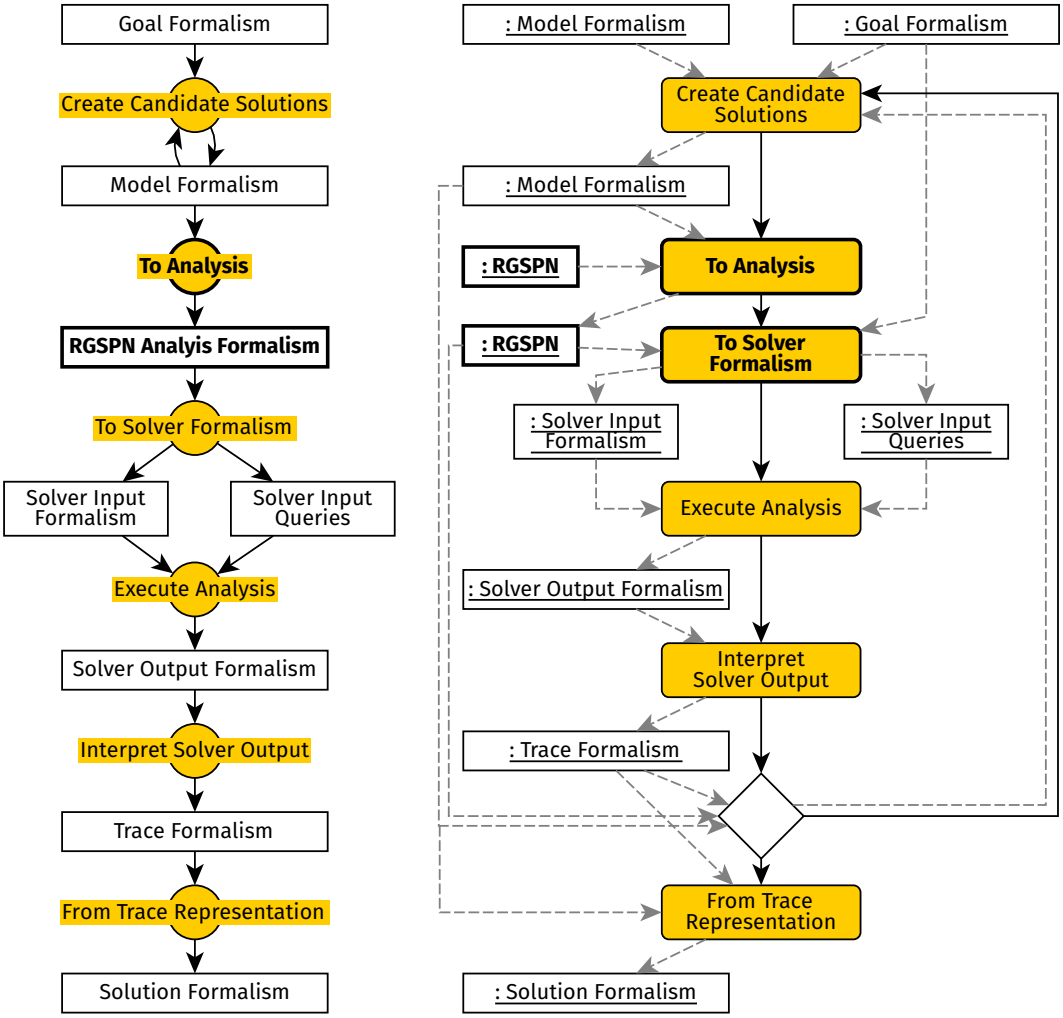


Figure 5.1 Formalism transformation graph and process model of the *Model Transformation* DSE pattern with RGSPN-based analysis. The components in *bold* were implemented in our work, while the rest of the components should be supplied by the DSE framework and stochastic analysis tool.

