Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 23–38
www.elsevier.com/locate/entcs

Parameter Dependencies for Component Reliability Specifications

Heiko Koziolek[1]

ABB Corporate Research
68526 Ladenburg, Germany

Franz Brosch[2,3]

Forschungszentrum Informatik (FZI) Karlsruhe
76137 Karlsruhe, Germany

Abstract

Predicting the reliability of a software system at an architectural level during early design stages can help to make systems more dependable and avoid costs for fixing the implementation. Existing reliability prediction methods for component-based systems use Markov models and assume that the software architect can provide the transition probabilities between individual components. This is however not possible if the components are black boxes, only at the design stage, or not available for testing. We propose a new modelling formalism that includes parameter dependencies into software component reliability specifications. It allows the software architect to only model a system-level usage profile (i.e., parameter values and call frequencies), which a tool then propagates to individual components to determine the transition probabilities of the Markov model. We demonstrate the applicability of our approach by modelling the reliability of a retail management system and conduct reliability predictions.

Keywords: CBSE, Reliability, Prediction, Usage Profile, Parameter Dependencies

1 Introduction

Formal techniques for analysing the properties of software design and software systems are useful not only for functional properties (e.g., correctness), but also for extra-functional properties (e.g., performance, reliability, security, safety, etc.). Predicting extra-functional properties of a system based on design models can help to avoid implementing software architectures that do not fulfil user requirements for timeliness and dependability. With this method, developers can save substantial

[1] Email: heiko.koziolek@de.abb.com

[2] Email: brosch@fzi.de

[3] This work is supported by the Seventh Framework Programme (FP7) of the European Comission, Grant Agreement ICT FP7-216556.

1571-0661© 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.09.026

costs for fixing an implementation based on a poor component-based software architecture.

For reliability predictions (i.e., probability of failure on demand (POFOD)) of a component-based system, software architects combine component specifications with failure probabilities given by (possibly third-party) component developers. However, component reliability – and particularly component interaction – depends on the usage profile assumed by the software architect. For example, unreliable but seldom used component services have little effect on the overall system reliability. The propagation of calls from one component to another may depend on the input parameter a certain service is called with. Therefore, component developers need to make the dependency between input parameters and the propagation of calls explicit in their component reliability specification, as they should not make assumptions about the users and the deployment environment of their components to keep them widely reusable.

Existing solutions for component-based reliability prediction (e.g., [3,6,5,19,15,2]) model the control flow in a component-based software architecture using Markov chains. They assume that the software architect constructing such a model can provide the transition probabilities in the Markov chains, which model the control flow propagation between individual components. This is not possible, if the software architect builds the prediction based only on component interface specifications or on implemented components viewed as a black box alone. From the component specification, the software architect does not know which required service a provided service calls upon invocation and how many calls are made.

We present a novel approach where component developers supply component reliability specifications based on so-called stochastic regular expressions (SRE), which extend regular expressions with probabilistic attributes. An SRE describes the call propagation through a component service in dependency to input parameter values. Software architects can compose these *parametrised* specifications using tools and add their application specific system-level usage profile. A transformation tool then solves the parameter dependencies inside the component reliability specifications and thereby deduces the component transition probabilities based on the system-level usage profile. With our approach, the software architect does not need to estimate transition probabilities inside the architecture model making reliability predictions more accurate.

We demonstrate the applicability of our approach by modelling the reliability of the component services of a retail management system supported by tools for the Palladio Component Model (PCM) [1]. They allow software architects to compose individual component specifications by different developers. We make predictions with varying system-level usage profiles and show the sensitivity of the overall system reliability to individual component failure probabilities and usage profile parameters.

The contribution of this paper is a modelling formalism to specify parameter dependencies in software component reliability specifications. Our approach of mod-

elling parameter dependencies can potentially be added to any existing component-based reliability prediction approach relying on the same assumptions thereby making it more accurate and flexible. Furthermore, our approach can be used for other compositional quality attributes (e.g., performance [9]).

The paper is organised as follows. Section 2 surveys related work. Section 3 describes the steps of our method and explains the involved developer roles. Section 4 defines our formalism on stochastic regular expressions and shows the transformation algorithm to derive Markov model from the formalism. Section 5 demonstrates our approach in a case study, followed by a discussion of assumptions in Section 6. Section 7 concludes the paper.

2 Related Work

The field of reliability prediction for component-based software architectures has been surveyed in [5,4,7]. The approach presented in this paper is in the class of state-based methods, which assume that the control flow between the components of a software architecture can be modelled using a Markov chain. Goseva et al. [5] state that most approaches rely on estimations of the transition probabilities between components. In our approach, no estimation is necessary, as each component specifies its call propagation in dependency to its own usage profile.

One of the first approaches for architecture-based reliability modelling was presented by Cheung in 1980 [3]. He emphasised the fact that a system is more reliable if the unreliable parts are rarely used. He introduced a Markov model to describe the control flow through a component-based software architecture. Cheung computes system reliability by analysing the Markov chain and incorporating the transition probabilities between different components, so that seldom used components only marginally contribute to system reliability.

Recently, several models based on Cheung's work have been introduced (e.g., [19,15]). Wang et al. [19] add special constructs to express architectural styles, while Sharma et al. [15] broaden the scope of architectural analysis and also analyse performance and security with a Markov model. None of the approaches provides special methods to determine the transition probabilities between individual components. Instead, they rely on testing data, which is not available for early design, or the software architect's intuition.

