



# Reliability prediction for component-based software architectures

Ralf H. Reussner<sup>a</sup>, Heinz W. Schmidt<sup>b,\*</sup>, Iman H. Poernomo<sup>a</sup>

<sup>a</sup> *Distributed Systems Technology Center, Melbourne, Australia*

<sup>b</sup> *School of Computer Science and Software Engineering, Monash University, 900 Dandenong Road, 3145 Caulfield, Vic., 3800, Australia*

Received 21 January 2002; received in revised form 9 April 2002; accepted 17 May 2002

## Abstract

One of the motivations for specifying software architectures explicitly is the use of high level structural design information for improved control and prediction of software system quality attributes. In this paper, we present an approach for determining the reliability of component-based software architectures.

Our method is based on rich architecture definition language (RADL) oriented towards modern industrial middleware platforms, such as Microsoft's .NET and Sun's EJB. Our methods involve parameterised contractual specifications based on state machines and thus permits efficient static analysis.

We show how RADL allows software architects to predict component reliability through compositional analysis of usage profiles and of environment component reliability. We illustrate our approach with an e-commerce example and report about empirical measurements which confirm our analytical reliability prediction through monitoring in our reliability test-bed. Our evaluation confirms that prediction accuracy for software components necessitates modelling the behaviour of binary components and the dependency of provided services on required components. Fortunately, our measurements also show that an abstract protocol view of that behaviour is sufficient to predict reliability with high accuracy. The reliability of a component most strongly depends on its environment. Therefore, we advocate a reliability model parameterized by required component reliability in a deployment context.

© 2002 Elsevier Science Inc. All rights reserved.

*Keywords:* Reliability; Availability; Component-based software; Software architecture

## 1. Introduction

Compositionality demands the possibility to reason about system properties based on just the external (or interface) abstractions and the architectural compositions (de Roeper et al., 1998). The use of software architectures for predicting quality attributes of the overall system is one of the original motivations in the field of software architecture (Shaw and Garlan, 1996). Software architecture is a high level abstraction of a software system: its components and their connections. Thus, architecture complements component definition which focuses on the individual components and their interfaces. Interface specifications are the hallmark of component-based software engineering (CBSE). Given

the connection between components and architecture, it is natural to extend contracts to the level of architectural specifications, and worthwhile to develop specialised methods for the prediction of quality attributes for component-based software architectures (Hamlet et al., 2001).

To be able to predict reliability, the component developer must make concrete assumptions about the deployment context. We make two observations.

*Unknown usage profile:* Design and implementation faults of software have a different impact on the reliability of the software, depending on how frequently the faulty code is executed. Different usage profiles of software arise in the context of different deployments but also as a result of changes of use—a kind of software aging.

*Unknown required context:* In CBSE components rely on other components in the environment. The exact properties of these components are not known until deployment. Such external, unknown components include

\* Corresponding author. Tel.: +61-3-9905-2479; fax: +61-9903-2863.

*E-mail addresses:* reussner@dstc.monash.edu.au (R.H. Reussner), heinz.schmidt@monash.edu.au, hws@monash.edu.au (H.W. Schmidt), imanp@dstc.monash.edu.au (I.H. Poernomo).

middleware (such as servers mapping web interfaces to back office data bases), operating systems, network and transport services, each potentially a point of failure if the component relies exclusively on it. Therefore, the reliability of a component depends on the reliability of its context.

Consequently, in our work we take the view that component-based interfaces and architectures need to be parameterised by the usage profile and the required components' reliability. Our approach uses the architectural composition of software to achieve compositionality for such parameterised reliability models. The binary deployment of components in CBSE implies executability. We make use of this fact to derive some of the required usage and reliability profiles automatically by execution. This leads to a partly empirical, execution-based approach to reliability evaluation and validation. The availability of (some) required components and potentially (some) usage profiles of components that are part of the “real” environment are clearly beneficial—provided the system architecture is tightly connected with the final implementation of the system.

The contribution of this paper is a novel method for predicting the reliability for component-based systems. Our prediction method overcomes the problem of missing usage and context information for components. Firstly, this is achieved by enhancing the solution given in (Hamlet et al., 2001). Hamlet proposes the separation of reliability and usage profile. But his methods focus on functions and require the component source. Our methods enable the user to compute directly the reliability of a component as a function of the usage profile. Secondly, we model parameterised contracts (Reussner, 2001b). These compute the protocols for services as a function of required services. This paper extends parameterised contracts to parameterised reliability contracts.

We start this paper with a brief summary of some fundamentals of reliability theory and the motivation for our component-oriented notions of reliability. Then we extend the component-oriented model to an architecture-based model of reliability, showing how to use the hierarchical composition to derive higher-level reliability models. Lastly, we empirically validate the quality of our predictions with data obtained from measurements on an example system.

## 2. Modelling reliability of software architectures

System reliability cannot be equated to software component reliability. Component interactions make a system more than the sum of its parts - and make system reliability a very complex design-specific function of external component reliabilities and the probability of rare human failure. This is shown for instance in the well

documented Therac-25 failure.<sup>1</sup> With the increasing interoperation and networking of software systems, the increasing capability and speed of communication between components and systems, errors can spread widely before humans can intervene. Fault-tolerance requires a systematic and formal approach to reliability. But component-based models for reliability, especially compositional ones are lacking. Software reliability is defined as the *probability of failure-free operation of a software system for a specified period of time in a specified environment*. It is a function of the *software faults* and its operational profile, i.e., the inputs to and the use of the software. For open systems, reliability is also a function of the reliability of essential required services in the deployment context of a software component.

