A Scenario-Based Reliability Analysis Approach for Component-Based Software

Sherif Yacoub, *Member, IEEE*, Bojan Cukic, *Member, IEEE*, and Hany H. Ammar, *Member, IEEE*

***Abstract*—This paper introduces a reliability model, and a reliability analysis technique for component-based software. The technique is named *Scenario-Based Reliability Analysis* (SBRA). Using scenarios of component interactions, we construct a probabilistic model named *Component-Dependency Graph* (CDG). Based on CDG, a reliability analysis algorithm is developed to analyze the reliability of the system as a function of reliabilities of its architectural constituents. An extension of the proposed model and algorithm is also developed for distributed software systems. The proposed approach has the following benefits:**

- **It is used to analyze the impact of variations and uncertainties in the reliability of individual components, subsystems, and links between components on the overall reliability estimate of the software system. This is particularly useful when the system is built partially or fully from existing off-the-shelf components.**
- **It is suitable for analyzing the reliability of distributed software systems because it incorporates link and delivery channel reliabilities.**
- **The technique is used to identify critical components, interfaces, and subsystems; and to investigate the sensitivity of the application reliability to these elements.**
- **The approach is applicable early in the development lifecycle, at the architecture level. Early detection of critical architecture elements, those that affect the overall reliability of the system the most, is useful in delegating resources in later development phases.**

***Index Terms*—Component-based software, component-dependency graphs (CDG), scenario-based reliability analysis (SRBA), software reliability analysis and modeling.**

ACRONYMS[1]

CBRE	Component Based Reliability Estimation
CBSE	Component-Based Software Engineering
CDG	Component Dependency Graph
COTS	Commercial Off-The-Shelf
DICOM	Digital Imaging and Communication in Medicine
DIMSE	DICOM Message Service Elements
EET	Extended Execution Time
IOD	Information Object Definition
MM	Markov Models
MSC	Message Sequence Charts
SBRA	Scenario-Based Reliability Analysis
SS_CDG	Subsystem Component Dependency Graph
UML	Unified Modeling Language

NOTATION

AE_{appl}	Average Execution Time of a Global Scenario
Ci	The ith component
E	A set of edges in a CDG
EC_i	The average execution time of the component C_i
ESS_m	The average execution time of the subsystem SS_m
N	A set of nodes in a CDG
n	A Node in a CDG, could be a component or a subsystem
NC_i	Name of component number i
NSS_m	Name of subsystem number m
PS_k	Probability of scenario number k
PT_{ij}	Probability of a transition from node n_i to node n_j
RC_i	A reliability estimate of component C_i
RSS_m	A reliability estimate of subsystem SSm
RT_{ij}	A reliability estimate of a transition from node n_i to node n_j
s	A start node in a CDG
S_k	Scenario number k
SS_m	Subsystem number m
t	A termination node in a CDG
T_{ij}	Name of the transition from node n_i to n_j

I. INTRODUCTION

COMPONENT-BASED SOFTWARE ENGINEERING (CBSE) is a specialized form of software reuse concerned with building software from existing components (including Commercial Off-The-Shelf components, COTS) by assembling them together in an interoperable manner. Achieving a highly reliable software application is a difficult task, even when high quality, pre-tested, and trusted software components are composed together [36]. As a result, several techniques have emerged to analyze the reliability of component-based applications. These can be categorized as:

- *System-level reliability estimation*. Reliability is estimated for the application as a whole.
- *Component-based reliability estimation*. The application reliability is estimated using the reliability of the individual components and their interconnection mechanisms.

Manuscript received June 26, 2000. This work was supported in part by grants to West Virginia University Research Corp. from the National Science Foundation Information Technology Research (ITR) Program grant CCR-0082574; from the NASA Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) managed through the NASA Independent Verification and Validation (IV&V) Facility, Fairmont, West Virginia; from NASA research grant NAG4-163; and from NSF CAREER award CCR-0093315. Associate Editor: R. A. Evans.

S. Yacoub is with the Hewlett-Packard Labs, Palo Alto, CA 94304 USA (e-mail: sherif.yacoub@hp.com).

B. Cukic and H. H. Ammar are with the Lane Department of Computer Science & Electrical Engineering, West Virginia University, Morgantown, WV 26506 USA(e-mail: cukic@csee.wvu.edu; hany.ammar@mail.wvu.edu).

Digital Object Identifier 10.1109/TR.2004.838034

[1]The singular and plural of an acronym are always spelled the same.

The first approach treats the software system as a unit. This approach is not the most suitable for component-based applications because it does not consider compositional properties of systems, and does not accommodate the reliability growth of individual components. The limitations of system-level approaches for component-based applications are discussed in [8]. As for the second approach, two issues arise. The first is about estimating the reliability of individual components, and the second is about analyzing the reliability of the application by aggregating the reliabilities of constituting components. This paper addresses the second problem, the analysis of the reliability of a component-based system as a function of its constituents. We are concerned with reliability analysis models for systems whose design is substantially based on independent execution scenarios. This work is motivated by the need to:

- Study the *sensitivity* of the application reliability to reliabilities of components and their interfaces. This guides the process of identifying components and interfaces with critical impacts on system reliability. We can analyze the effects of replacing components with new ones, with similar/same interfaces but improved reliability.
- Develop a probabilistic technique for reliability analysis which is applicable at the architecture level [33], [34]. Many reliability analysis techniques use test cases and fault injection. Using scenarios has the advantage of applicability in the *early phases* of development life cycle.
- Analyze the reliability of a component-based application even when the source code is not available (i.e. fault injection and seeding would not be applicable). This is frequently required by systems built of COTS components.
- Develop a reliability analysis technique that addresses issues related to the distributed nature of software systems, such as the *complexity* and hierarchical composition of subsystems. Moreover, physical distribution of components imposes the need for careful treatment of inter-component *link* reliabilities.

As mentioned above, the proposed technique is suitable for applications designed and analyzed using independent scenarios describing component interactions. Independent scenarios identify system execution paths. Each scenario terminates by returning the flow of control back to its starting conditions without carrying over state changes to the next scenario. Many reactive systems, such as feedback systems and simulation environments, have this property. The technique also assumes sequential execution of components; parallelism is outside the scope of the current study.