Hamlet et al. [6] specify reliabilities and call propagations for each individual component. Thus, the resulting models for each component shall be reusable across different architectures and system-level usage profiles. However, the approach requires the software architect to execute each component against the desired usage profile, which essentially reduces the approach to testing. The dependency of transition probabilities and input parameters is not made explicit in this approach.

Reussner et al. [14] took over the concept of parametrised contracts for software components to architecture-based reliability modelling. Each component specifies probabilities of calling required services. Therefore, an architecture model can be constructed by combining these individual specifications. The approach assumes

fixed transition probabilities and the models therefore have the usage profile implicitly encoded.

Cheung et al. [2] incorporate failure states into state-based models for individual components. They use information from requirements documents, simulation, and domain knowledge to train a hidden Markov model (HMM), which eventually delivers transition probabilities to failure states. However, the approach does not include the probability of control flow propagation to other components in relation to the usage profile.

3 Component Specification and Reliability Prediction

This section sketches the process model underlying our approach and describes the responsibilities for the involved developer roles.

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only [18]. A component specification consists of the provided and required interfaces of the component. This information is sufficient to assemble components and check their interoperability. However, additional information about each component is required to enable reasoning on extra-functional properties of a component-based architecture, such as performance, reliability, or availability.

To create a specification language that captures the additionally required information, it is necessary to consider the involved developer roles. In component-based software engineering (CBSE), there is a strict separation between *component developers* and *software architects*. Component developers implement components and provide functional and extra-functional specifications (i.e., models) for them. Software architects use these specifications to reason about planned architectures and later assemble the actual component implementations.

The component specifications for extra-functional properties provided by component developers need to describe how provided services of a component call required services in terms of probabilities, frequencies, and parameter values. This enables software architects to create a model of the control and data flow through the whole architecture by simply composing these specifications and not referring to component internals. In Section 4, we introduce a specification language based on stochastic regular expressions that lets component developers specify how provided services of a component call its required services via so-called external calls.

The propagation of calls from provided to required interfaces may depend on parameter values used as input to provided interface calls. For example, a component service could call a required service as many times as the length of a list supplied to it as an input parameter value. If, as in many existing component reliability specifications, the component specification does not reflect such dependencies, predictions based on the model are always fixed to a specific usage profile (i.e., a set of parameter values) assumed by the component developer. This is not desirable as software components should be reusable under varying usage profiles. The specification language introduced in Section 4 allows to express parameter dependencies

for the propagation of calls to required services. Therefore the system model can be adapted by different software architects to different usage profiles.

Fig. 1 depicts the process model underlying our approach. Each component developer provides a component specification, which includes extra-functional properties (i.e. in our case failure probabilities, etc.), call propagations to required services, and parameter dependencies of each provided service of the component. Methods to determine failure rates (e.g., [2]) are out of scope for this paper. If the component is already implemented, the component developers may use static code analysis techniques (e.g., in [8]) or dynamic monitoring (e.g., in [11]) to derive call propagations and parameter dependencies from their source code. Component specifications reside in public repositories, where software architects can obtain them.

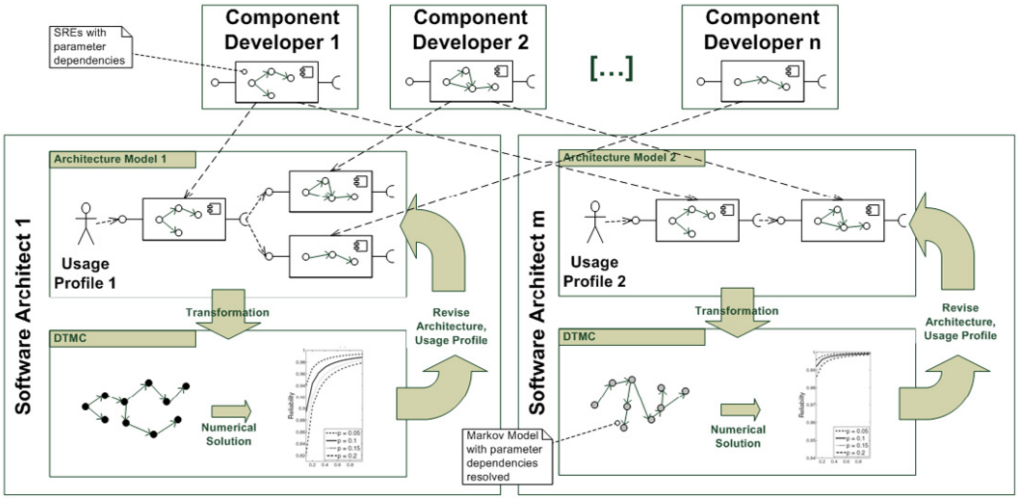

Fig. 1. Model-based Reliability Prediction for Component-Based Architectures

Each software architect assembles the component specifications in the same manner as later the component implementations. Additionally, each software architect provides a usage profile for the complete system (i.e., having direct interaction with the user or other systems). Based on this information, a tool traverses the architectural model and resolves the parameter dependencies in the component specifications.

For example, if the usage profile states that the length of a list supplied as an input parameter is always 10, and the component specification states that the called component calls 2 required services for each list item, the transformation tool determines that this component makes 20 calls to required services. As depicted in Fig. 1, another software architect could parametrise the model differently (e.g., list length 20) to get a different architecture model (i.e., 40 calls).