*Reliability, availability and mean-time between failure.* For measuring and predicting system reliability, we use the following basic notions (John D. Musa and Okumoto, 1987; Laprie and Kanoun, 1996): mean time to failure (MTTF) defines the average time to the next failure; mean time to repair (MTTR) is the average time it takes to diagnose and correct a fault, including any reassembly and restart times; *mean time between failures* (MTBF) is simply defined as  $MTBF = MTTF + MTTR$ ; *the failure rate* is the number of failures per unit time. It is reciprocal to MTBF. Another important concept closely related to reliability is *availability*. This is defined as *the probability of a system being available when needed*. Availability, or more specifically, *instantaneous availability*, is typically defined as the fraction of time during which a component or system is functioning acceptably, i.e., the uptime over the total service time

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$$

*Execution-based component reliability modeling, measurement and prediction.* For systems, in particular hardware systems, MTTF and MTTR are measured. For many systems, the failure rate and thus MTBF is constant—assuming that system changes can be ignored. The MTBF is then proportional to the length of time considered and is equated to the reliability. Moreover, repair times are often not meaningful for software, or, repairs may introduce faults. Therefore failure rate is more commonly used as a basis for software reliability measurement. Since a particular software component is not running all the time, we measure its reliability relative to execution time or the number of calls. This fits well with our notion of protocols of behaviour specified by finite state machines (FSMs) or Petri nets. The execution of a protocol successively “fires” transitions and failure rates are relative to the length of firing sequences.

<sup>1</sup> cf. for instance records of the comp.risks newsgroup.

*Operational profiles.* Reliability is typically measured over large numbers of *runs*. An execution, which might take months or years, is divided into these runs more or less naturally depending on the type of system. A run could be for example a single cycle in a closed-loop real-time control system or the execution of a single transaction in a transactional environment such as online banking. Runs give rise to *run types*: repetitions of similar executions. Probability distributions over run types, required inputs (such as account types or ranges of deposit and withdrawal amounts) and other environment and resource parameters define the *operational profile* of a software system. We model runs by execution traces and run types by state machines or Petri nets generating them. Since our models abstract from many details of the concrete binary component execution, we refer to *usage profiles* instead of operational profiles of the software.

### 2.1. Basic component service reliability model

From an interface perspective, service executions define the external behaviour of a component. At an abstract level, we regard a service execution as a *transition* of a protocol state machine representing requests to the component. The states of that protocol are control points constraining the possible orders of these requests. Ultimately, at the concrete implementation level, these services are realised by method executions. Fig. 1 shows separate substeps of these executions focusing on the transition across boundaries of components by external service calls. The elementary timeless transitions (vertical bars) in the figure are events characterizing the beginnings and ends of relevant states such as “*Method Execution*”. States are subject to different potential failures. For example, the reliability of Method Execution depends on the binary code of the method, of libraries, the operating system it runs on, the underlying hardware and so forth. In contrast, *Call of external Methods* typically relies on separate code and its underlying systems.

Since a failure-free execution must run through all these states, we can model its reliability as a product of separate reliability factors. Which factors should be considered? Although the above analysis identifies a

number of factors influencing the reliability of a service call, it is impractical or impossible to measure them all. We simplify the model by using the following observation. There are two kinds of factors: (a) constant factors, such as reliability of the method body code, reliability of call and return and (b) variable factors, such as the reliability of external method calls. Section 6 discusses how we determine constant factors. We now model the reliability of a method call with three figures:

- The method and connection specific reliability  $r_{cr}$  of a method call and method return: Intuitively  $r_{cr}$  is the product of the reliability of the correct call of the method and the correct return of the results. Since the reliabilities of call and return are dominated by connections and networks, it is useful to capture this in a single figure.
- The reliability of the method body  $r_{body}$  excluding the reliability of called methods: This factor reflects the quality and the length of the method’s code and the kind of operations it performs.
- The reliability of external method calls ( $r_e$ ): Since in general a method may call several external services and may have several branches and loops, obtaining a single number for the influence of the reliability of external services requires a profile for all possible execution traces through the code of the methods.

Putting them all together we obtain the service reliability

$$r_m := r_{cr} \cdot r_{body} \cdot r_e \quad (1)$$

### 3. Architectures and contractual interfaces

In this section we briefly describe rich architectural description language (RADL) and the contractual use of software components within this ADL. A more detailed treatment of RADL can be found in (Schmidt, 1998; Schmidt et al., 2001; Schmidt and Reussner, 2002). We concentrate on issues relevant for the reliability-prediction model presented afterwards. A more detailed discussion of parameterised contracts is given in Reussner (2001c) and Reussner (2001b). To ensure a tight relationship between the architectural model and the actual implementation of a system, RADL extends DARWIN (Magee et al., 1995) by adding constructs existing in industrial middleware platforms (such as for instance, server/containers, context-based interception, attribute-oriented configuration). RADL also uses rich interface definitions as advocated in (Krämer, 1998; Han, 2000; DeAlfaro and Henzinger, 2001; Reussner, 2001a) with the aim of capturing information useful for component assembly.

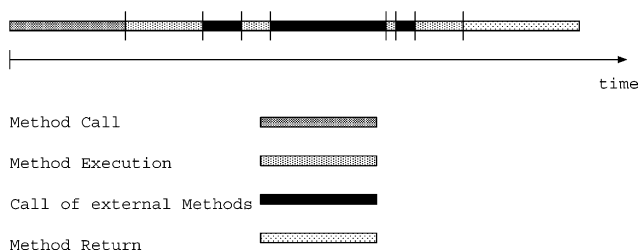


Fig. 1. Different states of a method call.

### 3.1. Gates, kens, bindings and mappings

We term our architectural entities *kens*; they are protection domains and provide views of policy rules and constraints. Kens are protected by *gates* controlling access and migration in and out of kens. The simplest kens, so-called *basic kens* are black-box components. The simplest gates form connections. Composite kens represent assemblies of kens, i.e., of components or recursively of assemblies. RADL supports different kinds of composition that are beyond the scope of this paper. An example configuration is shown in Fig. 2. which shows the basic ken `OnlineAccMgr` and the composite `BankSystemken`.

Gate specifications include *service signatures* describing how to call a component service (i.e. name, parameter order and types, return type and possibly thrown exceptions), *gate protocol FSMs* describing legitimate call sequences through gates, and *extra-functional service attributes* such as quality of service characteristics, reliability, timing, synchronisation, fault-tolerance, security, etc. Similar to DARWIN we distinguish *provided and required gates*, which allows detailed dependency modelling. A pair of compatible gates defines a legitimate, protected connection. Connections between a required gate and a provided gate are called *bindings*. Connections between two provided gates (or two required gates) are called *mappings*. The example is discussed in more detail below.

