Availability-driven Architectural Change Propagation through Bidirectional Model Transformations between UML and Petri Net Models

Vittorio Cortellessa, Romina Eramo, Michele Tucci
DISIM, University of L'Aquila, Italy
{*vittorio.cortellessa, romina.eramo, michele.tucci*}*@univaq.it*

***Abstract*—Software architecture is nowadays subject to frequent changes due to multiple reasons, such as evolution induced by new requirements. Architectural changes driven by non-functional requirements are particularly difficult to identify, because they attain quantitative analyses that are usually carried out with specific languages and tools. A considerable number of approaches, based on model transformations, have been proposed in the last decades to derive non-functional models from software architectural descriptions. However, there is a clear lack of automation in the backward path that brings the analysis results back to the software architecture. In this paper we address this problem in the context of software availability. We introduce a bidirectional model transformation between UML State Machines (SM), annotated with availability properties, and Generalized Stochastic Petri Nets (GSPN). Such transformation, implemented in the JTL language, is used both to derive a GSPN-based availability model from a SM-based software architecture and, after the analysis, to propagate back on the SM the changes carried out on the GSPN. We demonstrate the effectiveness of our approach on an Environmental Control System to which we apply well-known fault tolerance patterns aimed at improving its software availability.**

I. Introduction

The lifecycle of a software architecture spans overall the software development process, because it represents the earliest working artefact that evolves along the process, even after the software release. A software architecture can support different tasks, like test case generation [1], traceability [2], non-functional validation [3], etc. Hence, the architecture is subject to changes induced by decisions taken along the process [4].

Architectural changes driven by non-functional requirements are particularly difficult to identify, because they attain quantitative analyses that often cannot take place in the same context (i.e., languages and tools) where the architecture is designed. For example, very few ADLs embed constructs that allow to specify performance attributes, and even fewer ones are equipped with solvers leading performance indices out of an architecture specification.

Hence, in order to validate non-functional requirements on a software architecture, a considerable number of approaches, mostly based on model transformations, have been proposed in the last decades to generate non-functional models from software architectural descriptions [5], [6]. This generation step is also called *forward path*, and it is represented by the topmost steps of Fig. 1. However, the solution of generated models does not necessarily produce indices that satisfy the requirements, thus an iterative process is often required to modify/refactor the generated model on the basis of solution results. This process (hopefully) ends up when satisfactory indices are produced, and it is represented by the rightmost step of Fig. 1.

Thereafter, changes applied to non-functional models for sake of requirement satisfaction have to be propagated back to the software architecture, and this is represented by the bottommost step of Fig. 1, also called *backward path*. However, analyses results do not straightforwardly suggest what changes have to be made on the software architecture, hence this propagation is often based on the ability of experts that interpret the results. This clear lack of automation in the backward path represents a heavy limitation towards the construction of a round-trip process for non-functional validation of a software architecture.

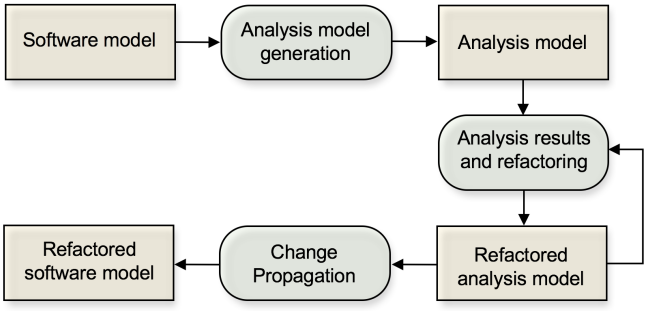


Figure 1: Round-trip non-functional analysis process

In this paper we work to introduce a first step to investigate this limitation in the context of availability at the architectural level that, in the last decade, has gained high interest from the research community (e.g., [7], [8]).

The basic idea beyond our approach is to exploit the power of bidirectional model transformations [9]. They are based on transformation languages that, once defined a transformation from a source to a target metamodel, with the help

This research was supported by the ECSEL-JU through the MegaMart2 project (grant agreement No 737494).

of traceability links, enable the inverse transformation from target to source metamodel. We introduce a forward path based on a bidirectional transformation between two specific modeling notations (i.e., one for architectural modeling and the other one for availability modeling), so that the backward path simply reduces to apply the inverse transformation to the availability model that satisfies the requirements. The changes made during the refactoring driven by analysis results are thus propagated back to the software architecture.

The bidirectional model transformation that we introduce in this paper works between UML State Machines (SM), annotated with availability properties, and Generalized Stochastic Petri Nets (GSPN). Such transformation, implemented in the JTL language, is thus used both to derive a GSPN availability model from a SM software architecture and, after the analysis, to propagate back on the UML SM model the changes made on the GSPN model. Such propagation consists in the generation of (possibly multiple) architectural refactoring solutions. Hence, we start investigating the suitability of bidirectional transformations in this context to progress towards a round-trip process for the assessment of software availability.

The rest of the paper is organized as follows: Sect. II introduces some background information on model-based availability analysis. Related work is presented in Sect. III. Sect. IV describes the proposed approach and its implementation. In Sect. V, we experiment our approach by applying a complete round-trip process for availability assessment on an example model. Finally, Sect. VI concludes the paper.

II. Model-based Availability Analysis

Availability can be simply described as the system readiness for providing correct service. It corresponds to the probability that the system is working within its specifications at a given instant [10]. In particular, the steady state availability can be expressed as the ratio between the value of MTTF (Mean Time To Failure) and the sum of MTTF and MTTR (Mean Time To Repair) values.

Stochastic Petri Nets (SPN) are a well-established formalism for modeling systems availability. In this paper, we consider an extension of SPN, called Generalized Stochastic Petri Nets (GSPN) [11]. Transitions defined in GSPN can be either immediate, when firings take no time, or timed, when associated delays are exponentially distributed. Immediate transitions fire with priority over timed transitions, and it is assumed that different priority levels can be defined over them. A weight is also associated to each immediate transition. When two or more immediate transitions are in conflict (e.g., because they have the same priority), the selection of the one that fires first is done using the associated weights. The delay associated with a timed transition is a random variable, distributed as a negative exponential, with a defined rate. When two or more timed transitions are in conflict, the selection of the one that fires first is done according to the race policy.