We propose a new technique called *Scenario-Based Reliability Analysis* (SBRA), which builds on scenarios used in the analysis phase of component-based system development. A *Component Dependency Graph* (CDG) is proposed as a new reliability analysis model. A CDG incorporates component and interaction usage probabilities, as well as their estimated reliabilities. An algorithm is developed to analyze the reliability of component-based applications using the information embedded in a CDG. In general, we are not concerned with estimating the reliability of the application over its execution lifetime. In our opinion, early lifecycle data is not sufficient for precise prediction of operational reliability. Instead, we concentrate our analysis to study the implications of component/interface/link reliabilities to the expected reliability of the software system, once it is built.

Section II summarizes related work on reliability analysis, and estimation of component-based software. Section III introduces the reliability analysis model, Component Dependency Graph (CDG). Section IV describes the proposed scenario-based analysis approach; constructing CDG, and applying an analysis algorithm which traverses these graphs. Section V describes the application of the model, and the algorithm to a case study. Section VI discusses the extensions of the reliability model, and the algorithm for distributed systems. Section VII describes the application of this approach to a distributed system example.

II. Background

Several reliability models and estimation techniques have been proposed to assess the reliability of component-based applications. Gokhale *et al.* [6] discuss the flexibility offered by discrete-event simulation to analyze component-based applications. Their approach relies on random generation of faults in components using a programmatic procedure which returns the inter-failure arrival time of a given component. The total number of failures is calculated for the application under simulation, and its reliability is estimated. This approach assumes the existence of a control flow graph of a program. The simulation approach assumes failure and repair rates for components, and uses them to generate failures in executing the application. It also assumes constant execution time per component interaction, and ignores failures in component interfaces and links (transition reliabilities). Sanyal *et al.* [18] introduce Program Dependency Graphs and Fault Propagation Analysis [19], [22] for analytical reliability estimation of component based-applications. The approach is code-based (reverse-engineering) where dependency graphs are generated from source code, which may not be available for off-the-shelf components.

Krishnamurthy *et al.* [11] assess the reliability of component-based applications using a technique called *Component Based Reliability Estimation* (CBRE). The approach is based on test information and test cases. For each test case, the execution path is identified. The path reliability is calculated using the reliability of the components assuming a series connection (using the independent failure assumption and perfect interfaces between components). This approach does not consider component interface faults, although they are considerable factors in reliability analysis of component-based software. In an attempt to borrow ideas from the hardware reliability engineering community, a research group at the University of Toronto [13], [25] proposes that designers should follow certain disciplines in software component development. They develop a set of design and interaction rules to minimize the interaction and dependence between components. The proposed rules facilitate modeling the component-based system as Markov chains and, hence, we can apply the same reliability analysis techniques as in the case of

hardware systems. A similar study of the failure dependency between components is discussed in [37]. However, such discipline is difficult to impose in practice.

The techniques discussed above are so called path-based approaches to reliability analysis of component-based software. Most path-based approaches assume that components fail independently, thus providing a pessimistic estimate of system reliability. Gokhale *et al.* [5] propose a technique which allows handling of dependent failures. The solution takes into account time-dependent representation of component reliability, and a fixed execution time per interaction. Everett [4] describes an approach which uses the Extended Execution Time (EET) reliability growth model to arrive at a composite reliability growth model for the testing period. The proposed CDG model and SBRA algorithm could be integrated into the "*superimpose component reliability*" step in Everett's framework.

Schneidewind [27] addresses the problem of assessing the reliability of distributed systems in which a set of client and server nodes remotely communicate with each other over a network. *Physical distribution* imposes the need to consider new factors in reliability analysis of software systems:

a) accounting for a possibility that the survivability of certain client and server components will be more critical to the system operation than others, and
b) accounting for the possibility of using redundant services across the network to allow system recovery if one of the critical nodes fails.

Several approaches use Markov chains to develop reliability models (Markov Models, MM) for software systems, for example [28], [29]. While the models we develop in this paper may look similar to MM in representation, they are conceptually different. MM are used to capture the system states, and the transitions from one state to another. Transitions are the results of component failures. The models that we develop here are based on representing the system as the composition of software components, not system states. Whereas state modeling focuses on the behavioral aspects of the system, we focus on capturing component interactions which represent system architecture. MM techniques have limited applications to complex systems when it becomes hard to identify the large number of system states.

Poore *et al.* [30] propose an approach which helps developers plan for software reliability analysis and certification. As part of their methodology, a component model is used to represent the system as a composition of components with transition probabilities. The work proposed in this paper shares similar objectives. Poore does not discuss how to obtain the component model nor its parameters, such as transition probabilities. He assumes that the analyst will construct the model based on domain experience. It is not clear in the approach how to obtain some model parameters such as transition probabilities. Our approach extends Poore's model by formally defining the component model, and a traversal algorithm to analyze system reliability as a function of component and transition reliabilities. We show how to use design and analysis scenarios to define the component model such that it is not heavily dependent on the domain analyst's intuition. Moreover, we address systems which are hierarchical in nature.

Goseva-Popstojanova and Trivedi [32] present a classification of architecture-based approaches to reliability assessment of component-based software systems. They identify three classes based on the methods used to describe the architecture, and aggregate the failure behavior of components and connectors. These classes are:

a) *state-based* where software architectures and failure behavior are represented as a Markov chain or a semi-Markov process;
b) *path-based* where reliability is estimated for set of execution scenarios [33], [35]; and
c) *additive models* which focus on estimating the time-dependent failure intensity of the system using components failure data.

The approach we present in this paper is a path-based approach, adequate for large-scale systems where the analysis of MM will be prohibitive due to the state space explosion problem. Our approach is also based on execution scenarios. A scenario is a set of component interactions triggered by a specific input stimulus [24]. One way to model scenarios is by using *sequence diagrams*. Sequence diagrams specify interactions between application entities in a time sequence manner. We adopt *Sequence Diagrams* as means of documenting a scenario for a component-based application. We also use the word *interaction* as a general term to refer generically to all possible types of collaboration between components. The notation of sequence diagrams involving components is similar to those used for Message Sequence Charts (MSC) [9] or interaction diagrams in the Unified Modeling Language (UML) [20], with some generalization of the terms. Modeling languages, in general, provide support to represent events (interactions between components) in sequence diagrams using an ordinal scale. Most languages are concerned with the order of invocations instead of the execution time of the component as a result of an invocation. Little research has considered annotating the sequence diagrams with execution times. For instance, Firley *et al.* [31] propose extensions to the UML to represent timed sequence diagrams. For the purpose of our study, we require that sequence diagrams be annotated with time stamps showing estimated component execution times. The majority of modeling tools provide support for annotations, which an analyst can use to capture execution time estimates in sequence diagrams. Scenarios are also related to the concept of operations and run-types used for the description of operational profiles [16]. Scenarios with specified input variables are similar to operation run-types. A generic scenario with fewer details about input values, but specific for an input (sub)domain, is similar to an operation with several run-types. Finally, a profile of the component execution probabilities assigned to scenarios is similar to the operational profile. Operational profiles have long been used to guide testing, development, and performance analysis by identifying frequently executed operations. Here, we use scenarios to derive a probabilistic model for the purpose of reliability analysis of component-based systems.

