Constructing Dependability Analysis Models of Reconfigurable Production Systems*

Kristóf Marussy[1,2] and István Majzik[2]

Abstract— **Model-driven engineering methodologies are often used for the design and integration of the software and hardware components in automated production systems. Architecture models in general purpose or domain specific modeling languages allow the modular and systematic construction of formal analysis models for the evaluation of dependability and performability. The on-line reconfiguration and fault handling actions in the system introduce changes in the architecture and parameters, and thus result in a multi-phased operation. We propose a mission automaton formalism to describe reconfigurations, fault handling and parameter changes over architecture models. By maintaining the architecture model, which is modified by the reconfigurations, along with the corresponding analysis models, a stochastic phased-mission system (PMS) is obtained and solved for dependability.**

I. Introduction

Model-driven engineering in manufacturing system design [1] is typically used to capture system architectures, behavior and possible reconfigurations. However, the extra-functional requirements, such as dependability (reliability, availability), as well as performance concerns (including timing and resource utilization) remain challenging to address [2], especially in the context of changing and evolving architectures [3], [4].

The viewpoints of dependability and performability are concerned with the probabilistic behaviors of systems, and often need formal stochastic models for rigorous analysis. Automated derivation of stochastic models from architecture models by model transformations [5], [6], [7] can integrate such analyses into multi-paradigm workflows.

In this work, we propose an approach to capture dynamic reconfigurations, fault handling and parameter changes on the level of the architecture model. To do this, we define a mission automaton formalism by extending Graph Transformation Abstract State Machines (GT+ASM) [8] with stochastic and timing properties. GT+ASM leverages graph pattern matching, which is the technique applied on the architecture model for the description of reconfigurations and fault handling. The changes described by the mission automaton result in multiple, non-overlapping phases of operation, that can be captured by stochastic modeling of phased-mission systems (PMS) [9]. We also define the (graph transformation based) construction of PMS analysis model from the architecture model and its evolution described by the mission automaton.

The PMS construction workflow is illustrated in Fig. 1. Domain modeling experts can construct or reuse the architecture modeling language, the graph patterns used for specifying changes, as well as the architecture models themselves. The modeling language is extended with run-time attributes to describe stochastic behaviors, such as failures and performance related events. Stochastic modeling experts specify the analysis model transformation in terms of Generalized Stochastic Petri Nets (GPSN) [10], which are especially amenable for automated, modular construction [11] and allow leveraging existing transformations, e.g. [5]. After the domain modeling expert specifies the mission automaton, its state space is automatically explored and GSPN models of the phases of operation are derived for PMS analysis.

We implemented a PMS analysis model construction tool based on change-driven target incremental execution [12], [13] of model transformations. Therefore PMSs even with a large number of phases can be created efficiently. We demonstrate our approach with a running example and case study of a flexible manufacturing system.

II. Preliminaries

A. Architecture models and metamodeling

In model-driven engineering graphs are used as formal descriptions of models, including UML, SysML and AADL artifacts [14], such as production system architectures.

Metamodels explicitly describe the abstract syntax of modeling languages, including the classes, references and attributes that comprise the language. An architecture model is an *instance model* of the architecture modeling language.

The Eclipse Modeling Framework (EMF) [15] is a defacto standard metamodeling technology, which we used in the implementation part of our work.

Example 1: The class diagram in Fig. 2 shows a metamodel for flexible manufacturing system (FMS) architectures based on the FMS performance model by Ciardo and Trivedi [16], which will serve as a running example throughout this paper.

The FMS contains *Machines*, which have *Capabilities* to perform *Tasks*. A *Task* takes *inputs* from a *Stockpile* to produce an *output*. *Machines* have reliability attributes *mttf* and *repairTime*, while *Capabilities* contain the *executionTime* of the *Task*. *Tasks* may be marked with *HIGH importance*, and the *initialSize* of a *Stockpile* can be specified.

*This paper is supported by the MTA-BME Cyber-Physical Systems Research Group and by the EFOP-3.6.2-16-2017-00013 grant of the European Union, co-financed by the European Social Fund.

