Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 279 (2) (2011) 3–16

www.elsevier.com/locate/entcs

Reputation-based Reliability Prediction of Service Compositions

Galina Besova[1], Heike Wehrheim[2]

Department of Computer Science
University of Paderborn
Germany

Annika Wagner[3]

Department of Computer Science
Fulda University of Applied Sciences
Germany

Abstract

Today, the concept of service oriented architectures provides a way of building integrated solutions out of existing services. To this end, services from different providers are composed using advanced orchestration and choreography techniques. However, while this principle allows for greater flexibility at a smaller cost, the use of third party services also includes a risk: Deployed services might not work as claimed by their providers. In this paper, we propose a technique for analyzing the expected reliability of service compositions based on ratings given by (previous) service users. Every service thereby comes with a *reputation*, and the analysis computes an overall reliability of a service composition from the reputations of its constituent services. The proposed model-driven approach proceeds by translating statechart models of service compositions into input for a probabilistic model checker (PRISM) using state-of-the-art model transformations techniques. The approach has been implemented as an Eclipse plug-in and is fully compliant with UML.

Keywords: Reliability prediction, service oriented architectures, probabilistic model checking, model transformations.

1 Introduction

Today, complex systems can be built by composing services to deliver integrated solutions, allowing greater flexibility, reuse of existing functionality, scalability, etc. Alike component-based systems, services can be obtained from different software providers by searching existing repositories for the required functionality. However, the use of third-party services entails a risk: Most often the time scale for building a

[1] Email: besova@mail.upb.de

[2] Email: wehrheim@uni-paderborn.de

[3] Email: annika.wagner@informatik.hs-fulda.de

1571-0661 © 2011 Elsevier B.V. Open access under CC BY-NC-ND license.

doi:10.1016/j.entcs.2011.11.008

new system does not allow for an extensive testing of an external service. A software designer thus has to rely on the provider's specification of the service functionality. A deviation of the actual behavior of the service from its specification thus only becomes apparent during execution of the constructed software.

In this paper, we propose an alternative to time-consuming testing or expensive formal analysis which is based on the use of a software provider's or service's *reputation*. The reputation can (for instance) be obtained by ratings of users of a service, like ratings for hotels and restaurants, or alternatively, by monitoring the service every time it is executed and recording whether it behaves as specified. The reputation is thus based on previous experience of users with the correctness of the service. Depending on the reputation of single services, the overall *expected reliability* of a service composition can be computed. Reliability herein is the probability of failure-free, correct operation of a service composition ("continuity of correct service" [9]). We take the reputation of a software provider as being an indication for the reliability of its services. The overall expected reliability of a service composition is then not just the average or sum of the reputations of its constituent services. Depending on the single service's effect on the overall behavior, a service with a bad reputation might or might not have a large influence on the reliability of the complete composition. In this paper we propose a technique for systematically computing the reliability of a service composition based on its *model* annotated with reputations of single services.

In our approach we follow a principle employed by a large number of techniques for the analysis of non-functional properties of component-based systems. In particular for performance analysis, a large body of work employing *model-driven* approaches has been developed in recent years [22,10,14]. Models of component-based systems are enhanced with information about performance attributes of single entities, and these enhanced models are afterwards translated into various sorts of analysis models (e.g. stochastic Petri nets, Markov chains, queuing networks). The actual performance analysis is then carried out using standard tools operating on such analysis models. In our setting, single services will be modeled using UML state machines [6]. The reputation of a single service is given as a numeric value in the range [0..1], describing the ratio of correct executions of the service. A service composition is then a choreography made up of reputation-annotated state machines. Such service composition models are translated into Markov decision processes, in which the probabilities of executing transitions are set according to the reputations and the type of events (send, receive or internal event). We generate Markov decision processes (MDPs) in the form of an input to the probabilistic model checker PRISM [18]. PRISM is then used to query the MDP for the probability of not reaching error states, and the answer to this query is the overall expected reliability of the service composition. For the generation of MDPs as input to PRISM we follow state-of-the-art model transformation techniques: using the language ATL [1] we define metamodel based rules for transforming UML state machine models to PRISM models. The required state machine metamodel was taken directly from the UML, and the PRISM metamodel had to be created. The model transformation

is part of a larger Eclipse-based tool which provides automation of our approach.