III. Component Dependency Graph (CDG)

Control flow graphs are the classical method of revealing the structure, decision points, and branches in program code [17]. We adapt the control flow graph principles to component-based applications to represent the architectural dependency between components and possible execution paths. We call this graph a *Component Dependency Graph*, CDG. In this section we define the graph, while the following section describes how to calculate graph attributes.

Definition 1: A Component Dependency Graph "CDG": A CDG is defined by *the tuple* $\langle N, E, s, t\rangle$, where $\langle N, E\rangle$ is a directed graph, s is the start node, t is a termination node. N is a set of nodes in the graph, $N = \{n_i\}$, $i = 1 \ldots |N|$, and E is a set of directed edges in the graph, $E = \{e_i\}$, $i = 1 \ldots |E|$.

Definition 2: A Node "n": $n \in N$ models a component C_i, and is defined by the tuple $\langle NC_i, RC_i, EC_i\rangle$, where: NC_i is the name of component C_i, RC_i is the reliability of component C_i, and EC_i is its average execution time of the component C_i.

Definition 3: Component Reliability "RC_i": RC_i is the probability that the component C_i executes correctly (failure free) upon invocation.

Definition 4: Average Execution Time of a Component "EC_i": EC_i is the average execution time of a component C_i.

Definition 5: A Directed Edge "e": $e \in E$ models the control flow transfer from one component to another. It is annotated by the tuple $\langle T_{ij}, RT_{ij}, PT_{ij}\rangle$ where: T_{ij} is the transition name from node n_i to n_j, denoted $\langle n_i, n_j\rangle$, RT_{ij} is the transition's reliability, and PT_{ij} is the transition's execution probability.

Definition 6: Transition Reliability "RT_{ij}": RT_{ij} is the probability that information sent from component C_i to component C_j is delivered error-free. This probability includes possible interface errors and possible channel delivery errors, as discussed in Section IV-A-3.

Definition 7: Transition Probability "PT_{ij}": PT_{ij} is the conditional probability that C_j will execute next given that C_i is currently executing. The sum of outgoing transition probabilities from any node must be unity.

Thus, a CDG is defined as follows:

$$
\begin{aligned}
&CDG = \langle N, E, s, t\rangle,\\
&N = \{n\},\ E = \{e\},\ s \text{ and } t \text{ are the start}\\
&\quad \text{and termination nodes}\\
&n = \langle NC_i, RC_i, EC_i\rangle,\\
&\mathrm{e} = \langle T_{ij}, RT_{ij}, PT_{ij}\rangle,\\
&T_{ij} = \langle n_i, n_j\rangle
\end{aligned}
$$

Fig. 1 shows the CDG of a system consisting of four components.

IV. Scenario-Based Reliability Analysis (SBRA)

We propose to analyze the reliability of a component-based software application in three steps:

1) estimation of the parameters used in the reliability model;
2) construction of the component dependency graph; and
3) application of the algorithm for reliability and sensitivity analysis.

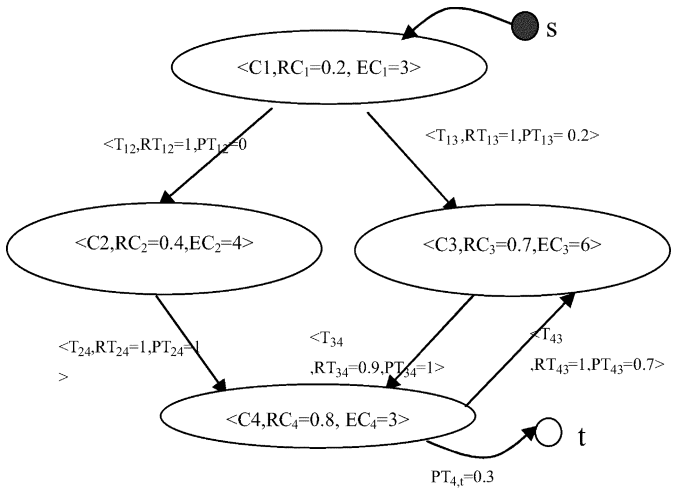

Fig. 1. A sample CDG.

The rest of this section describes these steps.

A. Parameter Estimation

1) Scenario-Related Parameters: **Scenario "$\mathbf{S_k}$"**. A scenario S_k from the set of the application scenarios S, $S_k \in S$ where $k = 1 \ldots |S|$, represents a sequence of component interactions. At the system level, scenarios are activated by specific input stimuli.

Scenario Probability "$\mathbf{PS_k}$". The dynamic behavior of a component-based application is specified using a set of scenarios. Each scenario is assigned the probability of execution. $\mathrm{PS_k}$ is the execution frequency of scenario k with respect to all other scenarios. The execution probabilities of all scenarios $\mathrm{S_k}$, $\mathrm{k} = 1 \ldots |\mathrm{S}|$, should sum to unity.

It is usually difficult to exhaustively document all possible scenarios, especially at early phases of the development life cycle. In such cases, to estimate PS_k, the concepts of operational profile and equivalence partitioning [15] could be used to specify scenario categories. Alternative solutions to identify/select scenarios to use in reliability analysis can be found in [26].

Average Execution Time of the application "$\mathbf{AE_{appl}}$". The average execution time of the application (AE_{appl}) is given by

$$AE_{\mathrm{appl}} = \sum_{\mathrm{k}=1}^{|\mathrm{S}|} \mathrm{PS_k} \times \mathrm{Time}(\mathrm{S_k}) \tag{1}$$

where PS_k is the probability of execution of scenario S_k, and $\mathrm{Time}(S_k)$ is the average execution time of scenario S_k.