*Contractually used components.* Contracted suppliers (which can be methods, objects, components) have pre- and postconditions associated to them. A precondition states what the supplier expects from its environment (called client of the supplier). For example, if the unit is a method, the preconditions may state assumptions about the method's parameters and its postcondition guarantees to the caller about the returned value or the state after return. The principal of design-by-contract of B. Meyer (1992) states, that the supplier guarantees the postcondition, if the client fulfills the precondition. This guarantee is usually conditional upon an invariant for the process, object or component modelled.

The abstract principle of parameterised contract is an generalisation of the principle of design-by-contract (Reussner 2001b,c). Instead of associating fixed invariants, pre- and postconditions to the supplier, in a parameterised contract the postcondition is parameterised by the precondition and vice versa. More over invariants are conditional. Their premise includes conditions on the environment such as interference constraints, quality of service requirements etc.

At the level of components, preconditions, i.e., assumptions about the environment, must be satisfied both at provided and required interfaces. This makes component contracts significantly different from object contracts, because the requirements include method

postconditions in required gates, usage profiles for provided gates and other extra-functional aspects. By similar arguments the component postconditions (i.e. guarantees to the environment) include service preconditions in required gates (which have to be guaranteed by the component in outgoing calls.).

Technically, parameterised contracts are based on mappings between provided and required gates and hence depend on the concrete interface model. A simplistic, signature-list based interface model would map each provided services to the set of external services  $ES(s)$  it requires. This represents a minimalist approach to dependencies. For example it is now possible to approximate the compound reliability by some product of those services in  $ES(s)$ . However, this does not reflect call frequencies, which make the reliability of a provided service dependent on the number of runs of the required components. As a result their reliability factors more than that of lesser used components and sometimes more than that of the component itself, because such runs can be unboundedly long call sequences resulting from loops. In this case the provided service liability is the limit of a series of increasingly long reliability products. The simplistic product of required services is hopelessly inaccurate in most cases. Our measurement confirms this analytical argument.

Because we already used FSMs for gate protocols for interoperability checking and component adaptation, we also utilise them to represent the effect of detailed design decisions on the reliability of provided services. Depending on this design different provided service calls give rise to different uses of required services. We call this a *service-effect FSM* (short service FSM). Note that service FSMs concentrate on this dependency between provision and requirements, not on implementation details. We view a service FSM as a minimalist abstraction of such reliability dependencies in architectures. We can now compose gate protocol FSMs and service FSMs to derive abstract behavioural models of kens parameterised by usage profiles and properties of required behaviour. When kens are composed, more concrete context-dependent use-reliability dependencies can be composed as well.

### 3.2. Modelling usage profiles with Markov chain models

Usage profiles are commonly modelled as Markov chains (Cheung, 1980; Whittaker and Thomason, 1994). Intuitively, a Markov chain is an FSM extended with transition-probability distributions. The transition function  $t$  is represented by the matrix  $P = ((p_{ij}))_{ij}$ , where  $p_{ij}$  is the transition probability from state  $i$  to state  $j$ . The defining property of Markov chains is the independence of the transition probabilities from previous transitions.

In RADL, Markov chains form the connection between usage profiles and external component reliabilities. Let us first consider an example. Fig. 3 depicts the usage of an online account. Every user logs in first. Then, 20% of the users have to retry one more time, 1% immediately log out, and the remaining 79% proceed to listing their accounts. Analogously one can interpret the data given for the other states. Thus we model a usage profile by probabilities  $u_{i,m}$  for calls to a provided service (method)  $m$  in state  $i$ . Now for two states  $i$  and  $j$  there may be several transitions with different methods  $m_k$  ( $0 \leq k \leq l$ ) of reliability  $r_{m_k}$ . Thus the probability of a failure-free transition by means of  $m_k$  is  $u_{i,m_k} r_{m_k}$  and the probability,  $s_{ij}$  of reaching  $j$  from  $i$  via any of the methods  $m_k$  without failure is the sum

$$s_{ij} = \sum_k u_{i,m_k} r_{m_k} \quad (2)$$

Because the usage profile of the  $m_k$  is a probability distribution ( $\sum_k u_{i,m_k} = 1$ ) and the method reliabilities  $r_{m_k} < 1$  we have that,  $s_{ij} < 1$ . The theory of Markov chains now lends itself to computing the probability of reaching  $j$  from  $i$  via any finite arbitrarily long sequence. In the case where there are loops, there are infinitely many such sequences. To this end, one computes the geometric series  $\sum_{k=0}^{\infty} S^k$  with  $S = ((s_{ij}))_{ij}$ . This series converges to  $(I - S)^{-1}$ , where  $I$  is the identity matrix. This matrix contains the probability of reaching the final state  $\hat{j}$  from the initial state  $\hat{i}$  as

$$r_{\text{system}} = (I - S)^{-1}(\hat{i}, \hat{j}) \quad (3)$$

For a protocol FSM with multiple final states we introduce an artificial super *final state*  $SFS$  reachable from any final state with reliability 1. More details are provided in Cheung's work (Cheung, 1980) which differs in associating reliability with states and usage with transitions. To apply Cheung's model directly we would have had to model all relevant intermediate states of method execution explicitly (see Fig. 1). Our method can be applied directly to protocol FSMs and scales better.

#### 4. Predicting reliability of software architectures

Kens have provided and required gates. Compound kens additionally have inner kens and their connection structure. To model connection reliability we use two constants  $r_m$  and  $r_b$  discriminating the independent reliabilities of mapping (local) and binding (typically via network connections).

##### 4.1. Reliability of basic kens

When we specify the reliability of a basic ken or one of its provided services,  $s$  as a function of reliabilities and usage, we do not want to limit the ken to a fixed

usage profile or a fixed environment. Rather we wish to compute its reliability in terms of the reliability of the external services used by,  $s$ . Therefore we formalise a basic ken as a triple consisting of provided gates (including protocol FSM), required gates (including protocol FSM) and service FSM usage profile for each provided service. The latter extends the service FSM to a Markov chain. The probabilities of the Markov chain model the essentials of internal resources and algorithmic decisions in the black box ken without revealing the details that determine those probabilities of call sequences to required gates.