The rest of the paper is organized as follows: In section 2 we describe the modeling and analysis formalisms used throughout the paper. Section 3 describes the transformation between the design and analysis models described in these formalisms. The tool support of the proposed approach is discussed in section 4. Section 5 gives an overview of related work and section 6 concludes the paper and gives some directions for future work.

2 Concepts

2.1 Modeling of choreographies

A service choreography consists of communicating services that perform activities and coordinate with each other by means of message exchange. Three basic activities that can be performed by a service in a composition include sending and receiving of messages, and internal activities. Communication can have synchronous or asynchronous character depending on the particular composition. In our setting, communication between the services is synchronous[4] - It is possible only if both communicating processes are able to execute the transitions with the sending and receiving events pair at the same time. This might require one of the communication parties to wait.

The modeling approach chosen in this work to describe compositions of communicating services is based on UML [6] state machines with a CSP-like [19] communication. Alternatively, other approaches like StoCharts [21] or Component-Interaction automata [11] could have been chosen as a basis for compositions modeling.

A model consists of one or more UML state machines that contain an arbitrary number of parallel composed statecharts representing services (separated by regions). Each statechart is described by its states and transitions, that can be triggered by an internal, receiving, or sending event which is indicated through labels *a*, *a?*, and *a!* respectively. For modeling these events we use the specific kinds of events provided by the UML for inter-process communication: *ExecutionEvent*, *ReceiveSignalEvent*, and *SendSignalEvent*.

Figure 1 illustrates the above discussed with an example of two services - *supplier* and *buyer*. Each service is presented by a statechart in its own state machine and region. They communicate on two occasions: first, to pass an order from the *buyer* to the *supplier*, and later, to inform the *buyer* about the order status. Note that the diagram additionally contains the reputations associated with the individual services. This is realized by a UML profile keeping our choreography models UML compliant.

[4] Asynchronous type can be modeled through an additional statechart that represents the communication channel.

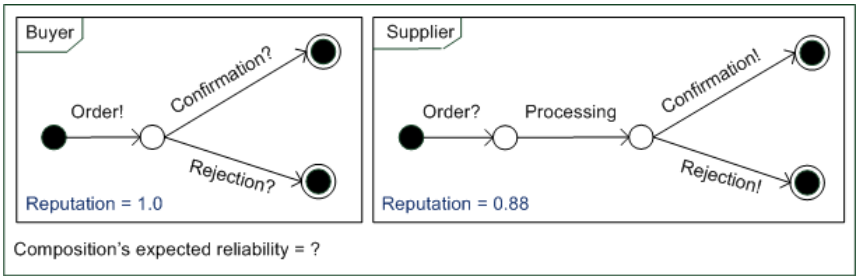

Fig. 1. Supplier and buyer choreography modeling example

2.2 Reliability analysis model

The modeling notation chosen in this work and described above is fairly straightforward and familiar to most software architects. However, it is mostly not supported by existing formal analysis techniques including model checking. Therefore, in order to analyze a choreography using model checking, its model has to be first transformed into the corresponding analysis model in accepted format. The model checker used in this work is the Probabilistic Symbolic Model Checker (PRISM) [3], therefore, its modeling language has to be introduced first.

The PRISM modeling language is based on the Reactive Modules formalism [7]. It allows description of a system as a Discrete Time Markov Chain (DTMC), Continuous Time Markov Chain (CTMC), or Markov Decision Process (MDP) model.

Main elements of an analysis model are modules and variables. Modules contain finite range local variables and commands. A PRISM command has the following form:

$$\texttt{[a] g -> } p_1\texttt{:(upd}_{11}\texttt{)\&...\&(upd}_{1m_1}\texttt{)+...+ } p_n\texttt{:(upd}_{n1}\texttt{)\&...\&(upd}_{nm_n}\texttt{);}$$

The command consists of two parts divided by the transition sign $\rightarrow$. The left hand side contains an action label a within the square brackets, which is used for processes synchronization, followed by a transition guard g. When the guard g is *true*, an action a, if not empty, forces simultaneous execution of all commands labeled with a within the model. The command's right hand side contains n possible mutually excluding variable updates sets each of which is equipped with a probability p_i. Each set contains m_i variable updates upd_{ij} which take place simultaneously.