2) Component-Related Parameters: **Component Reliability "$\mathbf{RC_i}$"**. Several techniques have been proposed to estimate the reliability of software components. We assume that the reliability of individual components has been calculated (numerical formula) or modeled (mathematical formula). We refer to this estimate as RC_i. Techniques for component reliability estimation include fault injection, testing, and retrospective analysis. Assessing the reliability of individual components, as mentioned earlier, is outside the scope of our discussion. Instead, assuming an estimate is available, we use it as a parameter to analyze the sensitivity of the application

reliability to variances in component reliabilities. Our system reliability model (CDG) can also accommodate the application of various software reliability growth models (SRGM) to components. In the analysis algorithm discussed in Section IV-C, the point reliability of the component could be substituted with a time-based model. In the SBRA algorithm, the average execution time of the component can be used in the component's SRGM to obtain an estimate. For instance, component reliability can be $RC_i(t)$, where t is the accumulated execution time, $t = M * EC_i$; and M is the M^{th} visit to the component during the execution of graph traversal algorithm.

Average Execution Time of a Component "$\mathbf{EC_i}$". The parameter EC_i represents the average execution time of component C_i. It is calculated using the following equation:

$$EC_i = \sum_{k=1}^{|S|} PS_k * \text{Time}(C_i)_{C_i \text{ in } S_k} \tag{2}$$

where

PS_k is the probability of execution of scenario S_k, Time(C_i) is the execution time of C_i, measured as the sum of its active times in the execution of scenario S_k.

The average execution times of components may not be available when applying SBRA methodology at the design level (coding has not yet started). In those cases, the analyst can make relative estimates of component execution times, for example, by comparing the component complexities. While this may be inaccurate, the methodology can be later used to study the effect of such uncertainties on the system reliability.

3) Transition Related Parameters: **Transition Reliability "$\mathbf{RT_{ij}}$".** The reliability of a transition from one component to another in the CDG model is the probability that the information is correctly delivered from the source component to the destination in the course of an execution. We do not describe in detail here how to calculate transition reliabilities. We are primarily concerned with incorporating them (as first class elements) in the model so that their effect on application reliability could be analyzed.

Estimating the transition reliability depends on two important factors: *Component Interfaces Reliability* and *Link Reliability*. Thus, $RT_{ij} = Interface\ Reliability * Link\ Reliability$. In the following, we briefly describe factors affecting interface and link reliabilities.

Component Interface Reliability is defined as the probability that two interacting components have matching interfaces. A component's interface defines how it interacts with other components. Interfaces describe the import and export relationships. A set of exported interfaces specifies the services that the component can provide. A set of imported interfaces specifies the services which this component requires from other (external) components, needed in an execution. A mismatch in an interface can be the result of an incompatibility in the structure or the sequence of messages exchanged between components, incompatibilities in data formats, types and message protocols, or misunderstood roles in an interaction (the client versus the server). Client/Server relationship, for example, is defined by import/export interface specifications. Formal specification of component interfaces is an approach to improve their reliability. Interface reliability improvement is an active research area [2], [21], and it deserves further study.

Link Reliability (also referred to as delivery channel reliability) is the probability of correct delivery of messages exchanged between components. This factor is essential in the case of component distribution across a network. Heimdahl *et al.* [7] estimate that almost 35% of faults in the system they studied were related to interface mismatches between components, and their underlying environments, which included communication channels. A message exchanged between components in a distributed environment is exposed to possible problems with the operating system calls, the underlying hardware technology, communication subsystems, and the physical network layer. Problems such as delays, congestion, physical failures, timing, and protocols affect link reliabilities. While their analysis is outside the scope of this paper, we incorporate their estimated effects in the SBRA algorithm.

Transition Probability "$\mathbf{PT_{ij}}$". PT_{ij} represents the probability of transition from C_i to C_j. It is calculated from the number of interactions between two components across the analysis scenarios as follows:

$$PT_{ij} = \sum_{k=1}^{|S|} PS_k * \left(\frac{|Interact(C_i, C_j)|}{|Interact(C_i, C_l)|_{l=1,\ldots,|N|}} \right)_{C_i, C_j, C_l \text{ in } S_k} \tag{3}$$

where

PS_k is the probability of execution of scenario S_k, $|N|$ is the number of components, and

$|Interact(C_i, C_j)|$ is the number of times C_i interacts with C_j in scenario S_k.

The sum of transition probabilities from any component should be unity.

Transition probabilities depend on the scenarios and the scenario profile. While uncertainties in the scenario profile affect the accuracy of PT_{ij} estimates [26], the proposed SBRA can be used to study the sensitivity of overall system reliability to changes in transition probabilities.

4) Data Sources: To calculate the above model parameters, the following data sources are required:

- A set of scenarios which capture interactions between components in the system. UML interaction diagram representation is adequate, if enhanced with timeline annotations.
- An initial scenario profile as estimated by the domain analyst. The analysis algorithm will be used to analyze the effect of uncertainties in the initial profile on the system reliability.
- Initial estimates for the component and transition reliabilities, or an SRGM model for each. The analysis algorithm is used to analyze the effect of uncertainties in these estimates.

B. CDG Construction

The following steps outline the process of constructing CDG graphs:

```
Procedure SBRA
Parameters
        consumes  CDG, AE_appl,
        produces  R_appl
Initialization:
        R_appl = 0, Time = 0, R_temp = 1
Algorithm
push tuple <C_1, RC_1, EC_1 >, Time, R_temp
while Stack not EMPTY do
        pop < C_i, RC_i , EC_i >, Time, R_temp
        if Time > AE_appl  or C_i = t; (terminating node)
                R_appl += R_temp      ;(an OR path)
        else
        ∀ < C_j ,RC_j , EC_j > ∈ children(C_i)
                push (<C_j, RC_j ,EC_j>, Time += EC_i ,
      R_temp = R_temp*RC_i*RT_ij *PT_ij )
          end
end while
```

Fig. 2. The SBRA algorithm.

- Using the application analysis scenarios, estimate the probability of execution of each scenario (PS_k) by estimating the frequency of execution of each scenario relative to all other scenarios.
- Estimate the reliability of components (RC_i) and interfaces (RT_{ij}), or decide on likely values for these reliabilities (see Sections IV-A-2 and IV-A-3).
- Calculate the average execution time for an application run (AE_{appl}) using the average execution time of a scenario, the scenario probability, and (1) (Section IV-A-1).
- For each scenario, calculate the execution time of each component (from the timeline of the sequence diagram), and the transition probability from one component to another.
- Calculate the average execution time of each component (EC_i) using the execution time of a component in each scenario, the probability of a scenario, and (2) (Section IV-A-2).
- Calculate the transition probability (PT_{ij}) for all scenarios using the probability of a scenario, the transition probabilities between components in each scenario, and (3) (Section IV-A-3).
- Construct the CDG according to the definitions in Section III.

