Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems

Genaína Rodrigues[1], David Rosenblum[1], and Sebastian Uchitel[2]

[1]Department of Computer Science,
London Software Systems,
University College London,
Gower Street, WC1E 6BT, UK
[2]Department of Computing,
London Software Systems,
Imperial College London,
180 Queen's Gate, SW7 2RH, U.K

Abstract. Scenarios are a popular means for capturing behavioural requirements of software systems early in the lifecycle. Scenarios show how components interact to provide system level functionality. If component reliability information is available, scenarios can be used to perform early system reliability assessment. In this paper we present a novel automated approach for predicting software system reliability. The approach involves extending a scenario specification to model (1) the probability of component *failure*, and (2) *scenario transition probabilities* derived from an operational profile of the system. From the extended scenario specification, probabilistic behaviour models are synthesized for each component and are then composed in parallel into a model for the system. Finally, a user-oriented reliability model described by Cheung is used to compute a reliability prediction from the system behaviour model. The contribution of this paper is a reliability prediction technique that takes into account the component structure exhibited in the scenarios and the concurrent nature of component-based systems. We also show how implied scenarios induced by the component structure and system behaviour described in the scenarios can be used to evolve the reliability prediction.

1 Introduction

Software reliability engineering is an important aspect of many system development efforts, and consequently there has been a great deal of research in this area [15, 10]. One important activity included in software reliability engineering is *reliability prediction* [11]. There has been much recent work in reliability engineering that has addressed reliability modeling and prediction of architecture- and component-based software [8, 19]. Components both simplify and complicate reliability prediction. They simplify because accurate component reliability estimates may be available to aid reliability prediction early in the development lifecycle. They complicate due to the need for a sound compositional approach

M. Cerioli (Ed.): FASE 2005, LNCS 3442, pp. 111–126, 2005.

to reliability prediction. A promising compositional approach to predicting reliability of component-based systems early in the lifecycle is to base the prediction on scenarios of system usage.

Scenarios have been widely adopted as a way to capture system behavioral requirements. *Message Sequence Charts* (MSCs) [9] and their UML counterpart, Sequence Diagrams (SDs) [16] are widely accepted notations for scenario-based specification.

There has been some previous work on using scenarios to predict the reliability of component-based software [4, 27], but they use imprecise, coarse-grained, sequential models of system architecture as the basis for prediction. In this paper, we present a novel scenario-based approach to reliability prediction in which a more precise, fine-grained, concurrent system architecture model is synthesised for computing a reliability prediction. The approach starts with a set of scenarios and a high-level message sequence chart (HMSC). The HMSC is annotated with *scenario transition probabilities* derived from an operational profile of the system [14], which accounts for the relative frequency with which system usage results in a transition from one scenario to another. We synthesise from the scenarios a deterministic probabilitistic behaviour model for each system component. Each component model is then extended to model the probability of component *failure*. The resulting probabilistic models are composed in parallel and used to predict the reliability of the component-based system according to Cheung's user-oriented reliability model [3].

The contribution of this paper is a reliability prediction technique that takes into account the component structure exhibited in the scenarios and the concurrent nature of component-based systems. We also show how as a result of this implied scenarios can impact the result of reliability analysis.

The paper is structured as follows: In Section 2, we briefly present some background information about the different elements of our approach. In Section 3 we describe in detail our scenario-based method for predicting software system reliability and an extensive illustration of our approach. In Section 4 we show how implied scenarios detection can be used to improve reliability prediction for concurrent software systems. In Section 5 we compare our approach to other efforts for analysing reliability of component-based software and discuss the main differences between our approach and other scenario-based reliability analysis models. Finally, in Section 6 we present our conclusions and discuss several future directions for our work.

2 Background

In this section we briefly review the two main concepts on which we base our method for predicting the reliability of component-based software: scenario specifications, and Cheung's user-oriented software reliability model. Note that we adopt Szyperski's definition of component as a unit of independent development and deployment. We further view components as being large-grained system entities (as opposed to small-scale components such as GUI widgets) for which one

may reasonably expect to have reliability data, which in turn can be established through reliability testing [6].

2.1 Scenarios

Scenario notations such as Message Sequence Charts [9] are used at early stages of development to document, elicit and describe system behaviour. Scenarios are partial descriptions of how components interact to provide system level functionality. A *scenario specification* is formed by composing multiple scenarios possibly from different stakeholders.

The underlying notion of scenario composition is that simple scenarios can be used as building blocks to describe new, more complex, scenarios. Simple sequences of behavior are described using *Basic Message Sequence Charts* (BMSCs). A BMSC is formed by vertical lines representing component time lines and horizontal arrows representing interactions between components. In this paper, we interpret each interaction as a synchronous communication between components. Because a BMSC can represent concurrent activity among the components it portrays, it denotes a partial ordering of activities, which in turn under an interleaving semantics determines a corresponding set of finite sequences of interactions.