Before providing an example of a PRISM module, the setting when more than one command can be executed at the same time needs to be discussed. In this case, the choice of one of the alternatives depends on the PRISM model type. DTMC and CTMC models assign equal probabilities to all alternatives, whereas MDP models simulate non-deterministic choice. In this work we focus on MDP models for service compositions since, compared to DTMC and CTMC, they additionally provide a mechanism for describing the cases when the choice between alternative execution paths of a service is not determined by a fixed probability distribution. This allows expressing influence of external environment on the system, changes of parameters within such environment, etc..

```
module M
  x : [0..1] init 1;
  y : [0..1] init 0; z : [0..1] init 0;

  [] x=1 -> 0.2: (y'=1) & (x'=0) + 0.8: (z'=1) & (x'=0);
  [a1] y=1 -> (x'=1) & (y'=0);
  [a2] z=1 -> (x'=1) & (z'=0);
endmodule
```

Fig. 2. PRISM module example

Figure 2 provides an example of a PRISM module M with three finite range variables x, y and z, and commands containing updates. The internal command with the guard $x = 1$ and without any synchronization takes place first, since the local variable x is initially set to 1. This command has two alternative update sets: the first assigns variables y to 1 and x to 0, and the second assigns variable z to 1 and x to 0. The first set has the probability of 0.2 and the second - 0.8. As variable x has been set to 0, the first command cannot be executed again. Instead, one of the commands with the guards $y = 1$ or $z = 1$, depending on the previously chosen update set, can now be executed. These commands synchronize with some module on action labels *a1* or *a2* and set variable x to 1, so the cycle can be repeated.

Modules that represent different interacting processes within the model, can be composed together in a process-algebraic expression. This expression should feature each module exactly once, and contain CSP-based operators including: parallel composition with full or partial synchronization over shared actions, asynchronous parallel composition, and operators for hiding and renaming of actions within the module. An analysis model described in PRISM modeling language is later translated by the model checker into a Markov model.

The described language allows definition of analysis models which can be used to model check various system properties including its reliability. The question to be discussed next, is the transformation of a design model of a choreography (section 2.1) into a model in the PRISM language.

3 Transformation concept

3.1 Reputation interpretation

To be able to analyze a service composition described in a design model, it has to be transformed into the analysis model in the selected format. In order to describe this transformation, it is necessary to define element mappings and analyze transition probabilities within the design model.

In this work we determine transition probabilities based on the reputation data of individual services and transition types. Given a reputation of a service as a whole, it is possible to choose between various interpretations of this information with regards to the probabilities of individual transitions. In this work we use the following interpretation:

- *Internal actions* of a service are not observable for other communication parties

and, therefore, do not directly influence its observed reliability. Hence, transitions due to internal actions are assumed to always occur, i.e., have probabilities of 1.

- *Message receipts* by a service are observed by other parties. Services are assumed to always accept messages, possibly discarding them later. Following this assumption, such transitions always occur, and also do not directly influence the observed reliability, i.e., have probabilities of 1.
- *Message sendings* by a service are observed by other communication parties, as they initiate communication. Their success probabilities influence the observed reliability of the containing service. Therefore, such transitions occur with the probability equal to the observed reputation of the containing services.

Additionally, it is assumed that a message sending failure causes service interruption with no possibility of repair.

For the example presented earlier in Figure 1 this interpretation would mean that all transitions are executed with the probability of 1 except of the *confirmation* and *rejection* sending, which are executed with the probability of 0.88. This interpretation combined with the knowledge of the design and analysis model formats can now be used to define the required transformation rules.

3.2 Transformation rules

The idea behind the proposed transformation rules is to create an analysis model, where each service is represented by its own PRISM module with the same name. These modules contain local variables needed to describe different states of their corresponding services, and commands to describe transitions between these states. A set of local variables of such a module always contains exactly one *start* variable and variables for unique final states of the service. The *start* variable initialized with 1 represents the start state of the service.

We use the following rules to transform a transition into one or more PRISM commands. Depending on the type of triggering event, one of the options in Figure 3 is chosen:

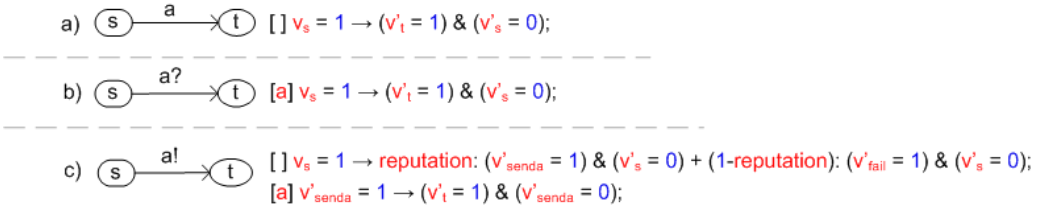

Fig. 3. Transformation concept for transitions with different triggering event types

Note that the definition of these rules is based on the interpretation of reputation for probabilities of individual transitions discussed in section 3.1.