*Parameterized contracts computing the reliability of basic kens.* To link the environment reliability of required gates to the reliability of services in provided gates, the parameterised contracts use the service FSMs and usage profiles of provided services. The example in Fig. 4 illustrates a service FSM usage profile of `listAccounts` in the `OnlineAccMgr` ken. Given the reliability of the external services called by `listAccounts`, we can compute the reliability of `listAccounts` by Eq. (3). In general, the service FSM usage profile  $u(i, m)$  (see above) and the reliabilities of required interfaces are combined to compute the missing reliabilities for the external services. As an example, let us assume the reliability value 0.9999 for the transition (methods) `getCustomerID`, `readTransaction`, `hasCreditCard`, and `getAvailableCredit` and assume reliability 0.9995 for other transitions of the `OnlineAccMgr` internal service FSM usage profile. According to the above definition of  $S$  we obtain:

$$S = \begin{pmatrix} 0 & 0.9999 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.949525 & 0 & 0 & 0.049995 & 0 \\ 0 & 0 & 0 & 0.9995 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9999 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.79992 & 0.2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (4)$$

By convention, the last row denotes the super final state, which can be reached from any final state with reliability 1 (Wang et al., 1999). The result according to Eq. (3) is a reliability of service `listAccount` of approximately  $(I - S)^{-1}(1, 7) = 0.99877$ . This corresponds to an MTTF of 816 calls.

*Computing the reliability of kens with usage profiles.* Given a ken with its provided-gate protocol FSM augmented by a usage profile and the reliability of its provided services (as computed by Eq. (2)) we can form the matrix  $S$ . Eq. (3) then delivers the overall reliability.

##### 4.2. Reliability of composite kens

Similar to the prediction of basic ken reliability we first determine the service reliability and then the overall reliability. This calculation differs only in the first step.

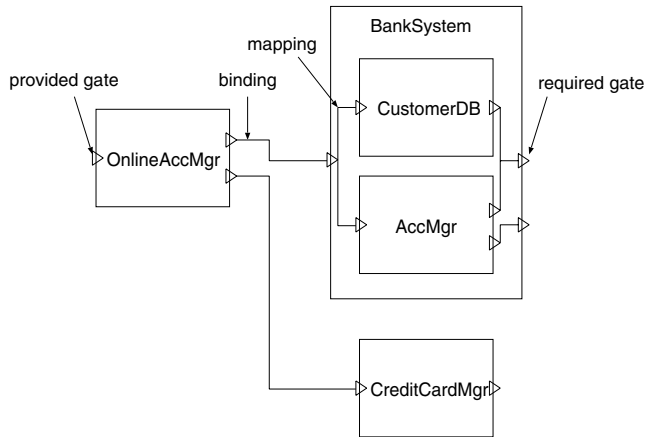


Fig. 2. Example: kens, gates, bindings and mappings.

Given a composite ken, the reliability  $r_s$  of one of its services,  $s$  is the product of two factors

$$r_s := r_m \cdot r_{\vartheta_{\text{inner}}} \quad (5)$$

where  $r_m$  is the reliability of the mapping (see Fig. 2) and,  $s$  is mapped to the service  $s_{\text{inner}}$  of the inner ken. We consider two cases.

1.  $K_{\text{inner}}$  is a basic ken: for computing the reliability of  $\vartheta_{\text{inner}}$  we use the service effect FSM of  $K_{\text{inner}}$  and proceed as described in Section 4.1 using Eqs. (2) and (3).
2.  $K_{\text{inner}}$  is a composite ken: we apply the computation of  $s_{\text{inner}}$  recursively using Eq. (5). (Note that the mapping reliabilities of the inner ken and the outer ken may differ.)

The recursion terminates since we assume that hierarchical kens also define distribution boundaries and hence form trees.

#### 4.3. Assumptions

Our reliability model makes a number of limiting assumptions that may restrict its applicability and require further enhancements: (a) the availability of data, and (b) assumptions concerning the mathematical model.

Table 1 summarises the data required for the respective architectural kens and which party ought to be providing the relevant model data. Our methods require the same data from a ken buyer (or user) regardless of whether the ken is basic or composite. This reflects a level of black-box character of the ken from the user standpoint even for composite kens.

In our model, we assume that the calling of services can be modelled with a Markov process. In other words, the path through the binary deployed control is unimportant. When a service is called, its reliability does not depend on methods previously called. This means that

Table 1  
Responsibility for minimal reliability properties

Party: Ken	Vendor: responsible for binary reliability	Buyer: responsible for context
Basic: black-b, component	Service effect FSM usage profile	required services reliability
Composite: gray-b assembly	Service effect FSM of inner basic kens; binding and mapping reliabilities	Required services reliability

failures of services are independent. This assumption is now widely accepted for well-designed highly dependable software (especially in combination with statistical testing) (Dyer, 1992; Whittaker and Thomason, 1994; Trammell, 1995). Error propagation can still invalidate this assumption. In our view, this is a motivation to use contract-centered CBSE and software architecture which separates independent failure factors and in practice promises considerable protection from this problem. The assumption of independent service reliability also requires the assumption of independent component reliability. While Cheung justifies this assumption through empirical investigations for modules (Cheung, 1980), counterexamples are also known: For increasing the reliability of airborne communications, redundant computers are installed under the assumption of their failure independence. In a recent aircraft failure, the triple redundant computer systems failed successively due to water leakage in the area where all three systems were installed in close proximity.

#### 5. Measurement and empirical evaluation

In our empirical validation of the reliability model of basic kens we insert a basic ken (associated with its service effect protocols) into a test-bed. The test-bed provides all methods required by the ken. There are two kinds of independent variables we can control:

- The reliability of the methods provided by the test-bed. They were programmed to fail with a given, adjustable probability.
- The usage profile of the service FSMs. At each branch we decide with a given, adjustable probability, which branch to take.

The dependent (observed) variables are the reliabilities of the services provided by the ken in the test-bed. We conduct the following experiments:

1. *Validation of prediction against measurements:* Independent variables: reliabilities of required methods. Fixed variables: usage model of the service effect FSMs. Measured variables: reliability of provided methods.