Three fundamental constructs for combining BMSCs are *vertical composition* (where two BMSCs can be combined sequentially), *alternative composition* (defining that the system could alternatively choose one of the BMSCs to follow) and *iterative composition* (which composes a BMSC sequentially with itself). The *high-level MSC* (HMSC) is a widely adopted syntactic construct for describing scenario composition. An HMSC is a directed graph, whose nodes refer to BMSCs and whose edges indicate the acceptable ordering of the BMSCs. HMSCs allow stakeholders to reuse scenarios within a specification and to introduce sequences, loops and alternatives of BMSCs. The semantics of an HMSC is the set of sequences of interactions that follow some maximal path through the HMSC.

Throughout this paper we use a variant of the Boiler Control system example presented by Uchitel et al. [25]. As shown in Figure 1, the Boiler Control system consists of four components: *Sensor*, *Control*, *Database* and *Actuator*. In the top portion of the figure, we depict the HMSC specification of the Boiler, which composes five BMSCs: *Initialise*, *Register*, *Analyse*, *Terminate* and *End*, which are depicted in Figure 1, excluding the upper-left corner where the HMSC is. Note that the variables appearing in curly brackets in the figure are an extension to MSCs that we explain in Section 3.

2.2 The Cheung User-Oriented Reliability Model

In order to predict software system reliability, we need a reliability model that expresses system reliability as a function of the reliability of the components and the frequency of utilization of those components. Using Cheung's approach [3], the reliability of the system can be computed as a function of both the deterministic properties of the structure of the program and the stochastic properties of the utilisation and failure of its components.

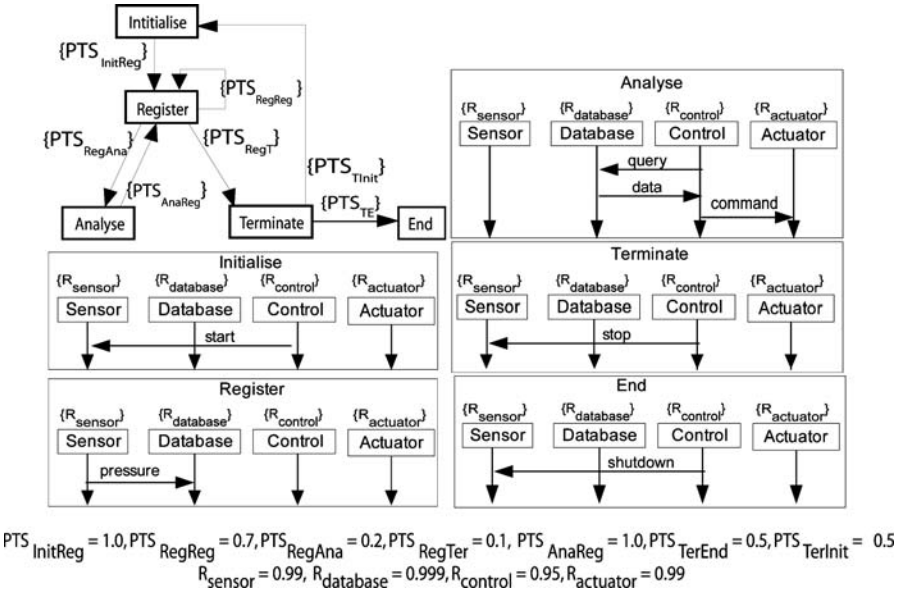

Fig. 1. The Message Sequence Chart Specification for the Boiler Control System, with Example Probability Values

Essentially, the Cheung model is a Markov reliability model that uses a program flow graph to represent the structure of the system. Every node N_i in the flow graph of the Cheung model represents a program module and a direct branch (N_i, N_j) represents a possible transfer of control from N_i to N_j. A probability P_{ij} that transition (N_i, N_j) will happen is attached to every directed branch. R_i is the reliability of node N_i. The original transition (N_i, N_j) in the flow graph is then modified into $R_i Pij$, which represents the probability that the execution of module N_i produces the correct result and control is transferred to module N_j. The reliability of the program is, therefore, the probability of reaching the correct termination of the program flow graph from its initial state in the following way: Let $N = \{C, F, N_1, N_2, ..., N_n\}$ be the states of the model, where N_1 is the start state of the program control flow graph, the N_i are intermediate states, N_n is the last (non-absorbing) state reached in any successful execution of the system, and C and F are absorbing states representing the terminal states Correct (to which there is a transition from N_n) and Fault. Let the transition matrix be M' where M'_{ij} represents the probability of transition from state i to state j:

$$\mathbf{M}' = \begin{array}{c} \\ C \\ F \\ N_1 \\ N_2 \\ \vdots \\ N_n \end{array} \begin{array}{c} \begin{array}{cccccccc} \mathrm{C} & \mathrm{F} & N_1 & N_2 & \dots & N_n \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1-R_1 & 0 & R_1P_{12} & \dots & R_1P_{1n} \\ 0 & 1-R_2 & 0 & R_2P_{22} & \dots & R_2P_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ R_n & 1-R_n & 0 & 0 & \dots & 0 \end{pmatrix} \end{array}$$