In all three cases the transition is preformed between the states s and t. Translated into the PRISM command, the first fact is represented by the boolean expression $v_s = 1$ as a guard. The fact, that the service has left the state s and entered

the state t, is represented by two updates $v'_s = 0$ and $v'_t = 1$ respectively, where v_s and v_t denote local variables corresponding to the service states s and t. Further event type specific details of the transformation rules can be combined under the corresponding sub items:

a) Transitions triggered by internal actions a are transformed into commands that contain mentioned updates $v'_s = 0$ and $v'_t = 1$ without any synchronization.

b) Transitions triggered by receiving events $a?$ are transformed into commands similar to the case a) that are, however, labeled with synchronization actions a. This is done to ensure that the receiving and sending (case c)) command pair, that represents communication between two services through a message a, is only executed synchronously.

c) Transitions triggered by sending events $a!$ are transformed into pairs of two subsequent commands. The first command represents two alternatives: a message will be sent with the probability equal to the *reliability* reputation of the containing service, or a failure with the complementary probability. If the first alternative is chosen, the variable update $v'_{senda} = 1$ is performed, representing the fact that the message will be sent. If the second alternative is chosen, the update $v'_{fail} = 1$ is performed, representing sending failure which makes further commands execution within the module impossible.

   The second command is only executed if the first probabilistic choice indicates that the message a will be sent. This command is labeled with the synchronization action a to ensure its simultaneous execution with the receiving command $a?$ (case b)).

   Figure 4 illustrates this transformation rule on an MDP fragment for transition $a!$. The fragment consists of three transitions: from state s either to one of the added intermediate states *senda* or *fail*, and from state *senda* to state t. The first two transitions with the complementary probabilities *reliability* and *1 - reliability* represent the two alternatives within the first command, whereas the last transition represents the second command and the actual sending of the message a with probability of 1.

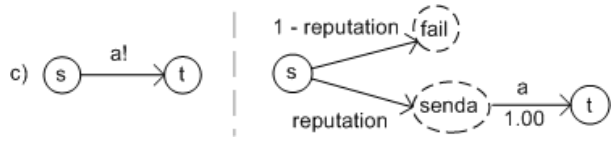

Fig. 4. Sending transition transformation into an MDP fragment

As the states within a design model are unnamed, we had to define a naming mechanism for the variables v_s, v_t, v_{senda} and v_{fail}. The following naming conventions have been used:

- The variable v_s is named depending on the location of the state s within the statechart as follows:
  - *start*, if s is an initial state.
  - variable that represent successful completion of an incoming transition (s', s),

if s is an intermediate state. Naming of such variables is discussed next in the context of variable v_t.

- The variable v_t is named depending on the type of the state t and of the transition (s,t) with the triggering event a as follows:
  · corresponding final state variable, if t is a final state.
  · a, if t is an intermediate state and (s,t) is an internal or receiving transition.
  · $aSent$, if t is an intermediate state and (s,t) is a sending transition.
- The variable v_{senda} is named a, where a is the name of the triggering event of the original sending transition $a!$.
- The variable v_{fail}, unlike v_{senda}, is shared by all commands within a module and is, therefore, simply named *fail*.

Note that the set of variables may contain a fail variable, if the reputation of the corresponding service is less than 100%. To comply with the requirements of the PRISM language all variable names are extended by the name of the containing module.

Figure 5 illustrates the result of this transformation for the supplier and buyer design model example. It contains two modules - *Buyer* and *Supplier*, described by their local variables and commands, and synchronized on shared actions: *order*, *confirmation*, and *rejection*. Module *Buyer* has variables *startBuyer*, *orderBuyer* and *orderBuyerSent* for the commands representing order sending transition, and a variable *finishBuyer* for the final state [5] . Module *Supplier* apart from start variable *startSupplier* contains a *failSupplier* variable, as its reputation is less than 100%. This module also has an *orderSupplier* variable for the order receiving command, and a *processingSupplier* variable for the processing command. Two variables *confirmationSupplier* and *rejectionSupplier* are added for the confirmation and rejection sending commands.