[1]Kristóf Marussy is with MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

[2]The authors are with Budapest University of Technology and Economics, Department of Measurement and Information Systems, Budapest, Hungary {marussy, majzik}@mit.bme.hu

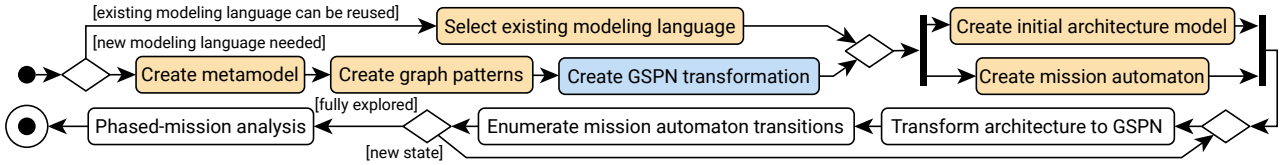

Fig. 1. Dependability analysis using mission automata, with modeling expert (orange), stochastic analysis expert (blue) and automated (white) activities.

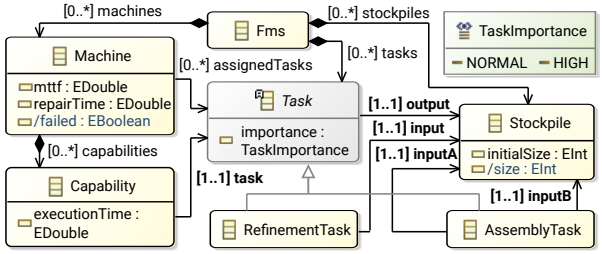

Fig. 2. Class diagram for the FMS architecture metamodel.

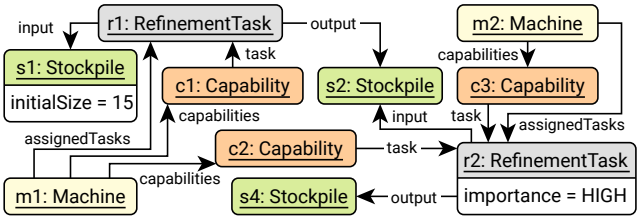

Fig. 3. Example FMS architecture instance model.

Run-time adaptation and reconfiguration may happen in response to changes in the *failed* states of machines and the *size* of *Stockpiles*. *Machines* may be added or removed, as well as *Tasks* may be *assigned* to *Machines*.

An example instance model, which is represented as an object graph, is shown in Fig. 3.

B. Graph patterns

State-of-the-art modeling toolchains often rely on model queries to retrieve fragments of interest from model, to specify model to model and model to text transformations, as well as to validate well-formedness constraints on models [17].

A *graph pattern* is a graph-like structure that represents a condition matched against an instance model. It can prescribe *structural* and *attribute constraints,* as well as *negative application conditions* on the pattern match.

Parameter variables match distinguished objects inside a pattern. Let $\mathcal{O}(M)$ denote the set of objects in an instance model M. The tuple $\langle o_1, \ldots, o_k \rangle \in \mathcal{O}(M)^k$ is a *match argument tuple* of the k-argument graph pattern $\phi(x_i, \ldots, x_k)$, written as $M \vDash \phi(o_1, \ldots, o_k)$, if ϕ matches M after binding each x_i inside ϕ to the corresponding o_i.

Example 2: Fig. 4 shows the graph pattern *qCanReplace*. A match argument tuple $\langle M, R, C, T \rangle$ expresses that the *Machine* R can replace M in performing the high-importance *Task* T, because it is currently not assigned to T or any other high-importance *Task*, but it has *Capability* C to perform T.

In Fig. 3, we have $\vDash$ *qCanReplace*$(m2, m1, c3, a1)$.

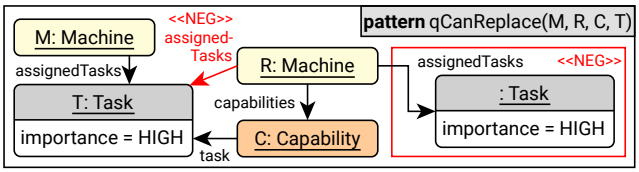

Fig. 4. Example graph pattern.

C. Generalized Stochastic Petri Nets

Generalized Stochastic Petri Nets (GSPN) are a commonly used formalisms for the dependability evaluation of asynchronous systems. Formally, they are directed bipartite graphs with a set of places P and transitions T [10].