Let M be the matrix obtained from M' by deleting the rows and columns corresponding to the absorbing states C and F. Let S be a matrix such that:

$$S = I + M + M^2 + M^3 + ... = \sum_{k=0}^{\infty} M^k = (I - M)^{-1}$$

where I is the identity matrix with same dimension of M. Cheung shows that the system reliability is $Rel = S(1,n) \times R_n$, which is the probability of successfully transitioning from N_1 to N_n in any execution times the probability of successfully reaching C from N_n. Equivalently, Cheung shows that $S(1,n)$ can be computed as

$$S(1,n) = (-1)^{n+1} \frac{|M|}{|I - M|} \quad (1)$$

where $|M|$ and $|I - M|$ represent the determinant of M and $I - M$, respectively. We refer the reader to Cheung [3] for further details on the description and derivation of these formulae.

In the next section, we show how we weave the concepts presented in this section into a method for predicting the reliability of component-based software.

3 Reliability Analysis Using Scenarios

In this section we describe a method to predict software system reliability as a function of component reliability estimates. We annotate a scenario specification with probabilistic properties and use a probabilistic labelled transition system (LTS) synthesised from the scenario specification for the software reliability prediction. The method is depicted in Figure 2 as five major steps: (1) annotation of the scenarios, (2) synthesis of the probabilistic LTS, (3) construction of the stochastic matrix, (4) system reliability prediction, and (5) implied scenario detection.

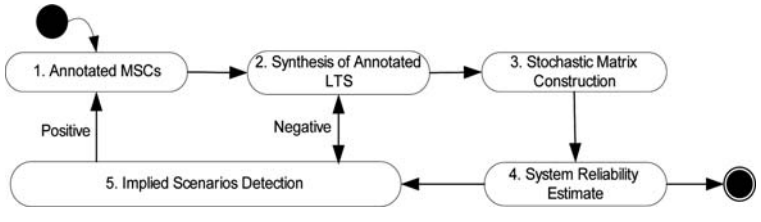

Fig. 2. The Reliability Prediction Method

Four key assumptions underlie our method:

1. The transfer of control between components has the Markov property, meaning that the transition from one execution state to another is dependent only on the source state and its available transitions and not on the past history

of state transitions. This is a traditional assumption that simplifies in work on reliability analysis and it greatly simplifies the computation of reliability estimates.
2. Failures are independent across transitions. Again, this assumption simplifies the computation of reliability estimates.
3. A message from component C to component C' represents an invocation by C of a service offered by C'. The reliability with which this service is performed is thus the reliability of C', $R_{C'}$. Additionally, the execution time of the invocation is assumed to be so short as not to be a factor in the component's reliability. In other words, $R_{C'}$ is the probability of successful completion of an invocation of any service offered by C', irrespective of the execution time of the service. This assumption is simply a modeling choice that is made without loss of generality. For instance, we could just as easily accommodate method-level reliabilities, and/or communication reliabilities (as is done, for instance, in Yacoub et.al [27])
4. There is only one initial and one final scenario for the system in the HMSC. Multiple initial and final scenarios can be combined by introducing a *super-initial* and a *super-final* scenario, analogously to the *super-initial state* and *super-final state* proposed by Wang et al. [26].

3.1 The Annotated Scenarios

In the first step, we annotate the scenarios (i.e., the HMSC and BMSCs) with two kinds of probabilities, *the probability of transitions between scenarios* PTS_{ij} and *the reliability of the components* R_C.

The transition probability PTS_{ij} is the probability that execution control transfers directly from scenario S_i to scenario S_j. This information would be normally derived from an operational profile for the system [14]. Thus, from scenario S_i, the sum of the probabilities PTS_{ij} for all successor scenarios S_j is equal to one. As the PTS_{ij} relates to the transition between scenarios, these probabilities are annotated on the corresponding edges of the HMSC, as shown on the HMSC of Figure 1.

The component reliabilities R_C are annotated on the BMSCs, as also shown in Figure 1. Without loss of generality, this paper uses coarse-grained, single values for the overall component reliabilities; in general, we could associate reliabilities with individual messages and/or segments of component timelines.

For the purposes of illustrating our method on the Boiler example, we use the values depicted in Figure 1 for the PTS_{ij}. The values for the PTS_{ij} are based on the assumption that the system executes the scenario *Register* (which causes sensor readings to be entered into the database) far more frequently than the scenarios *Analyse* and *Terminate*, and that when it does execute *Terminate* there is an equal probability of reinitialising and shutting down.

The values on Figure 1 for the reliability of the components reflect the assumption that the *Database* is a highly reliable commercial software product, that the *Sensor* and *Actuator* are components whose hardware interface to the sensed/actuated phenomena will eventually fail, and that *Control* is a complex software subsystem that still contains latent faults.