C. Reliability Analysis Algorithm

After constructing the CDG, we can analyze the reliability of the application as the function of the reliability of components and transitions using the algorithm shown in Fig. 2

The algorithm expands all branches of the CDG starting from the start node. The breadth expansions of the tree represent logical "OR" paths, and are hence treated as the summation of reliabilities weighted by the transition probability along each path. The depth of each path represents the sequential execution of components, the logical "AND", and is hence treated as the multiplication of reliabilities. The "AND" paths take into consideration the interface and link reliabilities $(\mathrm{RT_{ij}})$ between components. The depth expansion of a path terminates when the execution time of that path sums up to the average execution time of the application, or when the next node is a terminating node.

Due to the probabilistic nature of the dependency graph, when calculating the reliability of an application, the SBRA algorithm may loop between two or more components. However, these loops never lead to a deadlock by virtue of using the average execution time of the application to terminate the depth traversal of the graph. Therefore, deadlocks are not possible when executing the algorithm, and a termination of the algorithm execution is evident.

The reliability of an "AND" path is neither too pessimistic nor too optimistic. The path is either truncated with a termination node (a natural end of an application execution), or with an execution time limit, which is the average execution time of a scenario. The algorithm assumes that each component is wrapped such that failures in one component do not propagate to another component in the "AND" path [10].

V. Example 1: Waiting Queues Simulator

We illustrate the applicability of the proposed technique to a simple component-based application for simulating the behavior of waiting queues in which we deal with customers lining up at checkout counters at supermarkets, a self-serve car wash, etc. The application is built by composing software components developed as a part of an educational experiment in software reuse [1]. The domain was defined by a set of software components, a generic architecture for communicating components, and a set of possible execution scenarios. We limit our discussion here to a specific application, the checkout counter.

A. The Architecture

Fig. 3 describes the architecture of the application using the UML package diagram [20]. The architecture of the application is centered around a dynamic event list as the communication vehicle of events. In addition to the *EventList* component, the primary components in the architecture are *ArrivalGenerator*, a *QueuingFacility*, a *ServiceFacility*, a *Measurement* recorder, and a *ScheduleManager*. The analysis identified a set of six events which depict all scenarios in the execution of the application.

- Events are maintained in the *EventList*, and are sorted by arrival time. Each event triggers a specific execution scenario.
- *ArrivalGenerator* uses a distribution function to generate the next customer's arrival time. We used a random number generator with a uniform distribution of arrival times.
- *QueuingFacility* consists of a set of queue categories, where each queue category contains one or more queues. Events which indicate an action for a queue category or a queue are delegated to the queue facility, which, in turn, delegates the action to the appropriate queue category. In the case study, we used two categories for checkout counters, *Normal* and *Express*, and one queue for each category.
- The *ServiceFacility* component consists of a set of server categories, where each server category contains one or

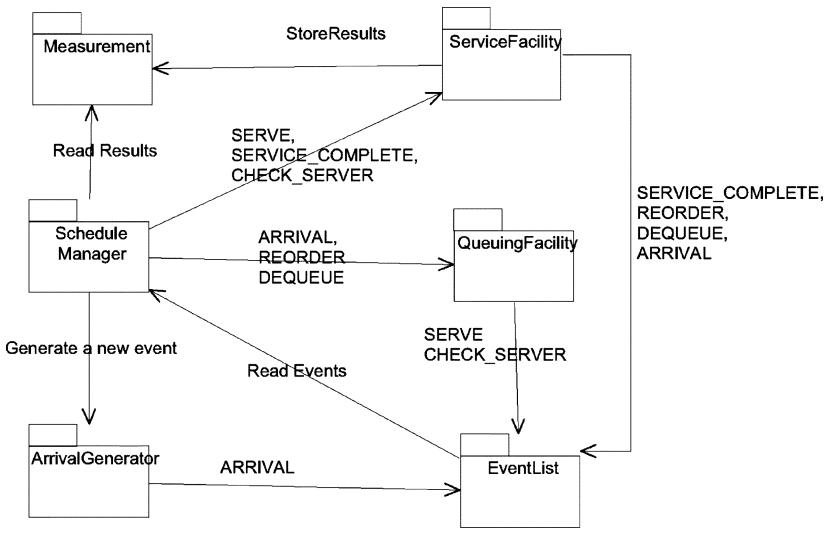

Fig. 3. The application architecture.

more servers. Events which indicate an action for a service category or a server are delegated to the *ServiceFacility*, which delegates the action to the appropriate service category.

- The *Measurement* component updates statistical information when a customer object completes the specified number of service units.
- The *ScheduleManager* component serves as the main routine for the simulation. It repeatedly dispatches events from the event list, and delegates actions based on the event type.

B. Scenarios

The interaction between components in the application is analyzed using six execution scenarios, each triggered by an event. The following summarizes the analysis scenarios, shown in Fig. 4.

- The ARRIVAL scenario describes the sequence of actions taken by components to process the arrival of a new customer to the queuing facility.
- The REORDER scenario is executed when it is required to reorder the customer in the queue category, or when a queue is empty and could accept customers from a busy queue.
- The DEQUEUE scenario is executed when a server is ready to accommodate a customer from its waiting queue.
- The SERVE scenario is executed when a customer is de-queued and needs to be served.
- The SERVICE_COMPLETE scenario is executed whenever a customer finishes its service at a station.
- The CHECK_SERVER scenario is executed to check whether a server is available to serve a customer.

C. The Component Dependency Graph

In this particular example, some of the parameters estimated below are calculated from the actual implementation of the system because the source code for the example was available when SBRA was applied. This may not be true for other applications. Early in the lifecycle, actual values for the model parameters may not be available. In such a case, feedback from the actual system implementation may be used to validate the assumptions made in the analysis phase about system model parameters, such as execution times and scenario profile.

Calculating PS_k. Based on the execution profile of the application, the probabilities of execution of the six scenarios are listed in Table I. These probabilities are calculated based on the numerous simulation runs for the application, and averaging over the executions of each scenario.

Calculating RC_i *and* RT_{ij}. In this experiment, we will assume that the reliabilities of components and transitions are known. We will use these parameters to discuss the sensitivity of the application's reliability to the variations in the reliabilities of components and transitions.

Calculating AE_{appl}. The average execution time of the application is calculated using the average execution time of each scenario, and the probability of execution of a scenario. Using (1) (see Section IV-A-1), AE_{appl} is 23.5.