2. *Test the influence of precision of service effect FSMs usage model:* Independent variables: precision of the usage profile of the service effect FSMs. Fixed variables: reliabilities of required methods. Measured variables: reliability of provided methods.

In this experiment, we check measured against predicted reliability of provided methods and study the influence of the precision of the usage profile on reliability.

The prediction is computed in each experiment for each method as it was done for the service `listAccounts` in Section 4.

5.1. *Testbed for validation*

The test-bed used for the empirical experiments was designed to offer maximum control over reliability and usage profiles. The main idea is to test a component (here the basic ken `OnlineAccMgr`) in an environment which has only services with a predefined reliability and to implement services of `OnlineAccMgr` as “dummy” methods, containing only calls to external methods and statements directing the control-flow according to a predefined usage profile.

The system is implemented in Java. To realise external (environmental) services with a predefined reliability we implemented all required services of `OnlineAccMgr` as methods throwing an exception with a predefined, adjustable probability (i.e., the reliability of the service). We deliberately omitted all other functionality to avoid other causes of exceptions, which would disturb the measurements. For example, the code of service `getSavingsAccountofAccMgr` is simply:

```
public void getSavingsAccounts()
    throws Exception
{
```

```
    if(Math.random() > relgetSavingsAccounts) }
        throw new Exception ();
    }
}
```

The test driver calls each service a predefined number of times  $N$ . (For the presented measurements  $N = 100000$ ) and counts the number of exceptions thrown  $N_e$ . From this, the MTTF ( $N/N_e$ ) can be computed for the duration (time unit) of a single service call. The reliability is  $1 - N_e/N$ .

5.2. *Sample system*

We measured the reliability of the following `OnlineAccMgr-ken` in our test-bed. The `OnlineAccMgr-ken` is a ken residing on internet-service host of a bank, providing online banking facilities. The model of the ken is motivated by the functionality provided by the online facility of one of the major Australian banks. In Fig. 3 we show the provided protocol of this ken. Also shown is an (estimated) usage profile for the ken, denoted by the probabilities given in squared brackets for each state transition. After the user logs in to the system (which includes the possibility of two failed trials), the system lists all accounts of the user with their balances. For a selected account, a pre-defined number of most recent transactions are listed. The user can scroll for further transactions and look at details of transactions, such as their receiver, ID, etc. At any time, the user can logout. When leaving the transaction-details view, the user can then scroll through the transactions of the selected account. The user also can select other accounts and proceed to inspecting their transactions without logging out.

The service effect protocol for `listAccounts` (Fig. 4) reads the bank-internal customer-ID of the user logged in. In the case where it is not provided (e.g., due to discrepancies of the online-banking department and

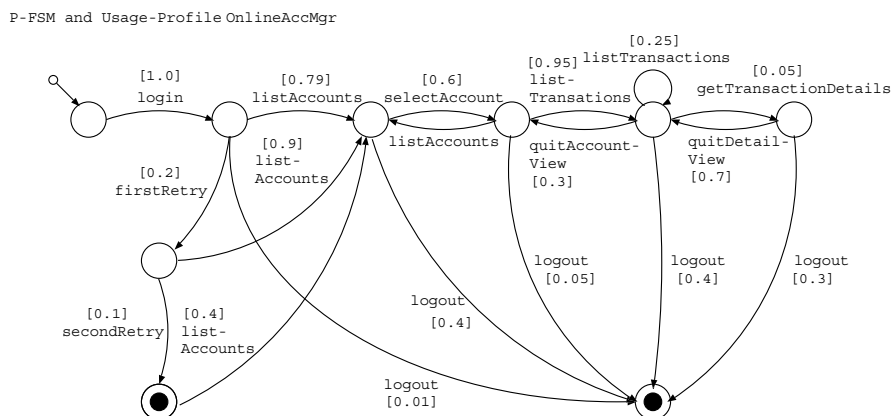


Fig. 3. Transition diagram of the P-FSM of the `onlineAccMgr` and annotated usage profile.

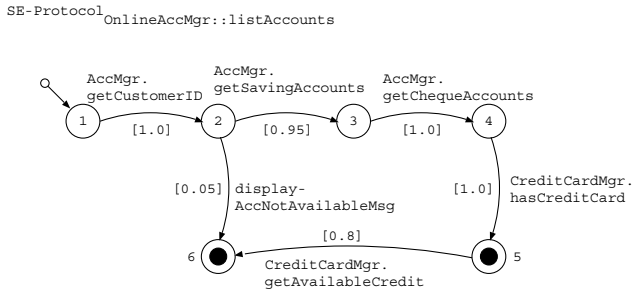


Fig. 4. Service effect protocol of the service `listAccounts` and annotated usage model.

the general customer database of the bank), an error message is issued. Otherwise, the system retrieves the savings or cheque account. If a credit card account (provided by a different branch of the bank or a third-party) is available, the current credit limit is also shown. Similarly, other services such as `listTransactions` are modelled with their usage profiles.

The different deployment contexts are now represented by different reliability vectors. Each maps required services to their context-dependent reliabilities.

### 5.3. Comparison of predicted and measured reliability

The deviation of the prediction from the measured values is small (below 1%) as shown in Table 2. The predicted reliability is computed as described in Section 4.1. For measurements and prediction we used the values of one reliability context (vector). The lower accuracy of service `listTransactions` is caused by its loop, because the measurements are all taken over finite sequences, while the prediction is based on the geometric sum in Eq. (3) and considers infinite sequences (as one cannot give an upper bound of loop executions). However, since the measured loop executions are finite, albeit large, one would expect inaccuracies in measuring true reliability. Measurement shows that the high accuracy of prediction is also maintained if the reliability of the external methods is low (e.g., 0.85) and the reliability prediction decreases accordingly. Note that in our example, moderate decreases in external reliability bring down the overall reliability of an `OnlineAccMgr`'s services to a practically unacceptable level (see Fig. 5).