3.2 Synthesis of the Probabilistic LTS

The second step of our method is to synthesise a probabilistic LTS from the annotated scenario specification. This step is an extension of the synthesis approach of Uchitel et al. [24], which consists of the following steps:

1. For each component C_i and each BMSC S_j, a *labelled transition system* (LTS) $C_i\_S_j$ is constructed by projecting the local behaviour of C_i within S_j. In particular, each message with an action a that C_i sends or receives in S_j is synthesised as a transition with action a in $C_i\_S_j$, and the sequence of transitions in $C_i\_S_j$ corresponds with the sequence of messages sent or received by C_i in S_j.
2. For each component C_i, the set of LTSs constructed for C_i in step 1 are composed into a *component LTS* for C_i according to the structure of the HMSC, with hidden transitions (τ actions) linking the final state of $C_i\_S_j$ to the start state of $C_i\_S_{j'}$ whenever there is a transition from S_j to $S_{j'}$ in the HMSC. The resulting LTS includes a new start state corresponding to the start state of the HMSC.
3. Each component LTS constructed in step 2 is reduced to a trace-equivalent deterministic, minimal LTS. This is consistent with the delayed choice semantics of the ITU MSC standard [9].
4. The architecture model for the system is taken as the parallel composition of the minimised component LTSs constructed in step 3.

Our extension of this approach exploits recent probabilistic extensions to the LTS formalism [2] and involves enhancements to each step listed above. The enhancements have the effect of mapping the probability annotations of the scenario specification into probability weights for transitions in the synthesised architecture model. In step 1, for each transition in a $C_i\_S_j$ representing the invocation of a service offered by C_i, an additional transition from the same source state is added with the target state being the global ERROR state. The resulting pair of transitions forms a probabilistic choice, with the former transition having probability R_{C_i} and the latter transition having probability $1 - R_{C_i}$.

In step 2, the scenario transition probabilities PTS_{ij} are mapped to probability weights on the hidden transitions linking the $C_i\_S_j$. Figure 3 illustrates the LTS of component *Control* that would be synthesised as a result of applying steps 1 and 2 of our synthesis method. Each shaded area contains an LTS synthesised in step 1 from a BMSC of Figure 1 and thus models the behaviour of *Control* within that BMSC. The transitions linking these different LTS are synthesised in step 2 and correspond to the transitions between BMSCs defined in the HMSC of Figure 1. Note that the probability weights on the τ transitions are the same as the corresponding transitions in the HMSC of Figure 1. Note also that because *data* is a message received by *Control* in scenario *Analyse*, it is synthesised as two transitions, the "successful" transition being weighted with probability R_{ctrl} and the transition to the ERROR state (labelled -1 in the figure) being weighted with probability $1 - R_{ctrl}$. This action only applies to transitions labelled with *data* as it is an application of assumption three we

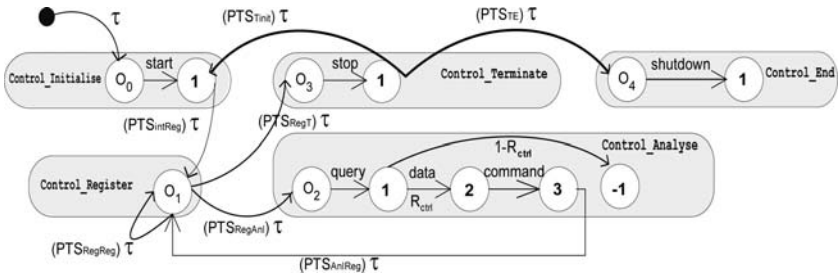

Fig. 3. Probabilistic LTS Synthesised for Component *Control*

explained earlier in this section. Note that the final state of the model is state 1 in the top right part of the figure.

Continuing with our extensions, in step 3, the probability weights must be handled correctly in the process of reducing each component LTS to its deterministic, minimal form. Intuitively, the elimination of a τ transition results in the merging of the transition's target state with its source state, with the outgoing transitions of the target state becoming outgoing transitions of the source state. Since there may be multiple τ transitions from the original source state (each with probability weight less than one), the probability weight of an eliminated τ transition must be "pushed" to the newly accumulated outgoing transitions, with the new weight on each such outgoing transition equal to its old weight times the weight on the eliminated τ transition. In the presence of τ self-loops (such as the τ self-loop on state 0 of *Control_Register* in Figure 4), it can be shown that such transitions can be eliminated entirely without any of the above merging or pushing of its weight. At the end of the elimination of outgoing τ transitions from a state, the weights on the outgoing transitions of the resulting state may not sum to one, in which case the weights must be normalised so that they do sum to one.

Using the example parameters presented previously in Figure 1, the resulting minimised LTS for component *Control* is depicted in Figure 4.