This method works recursively, as each component specification includes the (parametrised) usage profile for calling its required services. Therefore, with our specification language, it is possible to propagate requests at the system level to individual components and describe the control and data flow throughout the whole architecture. This is not possible with other approaches for reliability prediction

(e.g., [3,14]), which rely on the software architect to model the control flow based on provided and required interfaces only.

With the parameter dependencies resolved, the resulting model can be transformed into a discrete-time Markov chain (DTMC). Using existing techniques (e.g., [14]), the software architect can determine the probability of reaching the failure state of the Markov chain to get a reliability prediction (i.e., POFOD) for each component service at the system boundaries based on the failure probabilities of the components inside the architecture. This enables the software architect to assess the feasibility of reliability requirements.

If the predictions show that given reliability goals cannot be met, there are several options. The software architect can revise the architecture model for example by introducing redundancy or trying different architectural configurations. The software architect can also check the effect of using other functional equivalent components with different reliability properties. If possible, the modelled usage profile may need to be adapted or the reliability goals have to be changed. This method of model-based reasoning is supposed to be significantly more cost-efficient than building prototypes or relying on personal experience with similar systems only.

4 Parameter Dependencies in Component Reliability Specifications

This section describes our model for component reliability specification (Section 4.1), and system modelling (Section 4.2). Afterwards it explains the necessary transformation steps to deduce a Markov model from the specification (Section 4.3), and shows how to solve the Markov model to get a reliability prediction (Section 4.4).

4.1 Stochastic Regular Expressions for Component Specification

Our approach requires component developers to describe the reliability and control and data flow propagation of each provided service of a component using a stochastic regular expression (SRE). SREs consist of internal actions, external calls, a return value, and control flow statements as explained in the following. Additionally they use a language for arithmetic and boolean expressions. An example will be provided further below.

- **Internal Actions:** Let $I = \{1, ..., n\}, n \in \mathbb{N}$ be an index set and $IA = \{ia_1, ..., ia_n\}$ be a set of terminal symbols. ia_i refers to an *internal action* of the provided service modelled by the SRE. Let $fp : IA \rightarrow [0, 1]$ be a function, which assigns a *failure probability* to each internal action. Failure probabilities can be derived using statistical testing [12,2].
- **Expression Language:** Let L be a *language* for boolean and arithmetic expressions defined by the grammar in Appendix A. Let $S \subset L$ be the set of all strings in L, $B \subset L$ be the set of all boolean expressions in L, and $A \subset L$ the set of all arithmetic expressions in L. Let $\pi \in B$ be a boolean expression and $v \in A$ be an

arithmetic expression. Notice that L allows the specification of numbers as well as probability mass functions (PMF) to model values in a stochastic manner.

- **External Calls:** Let $J = \{1, ..., m\}, m \in \mathbb{N}$ be an index set and $EC = \{E_1, ..., E_m\}$ be a set of non-terminal symbols. E_j refers to an *external call* to a required service. E_j is a non-terminal symbol and can only be substituted by the SRE of the referenced required service after its composition to other components is fixed. We assume a call and return semantic (i.e., synchronous communication, the caller blocks until receiving an answer).

  Let $ei : EC \rightarrow (S \times A)$ and $eo : EC \rightarrow (S \times A)$ be two functions that assign parametric *input* and *output* expressions to an external call. For example $ei(E_1) = (a, x+1)$ means that the SRE calls the required service where the value $x + 1$ is assigned to the variable named a.
- **Return Value:** Let $rv \in \mathbb{N}$ denote a *return value*, which models the output produced by the modelled service, which is send back to the caller.

Let P be a non-terminal symbol. Then, the syntax of SREs in our approach is as follows:

$$P := ia_i \mid E_j \mid P \cdot P \mid P +_\pi P \mid P^\upsilon$$

The semantic of the above syntax is:

- $P \cdot P$ denotes a **sequence** of two symbols. The dot can be omitted.
- $P +_\pi P$ denotes a **branch** of two symbols. π represents a **branch condition** (e.g. $\pi := y < 0$, $y \in \mathbb{N}$, y denoting a variable) defined as a boolean expression of the language L.
- P^υ denotes a **loop**. υ represents a **parametric loop count** (e.g., $\upsilon := 3 + z, z \in \mathbb{N}$, z denoting a variable). Infinite loops are not allowed by SREs, the loop count is always bound.

Example: The following example shows a complete stochastic regular expression:

$$P := ia_1 \cdot (E_1 +_\pi E_2), \quad fp(ia_1) = 0.001, \quad ei(E_1) = (x, 2 - x), \quad eo(E_1) = \emptyset$$

$$ei(E_2) = (y, 27 * x + 3), \quad eo(E_2) = (z, x + 1), \quad \pi := y < 0, \quad rv := 0$$

It specifies that the service first executes some internal code (ia_1) with the failure probability 0.001 and then either executes an external calls E_1 or E_2 depending on the value of the input parameter y.

Rationale: Notice that an SRE is an *abstraction* of a component service's behaviour. It includes control flow constructs only if they influence external calls. Internal control flow of the service not changing the behaviour visible from outside are abstracted within internal actions. A single internal action may represent thousands of lines of code with a single failure probability. This abstraction focuses on necessary properties for a component-based reliability prediction (i.e., failure probabilities and call propagations) and makes the analysis of even complex components mathematically tractable.