A *marking* $m\colon P \to \mathbb{N}$ assigns *token* counts to the places. Starting from the initial marking m_0, if enough tokens are available at its input arcs, and no transition with higher priority is fireable, a transition may be *fired* to remove tokens from its input places and put tokens to its output places.

A continuous-time Markov chain (CTMC) represents the stochastic behaviors of the GSPN. *Timed* transitions are fired when an exponentially distributed delay with rate parameter $\lambda(t)$ has elapsed, while *immediate* transitions are fired immediately according to their priority $\Pi(t)$ and probability weight $\lambda(t)$ when they become enabled.

D. Phased-mission systems

Phased-mission systems are characterized by consecutive phases of operation caused by changes in system configuration or environment [18]. Modeling and analysis of PMSs are made more complex than single phased systems by the history of the system, such as degradation of the components, affecting subsequently occurring phases.

In state-based stochastic PMS models, each phase is described by a *lower level model*. The *upper level model* determines the length of each phase and the possible phase transitions. In order to propagate the history of the system, phase transitions map states of the lower level model associated with the source phase to the target phase.

III. Automated Analysis Model Construction

A. Dependability analysis models for static architectures

First we consider the construction of analysis models for failure processes of unchanging architectures, which will be called the *static* analysis model transformation. Creation of the *dynamic* analysis models that incorporate reconfigurations of the architecture models calls the static transformation as a subroutine.

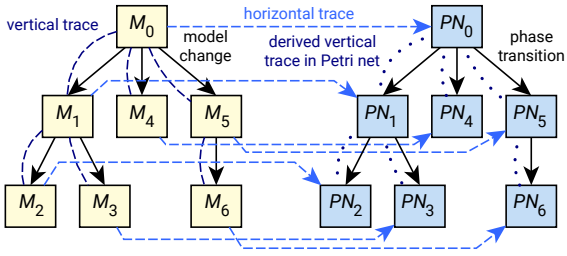

Fig. 5. Traceability relationships of the architecture and analysis models.

In case of architecture models, the typical approach is a modular transformation where patterns from the architecture model are systematically mapped (based on the types of elements) to interconnected model fragments in the analysis model. This process is facilitated by modular and compositional extensions to Petri nets [11]. For example, the modular Petri nets formalism [19] allows the assembly of large models by instantiating and connecting net fragments.

Model transformations tools, such as [20], [13], [21], construct target (right-side) models according to matches of precondition patterns in the source (left-side) models. The left side of a single transformation rule is a *precondition* graph pattern ϕ. The right side is a template for target model objects, in our case, a Petri net fragment [19]. For each match argument tuple $\langle x_1, \ldots, x_k \rangle$ of ϕ its right side is instantiated by adding a copy of it to the target model.

Traceability information relates the source and the target instance models. The *horizontal trace* hyperedges connect the objects of the match argument tuples on the left to the target model objects on the right.

By adding *local names* to objects within Petri net fragment templates, target model elements can be referenced through horizontal traces: $\phi(x_1, \ldots, x_k).p_j$ unambiguously identifies the place with local name p_j in the fragment instance associated with the rule ϕ and match arguments $\langle x_1, \ldots, x_k \rangle$.

B. Handling architecture model changes

No we look at architecture model changes that produce a new model M_2 from the original architecture M_1.

1) Vertical trace: The *vertical trace* relation $\sim$: $\mathcal{O}(M_1) \times \mathcal{O}(M_2)$ between the objects of M_1 and M_2 describes the objects preserved, created and removed. If $x \sim y$ holds, the modification preserved x as y in M_2. Created objects y have no corresponding x, while removed objects x have no y.

Given horizontal traceability information for M_1, M_2, we can derive vertical traces for their analysis models PN_1, PN_2. Let $\phi(x_1, \ldots, x_k).p \sim_{PN} \phi(y_1, \ldots, y_k).p$ when $x_i \sim_M y_i$ $(i = 1, \ldots, k)$, where $\sim_M$ and $\sim_{PN}$ are the vertical traces of the architecture and analysis models, respectively. That is, a Petri net node is preserved if the match arguments of the responsible rule activation were preserved by the architectural change.