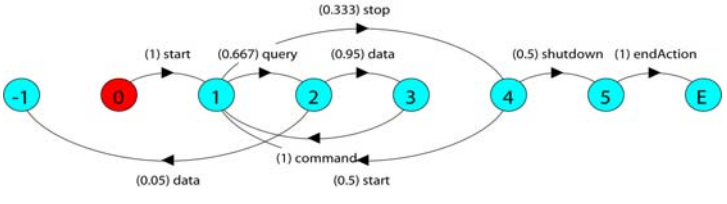

Fig. 4. Minimised Component LTS for Component *Control*

Finally, in step 4, the system architecture model is constructed as the parallel composition of the LTSs synthesized for each component. The probability weights of the composed LTS are computed according to the notion of *generative*

```
ArchitectureModel = Q0,
Q0    = ( (0.01) control.sensor.start -> ERROR
        | (0.99) control.sensor.start -> Q1),
Q1    = ( (0.001) sensor.database.pressure -> ERROR
        | (0.999) sensor.database.pressure -> Q2),
Q2    = ( (0.001) sensor.database.pressure -> ERROR
        | (0.809) sensor.database.pressure -> Q2
        | (0.152) control.database.query -> Q3
        | (0.038) control.sensor.stop -> Q10),
Q3    = ( (0.05) database.control.data -> ERROR
        | (0.95) database.control.data -> Q4),
Q4    = ( (0.005) control.actuator.command -> ERROR
        | (0.521) control.actuator.command -> Q5
        | (0.474) sensor.database.pressure -> Q9),
Q5    = ( (0.964) sensor.database.pressure -> Q2
        | (0.036) control.sensor.stop -> Q6),
Q6    = ( (0.005) control.sensor.start -> ERROR
        | (0.005) control.sensor.shutdown -> ERROR
        | (0.495) control.sensor.start -> Q1
        | (0.495) control.sensor.shutdown -> Q7),
Q7    = ( (1.0) endAction -> Q8),
Q8    = STOP,
Q9    = ( (0.006) control.actuator.command -> ERROR
        | (0.616) control.actuator.command -> Q2
        | (0.378) sensor.database.pressure -> Q9),
Q10   = ( (0.005) control.sensor.start -> ERROR
        | (0.005) control.sensor.shutdown -> ERROR
        | (0.495) control.sensor.shutdown -> Q7
        | (0.495) control.sensor.start -> Q11),
Q11   = ( (0.855) sensor.database.pressure -> Q2
        | (0.145) control.database.query -> Q12),
Q12   = ( (0.05) database.control.data -> ERROR
        | (0.95) database.control.data -> Q13),
Q13   = ( (0.005) control.actuator.command -> ERROR
        | (0.471) control.actuator.command -> Q1
        | (0.524) sensor.database.pressure -> Q9).
```

Fig. 5. The FSP of the Architecture Model

parallel composition defined by D'Argenio et al. [5]. At the end of this step, it follows that for each node of the synthesized architecture model, $\sum_{j=1}^{n} PA_{ij} = 1$, where n is the number of states in the LTS architecture model and PA_{ij} is the probability of transition between state S_i and S_j of the composed LTS. Otherwise, $PA_{ij} = 0$ if the transition (S_i, S_j) does not exist.

The architecture model for the Boiler Control system resulting from the application of all four steps of our extended synthesis method is depicted in Figure 5. For the sake of readability, we present the model in textual form as a specification expressed in FSP (Finite State Processes), the modelling notation of the LTSA tool (Labelled Transition System Analyser) [23]. FSP serves both as a modelling notation for end users, and as an intermediate form used in the automated synthesis of LTS models. As shown in the figure, a side-effect of the synthesis is the use of the auxiliary action *endAction* as the final action in a terminating path through the LTS.

3.3 Computing the Reliability Prediction

In this final step of our prediction method, the architecture model synthesised in the previous step is interpreted as a Markov model, and we apply the method of Cheung to compute the reliability prediction. In particular, the transition

$$\begin{pmatrix}
0 & 0.99 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.999 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.809 & 0.152 & 0 & 0 & 0 & 0 & 0 & 0 & 0.038 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.95 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.521 & 0 & 0 & 0 & 0.474 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.964 & 0 & 0 & 0 & 0.036 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.495 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.495 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.95 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.471 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.524 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.616 & 0 & 0 & 0 & 0 & 0 & 0 & 0.378 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.495 & 0.495 & 0 \\
0 & 0 & 0.855 & 0 & 0 & 0 & 0 & 0.145 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Fig. 6. The Matrix Derived from the Synthesized Boiler LTS

probability weights of the architecture model are mapped into a square transition matrix M' whose row entries sum to one. This is used as the matrix M' described in Section 2.2, with $N = \{E, -1, 0, 1, ..., n-1\}$ the set of states in the synthesised LTS, E the terminal state of correct execution (corresponding to state C described in Section 2.2), -1 the terminal fault state (state F of Section 2.2), and $n-1$ the state from which a transition to state E is made upon action *endAction* (state N_n of Section 2.2). Note that the numeric state labels produced by LTSA may need to be renumbered so that the state leading to state E is the highest numbered state, as required by Cheung's model.

In Figure 6 we depict the transition matrix derived from the synthesised architecture model presented in Figure 5; note that this is actually the reduced matrix M, with the rows and columns for states E and -1 eliminated as in Section 2.2. Additionally, we point out for the fact that the rows in the sparse matrix in Figure 6 will sum to one if we add the transitions to the *ERROR* state. Applying the Cheung model to that matrix, we compute the reliability prediction for the Boiler Control system as $Rel = 0.649 = 64.9\%$.