The included parameter dependencies for branch conditions and loop counts allow software architects to use varying usage profiles (i.e., the value assignments for the included variables). We restrict the values of parameters to integer values in this paper for clarity. This is not a general restriction. The formalism can be easily extended to other data types as shown in [9]. However, the value domain is always finite and discrete in any case. For large or unlimited value domains, a partitioning of the value domain into a manageable set of equivalence classes with similar reliability properties have to be found (also see [6]).

Using SREs instead of Markov models allows for structured control flow with clearly defined loop entry and exit points (also see [10]).

4.2 System Model

Usually, a whole use case spanning multiple SREs is the subject of a reliability prediction. Thus a complete system model consists of a set of SREs, with an initial usage profile.

Let $K = \{1, ..., r\}, r \in \mathbb{N}$ be an index set and $Services = \{P_1, \cdots, P_r\}$ be a set of connected SREs realising the functionality of one use case. The included SREs reference each other via their external calls. Let P_1 be the service called at the system boundaries.

Let $T = \{1, ..., t\}, t \in \mathbb{N}$ be an index set, Let $V = \{v_1, \cdots, v_t\} \subset S$ be the set of strings denoting the variable names specified in P_1, and let $N = \{n_1, \cdots, n_t\} \subset L$ be a set numbers or probability mass functions from L defining the initial values for the variables at the system boundaries. Then $UsageProfile = \{(v_1, n_1), \cdots, (v_t, n_t)\}$ denotes the system-level usage profile for the modelled use case.

Then, the system model is defined as: $SystemModel = \{UsageProfile, Services\}$.

4.3 Transformation

Once a software architect has assembled a set of SREs from a repository to realise a use case, and specified the usage profile to form a $SystemModel$, an automatic transformation (implemented as an Eclipse plugin) substitutes the variables in the first SRE with the values from the usage profile and propagates them through all connected SREs. The transformation algorithm in Listing 1 *traverses* the abstract syntax trees of the SREs of all participating component services.

Listing 1: Transformation Algorithm

```
Input:  P (SRE to transform)
        U (usage profile, i.e., a set of variables with values assigned)
        Z (pointer to the current state in the Markov model)
Output: Z (pointer to the current state in the Markov model)

Side Effect Input:  MM (initial empty markov model)
                    SS ∈ MM (start state in the markov model)
                    FS ∈ MM (failure state in the markov model
Side Effect Output: MM (modified markov model constructed in parallel)

pointer transform (P, U, Z) {
  //traverse abstract syntax tree of SRE P top-down
```

```
  switch(currentNode){
    case sequence with leftSRE, rightSRE:
      Z <- transform(leftSRE, U, Z);
      Z <- transform(rightSRE, U, Z);
      break;

    case branch with leftSRE, rightSRE, π as branch condition
      U1 <- evaluateStochastically(U,π); //new usage profile evaluated under
          the condition that π is true
      U2 <- evaluateStochastically(U,not π); //new usage profile evaluated
          under the condition that π is false
      π <- searchAndReplace(π, U); //replace variables in π with current
          values from usage profile
      bp <- solveExpression(π); // compute the branch probability

      S1 <- newState(); //initial state for the left branch
      S2 <- newState(); //initial state for the right branch
      S3 <- newState(); //state to join the branches again
      T1 <- newTransition(Z,S1,bp);
      T2 <- newTransition(Z,S2,1−bp);

      Z <- transform(leftSRE, U1, S1); // transform left branch
      T3 <- newTransition(Z,S3,1.0); // join branch
      Z <- transform(rightSRE, U2, S2); // transform right branch
      T4 <- newTransition(Z,S3,1.0); // join branch

      MM <- MM ∪S1 ∪ S2 ∪ S3 ∪ T1 ∪ T2 ∪ T3 ∪ T4;
      Z <- S3;
      break;

    case loop with body bodySRE, v as parametric loop count
      v <- searchAndReplace(v, U); //replace variables in v with current
          values from usage profile
      nl <- solveExpression(v); //compute number of iterations
      for i=1 to max(nl) do
        Z <- transform(bodySRE, U, Z); // unroll loops
      break;

    case ia with failure probability fp(ia):
      S1 <- newState();
      T1 <- newTransition(Z,S1,1−fp(ia));
      T2 <- newTransition(Z,FS,fp(ia)); // connect to failure state
      MM <- MM ∪S1 ∪T1 ∪T2;
      Z <- S1;
      break;

    case E with SRE as the called service and rv as the return value of SRE:
      input <- searchAndReplace(ei(E), U);
      U1 <- solveExpression(input); //add external call inputs to new usage
          profile
      Z <- transform(SRE, U1, Z);
      output <- searchAndReplace(eo(E), U ∪ rv);
      U <- U ∪ solveExpression(output); // add outputs to usage profile
      break;
  }
  return Z;
}
```

The algorithm creates a *Markov model* as a side effect while traversing the abstract syntax trees of the SREs. This Markov model contains a failure state, which is taken if an internal action fails. Therefore, the transition probability from a state representing an internal action ia_i to the failure state is its failure probability $fp(ia_i)$.

The Markov model contains a sequence of states for subsequent internal actions and transitions for branches. Additionally, the algorithm unrolls the loops within the SREs in the Markov model (i.e., it creates the states for the symbols inside a loop body for each loop count).

After substituting a variable name with its current value from the usage profile, the algorithm solves the resulting expression. For example, for a boolean expression $x < 0$ as a branch condition and a usage profile containing $P(x \geq 0) = 0.7$, the algorithm resolves a branch probability $bp = 1 - P(x \geq 0) = 0.3$ and uses it in the Markov model as the transition probability. Analogously, for an arithmetic expression $3+x$ as a parametric loop count and a usage profile $x = 5$, the algorithm resolves a loop count $lc = 8$ and uses it to unroll the states in the Markov model.