Horizontal and vertical traces are illustrated in Fig. 5. From the starting architecture M_0 and its analysis net PN_0 several other models are created by architectural changes. A vertical trace relation accompanies each modification. Each architecture M_i is connected to its Petri net PN_i by the horizontal trace hyperedges. The vertical trace between each adjacent PN_i, PN_j can be derived from the vertical trace between M_i, M_j and the corresponding horizontal traces.

2) Maintenance of markings: Along with the stochastic Petri net describing the failure processes of the production system, we also maintain the original architecture model so that it can be used to specify reconfigurations. Hence the states of the system are represented by pairs $\langle M_i, m_j \rangle$ of architectures M_i and markings m_j of the corresponding PN_i.

When the architecture changes add or remove objects, the restriction of $\sim_M$ to $\mathcal{O}(M) \cap \mathcal{O}(M')$, and hence the restriction of $\sim_{PN}$ to $PN \cap PN'$ is invertible. Upon such architecture change $M \mapsto M'$, the state change $\langle M, m \rangle \mapsto \langle M', m' \rangle$ updates the marking as follows: Tokens are *preserved* along the derived vertical traceability, i.e. $m'(p') = m(p)$, where $p \sim_{PN} p'$. *New places* q of PN' with no corresponding place in PN are set to their initial marking $m(q) = m_0(q)$ instead. Tokens on *deleted places* are lost.

While architecture element splits and merges are unusual in object-oriented modeling, they can be also handled. Upon a *split* with $p \sim_{PN} q_1$ and $p \sim_{PN} q_2$ $(q_1 \neq q_2)$, the tokens $m(p) = m'(q_1) = m'(q_2)$ are copied. *Merges* $p_1 \sim_{PN} q$, $p_2 \sim_{PN} q$ are only possible when $m(p_1) = m(p_2)$ so that $m'(q) = m(p_1) = m(p_2)$ is well-defined.

C. Augmenting architecture models with run-time features

For capturing the effects of run-time events, especially failures and execution of tasks, the *static* architecture model is extended by so-called *run-time attributes*, which are computed from the markings of the Petri net fragments that represent these events.

Precisely, run-time attributes are computed from the marking m_j in each system state $\langle M_i, m_j \rangle$ and are affixed to the objects of the architecture model M_i. Reconfiguration strategies can be formulated without knowing the details of the analysis model and the related transformation. Instead only the architecture model and its run-time extensions are accessed, promoting information hiding and modularization.

Observable run-time attributes are read-only attributes computed from the marking of the Petri net. In contrast, *controllable* run-time attributes can be modified by reconfigurations, causing changes in the Petri net marking. For simplicity, a controllable attribute must correspond to the number of tokens on a single place p, so that updating the attribute updates $m(p)$ directly.

To prevent the change of the Petri net marking from triggering the construction of a new analysis model, graph patterns may not depend on run-time attributes.

Example 3: The analysis model for the architecture in Fig. 3 is shown in Fig. 6.a. The static transformation incorporates failure models for machines into the FMS model adapted from [16]. The dashed rectangles correspond to the Petri net fragment instances and their horizontal traces.

The marking of *items* (for a stockpile) is the initial marking is the *initialSize* of the *Stockpile*. The timed transitions *deliver* and *putBack* model pallets which move work

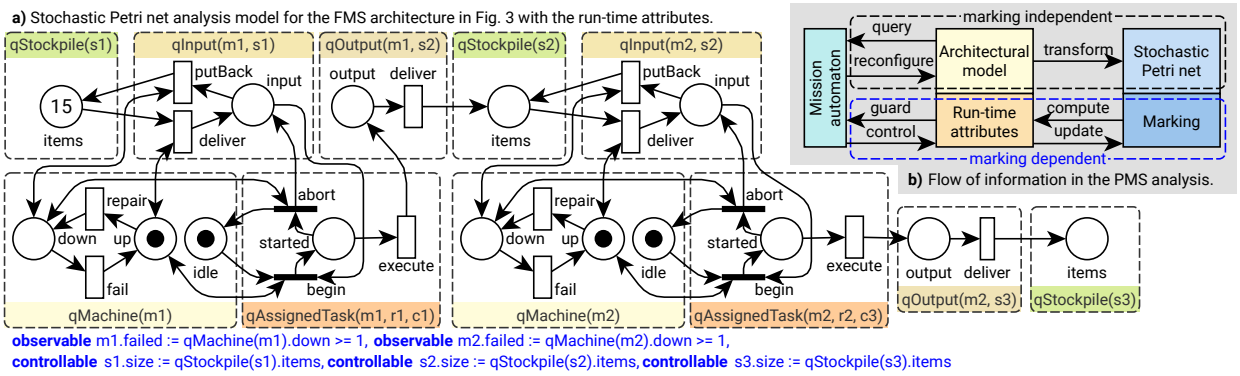

Fig. 6. a) Example Petri net analysis model for our running example and b) flow of information in the PMS analysis.

items between stockpiles and machines, while the immediate transitions *begin* and *abort* are responsible for starting tasks and aborting them when a fault occurs.