4 Implied Scenarios

Scenarios describe two aspects of a system. On the one hand, they describe a set of system traces the system is intended to exhibit. On the other, it describes the components that will provide system level functionality and their interfaces (the messages these components can use to interact between each other to provide system level functionality). In the example in Figure 1, we see that the Boiler Control System is expected to exhibit a trace *"start, pressure, query, data, command ..."* and that component Control interacts with *Database* only through messages *query* and *data*.

It has been shown [1, 25] that given a scenario specification, it may be impossible to build a set of components that communicate exclusively through the interfaces described and that exhibit only the specified traces when running in parallel. The additional unspecified traces that are exhibited by the composed system are all called implied scenarios and are the result of specifying the be-

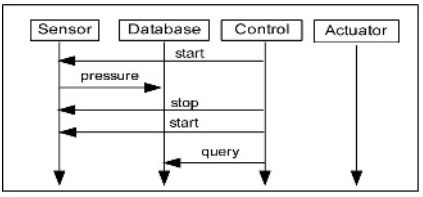

Fig. 7. Implied Scenario Detected

havior of a system from a global perspective yet expecting it to be provided by independent entities with a local system view. If the interaction mechanisms do not provide components with a rich enough local view of what is happening at a system level, they may not be able to enforce the intended system behavior. Effectively, what may occur is that each component may, from its local perspective, believe that it is behaving correctly, yet from a system perspective the behavior may not be what is intended.

The Boiler Control System of Figure 1 has implied scenarios, Figure 7 shows one of them. From the specification it is simple to see that after initialising *Sensor* there must be some pressure data registered into the *Database* before any queries can be done. However, in the implied scenario of Figure 7 a query is being performed immediately after start.

Why is this occurring? The cause is an inadequate architecture for the traces specified in the MSC specification. The *Control* component cannot observe when the Sensor has registered data in the *Database*, thus if it is to query the *Database* after data has been registered at least once, it must rely on the Database to enable and disable queries when appropriate. However, as the *Database* cannot tell when the *Sensor* has been turned on or off, it cannot distinguish a first registration of data from others. Thus, it cannot enable and disable queries appropriately. Succinctly, components do not have enough local information to prevent the system execution shown in Figure 7. Note that each component is behaving correctly from its local point of view, i.e. it is behaving according to some valid sequence of BMSCs. The problem is that each component is following a different sequence of BMSCs! The *Sensor*, *Control* and *Actuator* are going through scenarios *Initialise*, *Register*, *Terminate*, *Initialise*, *Analysis*, *Register*. However, the *Database* is performing *Initialise*, *Register*, *Analysis*, *Register*.

Implied scenarios indicate gaps in a scenario-based specification. They can represent intended system behaviour that was missing from the inherently partial scenario specification or undesired behaviour that should be avoided by changing the architecture of the system. Hence, implied scenarios need to be validated (identifying them as positive or negative system behaviour) and the scenario specification elaborated accordingly.

The existence of an implied scenario means that the reliability prediction for the Boiler Control System described above has been applied on a scenario specification that has a mismatch between behaviour and architecture. The behaviour model constructed in the previous section to predict reliability can exhibit behaviour (an implied scenairo) that has not yet been validated and that, acording

to whether it described intended or unintended system behaviour, can impact system reliability.

As an example, suppose that the rate at which the sensor checks pressure information and saves it in the database is high enough that the probability of occurence of the trace in Figure 7 is negligible. Then reliability should be predicted on the behaviour model of Figure 5 constrained in such a way that the implied scenario cannot occur. We can use the approach described in [25] to build such a constraint.

If we calculate the reliability of the resulting constrained model in the same way as described in Section 3 then we obtain 86.2%.

On the other hand, the implied scenario may be undesired behaviour that needs to be avoided through a change in the architecture of the system. In this case, different or additional components will be needed, and the reliability performance will have to be recalculated from scratch.

Either way shows that implied scenarios can impact the reliability prediction significantly and that they should be validated before reliability is calculated.

More generally, the existence of implied scenarios as a result of the close relation that exists between behaviour and architecture in scenario-based specifications supports our claim that taking into account behaviour and architecture when performing reliability prediction is important.

5 Discussion and Related Work

Several previous architecture-based approaches to reliability engineering of component-based systems have been reported. They can be divided into two main categories, *state-based* approaches and *path-based* approaches. Goševa-Popstojanova and Trivedi provide a comprehensive survey of the various approaches [8]. For the sake of brevity, we provide here a brief view of the approaches of greatest interest to the scope of this work.

State-based models [3, 7] use a control flow graph to represent the system architecture. In such models it is assumed that the transfer of control among the components can be modelled as a Markov chain, with future behaviour of the system dependent only on the current state and not on past behaviour. Gokhale et al. use a regression test suite to experimentally determine the architecture of the software and the reliabilities of its components. As described in Section 2, Cheung's model takes into account the reliability of each component and the operational profile. In general, relying the analysis of the software reliability on provided state-machines may not be accurate. In our model, the system states are generated by the LTSA based on the precision of a model checker. Although scenarios are provided as a basis for the analysis, we explore the expressiveness of the given scenarios by checking if the existence of implied scenarios that could impact negatively during the system execution.