In this work, availability analysis is conducted on a GSPN derived from a software architecture based on UML [12]. For sake of availability analysis, similarly to any other non-functional property, behavioral aspects of software architecture are essential. Hence, we assume that the behavior of a software architecture is described through a set of UML SMs.

However, UML does not natively provide support for availability modeling. Therefore, we rely on the "Dependability Analysis and Modeling" (DAM) profile [13] to enhance UML SMs models with availability annotations. DAM was designed on top of the standard "Modeling and Analysis of Real-time Embedded systems" profile [14], which extends UML to annotate models with information required to perform schedulability and performance analysis. Despite the ability to annotate behavioral models with availability properties, UML-DAM lacks the execution semantics to be formally analyzed. This is the reason why DAM-annotated UML models need to be transformed (e.g., in GSPN) for sake of analysis.

III. Related Work

Several approaches have been introduced in the last few years to derive analysis models from annotated software models. Bondavalli et al. [15] represents one of the first attempts at enriching a UML design to specify dependability aspects. The authors define UML extensions to automatically generate Stochastic Petri Net models for dependability analysis. High-level SPN models are derived from UML structural diagrams and later refined using UML behavioral specifications. The transformation relies upon an intermediate model, and no standard UML profiles are employed since none were available at the time of publication. In [16], Huszer et al. propose a transformation of UML statechart diagrams into Stochastic Reward Nets (SRN) to conduct a performance and dependability analyis. The transformation is defined as a set of SRN patterns, and the dependability analysis is performed under erroneous state and faulty behavior assumptions. Mustafiz et al. [17] also present a mapping between a probabilistic extension of statecharts and a Markov chain model for quantitative assessment of safety and reliability. Bernardi et al. [18] propose a transformation of UML sequence, statechart and deployment diagrams into a GSPN model for performability analysis. Software models are annotated using the former standard UML SPT profile. Our bidirectional transformation is based on the mechanisms related to statechart transformation as formally specified in [18], which we have implemented in JTL.

In the context of bidirectional model transformations, a round-trip engineering process between models representing different views of the same system is formally defined in [19]. In the general context of partial and non-injective

model transformations, modifications on target models are considered valid if they can be translated into a corresponding exact change to the source. In the performance analysis domain, in a previous paper [20] we have introduced a similar approach to the one presented in this paper. In particular, we have defined a bidirectional model transformation between UML software models and Queueing Network (QN) performance models. The forward transformation path generates the performance model from the initial software model, whereas the backward one is used to generate, after the analysis, a new software model from the modified version of the performance model. In [21] two methods to tackle the problem of deriving architectural changes from model-based performance analysis results have been compared: (i) to perform refactoring on the software side by detecting and solving performance antipatterns, or (ii) to modify the analysis model using bidirectional model transformations to induce architectural changes. This represents an interesting study for reasoning on the pros and cons of modifying a non-functional model as opposite to apply modifications to a software architectural model. To the best of our knowledge, this is the first paper proposing an automated propagation of changes performed on an availability model back to the architectural model. Even though the scope of this paper is clearly limited to the modeling notation context considered here (i.e., UML statecharts and GSPNs), our approach represents a first step towards the usage of bidirectional transformations for closing a round-trip process for software availability modeling and analysis.

IV. Our Approach

Model Driven Engineering (MDE) [22] leverages intellectual property and business logic from source code into high-level specifications enabling more accurate analyses. In general, an application domain is consistently analyzed and engineered by means of a *metamodel*, i.e., a coherent set of interrelated concepts. A model is said to *conform* to a metamodel: it is expressed by the concepts encoded in the metamodel. Constraints are expressed at the meta-level, and model transformations are based on source and target metamodels. With the introduction of model-driven techniques in the software lifecycle, the analysis of quality attributes has became effective by means of automated transformations from software artifacts to analysis models [5].

This paper proposes a model-driven approach to support designers in their availability analysis process that involves the back propagation of results to the software architecture (as introduced in Sect. I). The approach makes use of bidirectional model transformations written in JTL (that will be presented in the next section). In contrast to unidirectional languages, bidirectional transformation languages allow to describe both forward and backward transformations simultaneously, so that the consistency of the transformation can be guaranteed by construction [23]. Our approach focuses on analysis models designed by means of GSPN, whereas the software architecture is modeled by means of UML and, in particular, for the behavioral aspects, State Machines annotated via DAM are considered (UMLSM).

The overall approach is reported in Fig. 2. The *UMLSM-GSPN* bidirectional transformation (that will be presented later in this section) is defined once, and it is embedded into the JTL engine that enables its execution in both forward and backward directions. In order to execute the *UMLSM-GSPN* in forward direction, a DAM-annotated SM model is taken as input to the JTL engine, and a GSPN model is produced as output. The generated GSPN model is solved in order to obtain a set of indices that have to be interpreted. Thereafter, the GSPN model is iteratively refactored until availability requirements are satisfied[1]. In order to propagate changes applied to the GSPN model back to the software architecural model, the *UMLSM-GSPN* bidirectional transformation is executed in backward direction. In this case, the JTL engine takes as input the modified GSPN model and produces as output DAM-annotated SM models.

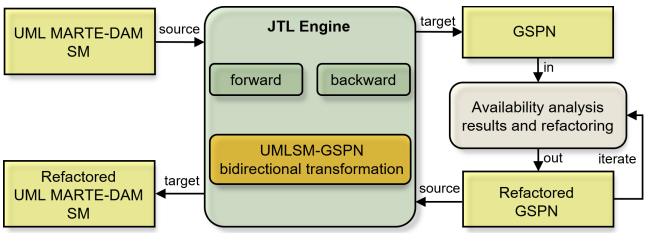


Figure 2: Overall approach

The rest of this section describes technological and technical details of the approach.