The function `solveExpression` supports computing all arithmetic and boolean operations specifiable by the expression language L. They can be arbitrarily complex and involve constants as well as probability mass functions.

Upon traversing an external call E_j, the algorithm creates a new usage profile for the called SRE. It first resolves the parameter dependencies on the inputs of the call defined by the function $ei(E_j)$. Then it uses these values as new usage profile. After traversing the called SRE, the algorithm resolves parameter dependencies on the outputs of the call defined by the function $eo(E_j)$. The parametric dependencies of these output values may refer to the return value rv of the called SRE.

For the first invocation, the SRE P_1 of a $SystemModel$ is given as input parameter as well as the $UsageProfile$ of the $SystemModel$.

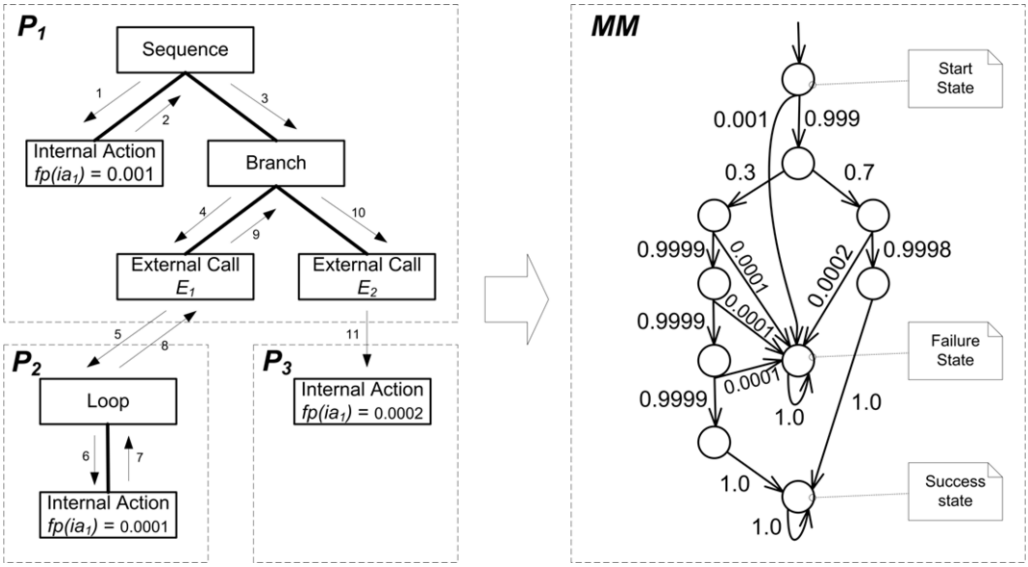

Fig. 2. Transformation Example: SRE Traversal (left), Resulting Markov Model (right)

Example: As an example, we use the SRE P from Section 4.1 and connect its external calls E_1 and E_2 to the following two SREs P_2 and P_3 to form a set of $Services$.

$$P_2 := (ia_1)^v, \quad fp(ia_1) = 0.001, \quad v = x + 2, \quad rv = 0$$
$$P_3 := ia_1, \quad fp(ia_1) = 0.002 \quad rv = 0$$

Together with the $UsageProfile = \{(x, 1), (P(y = -2) = 0.3), (P(y = 4) = 0.7)\}$, we form the $SystemModel = \{UsageProfile, \{P_1, P_2, P_3\}\}$. Running the

transformation algorithm in Listing 1 initiates the traversal of the abstract syntax trees of the SREs depicted on the left hand side of Fig. 2. By solving the parameter dependencies for the branch condition $y < 0$ and the parameteric loop count $x + 2$ with the given $UsageProfile$ the algorithm calculates a branch probability of 0.3 to call E_1 and a loop count of 3.

After termination, the algorithm has created the Markov model on the right hand side of Fig. 2. It includes a start state, a failure state, and a success state.

4.4 Reliability Prediction

With the Markov model created by the transformation, we can now compute the reliability of the overall system model (i.e., for the modelled use case). In our approach we define the reliability as $R := 1 - POFOD$, where POFOD is the probability of failure on demand. It can be computed from the Markov model by calculating the probability of reaching the success state from the start state [17].

Example: The transition matrix for the Markov chain generated in the former example is

$$P = \left(\begin{array}{cccccccc|cc} 0 & 0.999 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.001 \\ 0 & 0 & 0.3 & 0.7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9999 & 0 & 0 & 0 & 0 & 0.0001 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9998 & 0 & 0.0002 \\ 0 & 0 & 0 & 0 & 0 & 0.9999 & 0 & 0 & 0 & 0.0001 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.9999 & 0 & 0 & 0.0001 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \end{array}\right)$$

where the second to last column refers to transition probabilities to the success state and the last column refers to transition probabilities to the failure state. As this is an absorbing Markov chain, we can compute the probability $b_{i,j}$ that the chain will be absorbed in the absorbing state s_j if it starts in the transient state s_i (cf. [17]). Let $N = (I - Q)^{-1}$ be the fundamental matrix computed from the identity matrix I and the upper left transition matrix Q. Let R be the upper right transition matrix from the canonical form of P. Then, the absorption probabilities are determined by the matrix B with the entries $b_{i,j}$, which is given by:

$$B = NR = (I - Q)^{-1}R =$$

$$\left(\begin{array}{cccccccc} 1 & 0.999 & 0.2997 & 0.6993 & 0.29967 & 0.29964 & 0.29961 & 0.69916 \\ 0 & 1 & 0.3 & 0.7 & 0.29997 & 0.29994 & 0.29991 & 0.69986 \\ 0 & 0 & 1 & 0 & 0.9999 & 0.9998 & 0.9997 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0.9998 \\ 0 & 0 & 0 & 0 & 1 & 0.9999 & 0.9998 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0.9999 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \left(\begin{array}{cc} 0 & 0.001 \\ 0 & 0 \\ 0 & 0.0001 \\ 0 & 0.0002 \\ 0 & 0.0001 \\ 0 & 0.0001 \\ 1.0 & 0 \\ 1.0 & 0 \end{array}\right) = \left(\begin{array}{cc} 0.99877 & 0.001229761 \\ 0.99977 & 0.000229991 \\ 0.9997 & 0.00029997 \\ 0.9998 & 0.0002 \\ 0.9998 & 0.0002 \\ 0.999 & 0.0001 \\ 1.0 & 0 \\ 1.0 & 0 \end{array}\right)$$

Thus, the reliability (i.e., the probability of reaching the success state) of the example under the given usage profile is $R = b_{0,0} = 0.99877$.

5 Case Study

To illustrate our approach, we take a retail management system as an example, which has been designed in a service-oriented way. It is taken from the SLA@SOI research project [16] and based on the Common Component Modelling Example [13]. The system enables a retail chain to handle its sales and manage its inventory data. Fig. 3 shows parts of the system architecture.

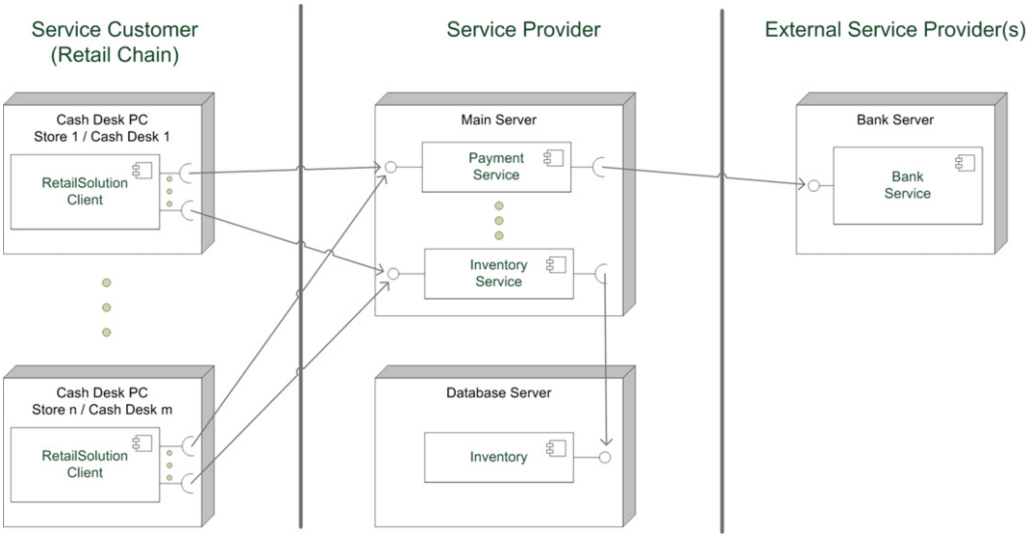

Fig. 3. System Architecture of the Service-oriented Retail Management System (excerpt)

The retail chain takes the role of the service customer. It operates several stores, each of which contains multiple cash desk lines. At each cash desk line, a dedicated retail solution client handles the individual sales processes. The service provider offers a range of services to the retail solution clients, such as a payment service to handle payments by cash or by credit card, and an inventory service to update stock information in the inventory. The service provider in turn uses services of external providers. Fig. 3 shows a bank service used by the payment service as an example.

We focus on the service operation `bookSale()` that belongs to the inventory service. The retail solution clients invoke this operation at the end of each sales process in order to update the stock information. The stochastic regular expression which corresponds to the `bookSale()` execution looks as follows:

$$P_1 = P_{bookSale} := E_2(ia_1 +_\gamma (E_3 +_\pi E_4))^\upsilon E_5, \quad \upsilon := x, \quad \gamma := (y = 0), \quad \pi := (z = 0)$$

$$P_2 := ia_2, \quad P_3 := ia_3, \quad P_4 := ia_4, \quad P_5 := ia_5$$

The operation takes a list of all sold items as an input parameter, and contains a loop to iterate through this list. The iteration number υ of the loop equals the number x of items in the list. Within the loop, stock information is updated by external calls E_3 and E_4 to the inventory component service operations P_3 and P_4.

Additionally, two external calls E_2 and E_5 to service operations P_2 and P_5 (to set up and to commit the database transaction) surround the loop over all sold items. All inventory component service operations are modelled through single internal actions ia_2, ia_3, ia_4 and ia_5.

The control flow of the `bookSale()` operation distinguishes several cases. First, an item may be rated as small goods that do not have to be declared in the inventory. A branch located in the loop body reflects this fact. If an item is small goods (which corresponds to y having value 0), there is no need for stock information update. Rather, some internal calculations ia_1 are done. The probability $P(y = 0)$ is part of the usage profile. Second, an item can be identified by a number or by a name, which results in two different types of database queries P_3 and P_4. Again, the usage profile determines the probabilities of both alternatives through $P(z = 0)$.