Path-based models [20, 27] compute the reliability of the system by enumerating possible execution paths of the program. The scenario-based method of Yacoub et al. [27] is perhaps closest in spirit to our own approach. In many ways

their method is a hybrid approach in which a state-based model of the system is constructed from a scenario specification (a set of basic scenarios plus a graph representing the composition of basic scenarios), and then paths through the model are enumerated until a threshold execution time is reached along each path. Their approach reveals the pitfalls of using imprecise, coarse-grained behaviour models of system architecture. The model used in their approach is the *component dependence graph* (CDG), a state-machine model in which the states represent execution inside a particular component (with one state per component), and the transitions represent the transfer of control from one component to another (with a transition from one component to another representing a merge of all messages sent by the former to the latter in the scenarios). Because the representation of component behaviour in the CDG is at the level of whole components, it is an inherently sequential model of system behavior in which one component executes at a time, meaning that any concurrency inherent in the scenario specification is lost. Furthermore, a CDG can exhibit sequences of component transitions not found in the scenarios from which it is derived. In a sense such sequences are implied scenarios, but they arise not as an artefact of components having limited local knowledge of global behaviour. Instead, they are merely a consequence of modelling the system architecture imprecisely at the granularity of whole components rather than at the granularity of the component interactions specified in the scenarios. Finally, it can happen that a component in a CDG is represented by an absorbing state, even though the scenario specification itself is able to progress beyond any interactions with the "absorbing" component. Indeed, we attempted to model the Boiler Control system using the approach of Yacoub et al., with the result that the *Actuator* was an absorbing component from which we had to add transitions artificially to other components in order to construct a model that was able to progress to the final state.

In previous work we show how reliability engineering of component-based software systems can be carried out following a model-driven approach [17, 18]. It would be fair to say that the Unified Modeling Language (UML) has had a considerable influence to make viable model driven analysis approach such as [22], where design and analysis of software architecture can be specified, visualized, constructed and documented using one common notation. Since its first version, UML has been enriched in order to become more precise syntactically and semantically. The ultimate goal is to support automated or semi-automated transformation of design models to code, raising the level of abstraction at which automated code generation is applied. A major challenge for model-driven development will be finding ways of enforcing or preserving properties established early in development, particularly non-functional properties such as reliability predictions.

Other work can be situated in the area of a model-driven analysis technique: [12, 13, 4]. These approaches also propose a framework for automatic generation of reliability models from software specifications, bringing reliability analysis to early stages of the software lifecycle. István et al. [12, 13] shed some light on ways to fully automate dependability analysis, applied to the Fault-Tolerant

CORBA, using graph transformations into their VIATRA framework. The work from Singh et al. [21] provides a prediction algorithm to analyse the reliability of the system prior to its construction. Their approach requires the user to provide global behavior scenarios other than the local behavior of the components interactions. However, this feature may turn out to be unsuitable for the system modularity and therefore hindering systems maintainability.

6 Conclusion and Future Work

In this paper, we have presented a framework to quantitatively assess software reliability using scenario specifications, thus applicable to early phases of the software life cycle. Our major contribution lies on a reliability prediction technique that takes into account the component structure exhibited in the scenarios and the concurrent nature of component-based systems.

In the approach we present, we have extended scenario specification to model the probability of component *failure*, and *scenario transition probabilities* derived from an operational profile of the system. From the extended scenario specification, probabilisitic behaviour models were synthesised for each componenet and then composed in parallel into a model for the system. The Cheung model for software reliability was then used to compute a reliability prediction from the system behaviour model. The importance of implied scenarios detection in the software reliability analysis was then addressed so that the intended system behaviour could be enforced despite the local view of the components. We numerically showed how the detection of implied scenarios can improve the reliability assurance of the software system.

For future work, we will use our framework to enhance software system reliability using software architecture models. In doing this, we can use our framework for the purpose of model driven development to construct deployment profiles and generate implementation code configured to the desired reliability assurance for software systems. Another promising direction includes the use of the synthesized component LTS to predict component reliability. This may be useful when there are uncertainties associated with a component s operational profile coming out from lack of implementation artifacts. In Section 4 we presented initial evidence of how important is to consider implied scenarios when assessing provided scenario specifications for reliability. However, additional work is needed to explore methods and techniques that can fully reveal the effect of implied scenarios on system reliability. Finally, we plan to apply our approach on case studies of larger, more realistic systems in order to evaluate its scalability and the accuracy of the predictions it produces.

Acknowledgment

David Rosenblum holds a Wolfson Research Merit Award from the Royal Society. Sebastian Uchitel was partially funded by EPSRC grant READS GR/S03270/01 and Genaína Rodrigues was funded by CAPES, under grant number 108201-9.