A. JTL overview

JTL [24], [25] is a constraint-based model transformation language specifically tailored to support bidirectionality and non-determinism. The implementation relies on the Answer Set Programming (ASP) [26], which is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. JTL adopts a QVT-R [27] like syntax and allows a declarative specification of relationships between MOF models. The mechanism of transformation is rule-based like QVT-R. The language supports object pattern matching, and implicitly creates trace instances to record what occurred during a transformation execution. A transformation between candidate models is specified as a set of *relation*s that must hold for the transformation to be successful: in particular, it is defined by two *domain*s and includes a pair of *when* and *where* predicates that specify the pre- and post- conditions that must be satisfied by elements of the candidate models. When a bidirectional

[1]Note that in case of non-satisfiable requirements, this loop has to be broken by designers, and the software architecture has to be fully re-conceived.

transformation is invoked for the enforcement, it is executed in a specific direction by selecting one of the candidate models as the target by means of a run configuration. The JTL engine finds and generates, in a single execution, all the possible models which are consistent with the transformation rules by a deductive process.

The JTL tool[2] has been implemented within the Eclipse framework and mainly exploits the Eclipse Modeling Framework (EMF)[3]. Moreover, it makes use of the DLV[4] solver [28] (which has been wrapped and integrated in the overall environment) to execute transformations in both forward and backward directions.

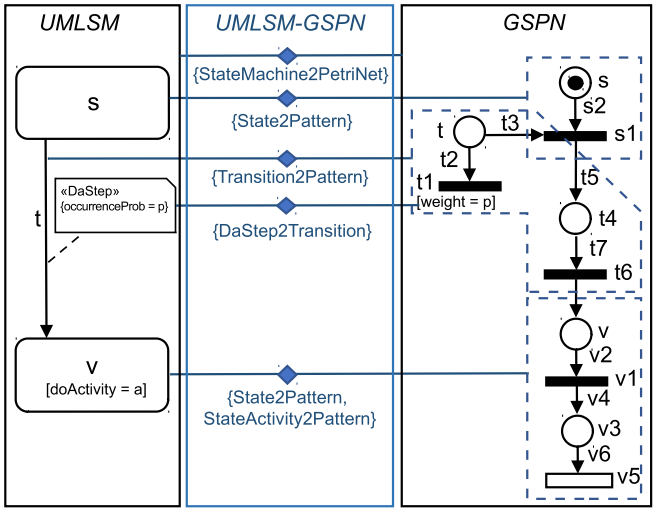


Figure 3: Relationships between UML SM and GSPN

B. UMLSM-GSPN bidirectional transformation

The *UMLSM-GSPN* bidirectional transformation has been implemented within the JTL framework. Such implementation considers the formal definition of the unidirectional translation of UML SM in GSPN provided in [13]. Starting from the latter, the relationships between UML SM and GSPN are deduced and then completed in order to define the bidirectional mapping between the two notations. The complexity of the latter task is high, because the unique bidirectional transformation has to guarantee the syntactic and semantic consistency of source and target models in both directions.

A portion of the *UMLSM-GSPN* bidirectional mapping is graphically shown as a set of relationships in Fig. 3. The left-hand side of the figure depicts the *UML SM* domain, the right-hand side depicts the *GSPN* domain, whereas the relationships are shown in the middle. The figure describes two different transformations of states, depending on whether the state is simple (see the state `s`) or includes activities (see the state `v`), a translation for transitions (see the transition `t`) and a translation for the stereotype `<<DaStep>>`.

[2]JTL Tool: https://jtl.di.univaq.it
[3]Eclipse Modeling Framework: https://www.eclipse.org/modeling/emf/
[4]DLV-complex: https://www.mat.unical.it/dlv-complex/

For sake of detail illustration, a fragment of the *UMLSM-GSPN* bidirectional transformation implemented via JTL is depicted in Fig. 4. As said in Sec. IV-A, the transformation is specified by means of a set of *relations* among elements of the two involved *domains*; they represents the transformation rules that can be executed in both the directions. The first line of the listing declares the variable `uml` that matches models conform to the UML SM metamodel and the variable `pn` that matches models conform to the GSPN metamodel (based on the standard Petri Net Markup Language (PNML) [29]). The main relations specified in the transformation are described as follows:

- `StateMachine2PetriNet` generates a container element of type `PetriNet` with attribute `id` from an element of type `StateMachine` with attribute `name`, and vice-versa in the opposite direction. Moreover, the correspondence between the reference `region` of type `Region` and the reference `pages` of type `Page` is defined.
- `State2Pattern` maps simple states to a specific pattern. Since a single element in the UML SM domain induces the creation of a list of elements in the GSPN domain, the relation enforces multiple patterns. In particular, for each UML SM `State` in a `Region` (see the reference `subvertex`), the following GSPN elements (see the references `objects`) are created: an element `s` of type `Place`, an element `s1` of type `Transition` (of kind "immediate"), and an element `s2` of type `Arc` that links `s` and `s1`. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent `State` is generated;
- `StateActivity2Pattern` considers states that involve elements of type `Activity` and add a pattern of elements to the base pattern defined for simple states. In particular, the following elements are added: `s3` of type `Place`, `s4` of type `Arc` that links the previously created transition `s2` and the place `s4`, `s5` of type `Transition` (of kind "exponential"), and `s6` of type `Arc` that links `s4` and `s5`. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent `State` is generated;
- `Transition2Pattern` relates transitions to a specific pattern. In particular, for each UML SM `Transition` in a `Region` (see the reference `transition`), the following GSPN elements (see the references `objects`) are created: an element `t` of type `Place`, an element `t1` of type `Transition` (of kind "immediate"), an element `t2` of type `Arc` that links `t` and `t1`, an element `t3` of type `Arc` that links `t` and the transition `s1` (created from a simple state), an element `t4` of type `Place`, an element `t5` of type `Arc` that links `s1` and `t4`, an element `t6` of type `Transition` (of kind "immediate"), and an element `t7` of type `Arc` that links `t4` and `t6`. In the opposite direction, for each occurrence of the described GSPN pattern a correspondent `Transition` is generated;

Figure 4: A fragment of UMLSM-GSPN

- `DaStep2Transition` relates UML SM transitions annotated with the stereotype `DaStep` from the profile DAM and GSPN transitions (of kind "immediate"). Moreover, the value of the attribute `occurrenceProb` is mapped to attribute `weight`, and vice-versa.