Calculating EC_i. The average execution time of each component is calculated using (2) (Section IV-A-2). Table II lists the average execution time of each component.

Calculating PT_{ij}. Using the analysis scenarios, the scenario probabilities (Table I) and (3) (Section IV-A-3), the transition probabilities PT_{ij} are calculated. Using the definitions of Section III, the CDG, shown in Fig. 5, was constructed.

D. Applying the SBRA Algorithm

We have implemented the SBRA algorithm defined in Section IV-C, and applied it to the CDG of the application in Fig. 5. The following describes the types of analyses we conducted.

A) Reliability of the application as a function of component reliability. Using the SBRA algorithm, we are able to investigate the variation in the reliability of the application as a function of the reliability of individual components. The graph in Fig. 6 shows the application reliability as the function of varying the reliability of one component, while the reliabilities of other components are fixed to 1.0, for the sake of comparison.

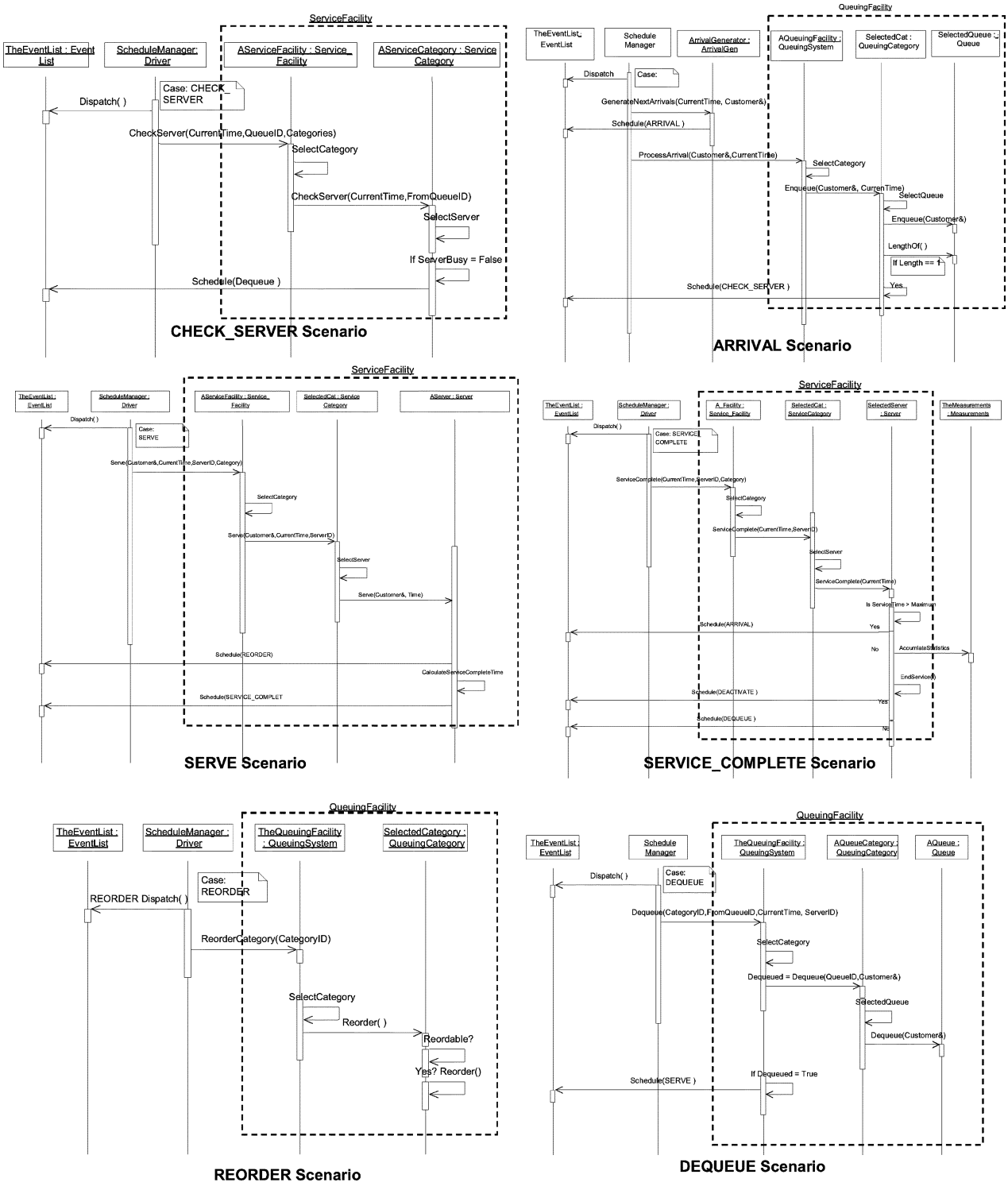

Fig. 4. Scenarios for the waiting queue simulator.

B) Reliability of the application as function of transition reliability. Using the SBRA algorithm, we are able to investigate the variation in the reliability of the application as a function of the reliability of transitions (interfaces and delivery channels) between components. Fig. 7 shows the reliability of the application as a function of the reliability of a transition (one at a time), while the reliabilities of other components and transitions are fixed (equal to 1 for the sake of comparison).

C) Reliability of the application as a function of the scenario profile. The change in the manner an application uses a domain component has a substantial effect on the sensitivity of the application reliability. Our model accounts for changes in component usage through the variations in scenario execution probabilities (PS_k in Section IV-A-1). For our case study, the usage of the components differs from one application to another, e.g., supermarket, immigration posts, car wash service, etc. For illus-

TABLE I
AVERAGE EXECUTION TIME OF SCENARIOS

Scenario Name	*Probability of a Scenario (PS_k)*	*Average Execution Time of a Scenario Time(S_k))*
ARRIVAL	0.145	41.43
SERVE	0.145	34.46
DEQUEUE	0.28	14.02
SERVICE_COMPLETE	0.145	31.93
CHECK_SERVER	0.14	20.83
REORDER	0.145	6.06

TABLE II
AVERAGE EXECUTION TIME OF EACH COMPONENT IN THE CASE STUDY

Component Name	*Average Execution Time of a component EC_I*
Generator	2
SimulatorDriver	5
ServiceFacility	7
QueuingFacility	5
EventList	14
Measurement	0.8