The commands within both modules can be derived by application of the discussed transformation rules to transitions, and usage of appropriate variables and synchronization actions. The resulting model can now be analyzed in PRISM to check various properties of the composition. These properties have to be formalized in the PRISM properties specification language. In this work, in order to obtain the expected reliability, we expressed it as *a probability of not reaching a failure state during the lifetime of the system* or formally, for an MDP system model containing k failure states:

$$\mathtt{P}_{min=?}\mathtt{[!(F\ (fail}_1 = 1\mathtt{||...||fail}_k = 1\mathtt{))]}$$

$$\mathtt{P}_{max=?}\mathtt{[!(F\ (fail}_1 = 1\mathtt{||...||fail}_k = 1\mathtt{))]}$$

The value of these properties for the above example is equal to 0.88.

The proposed rules enable step-by-step creation of comprehensible analysis models of service compositions for further analysis with the PRISM model checker. However, to facilitate application of the proposed reliability prediction approach, it

[5] When the final state has no name, the corresponding variable is named *finish*.

```
module Buyer
// local variables
startBuyer: [0..1] init 1;      orderBuyer: [0..1] init 0;
orderBuyerSent: [0..1] init 0;  finishBuyer: [0..1] init 0;

// order!
[] startBuyer=1 -> (orderBuyer'=1)&(startBuyer'=0);
[order] orderBuyer=1 -> (orderBuyerSent'=1)&(orderBuyer'=0);
// rejection?
[rejection] orderBuyerSent=1 -> (finishBuyer'=1)&(orderBuyerSent'=0);
// confirmation?
[confirmation] orderBuyerSent=1 -> (finishBuyer'=1)&(orderBuyerSent'=0);
endmodule

module Supplier
// local variables
startSupplier: [0..1] init 1; failSupplier: [0..1] init 0; orderSupplier: [0..1] init 0;
processingSupplier: [0..1] init 0; confirmationSupplier: [0..1] init 0;
rejectionSupplier: [0..1] init 0;  finishSupplier: [0..1] init 0;

// order?
[order] startSupplier=1 -> (orderSupplier'=1)&(startSupplier'=0);
// processing
[] orderSupplier=1 -> (processingSupplier'=1)&(orderSupplier'=0);
// confirmation!
[] processingSupplier=1 -> 0.88:(confirmationSupplier'=1)&(processingSupplier'=0) +
                           (1 - 0.88):(failSupplier'=1)&(processingSupplier'=0);
[confirmation] confirmationSupplier=1 -> (finishSupplier'=1)&(confirmationSupplier'=0);
// rejection!
[] processingSupplier=1 -> 0.88:(rejectionSupplier'=1)&(processingSupplier'=0) +
                           (1 - 0.88):(failSupplier'=1)&(processingSupplier'=0);
[rejection] rejectionSupplier=1 -> (finishSupplier'=1)&(rejectionSupplier'=0);
endmodule
```

Fig. 5. Analysis model for the supplier and buyer example

is necessary to provide required tool support. Therefore, the tool support, which development was carried out as part of this work, will be discussed next.

4 Tool support

As already mentioned, the reliability prediction approach proposed in this work contains several steps, which are summarized in Figure 6.

First, a service composition is modeled as described in section 2.1, and annotated with reputations. Then this design model is transformed into the analysis model via application of the transformation rules informally explained in section 3.2. Finally, the resulting model together with the model-specific reliability property specification is analyzed using the PRISM model checker. The result of such an analysis provides the reliability value of the service composition described in the initial design model.

Tool support for some of these steps like UML modeling and model checking with PRISM already exist. Other steps like model annotation and transformation required development of appropriate supporting mechanisms.

The Eclipse platform with its flexible plug-in based architecture and numerous useful third-party plug-ins has been chosen as a development and application platform for our approach. This choice allows the reuse of already existing UML2 conform modeling tools (e.g. UML2 Tools [5], TOPCASED [4]) realized as Eclipse plug-ins, to support the design model definition. The following steps of our approach are not directly supported in Eclipse, however, various plug-ins significantly simplified the development process of our transformation tool.