Our method allows assigning failure probabilities to internal actions. For example, a call to an inventory component service operation might fail - the underlying database might be temporarily unavailable, overloaded, or the request sent to the database might be lost. Our approach includes the propagation of the usage profile from the inventory service component to the inventory component: calling the `bookSale()` operation with a list of x items results in an expected call sequence of $P(y \neq 0) * P(z = 0) * x$ and $P(y \neq 0) * P(z \neq 0) * x$ calls to the two stock update operations, surrounded by one call for transaction setup and one call for transaction commit.

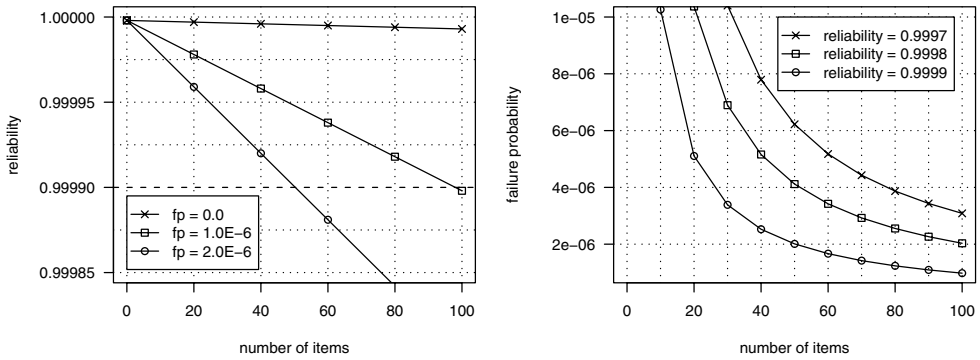

Fig. 4. Reliability of the `bookSale()` operation

In our case study, we are interested in analysing the relation between the usage profile of the `bookSale()` operation and its reliability. We assume the failure probabilities of the internal actions involved to be 10^{-6}, except the failure probabilities of stock information update ($fp(ia_3)$, $fp(ia_4)$), which we vary between 0 and $2 * 10^{-6}$. We set small goods to 5% and items identified by name to 20%.

For our reliability calculation, we have modelled the retail management system using tools of the Palladio Component Model [1]. We have implemented an additional algorithm for Markov Chain generation and calculation of the resulting reliability.

The left hand side of Fig. 4 shows the results. Evidently, the reliability decreases with a growing number of items. A longer list of items results in a higher number

of loop iterations, leading to more stock update operations. Only in the theoretical case that the stock update cannot fail at all, the reliability is nearly independent of the number of items. In that case, it is determined only by the potential failure of transaction setup and commit, as well as the internal calculation ia_1.

If we set a goal of 99.99% for the `bookSale()` reliability, the figure shows that this goal is reached with a stock update failure probability of 10^{-6} and at most 98 items. If the failure probability rises to $2 * 10^{-6}$, the goal is only reached with a number of items smaller than 50.

The right hand side of Fig. 4 shows the relation between the number of items and the failure probability of database queries for varying reliability goals. Generally, the acceptable failure probability decreases for higher reliability goals and for a higher number of sold items. If, for instance, 60 items are sold, a failure probability of $2 * 10^{-6}$ would be acceptable to reach 99.98% reliability, but it would violate a reliability goal of 99.99%.

6 Assumptions and Limitations

In this section, we discuss assumptions and limitations of our approach. Presumably, the most critical assumption lies in the determination of failure probabilities for internal actions. The predicted reliability can only be as close to reality as the failure probabilities given as an input to the method. Cheung et al. [2] propose estimating these probabilities through domain experts. Lyu et al. [12] suggest using statistical testing. If probabilities of failure are only roughly estimated, the approach can still be valuable in comparing architectural alternatives, or determining acceptable ranges for failure probability.

Besides individual failure probabilities, the parametric dependencies of the control flow also need to be given as an input to reliability prediction. The specification of the internal behaviour of components given by the component developer or determined through a reverse engineering process can serve as a source of information [8]. If no specification exists and the source code is not available, it might be still possible to reconstruct the parametric dependencies by monitoring the inputs and outputs of the component by running it as a black box in a test-bed [11].

Our method abstracts from the concrete faults that may cause a failure. A service execution may alter the state of a component, such as the values of variables owned by the component, in an invalid way. The faulty component state might not immediately result in a failure, but at a later point in time. Such coherences cannot be explicitly expressed by our approach. The same is true for the probability of failure due to usage of certain resources or message loss over a certain communication link. Also, failures caused by concurrency are not explicitly captured by the SREs. Our approach aggregates the distinct causes of failure into one probability value, which tends to decrease accuracy of prediction.

If an internal action fails, we assume that the corresponding service call is also rated as being failed. We do not consider the possibility that the system handles the failure and might successfully finish service execution even though the failure

happened. Furthermore, we assume the failure probabilities of internal actions to be stochastically independent. Currently the failure probability values are fixed constants. They cannot be adapted to take factors like system or component state at run-time, previous service calls or the control flow of the current call into account. Such considerations are left as a topic for future work.

7 Conclusions

We presented a novel formalism to model software component reliability specifications. The formalisms allows to specify the control flow propagation of a software component from provided to required services in dependency to input parameter values. Software architects compose these specifications from individual component developers, use a tool to transform the specification into a Markov model parametrised for their specific usage profile, and can then conduct reliability predictions for their design.