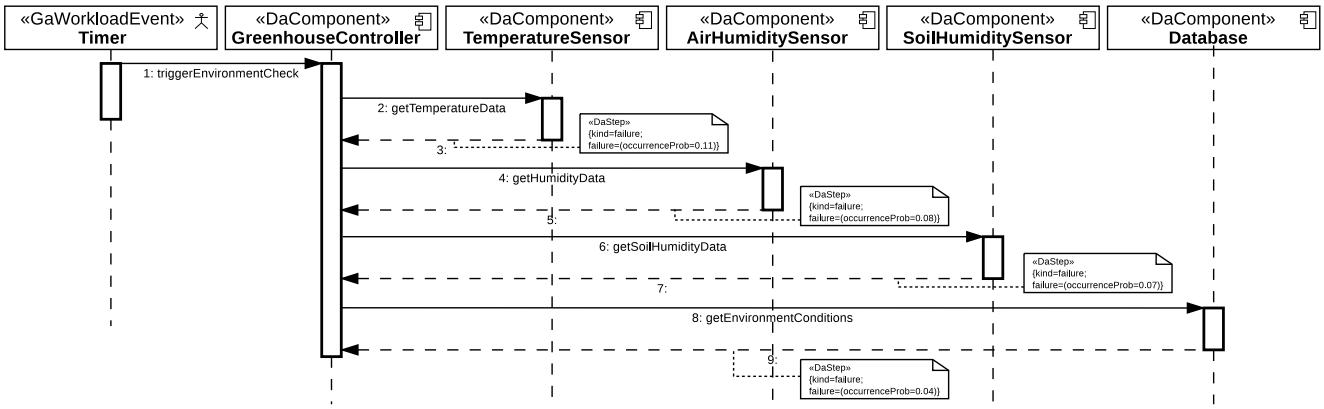
Next section shows how the bidirectional transformation defined above can be applied to an example model.

V. Example Application

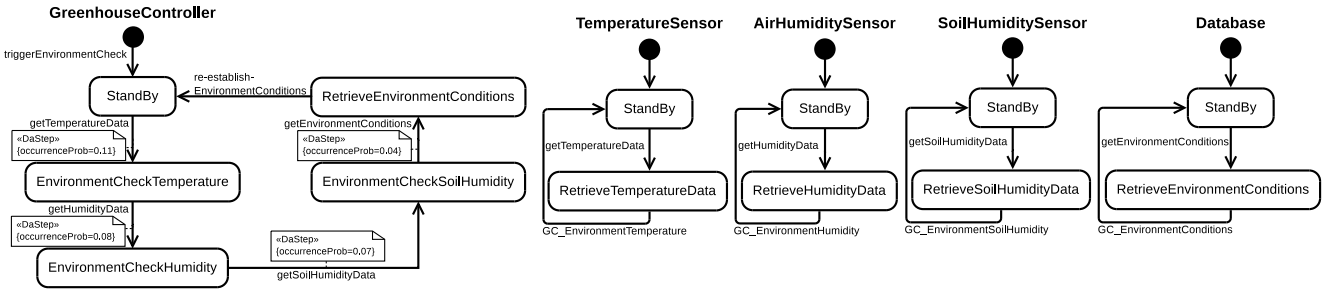
In order to experiment the proposed approach in practice, we consider a software system for the environmental control of a botanical garden. The Environmental Control System (ECS) is responsible for the automated management of the artificial habitat preserved in greenhouses. A network of sensors periodically checks air temperature, air humidity and soil humidity inside greenhouses. When sensors detect values exceeding the thresholds defined for a given greenhouse, the system automatically restores the environment conditions activating irrigation and air conditioning systems as required.

ECS consists of five software components: *GreenhouseController* is responsible for checking environment conditions; *TemperatureSensor*, *AirHumiditySensor* and *SoilHumiditySensor* are able to respectively measure air temperature, air humidity and soil humidity; *Database* is queried to retrieve the condition threshold defined for each monitored greenhouse. Each software component is deployed on a different platform device and, for sake of availability, it is considered as a single unit of failure.

We consider a use case scenario of ECS in which a timer periodically activates a procedure to check environment conditions. The corresponding behavioral view is defined in the UML Sequence Diagram in Fig. 5a. Labels are included to emphasize the application of the *DaStep* stereotype to every return message that follows a request from *Green-*



(a) UML Sequence Diagram



(b) UML State Machines

Figure 5: Behavioral view of ECS case study

houseController to any other component in the system. This stereotype is used here to define system failure modes and the probability of failures occurring in the scenario, as follows: attribute *kind* is set to *failure*, while attribute *failure* of type *DaFailure* is used to set the failure probability as the *occurrenceProb* real value. Failure modes are associated, in this way, to the losses of return messages due to intermittent network problems.

A SM describing the internal behavior of each software component is derived from the Sequence Diagram [30]. These five state machines, shown in Fig. 5b, represent the starting set of source models on which the approach proposed in Sect. IV is applied.

A. Analysis model generation

The first operational step of our approach consists in the execution of the transformation presented in Sect. IV-B in the forward direction. From the set of five UML SMs of ECS, this execution generates a set of five GSPNs. Each of these target models represents a subnet of the GSPN of the entire system on which the availability analysis will be performed.