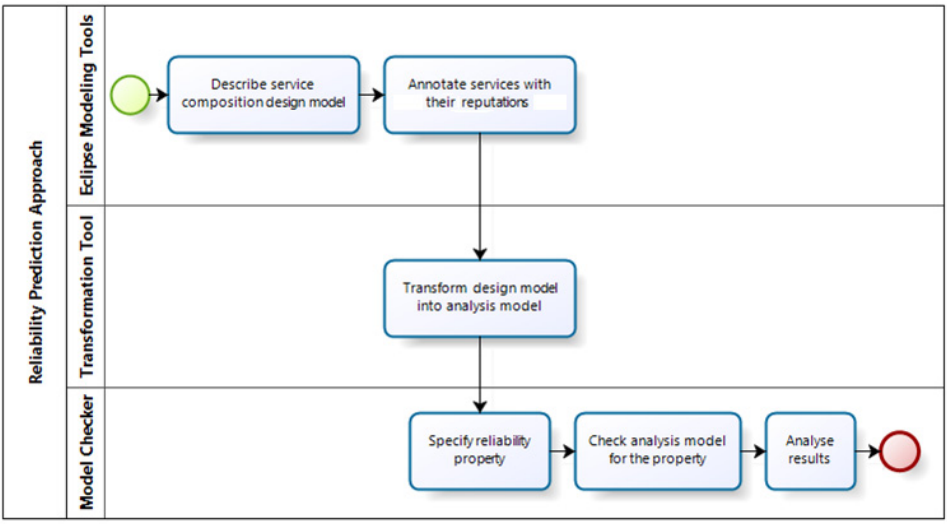

Fig. 6. Reliability prediction approach steps

First of all, we rely on the Eclipse Modeling Framework (EMF) for storing and retrieving our models. Moreover, the ATL transformation language [1](supported by a third-party plug-in) was used to define and apply the model transformation rules. Finally, we used the JET-template model-to-text engine [2] which allowed us to generate a textual representation of the transformed model. The PRISM model checker is, unfortunately, not integrated within the Eclipse platform, therefore, the analysis model produced by our tool has to be imported manually. The last two steps of the model-to-model transformation and the model-to-text transformation have been integrated in our tool.

Figure 7 illustrates the transformation principle and the artifacts needed for its implementation. It demonstrates, that the definition of the ATL transformation rules and JET-templates required UML and PRISM language metamodels. The later was also developed in this work.

The last point to be mentioned is the annotation of design models. Our approach takes advantage of an existing lightweight UML extension mechanism by means of profiles. To enable annotation of UML statecharts with reputations, we defined a UML profile that contains a stereotype for state machine regions allowing these regions to carry a so-called tagged value storing a reliability value. As regions are used to separate services within one state machine, each region requires a reputation value.

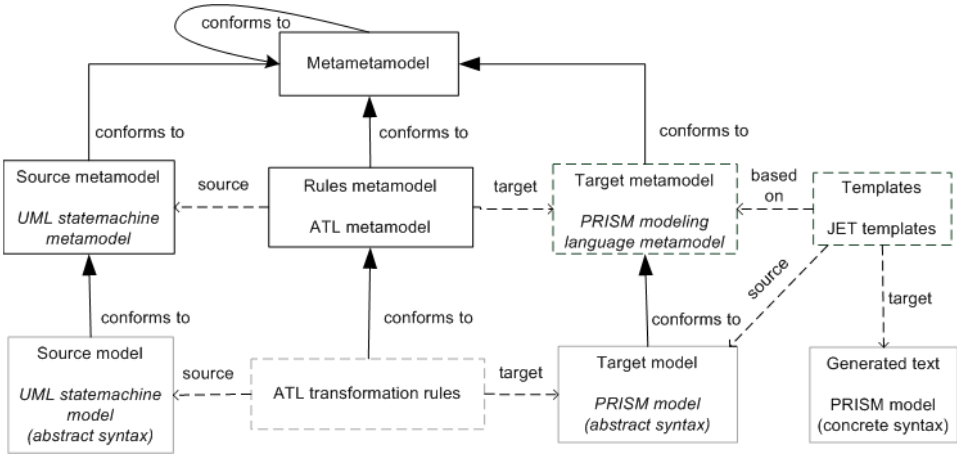

Fig. 7. Metamodel based model transformation principle

5 Related work

Prediction of reliability has always attracted research interest. Numerous approaches have been proposed to address the growing complexity of component-based [28,15,23,16,24,25] and service-oriented systems [17,13,29,31,20,8,14,26,32,30,27,12].