IV. Mission Automata

In-place graph transformation rules (GT) controlled by an abstract state machine (ASM) were proposed [8] as a mathematically precise description of model changes. In this section we define a stochastic and timed variant of GT+ASM for reconfigurations of automated production systems.

Informally, the *mission automaton* is a GT+ASM running along the stochastic Petri net analysis model. Transitions in the automaton may be triggered by changes of run-time attributes in the analysis model or by the elapsing time. Actions attached to transitions may reconfigure the architecture model, as well as update a set of global variables.

Fig. 6.b illustrates the data flow between the mission automaton, the architecture models and the stochastic Petri nets. In the mission automaton, interactions between the static elements and run-time attributes of the architecture model are avoided by forbidding access to the run-time attributes in actions that modify the static architecture elements. Instead, as described in the next section, a specific run-time attribute update action is offered with limited control flow.

Therefore, the parts of mission automaton that depend solely on the static architecture model are separated from those that also depend on run-time attributes, and hence the Petri net marking. Thus the state space of the mission automaton can be overapproximated without exploring the state spaces of the derived Petri nets. State space and probability distribution handling is delegated to a PMS analysis tool.

A. Formal definition

A *mission automaton* is a 5-tuple $\langle L, \ell_0, F, G, T \rangle$, where L is the set of *locations,* $\ell_0 \in L$ is the *initial location,* $F \subseteq L$ is the set of *final locations,* G is the set of *global variables,* and T is the set of *transitions*. A transition $\langle \ell_1, \ell_2, V, \phi, \underline{x}, \tau, \underline{\alpha} \rangle \in T$ from $\ell_1 \in L$ to $\ell_2 \in L$ is equipped with a set of local variables V, a k-parameter *precondition pattern (guard)* ϕ, a list of *parameters* $\underline{x}$, a *trigger* τ and list of *actions* $\underline{\alpha}$. The parameters $\underline{x} = \langle x_1, \ldots, x_k \rangle$ are global or local variables ($x_i \in G \cup V$), such that each local variable $v \in V$ appears at least once in $\underline{x}$. Variables will be bound to objects of the architecture model.

1) Expressions: Algebraic and logical *expressions* can be formed using the attributes of objects $\mathcal{O}(M)$ pointed by the variables. We do not specify syntax and semantics for the expressions other than the attribute references $x.a$, where a is an attribute from the architectural metamodel and $x \in G \cup V$.

An expression is *run-time attribute dependent* if it contains an attribute reference $x.a$ where a is a run-time attribute.

2) Triggers: The trigger τ may be a *a) state-based* trigger "**on** e **weight** w", where e and w are (possibly run-time attribute, and thus marking dependent) Boolean and *weight* expressions, respectively; or a *b) timed* trigger "**after** d", where the delay expression d is not dependent on the value of any run-time attribute.

3) Actions: Each action α_i of $\underline{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle$ is either a *a) variable update* "$g := x$", where $g \in G$ and $x \in G \cup V$; an *b) attribute update* "$x.a := e$", where $x \in G \cup V$, a is a controllable attribute and e is a (possibly run-time attribute dependent) *value* expression; or a *c) model manipulation action.* Model manipulation actions modify the architecture model by adding or removing objects or references, as well as changing static attributes. Due to space constraints, we refer to an an existing *host language*[1] for semantics model mutation, as well as algebraic graph transformations for a precise mathematical foundation [22], [8].

Example 4: Fig. 7 shows a mission automaton for FMS reconfiguration, which showcases the expressiveness of our formalism. Transitions are denoted as $\tau\,[\phi(\underline{x})]/\underline{\alpha}$, where the guard is omitted if it is the trivial graph pattern (which has 0 parameters and holds always).