The composition of subnets obtained in this step is based on interactions among components, as appearing in the Sequence Diagram. In particular, for each interaction, an arc is created that leaves the transition of the calling subnet and enters an additional place that represents the interface between the subnets. In turn, the latter place is the source of an arc towards the transition of the called subnet that represents the interface entry point. Return messages from a call are treated in a similar way, with the only difference that from the additional place, in this case, a transition modeling the possibility of losing the message can also be reached. This mechanism is to model the failure mode that we devise in this example, namely the loss of return messages.

The GSPN outcoming from this forward direction is reported in Fig. 6. Subnets resulting from the execution of the transformation are highlighted by different color backgrounds, and they are labeled with the name of the SM they are generated from. In the GSPN model, for sake of readability, all pairs of place *t* and transition *t1* defined in Fig.3 have been collapsed without losing information.

Places `requesting` and `requestEnvironmentCheck`, as well as transitions `arrival` and `GC.triggerEnvironmentCheck`, have been included in the GSPN to model the workload generated by the *Timer* and the initial marking.

B. Analysis results and refactoring

Given an initial marking of a GSPN, and provided that every place of the net is bounded, the reachability set is the set of all the markings reachable by sequences of transition firings from the initial one. The reachability graph associated to a GSPN is a directed graph whose nodes are the markings in the reachability set and each arc, connecting a marking M