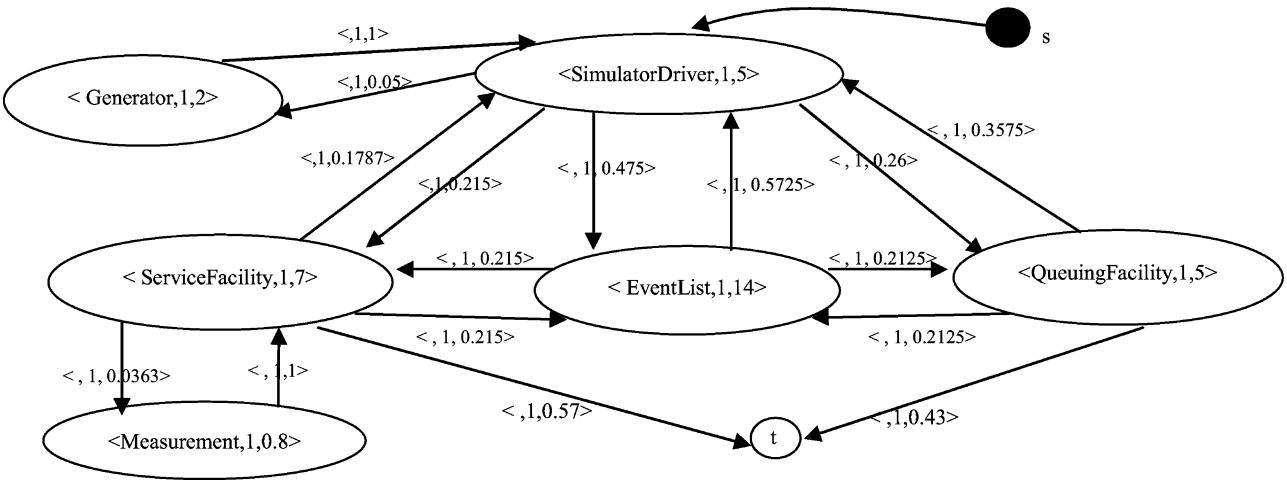

Fig. 5. CDG of the waiting queues simulator.

tration, we consider four cases. Varying the maximum requested service time, and the maximum limit on the period between customers inter-arrival time, generated these cases. The profiles for the four cases are shown in Table III.

We can analyze the reliability of the application as a function of component reliabilities with different usage profiles. We selected the three components *EventList*, *QueueFacility* and *ServiceFacility*. The ensuing application reliabilities are plotted in Fig. 8.

E. Results

- From Fig. 6, the application reliability varies significantly with the variation in the reliability of *SimulatorDriver* and *EventList*. As the reliability of these components decreases slightly, the system reliability decreases dramatically. This is due to the fact that these two components are at the core of the simulation application and, therefore, any faults in these components will affect the correct operation of the application.
- From Fig. 6, the reliability of the application does not vary significantly with the variation in the reliability of the *Measurement* component. This is due to the nature of that component (it records simulation results), being invoked a few times to record and retrieve statistics.
- From Fig. 7, transition reliabilities can significantly affect the reliability of the application. For example, the interface and/or link between the *SimulatorDriver* and *EventList* components can significantly deteriorate the reliability of the overall application if there are mismatches or errors in the data flow.
- From Fig. 8, the sensitivity of the application reliability to changes in the component reliabilities varies with the usage of components. For example, the application reliability becomes more sensitive to the reliability of the components *EventList* and *QueueFacility* for *Profile1* than

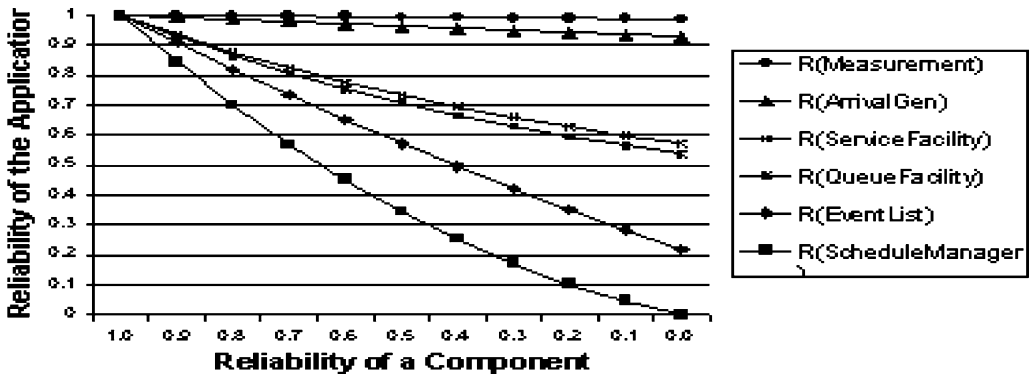

Fig. 6. Application reliability as function of component reliabilities (one at a time).

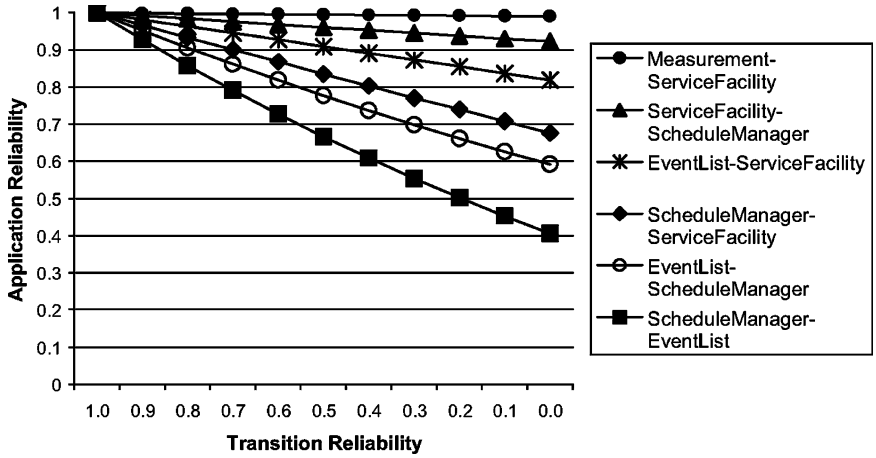

Fig. 7. Application reliability as function of transition reliabilities (one at a time).