Most of these approaches rely on some model of a system architecture expressed in a specification language like UML([14], our method), BPEL ([31,8]), WSCI ([27]), etc.. In the case of orchestrated services workflow diagrams (e.g. UML activity diagrams [14]) are generally used. Service choreographies, on the other hand, are modeled through the specification of communication between the parties (e.g. WSCI specification [27], communicating state machines in our case). The further choice of a concrete specification language depends on the desired level of abstraction. Additionally, most approaches require information on reliability of individual services/components. In some methods actual reliabilities are required, whereas other approaches including our rely on reputations.

Depending on the technique provided system models are either directly analyzed using reduction rules to compute QoS [13,20,30], or transformed into some kind of stochastic model [29,31,8,14,26,27,12] for further analysis, like in our case. The most widely used stochastic models for this purpose include Markov models and stochastic Petri nets with corresponding analysis algorithms. For instance, Zhong and Qi [31] consider BPEL specifications and transform them into stochastic Petri nets for analysis. Gallotti et al. [14] consider UML activity diagrams of an orchestration extended with QoS properties, and transform them into Markov models. Our approach is similar to [14] as it also uses Markov model and PRISM model checker for reliability analysis, however, it is focused on service *choreographies*, and, therefore, considers different UML diagrams used for communicating services.

Xia et al. [27] propose the only other approach, that we are aware of except of our, that considers choreographies. It is based on WSCI specifications of compositions, which are translated into a General Stochastic Petri net for reliability prediction. Compared to our approach, the authors focus on models at a different abstraction

level and apply other analysis techniques. This allows complementary usage of both methods at different system development stages.

Some authors [30,24] also address the problem of obtaining required reliability information for individual services/components. Zheng and Lyu [30] propose collaborative mechanism for predicting reliability of a service for a user based on the data collected from similar users, who have used this service. According to the authors, this mechanism demonstrates better reliability prediction accuracy than other approaches, however, it can only be applied if the failure data of services is available. This requires the service to be implemented and deployed. Roshandel, et al. [24] propose an architecture-based mechanism for reliability prediction of a component using Markov models, which makes it similar to some system-level approaches and does not require a component to be implemented. This is achieved due to the Hidden Markov models used to address the lack of an operational profile. Our approach, like other mentioned techniques except of [30,25], assumes service reputations to be supplied by some reputation provider. Such a provider could be based on these mechanisms.

Later, in [25] the authors extend [24] to estimate system reliability. This approach is based on a system model very similar to ours. It describes communicating components as a set of concurrent state machine containing interaction protocols of components. This model is transformed into a Dynamic Bayesian Network that includes reliabilities of individual components. Compared to [25], our approach does not associate service reliability with its probability of start, instead sending transitions are assumed to carry reliability-relevant probabilities. Unlike [25], where component-based systems are considered, we assume that services have no failure dependencies.

With respect to existing approaches, our method represents a first attempt to develop methodology and tool support for predicting reliability of service choreographies at the early design stage based on formal methods.

6 Conclusion

In this paper we have proposed a technique for computing the expected reliability of service choreographies based on reputations of single services. The technique involved transforming metamodel instances of UML state machines into Markov decision processes in the form of an input to PRISM. The probabilistic model checker PRISM could then be used to determine the expected reliability. The approach has been implemented on the basis of state-of-the-art model transformation techniques and is UML compliant.

In the future, we intend to investigate how different forms of information about the correctness of services, some obtained by monitoring, some by a formal analysis, can be combined for reliability prediction. Furthermore, we will evaluate our approach on realistic case studies. This will in particular show whether our interpretation of reputations as being probabilities of sending transitions is the right choice for choreographies, or whether other choices, possibly depending on the application

domain, are valid as well.

References

[1] Atlas Transformation Language (ATL). http://www.eclipse.org/atl.

[2] Java Emitter Templates (JET). www.eclipse.org/emft/projects/jet.

[3] Probabilistic Symbolic Model Checker (PRISM). http://www.prismmodelchecker.org.

[4] TOPCASED Toolkit. http://www.topcased.org/.

[5] UML2 Tools. http://www.eclipse.org/modeling/mdt/uml2tools.

[6] Unified Modeling Language (UML) Version 2.1.1. http://www.omg.org/spec/UML/2.1.1, August 2007.

[7] R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 207–218, Washington, DC, USA, 1996. IEEE Computer Society.