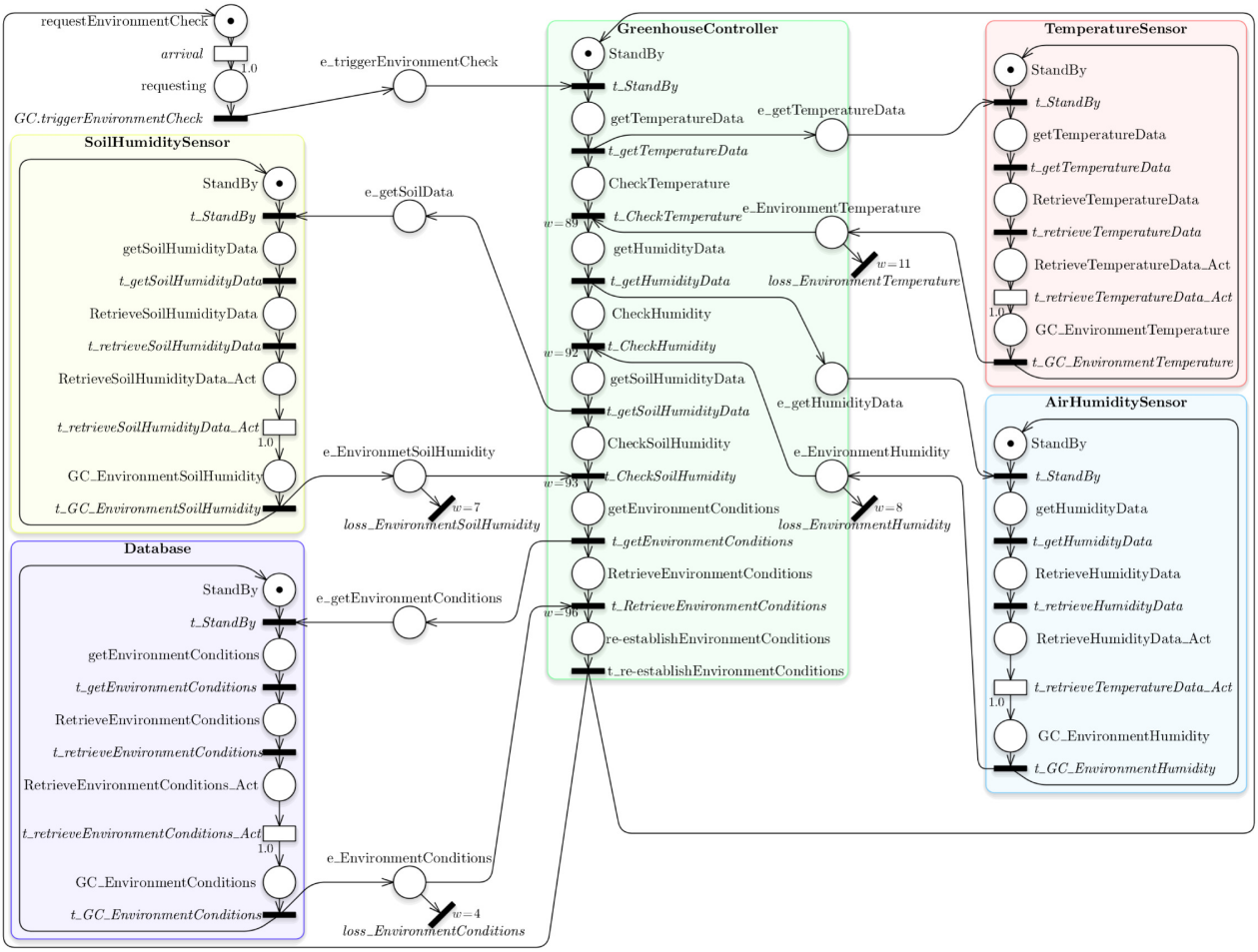


Figure 6: GSPN obtained by the forward execution of the transformation

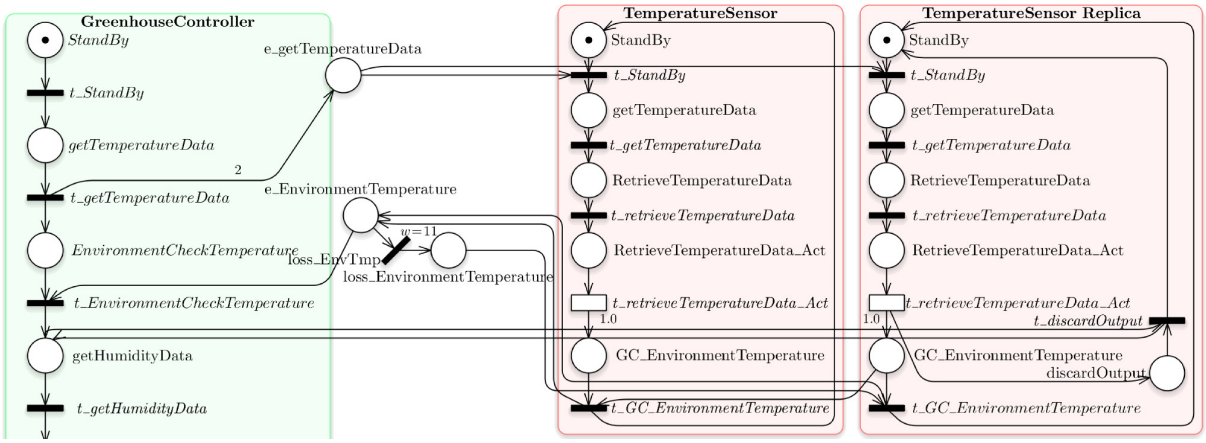
to a M' one, represents the firing of a transition enabled in M and leading to M'.

In general, availability metrics of an GSPN model can be defined as reward functions on the reachability graph [31]. Let M^0 be the initial marking, and $r_M = \{1 \text{ if } M \in O,\ 0 \text{ if } M \in F\}$ be a state reward function that partitions the set of reachable markings $RS(M^0)$ into two sets: O, the set of operational system states, and F, the set of system failure states. The probability of the system being in marking M at time instant t can be expressed as $\sigma_M(t) = Pr\{X(t) = M\}$. Steady state probability can be computed as $\sigma_M = lim_{t\to\infty}\sigma_M(t)$, and it represents the probability of the system being in marking M at any time instant $t > 0$. The steady state availability of the GSPN is then defined taking into account the reward function and the steady state probabilities of individual markings introduced before, as follows : $A_\infty = \sum_{M\in RS(M_0)} r_M\sigma_M = \sum_{M\in O}\sigma_M$. The value of A_∞ is to be interpreted as the percentage of time the system is not in a failure state after running for a sufficiently long time.

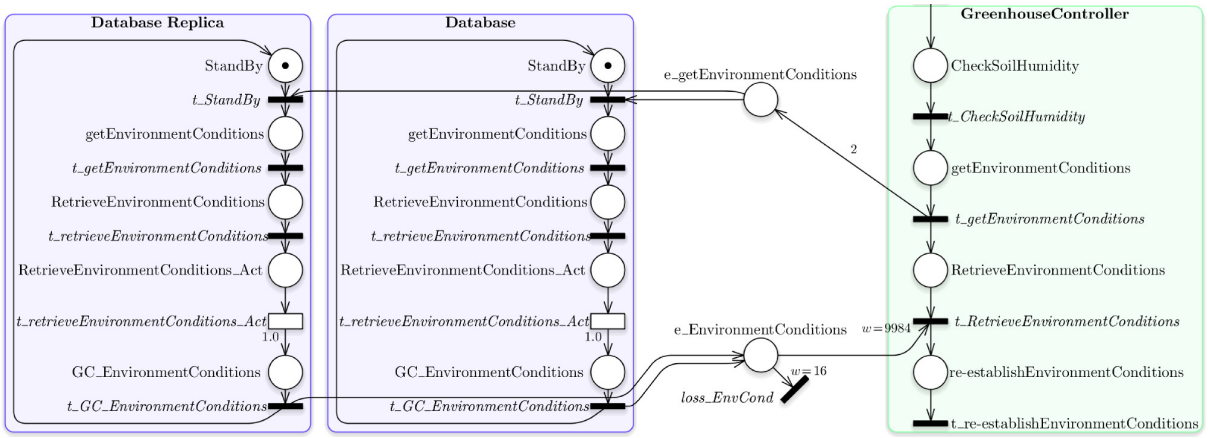
System failure mode needs to be defined in order to discern operational states from failure ones, and to exclusively assign the related markings to one among the O and F subsets of reachable markings. The system is considered to be in a failure state when a return message from a component is lost. As a consequence, in the GSPN obtained from the previous step, we define as failure states the markings reached from firing the following transitions: `loss_EnvironmentTemperature`, `loss_EnvironmentHumidity`, `loss_EnvironmentSoilHumidity` and `loss_EnvironmentConditions`.

The GreatSPN solver [32] is used to derive the reachability graph of markings in the net and to compute the corresponding values of σ_M. In the initial marking of the net a token appears in the `StandBy` place of each component subnet, as well as in the `requestEnvironmentCheck` place that generates the workload. Immediate transitions representing the loss of return messages are marked in Figure 6 with weights derived from the loss probabilities. The steady state availability index of the system obtained with the shown parameterization is $A_\infty = 0.987241$.

Now, in order to improve the system availability, we tentatively apply a fault tolerance refactoring technique to the *TemperatureSensor* component. The *Semi-Active Replication*



(a) Semi-Active Replication applied on *TemperatureSensor*



(b) Active Replication applied on *Database*

Figure 7: Detail of the refactored GSPN

technique, as defined in [33], requires the creation of a replica of the unit of failure, which in our case we call *TemperatureSensorReplica*. This replica provides the same functionalities as the original one, but it is deployed on a different node to assure failure independence among them. The net effect of this refactoring is to decrease the probability of losing a return message from the *TemperatureSensor* unit. The region of the GSPN involved in the refactoring is shown in Fig. 7a.

After such refactoring, the *GreenhouseController* component has to send data requests to the *TemperatureSensor* unit and to its replica at the same time. *TemperatureSensorReplica*, after retrieving temperature data, awaits *GreenhouseController* to acknowledge the reception of a return message from *TemperatureSensor*. This event triggers the `t_discardOutput` transition in the replica, so that the latter is not in charge of sending its data. When a return message is lost, *GreenhouseController* requests the replica to send retrieved data by activating the `GC.EnvironmentTemperature` transition.