5.2.1 Specification environment

5.2.2 Transformation execution

5.3 Evaluation of incremental transformations

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

RQ1 How does the initial batch transformation from the engineering DSL to the formal

Table 5.1 Source model, abstract net and concrete net sizees 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

stochastic model (GSPN) scale with respect to size of the input model?

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

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

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

5.3.1 Measurement setup

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

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

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

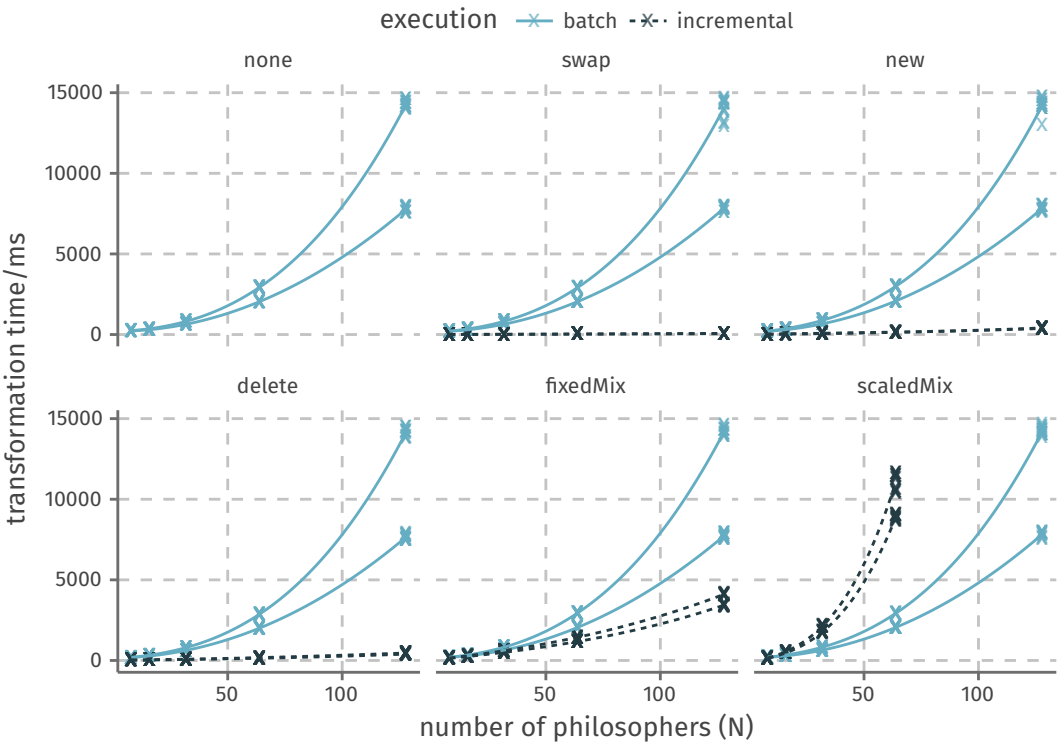


Figure 5.2 Execution times of transformations.

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

5.3.2 Results

The execution time of the transformations on the various model sizes and change operations is shown in the scatter plot in Figure 5.2. 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

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

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 5.2, 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 the construction of an appropriate RETE net **[TODO: cite?]** and the traversal of the source model to populate the base relations.

2. The *view transformation* phase fires the transformation rules on the VIATRA Event-driven Virtual Machine (EVM) to construct the abstract RGSPN model with references.

3. The *concretizer query* phase traverses the abstract net to prepare the precondition queries of the concretizer transformation.

4. The *concretizer transformation* phase is ran on the EVM to resolve references and inline expression in the abstract net to construct the concrete RGSPN target model.