The global variables *Mat* and *Prod* should be initialized to the stockpiles of raw materials and final products, respectively. When the FMS completes 7 products, the automaton moves to the *rampedUp* location from *initial*, and a model manipulation action adds 2 more machines. However, if some machine M responsible for a *HIGH* importance task T fails before ramping up, the transition to *replaced* is taken instead. The pattern *qCanReplace* (Fig. 4) also binds local

[1]The Xbase language and the Xtext framework (`https://www.eclipse.org/xtend/`) were used as the hosts in our implementation.

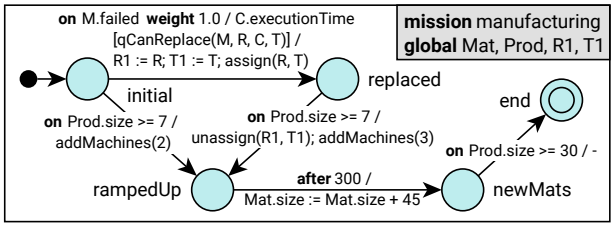

Fig. 7. Example mission automaton.

variables to the replacement machine R and its capability C, so we can give faster machines larger weight among multiple possible replacements. R and T are saved to global variables *R1* and *T1*, and a model manipulation action assigns T to R. When ramping up, this assignment is removed, and 3 more machines are added. 300 time units later, 45 more raw materials are added by an attribute update action. The final location *end* is reached after the FMS completes 30 products.

B. Phased-mission analysis

Analysis of architecture models and mission automata is performed in two steps. Firstly, the mission automaton is *unfolded* by taking into account the potential instantiations of its transitions, as well as the modifications of the architecture model instance. The unfolded mission automaton and architecture model configurations form a tree (Fig. 5).

The (indirectly) marking dependent behavior of the mission automaton is overapproximated by ignoring triggers in the automaton. Each mission automaton transition is considered fireable, regardless of the reachable markings of the stochastic Petri net analysis model.

Dependence on the static and run-time parts of the architecture models is *erased* from the triggers and actions by substituting architectural concepts with their Petri net representations in the analysis model.

Secondly, the tree of architecture configurations is turned into a PMS for analysis. The *upper level* model is the unfolded mission automaton, where transitions between phases of operation are governed by the trigger expressions. The *lower level* models are the stochastic Petri nets that describe the behavior of the system during a phase. As the mission completes successfully upon reaching a final location, the corresponding phases are marked in the upper level model.

The hierarchical methodology proposed by Mura and Bondavalli [9], [23] can be adapted for the PMS analysis. We direct the reader to our report [24] for technical details.

V. Evaluation

We implemented a domain-specific language for mission automata, as well as the mission automaton unfolding and erasure algorithm from Section IV-B, using the VIATRA incremental transformation engine [8], [13], [17] for EMF models and the Xtext language engineering framework.

We evaluate the scalability of our analysis model construction approach in the context of incremental analysis model transformations. Reachable configurations are explored during the unfolding in a depth-first manner. Exploiting the transaction support of EMF both forward moves and backtracking modify the architecture model in-place. Hence the static transformation, which is also powered by VIATRA, can be executed in a change-driven style.

TABLE I
Measurement results.

	initial phase		total phases			
N	$\lvert PN_0 \rvert$	time/ms	phases	$\sum_i \lvert PN_i \rvert$	time/ms	ph.t./ms
1	41	186	8	415	303	16.6
2	78	185	20	1818	426	12.7
4	152	213	68	11 278	1022	12.1
6	226	247	148	35 530	2150	12.9
8	300	281	260	81 678	3955	14.2
10	374	319	404	156 826	6267	14.8
12	448	353	580	268 078	9346	15.5
14	522	390	788	422 538	13 729	16.9
16	596	426	1028	627 310	19 481	18.6

Setup: The mission automaton from Fig. 7 was unfolded with scaled versions of Fig. 3, such that the initial architecture contains N copies of the machines *m1* and *m2*, respectively. The query *qCanReplace* matches N^2 pairs of machines, thereby increasing the number of phases.

The experiments were ran on a computer with an Intel Core i7-5700HQ 2.7 GHz CPU with 4 GiB of heap space reserved for the Java 1.8.0u144 virtual machine. Each experiment was repeated 30 times after 10 warm-up repetitions.