Upon performing the same analysis made on the starting

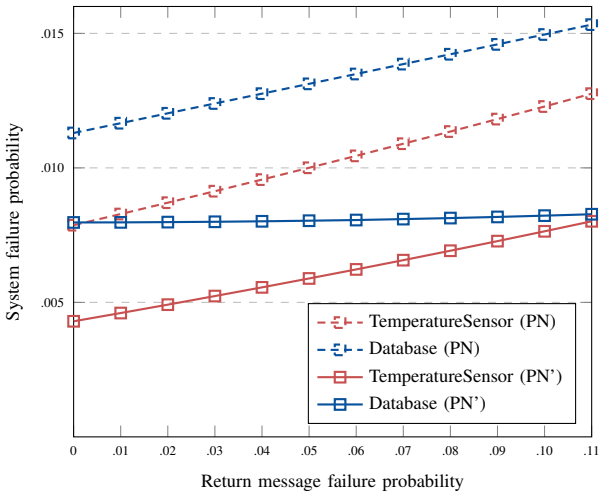


Figure 8: Sensitivity analysis of original (PN) and refactored (PN') GSPN

GSPN, the steady state availability index of the refactored GSPN is now $A_\infty = 0.988186$.

In order to further improve the system availability, the

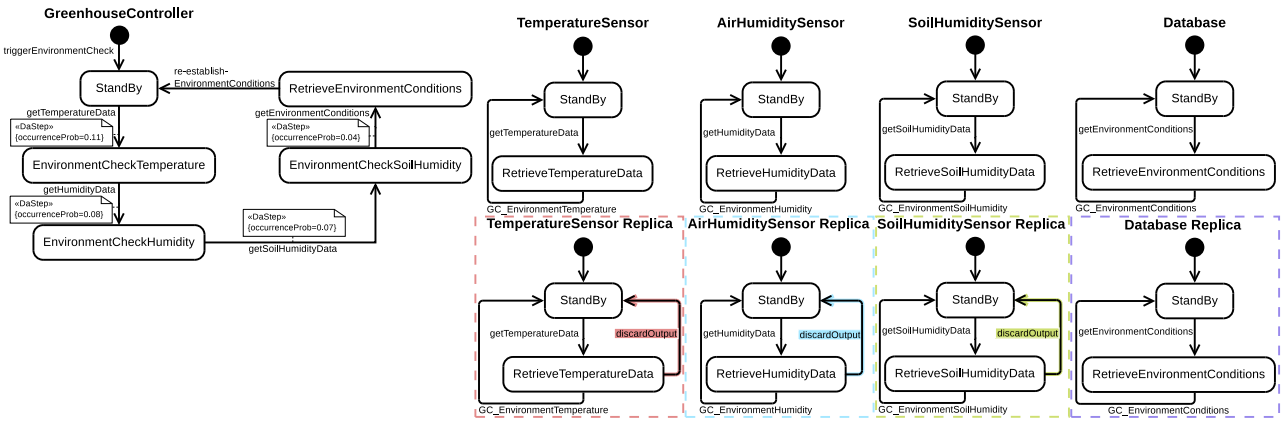


Figure 9: UML State Machines obtained in the change propagation step

same fault tolerance technique has then been extended to all other components affected by the same failure mode, namely: *AirHumiditySensor*, *SoilHumiditySensor* and *Database*. The steady state availability index obtained after such extended refactoring is $A_{\infty} = 0.991044$.

Always with the availability improvement intent, we have further refactored the latest GSPN by replacing the fault tolerance mechanism applied to *Database* with a more powerful one. In particular, we have applied the *Active Replication* pattern [33], that is an error masking technique requiring the use of a group of replicas where all members of the group actively receive and process, along with the replicated component, every input. As opposite to the *Semi-Active Replication* pattern, all replicas in this pattern deliver their outputs without waiting for a trigger.

Fig. 7b reports the application of the *Active Replication* pattern to the *Database*, where for sake of simplicity we have introduced only one replica. As a further consequence, the probability of losing a return message from *Database* is now 0.0016, because it results from the combination of 0.04 loss probabilities of *Database* and its replica. The steady state availability index of this latter refactored GSPN has again improved to: $A_{\infty} = 0.991986$.

Sensitivity analysis was carried out by varying the failure probability related to *TemperatureSensor* and *Database* in the PN original net and the PN' refactored one. The results are graphically reported in Fig. 8, where the x-axis represents the component failure probabilities, and the y-axis the system failure one (i.e., $1 - A_{\infty}$).

First, in both PN and PN' cases the system failure probability increases more quickly when the failure probability associated to *TemperatureSensor* increases, as compared to the same increase of the *Database* probability. This is due to the fact that, in our scenario, the former component is invoked before the latter one, thus potentially causing an early failure of the system.

Moreover, the PN' refactored net clearly shows a lower system failure probability as compared to the PN one for any considered point on the x-axis, due to the fault tolerance mechanisms introduced in the former one.

However, the sensitivity of the system failure probability to the *Database* variation in PN' is negligible for the considered range. This confirms that the *Active Replication* pattern brings a sensible improvement to the system, even in case one only replica is allocated.

C. Change propagation

After the analysis and refactoring step, the bidirectional transformation is applied in backward direction on the refactored GSPN model. As a consequence, a new UML SM model embedding the propagated changes is generated, as shown in Fig. 9.

As said, a new software component replica of each sensor and of database has been introduced in the GSPN model. The back propagation of these changes generates four additional state machines (i.e., one for each replica, outlined by a dashed line in the figure) by enforcing the `StateMachine2PetriNet` relation and its triggered relations. The state machines corresponding to the original components are instead restored without any modification. In addition, each state machine produced by the semi-active replication pattern (i.e., all sensors replicas) includes a new `discardOutput` transition (highlighted in the figure) that represents the case in which no failure occurs in the original component and, as a consequence, the data computed by the replica must be discarded. The change performed in the GSPN model (i.e., the creation of the `discardOutput` place and the `t_discardOutput` transition in every replica) matches the rule `Transition2Pattern` and, as a result, the `discardOutput` transition is generated in the UML model.