Table 2  
Predicted versus, measured reliability for services of `OnlineAccMgr`

Service	Pred. rel.	Meas. rel.	Difference	Error (%)
<code>listAccounts</code>	0.99877	0.99877	–	–
<code>listTransactions</code>	0.98705	0.99397	0.00692	0.70
<code>getTransactionDetails</code>	0.99800	0.99813	0.00013	0.01

The  $x$ -axis shows the scaling factor for external reliabilities. A constant factor is used to fix the relationship between the reliability of different required components while scaling them all down. The reliability of service `listTransactions` drops much faster than that of any other service, due to its internal loop. This suggests strongly that (a) *service effects matter for reliability* and one should push vendors to include adequate information about them, and (b) *the external reliability values are crucial*. Because of non-linear dependencies resulting from loops, small differences in the external reliability (e.g., 0.02) give rise to considerable variation (50%). This is a strong argument for deploying parameterised contracts to compute context-dependent reliability of components.

Fig. 6 shows that *the effect of imprecise service effect usage profiles is insignificant compared to exact reliabilities for required components*. We compute the difference between predicted and measured reliability as a function of varying usage profile. The measured reliability remains fixed (Table 2). This is interesting when considering the strong influence of the environment reliability on the services reliability as shown in Fig. 5. The weak influence of the precision of the usage profile on service reliability is due to similar call-sequence structure. The influence is more significant in examples with more complex call-sequence variation. These findings suggest that a rough and ready estimation or even some educated guesses combined with calibration by automated methods could well be sufficient for service FSM usage profiles. *In practice, the structure of the service FSM is the key: the internal resource usage probability is less important*. While we need larger empirical studies, these results are very promising for trusted and predictable assembly, because vendors can generate and deliver the FSM tables with their binary component without revealing internals.

## 6. Gathering usage profiles and reliability data

Our methods require empirical data such as reliabilities of external services, bindings, mappings and usage profiles. We suggest three methods to acquire these data systematically: modelling, monitoring and estimates. For modelling, usage profiles are explicitly specified and collected piecemeal as part of the stepwise development process: for example, as part of test design and execution. Provided gate FSM usage profiles are particularly beneficial for overall architecture reliability prediction and other verification, testing and tuning tasks (Dyer, 1992). The crucial reliability data for required services can also be obtained by monitoring and gathering failure statistics. Often, monitoring can be achieved with little overhead. In the absence of hard data, experts may be able to give reasonable estimates about reliability of systems they use frequently.

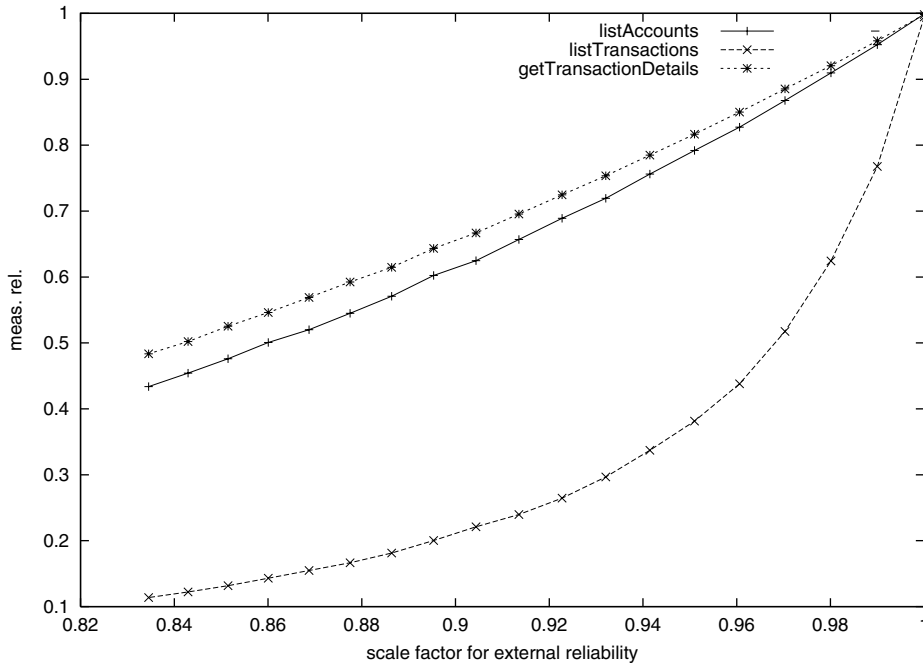


Fig. 5. Comparing the reliability of services depending of the external services' reliabilities.

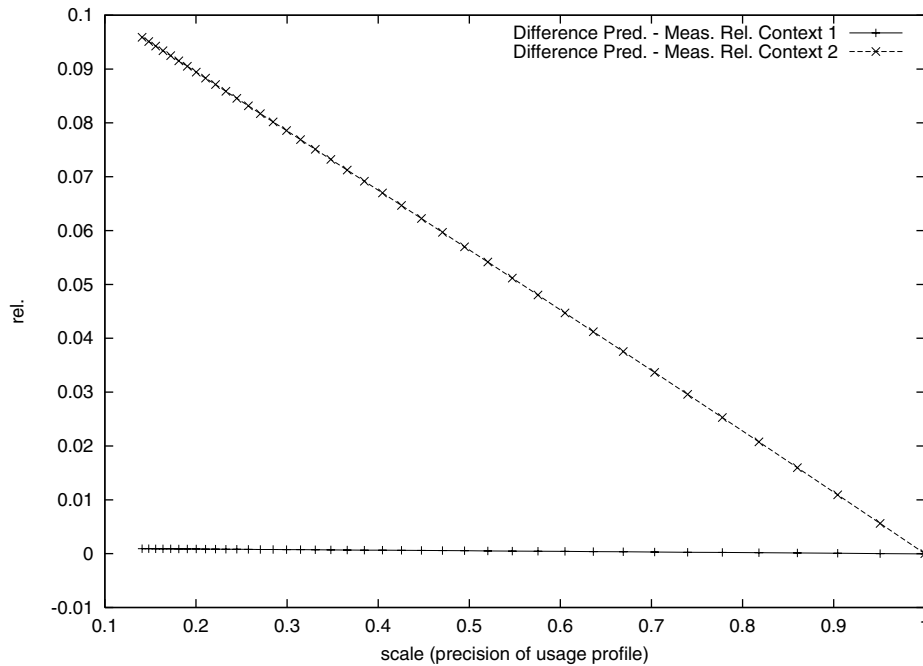


Fig. 6. Differences between predicted and measured reliability of service listAccounts varied over the precision of the usage profile of the service effect FSMs for both contexts of external reliability.