Results: Table I presents the number of elements in the Petri nets obtained by unfolding, as well as the median running times. We report the running time of the transformation of the initial phase, including the execution of the static transformation in batch mode. We also report the total running time, and the average incremental execution time for the non-initial phases (ph.t.).

Discussion: The sizes of the analysis models for a single phase grew linearly as machines were added to the architecture, while the number of phases in the upper level model grew quadratically, along with the execution time of the full PMS construction. The unfolding could take advantage of incremental execution of the static transformation. Thus no more than 20 ms per non-initial analysis model was taken. Total execution time remained below 20 seconds.

VI. Related work

Methods for the construction of stochastic analysis models are widespread in the evaluation of architecture models, especially for component-based design [6], [7]. The Palladio component model [25] was suggested as a specialized modeling language for performance prediction. Recently, the Renew metamodeling framework [26] was proposed for adding Petri net semantics to modeling languages. In [27] coevolution of architectures and fault trees were investigated.

A dependability-driven design methodology for embedded systems was suggested in [28], while [29], [4] investigated adaptation for dependability by agent-based planning.

The two main groups of dependability analysis techniques for phased-mission systems are state-space based and combinatorial approaches [30]. State-space based analysis

may happen by solving stochastic models for individual phases and propagating the results [18], [9], as well as with deterministic and stochastic Petri nets models [23]. Combinatorial approaches can take advantage of efficient data structures [31], [30], but may require more restricted models, such as fault trees or block diagrams.

Combinations of CTMCs were investigated with timed and hybrid automata [32], [33], [34]. While these approaches are significantly more expressive for linear-time properties than ours, they do not support the reconfiguration of the model.

VII. Conclusions and future work

We presented 1) a mission automaton formalism for specifying reconfigurations and fault handling in production system architectures and 2) a mapping from these to stochastic PMS analysis models. The dependence between the phase transitions in the mission and the events represented in the analysis model is abstracted by incorporating run-time attributes into the architecture model. According to our empirical evaluation, good scalability can be provided by our incremental analysis model construction.

Possible extensions include relaxing the separation between the static and the run-time attribute dependent parts of the mission automata, for example, by adopting stochastic timed automata model checking techniques [33]. Graph-based state encoding [35] during mission unfolding may reduce the PMS from a tree to a directed graph, potentially decreasing the number of phases to be analyzed by merging equivalent phases. Support for more advanced transformation chains [12] could increase applicability in complex multi-paradigm modeling scenarios.

Acknowledgment. We would like to thank Gábor Bergmann and András Vörös for their insightful discussions and comments.

References

[1] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Assessing the state-of-practice of model-based engineering in the embedded systems domain," pp. 166–182, 2014.

[2] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *J. Syst. Softw.*, vol. 110, pp. 54–84, 2015.

[3] M. Felici, "Taxonomy of evolution and dependability," in *Proc. 2nd Workshop Unanticip. Softw. Evol.*, pp. 95–104.

[4] S. Rehberger, L. Spreiter, and B. Vogel-Heuser, "An agent-based approach for dependable planning of production sequences in automated production systems," *Automatisierungstechnik*, vol. 65, no. 11, pp. 766–778, 2017.

[5] A. Bondavalli, I. Majzik, and I. Mura, "Automatic dependability analysis for supporting design decisions in UML," in *Proc. 4th IEEE Int. Symp. High-Assur. Syst. Eng.* IEEE, 1999.

[6] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, 2010.

[7] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and analysis of software systems specified with UML," *ACM Comput. Surv.*, vol. 45, no. 1, 2012.

[8] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Sci. Comput. Program.*, vol. 68, no. 3, 2007.

[9] I. Mura and A. Bondavalli, "Hierarchical modeling and evaluation of phased-mission systems," *IEEE Trans. Rel.*, vol. 48, no. 4, pp. 360–368, 1999.

[10] M. A. Marsan, G. Conte, and G. Balbo, "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems," *ACM Trans. Comput. Syst.*, vol. 2, no. 2, 1984.

[11] A. Marechal and D. Buchs, "Generalizing the compositions of Petri nets modules," *Fundam. Inform.*, vol. 137, no. 1, pp. 87–116, 2015.