Our approach helps component developers by providing a language to specify component reliability. It helps software architects, who can quickly assemble component reliability specification, and parametrise them for different usage profiles without knowing the internals of the components. Similar existing component reliability prediction approaches can adopt our method of modelling to make their prediction more accurate and more flexible. By assessing the reliability of a system design during early development stages, potentially high costs for late life-cycle fixings of a system can be avoided.

Our approach is still in an initial stage. We plan to add information about the underlying hardware resources to our model to allow to make more refined predictions. Furthermore, we will add reliability models for network connections and connectors between components. Another important direction is the determination of valid failure probabilities. We plan to incorporate existing approaches for determining failure probabilities into our own method and validate it on a large real-life case study.

References

[1] Becker, S., H. Koziolek and R. Reussner, *The Palladio Component Model for Model-Driven Performance Prediction*, Journal of Systems and Software **82** (2009), pp. 3–22.

[2] Cheung, L., R. Roshandel, N. Medvidovic and L. Golubchik, *Early prediction of software component reliability*, in: *ICSE '08: Proceedings of the 30th international conference on Software engineering* (2008), pp. 111–120.

[3] Cheung, R. C., *A user-oriented software reliability model*, IEEE Trans. Softw. Eng. **6** (1980), pp. 118–125.

[4] Gokhale, S. S., *Architecture-based software reliability analysis: Overview and limitations*, IEEE Trans. on Dependable and Secure Computing **4** (2007), pp. 32–40.

[5] Goseva-Popstojanova, K. and K. S. Trivedi, *Architecture-based approaches to software reliability prediction*, Computers & Mathematics with Applications **46** (2003), pp. 1023–1036.

[6] Hamlet, D., D. Mason and D. Woit, *Theory of software reliability based on components*, in: *Proc. 23rd Int. Conf. on Software Engineering (ICSE '01)* (2001), pp. 361–370.

[7] Immonen, A. and E. Niemel, *Survey of reliability and availability prediction methods from the viewpoint of software architecture*, Journal on Softw. Syst. Model. **7** (2008), pp. 49–65.

[8] Kappler, T., H. Koziolek, K. Krogmann and R. Reussner, *Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering*, in: *Proc. Software Engineering 2008 (SE'08)*, LNI **121** (2008), pp. 140–154.

[9] Koziolek, H., "Parameter Dependencies for Reusable Performance Specifications of Software Components," Ph.D. thesis, Department of Computing Science, University of Oldenburg, Germany (2008).

[10] Koziolek, H. and V. Firus, *Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation*, in: J. Kuester-Filipe, I. H. Poernomo and R. Reussner, editors, *Proc. 3rd International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'06)*, ENTCS **176** (2006), pp. 69–87.

[11] Kuperberg, M., K. Krogmann and R. Reussner, *Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models*, in: *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE 2008), Karlsruhe, Germany, 14th-17th October 2008*, LNCS **5282** (2008), pp. 48–63.

[12] Lyu, M. R., *Software reliability engineering: A roadmap*, in: *FOSE '07: 2007 Future of Software Engineering* (2007), pp. 153–170.

[13] Rausch, A., R. Reussner, R. Mirandola and F. Plášil, editors, "The Common Component Modeling Example: Comparing Software Component Models," LNCS **5153**, Springer, 2008.

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

[15] Sharma, V. S. and K. S. Trivedi, *Quantifying software performance, reliability and security: An architecture-based approach*, Journal of Systems and Software **80** (2007), pp. 493–509.

[16] SLA@SOI, *Empowering the Service Economy with SLA-aware Infrastructures*, ICT: FP7-216556 (2008), http://www.sla-at-soi.eu/.

[17] Stewart, W., "Introduction to the Numerical Solution of Markov Chains," Princeton University Press, 1994.

[18] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[19] Wang, W.-L., D. Pan and M.-H. Chen, *Architecture-based software reliability modeling*, Journal of Systems and Software **79** (2006), pp. 132–146.

A Stochastic Expression Language

The following depicts the EBNF of the stochastic expression language:

$$
\begin{aligned}
&< expression > ::= < compare-expr > \\
&< compare-expr > ::= < sum-expr > | < sum-expr > < compare-op > < sum-expr > \\
&< compare-op > ::= " < " \mid " > " \mid " \leq " \mid " \geq " \mid " = " \mid " \neq " \\
&< sum-expr > ::= < prod-expr > | < prod-expr > < sum-op > < prod-expr > \\
&< sum-op > ::= " + " \mid " - " \\
&< prod-expr > ::= < atom > | < atom > < prod-op > < atom > \\
&< prod-op > ::= " * " \mid " / " \\
&< atom > ::= < number > | < string > | < id > | < pmf > \mid "(" < compare-expr > ")" \\
&< pmf > ::= "PMF[" < samples > "]" \\
&< samples > ::= < sample > \mid < sample > < samples > \\
&< sample > ::= "(" < number > ";" < number > ")" \\
&< number > ::= < digit > | < digit > < number > \\
&< digit > ::= "0" \mid ... \mid "9" \\
&< string > ::= < char > | < char > < string > \\
&< char > ::= "a" \mid ... \mid "z" \\
&< id > ::= < ucchar > | < ucchar > < id > \\
&< ucchar > ::= "A" \mid ... \mid "Z"
\end{aligned}
$$