[8] D. Ardagna, C. Ghezzi, and R. Mirandola. Model Driven QoS Analyses of Composed Web Services. In Petri Mähönen, Klaus Pohl, and Thierry Priol, editors, *ServiceWave*, volume 5377 of *Lecture Notes in Computer Science*, pages 299–311. Springer, 2008.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[10] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In Vittorio Cortellessa, Sebastián Uchitel, and Daniel Yankelevich, editors, *WOSP*, pages 54–65. ACM, 2007.

[11] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. In *Proceedings of the 2005 conference on Specification and verification of component-based systems*, SAVCBS '05, New York, NY, USA, 2005. ACM.

[12] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimisation in Service-Based Systems. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[13] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281 – 308, 2004.

[14] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures*, volume 5281 of *QoSA '08*, pages 119–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] S. S. Gokhale and K. S. Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *ISSRE*, pages 64–78. IEEE Computer Society, 2002.

[16] K. Goseva-Popstojanova and S. Kamavaram. Software reliability estimation under uncertainty: Generalization of the method of moments. In *HASE*, pages 209–218. IEEE Computer Society, 2004.

[17] V. Grassi. Architecture-based reliability prediction for service-oriented computing. In Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky, editors, *WADS*, volume 3549 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2004.

[18] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[20] S. Hwang, H. Wang, J. Tang, and J. Srivastava. A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. *Inf. Sci.*, 177(23):5484–5503, 2007.

[21] D. N. Jansen and H. Hermanns. QoS modelling and analysis with UML-statecharts: the StoCharts approach. *ACM SIGMETRICS Performance Evaluation Review*, 32:28–33, 2005.

[22] H. Koziolek and R. Reussner. A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *SIPEW*, volume 5119 of *Lecture Notes in Computer Science*, pages 58–78. Springer, 2008.

[23] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reliability prediction for component-based software architectures. *J. Syst. Softw.*, 66:241–252, June 2003.

[24] R. Roshandel, S. Banerjee, L. Cheung, N. Medvidovic, and L. Golubchik. Estimating software component reliability by leveraging architectural models. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 853–856. ACM, 2006.

[25] R. Roshandel, N. Medvidovic, and L. Golubchik. A bayesian model for predicting reliability of software systems at the architectural level. In Sven Overhage, Clemens A. Szyperski, Ralf Reussner, and Judith A. Stafford, editors, *QoSA*, volume 4880 of *Lecture Notes in Computer Science*, pages 108–126. Springer, 2007.

[26] N. Sato and K. S. Trivedi. Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2007.

[27] Y. Xia, J. Chen, M. Zhou, and Y. Huang. A Petri-Net-Based Approach to QoS Estimation of Web Service Choreographies. In Lei Chen, Chengfei Liu, Xiao Zhang, Shan Wang, Darijus Strasunskas, Stein Tomassen, Jinghai Rao, Wen-Syan Li, K. Candan, Dickson Chiu, Yi Zhuang, Clarence Ellis, and Kwang-Hoon Kim, editors, *Advances in Web and Network Technologies, and Information Management*, volume 5731 of *Lecture Notes in Computer Science*, pages 113–124. Springer Berlin / Heidelberg, 2009.

[28] S. M. Yacoub, B. Cukic, and H. H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(4):465–480, 2004.

[29] A. Zarras, P. Vassiliadis, and V. Issarny. Model-driven dependability analysis of webservices. In Robert Meersman and Zahir Tari, editors, *CoopIS/DOA/ODBASE (2)*, volume 3291 of *Lecture Notes in Computer Science*, pages 1608–1625. Springer, 2004.

[30] Z. Zheng and M.R. Lyu. Collaborative reliability prediction of service-oriented systems. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE (1)*, pages 35–44. ACM, 2010.

[31] D. Zhong and Z. Qi. A petri net based approach for reliability prediction of web services. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (1)*, volume 4277 of *Lecture Notes in Computer Science*, pages 116–125. Springer, 2006.

[32] B. Zhou, K. Yin, S. Zhang, H. Jiang, and A. J. Kavs. A tree-based reliability model for composite web service with common-cause failures. In Paolo Bellavista, Ruay-Shiung Chang, Han-Chieh Chao, Shin-Feng Lin, and Peter M. A. Sloot, editors, *GPC*, volume 6104 of *Lecture Notes in Computer Science*, pages 418–429. Springer, 2010.