[12] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, 2006.

[13] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró, "VIATRA 3: A reactive model transformation platform," in *ICMT 2015*, ser. LNCS, vol. 9152. Springer, 2015, pp. 101–110.

[14] J. Bézivin and O. Gerbé, "Towards a precise definition of the OMG/MDA framework," in *Proc. 16th IEEE Int. Conf. Autom. Softw. Eng.* IEEE, 2001.

[15] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison-Wesley Prof., 2009.

[16] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic reward net models," *Perform. Eval.*, vol. 18, no. 1, pp. 37–59, 1993.

[17] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, "EMF-IncQuery: An integrated development environment for live model queries," *Sci. Comput. Program.*, vol. 98, no. 1, pp. 80–99, Feb. 2015.

[18] A. K. Somani, J. A. Ritcey, and S. H. L. Au, "Computationally-efficient phased-mission reliability analysis for systems with variable configurations," *IEEE Trans. Rel.*, vol. 41, no. 4, pp. 504–511, 1992.

[19] E. Kindler and L. Petrucci, "Towards a standard for modular Petri nets: A formalisation," in *PETRI NETS 2009*, ser. LNCS, vol. 5606. Springer, 2009, pp. 43–62.

[20] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1–2, pp. 31–39, 2008.

[21] *MOF Query/View/Transformation Specification*, Object Management Group Std., Rev. 1.3.

[22] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[23] I. Mura and A. Bondavalli, "Markov regenerative stochastic Petri nets to model and evaluate phased mission systems dependability," *IEEE Trans. Comput.*, vol. 50, no. 12, pp. 1337–1351, 2001.

[24] K. Marussy and I. Majzik, "Constructing phased-mission systems for dependability analysis of reconfigurable production systems," Tech. Rep., 2018. [Online]. Available: http://doi.org/10.5281/zenodo.1290661

[25] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, 2009.

[26] D. Mosteller, L. Cabac, and M. Haustermann, "Integrating Petri net semantics in a model-driven approach: The Renew meta-modeling and transformation framework," in *TOPNOC XI*, ser. LNCS. Springer, 2016, vol. 9930, pp. 92–113.

[27] S. Getir, L. Grunske, A. van Hoorn, T. Kehrer, Y. Noller, and M. Tichy, "Supporting semi-automatic co-evolution of architecture and fault tree models," *J. Syst. Softw.*, vol. 142, pp. 115–135, 2018.

[28] N. Suri, A. Jhumka, M. Hiller, A. Pataricza, S. Islam, and C. Sârbu, "A software integration approach for designing and assessing dependable embedded systems," *J. Syst. Softw.*, vol. 83, pp. 1780–1800, 2010.

[29] X.-L. Hoang, A. Fay, P. Marks, and M. Weyrich, "Generation and impact analysis of adaptation options for automated manufacturing machines," in *22nd IEEE Int. Conf. Emerg. Technol. Fact. Autom.* IEEE, 2017.

[30] D. Wang and K. S. Trivedi, "Reliability analysis of phased-mission system with independent component repairs," *IEEE Trans. Rel.*, vol. 56, no. 3, pp. 540–551, 2007.

[31] L. Xing, "Reliability evaluation of phased-mission systems with imperfect fault coverage and common-cause failures," *IEEE Trans. Rel.*, vol. 56, no. 1, pp. 58–68, 2007.

[32] S. Donatelli, S. Haddad, and J. Sproston, "Model checking timed and stochastic properties with CSL^{TA}," *IEEE Trans. Softw. Eng.*, vol. 35, no. 2, pp. 224–240, 2009.

[33] T. Chen, T. Han, J. P. Katoen, and A. Mereacre, "Model checking of continuous-time Markov chains against timed automata specifications," *Log. Method. Comput. Sci.*, vol. 7, no. 1, 2011.

[34] P. Ballarini, B. Barbot, M. Duflot, S. Haddad, and N. Pekergin, "HASL: A new approach for performance evaluation and model checking from concepts to experimentation," *Perform. Eval.*, vol. 90, pp. 53–77, 2015.

[35] Á. Hegedüs, Á. Horváth, and D. Varró, "A model-driven framework for guided design space exploration," *Autom. Softw. Eng.*, vol. 22, no. 3, pp. 399–436, 2013.