In practice, a combination of these approaches is usually appropriate. Educated guesses can be combined with statistically derived usage profiles and systematically-acquired test information. Our experiments show, even if the failure rate of the different methods varies between 10% and 0.0001%, the precision of the usage profile does not significantly affect the accuracy of the predictions.

### 7. Related work

Reliability theory for hardware components does not apply directly to software components. While software reliability theory is well-developed, it has evolved for large closed systems. To date there is a absence of reliability methods for component-based software engineering, especially for open distributed software.

Existing methods, like Cleanroom (Mills et al., 1987; Dyer, 1992), advocate statistical testing, where the reliability of software is determined before the system is shipped. However, this approach is neither compositional nor takes architectural specifications into account.

The deployment of usage profiles was also strongly advocated in the cleanroom approach by Mills et al. (Mills et al., 1987). (Hamlet et al., 2001) advocates a statistical view of component-oriented architectures. He clearly identifies the need for usage profiles but focuses mainly on functions and requires source code for his computations. Our work builds on Hamlet's but our methods work with binary black-box components and architectural assembly. Moreover they recognise the strong influence of required components reliability.

The work closest to ours is perhaps that of (Cheung, 1980; Wang et al., 1999), who also focus on architectural structures. However, firstly, their architectural compositions are derived from relatively low-level control compositions for functional components, such as pipeline, sequence, alternative and parallel compositions, compromising the black-box principle. Furthermore, their prediction model is mostly concerned with the flow of control across module boundaries in sequential software systems and the failure of components in chains of control. In contrast, we discriminate abstractly the different kinds of component failures and gain considerably more accuracy of prediction as a result. Most importantly our model is applicable to open distributed systems. Moreover, following Cheung's ideas, their method actually uses determinants (complexity  $O(n!)$ ) while we use inversion (complexity  $O(n^3)$ ) in our implementation. In the meantime efficient algorithms to solve special sparse matrix-inversion problems have been developed.

A syntax for defining and specifying quality of service attributes is given by Frolund and Koistinen in (Frolund and Koistinen, 1998). Similar to our work, they emphasise the contractual use of quality of service attributes. In contrast to our approach their reliability is specified as constant while we use parameterised contracts (Reussner, 1999) for computing context-dependent component reliability.

While FSMs are easy to understand and have been widely used in protocol specification and testing, their main attractiveness for our work derives from algorithms for inclusion checking and other computations. Since FSMs are quite limited in expressiveness, the question arises of whether they are suitable for the modelling and analysis of service reliability dependencies. While this article also verifies this question empirically, it should be noted that various more powerful extensions of FSMs exist. Some inherit the efficiency from normal FSMs (Freudig, 1998; Reussner, 2001b).

Since every analysis method depends strongly on the availability of the required data, we undertook some

work to generate service-effect FSMs from control-flow analysis data (Hunzelmann, 2001). Alternatively, tool support also exists for deriving FSMs from message sequence charts (Wydaeghe, in press).

## 8. Conclusions

We presented a novel approach to predicting the reliability of component-based software architectures. Our model is parameterised by usage profiles and the reliability of required components.

Our approach models dependencies between service reliability as state machines connecting provided gate protocols and required gate protocols. Partially statistical usage profiles are required for our prediction. Using concepts of the RADL architecture definition, we embedded reliability reasoning into software architecture composition and showed how parameterised contracts overcome the openness problem of component-based architecture. Some reliability calculations are performed when components or entire assemblies – so-called kens – are deployed into operational contexts. The methods in this paper are strongly compositional. That is, our parameterised reliabilities are computed following the compositional structure of architectures and ignore component internals.

The article develops a combination of analytical and empirical methods. Our empirical validation shows that our prediction is extremely accurate. The article also conducts some sensitivity experiments. The outcomes strongly suggest that service and hence component reliability is dominated by the reliability of required components, that the dependency between provided and required gates must be modelled for accurate prediction, and finally, that internal resource usage profiles are less relevant for the reliability of a wide class of components.

Our methods have a precise semantics in automata theory and the theory of Markov chains. We have shown the benefit of formal specification and modeling of component-environment dependencies and their need for accurate reliability prediction. Our measurements strongly suggest the delivery of component reliability specifications with deployed components. We believe our methods are practical because the results can be computed reasonably efficiently and, more importantly, relevant statistical data can be obtained analytically from monitoring and measuring binary deployed components.

The paper evaluates the methods empirically using a running online banking example in our reliability test-bed. The results confirm a high accuracy of our reliability prediction based on service effects state machines and Markov chains.

Limitations of the work were identified and related and future work was sketched.

## Acknowledgement

Partial support of the finalisation of this work by IRISA/INRIA as part of a sabbatical visit is gratefully acknowledged by the second author.