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

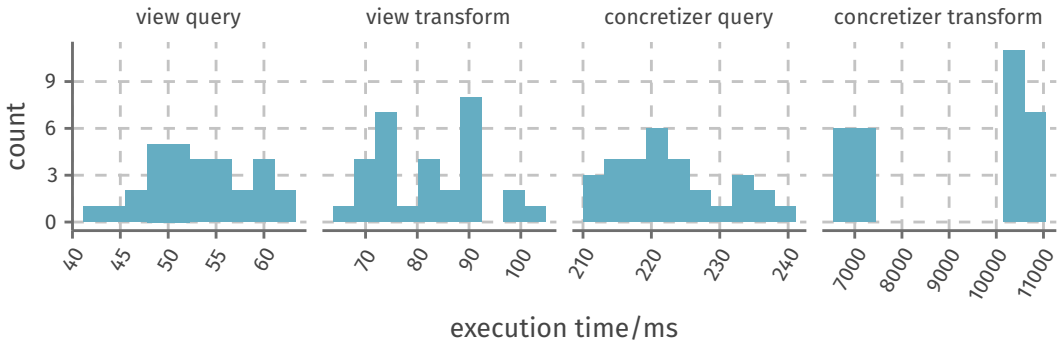
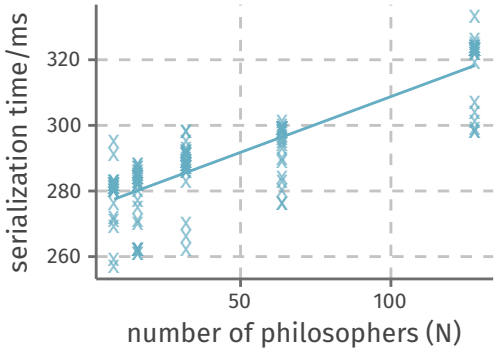


Figure 5.3 Execution times of batch transformation phases with $N = 128$ philosophers.

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

The histogram of the transformation phases with 30 iterations is shown in Figure 5.3. 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 significats 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 the creation and removal of objects has larger overhead than the modification of references in the source model and the transformed 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 hyphotesize 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 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 corrent 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 enviroment was also controlled manually to ensure that subsequent iterations did not interfere.

Despite these attemps, 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 insteand of being caused by interferences.

As we conducted our expriments 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

Babar, Junaid, Marco Beccuti, Susanna Donatelli, and Andrew S. Miner (2010). "GreatSPN Enhanced with Decision Diagram Data Structures". In: *PETRI NETS 2010*. Vol. 6128. LNCS. 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. Soft.* 82 (1), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066.

Blake, James T., Andrew L. Reibman, and Kishor S. Trivedi (1988). "Sensitivity analysis of reliability and performability measures for multiprocessor systems". In: vol. 16. 1. ACM, pp. 177–186. DOI: 10.1145/1007771.55616.

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 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.

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: *Dependable Systems & Networks*. IEEE. DOI: 10.1109/DSN.2009.5270318.

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.

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.

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.

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*. Vol. 1786. LNCS. 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.

Kang, Eunsuk, Ethan Jackson, and Wolfram Schulte (2010). "An Approach for Effective Design Space Exploration". In: *Monterey Workshop 2010*. Vol. 6662. LNCS. 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. 14. 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 2017*. Vol. 5606. LNCS, 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.

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.

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.

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

Murata, Tadao (1989). "Petri nets: Properties, analysis and applications". In: *Proc. IEEE* 77.4, pp. 541–580. DOI: 10.1109/5.24143.

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*. Vol. 9938. LNCS. 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). In: *EEF School 2000*. LNCS 2090. Springer, pp. 315–343. DOI: 10.1007/3-540-44667-2_9.

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.

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*. Vol. 1340. CEUR Workshop Proceedings, pp. 42–51. URL: http://ceur-ws.org/Vol-1340/paper6.pdf.

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

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

Vörös, András, Dániel Darvas, Ákos Hajdu, Attila Klenik, Kristóf Marussy, Vince Molnár, Tamás Bartha, and István Majzik (2017). “Industrial applications of the PetriDotNet modelling and analysis tool”. In: *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.