Finally, the obtained model is consistent with respect to the consistency relation defined in the transformation, and it is compliant with the source metamodel.

VI. Conclusions

In this paper, we proposed a model-driven approach to work towards a round-trip process for the assessment of software availability through bidirectional model transformations. We

introduced a bidirectional transformation, implemented in the JTL language, able to: (i) translate UML SM, annotated with availability properties, into GSPN, and (ii) propagate modifications carried out on the GSPN model back to the original architectural model. Our experimentation has shown the potential of our approach to automatically propagate changes on the software architecture, so to provide architectural alternatives that may satisfy availability requirements.

The current implementation of UMLSM-GSPN transformation only applies to State Machines of single components. In a short-term view we plan to extend the transformation to interactions among components.

We would like to remark that the change propagation from analysis to architectural models may consist in the generation of multiple architectural alternatives. In fact, this is a non-bijective process because a specific refactored pattern in a GSPN can be mapped in more than one solution in the architectural model. Thus, a major challenge for the future is to study the human-assisted process of choice among the suggested alternatives.

Finally, the suitability of bidirectional transformations in the availability domain shall be consolidated by introducing transformations among different architectural notations and analysis metamodels.

REFERENCES

[1] H. Muccini, A. Bertolino, and P. Inverardi, "Using software architecture for code testing," *IEEE Trans. Software Eng.*, vol. 30, no. 3, pp. 160–171, 2004.

[2] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software*, vol. 80, no. 6, pp. 918–934, 2007.

[3] V. Cortellessa, A. D. Marco, and P. Inverardi, "Non-functional modeling and validation in model-driven architecture," in *WICSA*, 2007, p. 25.

[4] D. Garlan, J. M. Barnes, B. R. Schmerl, and O. Celiku, "Evolution styles: Foundations and tool support for software architecture evolution," in *WICSA/ECSA*, 2009, pp. 131–140.

[5] V. Cortellessa, A. D. Marco, and P. Inverardi, *Model-Based Software Performance Analysis*. Springer, 2011.

[6] S. Bernardi, J. Merseguer, and D. C. Petriu, *Model-Driven Dependability Assessment of Software Systems*. Springer, 2013.

[7] H. Sözer, M. Stoelinga, H. Boudali, and M. Aksit, "Availability analysis of software architecture decomposition alternatives for local recovery," *Software Quality Journal*, vol. 25, no. 2, pp. 553–579, 2017.

[8] T. Pitakrat, D. Okanovic, A. van Hoorn, and L. Grunske, "An architecture-aware approach to hierarchical online failure prediction," in *QoSA*, 2016, pp. 60–69.

[9] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu, "Feature-based classification of bidirectional transformation approaches," in *SOSYM*, 2015, pp. 1–22.

[10] E. de Souza e Silva and H. R. Gail, "Calculating availability and performability measures of repairable computer systems using randomization," *J. ACM*, vol. 36, no. 1, pp. 171–193, 1989.

[11] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., 1994.

[12] "Unified modeling language," OMG, 2015, version 2.5.

[13] S. Bernardi, J. Merseguer, and D. C. Petriu, "A dependability profile within MARTE," *Software and System Modeling*, vol. 10, no. 3, pp. 313–336, 2011.

[14] "A UML profile for MARTE: modeling and analysis of real-time embedded systems," OMG, 2008.

[15] A. Bondavalli, M. D. Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia, "Dependability analysis in the early phases of uml-based system design," *Computer Systems Science And Engineering*, vol. 16, pp. 265 – 275, 2001 2001.

[16] G. Huszerl, K. Kosmidis, M. D. Cin, I. Majzik, and A. Pataricza, "Quantitative analysis of uml statechart models of dependable systems," *The Computer Journal*, vol. 45, pp. 260–277, 2000.

[17] S. Mustafiz, X. Sun, J. Kienzle, and H. Vangheluwe, "Model-driven assessment of system dependability," *Software & Systems Modeling*, vol. 7, no. 4, pp. 487–502, Oct 2008.

[18] S. Bernardi and J. Merseguer, "QoS Assessment via Stochastic Analysis," *IEEE Internet Computing*, vol. 10, pp. 32–42, 2006.

[19] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," in *Theory and Practice of Model Transformations*, 2008, pp. 31–45.

[20] R. Eramo, V. Cortellessa, A. Pierantonio, and M. Tucci, "Performance-driven architectural refactoring through bidirectional model transformations," in *QoSA*, 2012, pp. 55–60.

[21] D. Arcelli and V. Cortellessa, "Software model refactoring based on performance analysis: better working on software or performance side?" in *FESCA*, 2013, pp. 33–47.

[22] D. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[23] P. Stevens, "A Landscape of Bidirectional Model Transformations," in *GTTSE*. Springer, 2008, pp. 408–424.

[24] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "JTL: a bidirectional and change propagating transformation language," in *SLE10*, 2010, pp. 183–202.

[25] R. Eramo, A. Pierantonio, and G. Rosa, "Managing uncertainty in bidirectional model transformations," in *SLE*, 2015, pp. 49–58.

[26] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming," in *ICLP*, 1988, pp. 1070–1080.

[27] "MOF Query/View/Transformation - QVT," OMG, 2016.

[28] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV System for Knowledge Representation and Reasoning," *TOCL*, 2004.

[29] M. Weber and E. Kindler, "The petri net markup language," in *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets*, 2003, pp. 124–144.

[30] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *ICSE*, 2000, pp. 314–323.

[31] K. Goseva-Popstojanova and K. S. Trivedi, "Stochastic modeling formalisms for dependability, performance and performability," in *Performance Evaluation: Origins and Directions*, 2000, pp. 403–422.

[32] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo, "Greatspn 1.7: Graphical editor and analyzer for timed and stochastic petri nets," *Performance Evaluation*, vol. 24, no. 1-2, pp. 47–68, 1995.

[33] T. Saridakis, "A system of patterns for fault tolerance," in *Proceedings of 2002 EuroPLoP Conference*, 2002.