## References

- Cheung, R.C., 1980. A user-oriented software reliability model. *IEEE Transactions on Software Engineering* 6 (2), 118–125, Special collection from COMPSAC '78.
- DeAlfaro, L., Henzinger, T.A., 2001. Interface automata. In: Gruhn, V. (Ed.), *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, vol. 26, vol. 5 of *SOFTWARE ENGINEERING NOTES*, New York, ACM Press, pp. 109–120.
- de Roeber, W.-R., Langmaack, H., Pnueli, A., 1998. Compositionality: the significant difference. In: *International Symposium, COMPOS'97*, Bad Malente, Springer-Verlag, Germany.
- Dyer, M., 1992. The cleanroom approach to quality software development. In: *Series in Software Engineering Practice*. Wiley & Sons, New York, NY, USA.
- Freudig, J., 1998. Konformitätsprüfung jenseits von Typanalyse. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany.
- Frolund, S., Koistinen, J., 1998. Quality-of-service specification in distributed object systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory.
- Hamlet, D., Mason, D., Voit, D., 2001. Theory of software reliability based on components. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE-01)*, IEEE Computer Society, Los Alamitos, California, pp. 361–370.
- Han, J., 2000. Temporal logic based specification of component interaction protocols. In: *Proceedings of the 2nd Workshop of Object Interoperability at ECOOP 2000*, Cannes, France.
- Hunzelmann, G., 2001. Generierung von Protokollinformation für Softwarekomponentenschnittstellen aus annotiertem Java-Code. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany.
- John D. Musa, A.I., Okumoto, K., 1987. *Software Reliability—Measurement, Prediction, Application*. McGraw-Hill, New York.
- Krämer, B., 1998. Synchronization constraints in object interfaces. In: Krämer, B., Papazoglou, M.P., Schmidt, H.W. (Eds.), *Information Systems Interoperability*. Research Studies Press, Taunton, England, pp. 111–141.
- Laprie, J.-C., Kanoun, K., 1996. Software reliability and system reliability. In: Lyu, M.R. (Ed.), *Handbook of Software Engineering Reliability*. McGraw-Hill, New York, pp. 27–69.
- Magee, J., Dulay, N., Eisenbach, S., Krämer, J., 1995. Specifying distributed software architectures. In: *Proceedings of ESEC '95—5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, Sitges, Spain. Springer-Verlag, Berlin, Germany, pp. 137–153.
- Meyer, B., 1992. Applying design by contract. *IEEE Computer* 25 (10), 40–51.
- Mills, H.D., Dyer, M., Linger, R., 1987. Cleanroom software engineering. *IEEE Software* 4 (5), 19–25.
- Reussner, R.H., 1999. Dynamic types for software components. In: *Companion of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, extended abstract.
- Reussner, R.H., 2001a. Enhanced component interfaces to support dynamic adaptation and extension. In: *34th Hawaii International Conference on System Sciences*, IEEE.
- Reussner, R.H., 2001b. Parametrisierte Verträge zur Protokolladaptation bei Software-Komponenten. Logos Verlag, Berlin.
- Reussner, R.H., 2001c. The use of parameterised contracts for architecting systems with software components. In: Week, W., Bosch, J., Szyperski, C., (Eds.), *Proceedings of the Sixth international Workshop on Component-Oriented Programming (WCOP'01)*.
- Schmidt, H.W., 1998. Compatibility of interoperable objects. In: Krämer, B., Papazoglou, M.P., Schmidt, H.W., (Eds.), *Information Systems interoperability*, Research Studies Press, Taunton, England, pp. 143–181.
- Schmidt, H.W., Poernomo, I., Reussner, R.H., 2001. Trust-by-contract: modelling, analysing and predicting behaviour in software architectures. *Journal of Integrated Design and Process Science* 5 (3), 25–51.
- Schmidt, H.W., Reussner, R.H., 2002. Generating adapters for concurrent component protocol synchronisation. In: *Accepted for the Proceedings of the Fifth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*.
- Shaw, M., Garlan, D., 1996. *Software Architecture*. Prentice Hall, Englewood Cliffs, NJ, USA.
- Trammell, C., 1995. Quantifying the reliability of software: statistical testing based on a usage model. In: *Proceedings of the Second IEEE international Symposium on Software Engineering Standards*, p. 208.
- Wang, W.-L., Wu, Y., Chen, M.-H., 1999. An architecture-based software reliability model. In: *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, 16–17 December, Hong Kong, China, IEEE.
- Whittaker, J.A., Thomason, M.G., 1994. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering* 20 (10), 812–824.
- Wydaeghe, B. Component composition based on composition patterns and usage scenarios. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, Belgium.

**Ralf Reussner** received his diploma in computer science in 1997 from the Universität Karlsruhe (T.H.), Germany. He was member of the software engineering group at the University of Kaiserslautern 1997–1998. From 1998 to 2001 he received a research fellowship from the National German Research Council (DFG) and received his PhD in computer science in 2001 from the Universität Karlsruhe (T.H.). In his PhD thesis he introduced the abstract concept of parameterised contracts and worked on extensions of finite state machines with decidable inclusion problem. Since then he has been a Senior Research Scientist at the Distributed Systems Technology Centre Pty Ltd. (DSTC), Melbourne, Australia, where he works on mechanisms for predicting properties of component-based software architectures. He published a book and about thirty refereed articles on software engineering for distributed and parallel applications including work on benchmarking parallel applications. In this context he maintains the open-source project SKaMPI. Ralf consults in the area of distributed application developments and he held several invited talks about his research in academia and industry. Besides serving on several program committees, he organised workshops on ECOOP and TOOLS conferences. He is member of the ACM, IEEE, and the GI e.V.

**Heinz W. Schmidt** is Professor of Software Engineering at Monash University, Melbourne, Australia. He has over twenty years experience with object-oriented or component-oriented systems, methods and languages in research, education and practice. Heinz has authored or co-authored over fifty refereed articles, supervised over twenty-five higher-degree research (Masters and PhD) students, and lectures in software engineering, programming languages and distributed systems. Before joining Monash he conducted research in these areas at the German National Research Centre for Information Technology (GMD), the International Computer Science Institute at the University California Berkeley, USA, the CSIRO Canberra, Australia, and Australian National University, Canberra, Australia. Heinz received his PhD (Dr-Ing.) degree from Bremen University, Germany, and completed his MSc (Dipl.-Inform.) degree at Bonn University, Germany. He is a member of the ACM, IEEE and VDI. He chaired and co-chaired object-oriented and software engineering conferences and workshops, and served on several program committees.

**Iman Poernomo** completed a Bachelor of Arts (Philosophy major) in 1997 and an Bachelor of Science (Honours in Pure Mathematics) in 1998 from Monash University, Australia. He went on to a PhD at the school of Computer Science and Software Engineering (SCSSE), Monash University. He was a research assistant for the proofs-as-programs group at SCSSE between 1999–2001. His PhD research was concerned with automated software engineering using algebraic specifications and logics of programming languages. In 2002, he com-

menced work as a Senior Research Scientist at the Distributed Systems Technology Centre Pty Ltd. (DSTC), Melbourne, Australia, where he works on specification and analysis of component-based software architectures. In this position, he has consulted in the areas of middleware for the enterprise and enterprise systems modeling. He is the author of 11 refereed articles in the fields of algebraic specification, proof-theory and component-based software engineering.