We would like to thank Rami Bahsoon, Philip Cook and the anonymous referees for their helpful suggestions on improving the manuscript.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. of the* 22^{nd} *ICSE*, pages 304–313. ACM Press, 2000.
2. T. Ayles, A. Field, J. Magee, and A. Bennett. Adding performance evaluation to the LTSA tool (tool demonstration). In *Proc. 13th Performance Tools*, September 2003.
3. R. C. Cheung. A User-Oriented Software Reliability Model. In *IEEE Transactions on Software Engineering*, volume 6(2), pages 118–125. IEEE, Mar. 1980.
4. V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of uml based software models. In *Proceedings of the* 3^{rd} *WOSP*, pages 302–309. ACM Press, 2002.
5. P. R. D'Argenio, H. Hermanns, and J.-P. Katoen. On generative parallel composition. In C. Baier, M. Huth, M. Kwiatkowska, and M. Ryan, editors, *Electronic Notes in Theoretical Computer Science*, volume 22. Elsevier, 2000.
6. P. Frankl, R. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, 1998.
7. S. Gokhale, M. Lyu, and K. Trivedi. Reliability Simulation of Component Based Software Systems. In *Reliability Simulation of Component Based Sofware Systems*, pages 192–201. Proc. of the 9^{th} ISSRE, 1998.
8. K. Goševa-Popstojanova and K. S. Trivedi. Architecture-Based Approach to Reliability Assessment of Software Systems. In *Performance Evaluation Journal*. Elsevier Science, 2001.
9. ITU. ITU-T Recommendation Z.120 Message Sequence Charts (MSC'99). Technical report, ITU Telecommunication Standardization Sector, Geneva, 1996.
10. M. R. Lyu. *Software Reliability Modeling*. World Scientific Publishing Company, 1991.
11. M. R. Lyu. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill, 1996.
12. I. Majzik and G. Huszerl. Towards dependability modeling of FT-CORBA architectures. In *Proc.* 4^{th} *EDCC, Toulouse)*, pages 121–139. Springer–Verlag, 2002.
13. I. Majzik, A. Pataricza, and A. Bondavalli. Stochastic Dependability Analysis of System Architecture Based on UML Models. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems, LNCS–2667*, pages 219–244. Springer Verlag, 2003.
14. J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, 1993.
15. J. D. Musa, A. Iannino, and K. Okumoto. *Software reliability: measurement, prediction, application*. McGraw-Hill, Inc., 1987.
16. Object Management Group. Unified Modeling Language Specification version 2.0:Superstructure. Technical report, http://www.omg.org/docs/ptc/03-08-02.pdf, 2003.
17. G. Rodrigues. A Model Driven Approach for Software Systems Reliability. In *Proc. of the Doctoral Symposium of the* 26^{th} *ICSE, May 2004 - Edinburgh, Scotland*. IEEE Computer Society, May 2004.

18. G. Rodrigues, G. Roberts, and W. Emmerich. Reliability Support for the Model Driven Architecture. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *To Appear in: Architecting Dependable Systems II –LNCS*. Springer Verlag, 2004.
19. R. Roshandel and N. Medvidovic. Toward Archtitecture-Based Reliability Estimation. In *ICSE/WADS 2004, Edinburgh, UK.*, pages 2–6. IEEE Computer Society, May 2003.
20. M. Shooman. Structural Models for Software Reliability Prediction. In *Proc. of the* 2^{nd} *ICSE*, pages 268–280, 1976.
21. H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj. A bayesian approach to reliability prediction and assessment of component based systems. In *Proc. of the* 12^{th} *IEEE ISSRE*, pages 12–21. IEEE, 2001.
22. J. Skene and W. Emmerich. A Model Driven Architecture Approach to Analysis of Non-Functional Properties of Software Architecture. In *Proc. of the* 18^{th} *ASE. Toronto, CA*. IEEE Computer Society, Oct. 2001.
23. S. Uchitel, R. Chatley, J. Kramer, and J.Magee. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In *Proc. of* 9^{th} *TACAS, Warsaw*, Apr. 2003.
24. S. Uchitel, J. Kramer, and J.Magee. Synthesis on Behavioral Models from Scenarios. In *IEEE Transactions on Software Engineering*, volume 29(2), pages 99–115. IEEE, Feb. 2003.
25. S. Uchitel, J. Kramer, and J.Magee. Incremental Elaboration of Scenarios-Based Specifications and Behavior Models Using Implied Scenarios. In *ACM Transactions on Software Engineering and Methodologies*, volume 13(1), pages 37–85. ACM Press, Jan. 2004.
26. W. L. Wang, Y. Wu, and M. H. Chen. An Architecture-Based Software Reliability Model. In *Proc. Pacific Rim International Symposium on Dependable Computing. Washington, DC , USA*, pages 143–150. IEEE Computer Society, 1999.
27. S. M. Yacoub, B. Cukic, and H. H. Ammar. Scenario-Based Reliability Analysis of Component-Based Software. In *Proc. of the* 10^{th} *ISSRE, Boca Raton, FL, USA*. IEEE, Nov. 1999.