TABLE III
SCENARIO PROBABILITIES FOR FOUR DIFFERENT PROFILES

	ARRIVAL	*SERVE*	*DEQUEUE*	*SERVICE_COMPLETE*	*CHECK_SERVER*	*REORDER*
Profile1	0.950	0.011	0.011	0.011	0.006	0.011
Profile2	0.704	0.073	0.075	0.072	0.003	0.073
Profile3	0.553	0.107	0.115	0.107	0.011	0.107
Profile4	0.145	0.145	0.28	0.145	0.14	0.145

in the case of *Profile2*, *Profile3*, and *Profile4* (Fig. 8(a), Fig. 8(b)). On the contrary, the application reliability is less sensitive to the reliability of the component *Service-Facility* for *Profile1* than for *Profiles 2*, *3*, and *4* (see Fig. 8(c)).

The above results are logically sound for the application under consideration. In a simulation environment, we expect that the event dispatcher is the reliability bottleneck because a failure in that component affects the system operation. Results from the experiment illustrate that our model and method automatically derives the same conclusion; i.e., failures in the event dispatcher would significantly affect the overall system reliability.

VI. EXTENSIONS FOR DISTRIBUTED SYSTEMS

Distributed systems often require many services available as course-grained or fine-grained design components. From this viewpoint, the development of many distributed systems is concerned with integrating components, and the resulting applications are component-based systems. The distributed nature of software systems imposes additional requirements on reliability analysis models and techniques. Due to their complexity, distributed systems are often hierarchical in nature. Each subsystem is further decomposed into components or other subsystems. Reliability analysis techniques should incorporate the concepts of hierarchy and subsystem decomposition, i.e., we should be able to assess the sensitivity of the system reliability as the function of components or subsystems, and assess the sensitivity of subsystem reliabilities as a function of the reliability of their descendents. Another requirement imposed on a reliability analysis technique for distributed systems is the ability to deal with physically distributed components. The distribution of components across the network makes the link or channel reliabilities even more critical factors in reliability analysis. Reliability analysis techniques for distributed component-based systems must account for link reliabilities. In this section, we extend the application of CDG models and the SBRA algorithm to analyze complex systems which are hierarchical and distributed in nature. The extension incorporates subsystem concepts and hierarchical CDG.

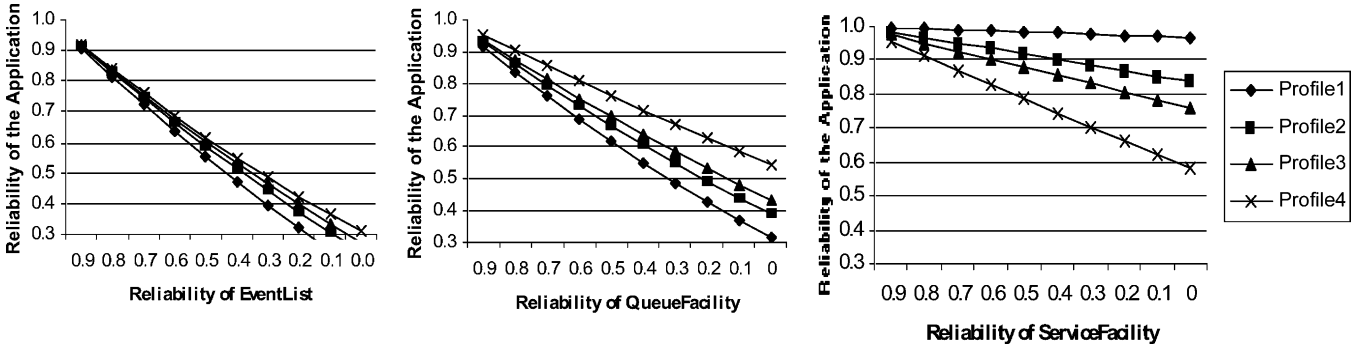

Fig. 8. Application reliability as function of scenario profiles.

A. Defining CDG for Distributed Systems

Reliability analysis based on flattening a hierarchically designed complex system would be hard to conduct because of the large number of components involved. Thus, it is easier to analyze the reliability of the system by incorporating the concepts of layers or subsystems. A layer is a logical concept. It provides specific services to the rest of the system [12]. A subsystem indicates an "*is composed of*" relationship. In the following, we use the term subsystem to refer to a composition of components. A layer is also considered a subsystem if it is composed of a set of components.

For our model, a system is defined by a set of subsystems, components, and transitions. A component dependency graph is used as a probabilistic reliability model. The CDG definition is similar to that of Section III, with different interpretation of nodes.

Node "n". A node n models a subsystem SS_m, or a component C_i, $n = \langle C_i \rangle | \langle SS_m \rangle$, where C_i is the i^{th} component, SS_m is the m^{th} subsystem modeled by a separate CDG called SS_CDG, defined later.

Subsystem "SS_m". A subsystem is a structural viewpoint of a set of components bound together to interact and deliver certain functionality. A subsystem is treated as a component with composite nature. It consists of other components, thus preserving the component approach to reliability analysis. For the system level analysis, a subsystem SS_m is defined by the tuple $\langle NSS_m, RSS_m, ESS_m \rangle$, where NSS_m is the subsystem's name, RSS_m is the subsystem's reliability, and ESS_m is its execution time. A subsystem is defined by a separate SS_CDG.

Subsystem Reliability "RSS_m". RSS_m is the probability that the subsystem SS_m provides failure free services for other subsystems or components upon an invocation. RSS_m is the function of the reliability of its individual components, and their link reliabilities.

Execution Time of a Subsystem "ESS_m". ESS_m is the execution time of a subsystem SS_m. The execution time of a subsystem varies according to the type of service it provides, the scenario which activates this service, and the average execution times of its constituting components.

A Subsystem Component Dependency Graph (SS_CDG). An SS_CDG is a component dependency graph which represents the control flow dependency of the subsystem's components. It has all the parameters defined for the system CDG and, hence, CDG become hierarchical in nature. Additionally, SS_CDG has a terminate-and-return node which returns execution to the parent CDG graph. To summarize, $SS\_CDG = \langle CDG, t_r \rangle$ where $CDG = \langle N, E, s, t \rangle$, and $\mathrm{t_r}$ is the termination node for the subsystem which indicates return to the calling system's CDG.

Transition Reliability "RT_{ij}". RT_{ij} is the probability that the data and/or control sequence sent from component (subsystem) C_i (SS_i) to component (subsystem) C_j (SS_j) is delivered error-free. This probability includes possible interface errors, and possible channel delivery errors.

Transition Probability "PT_{ij}". PT_{ij} is the conditional probability that C_j (SS_j) executes next, given that C_i (SS_i) is currently executing.

Thus, a CDG is defined as follows: $CDG = \langle N, E, s, t \rangle$

$N = \{n\},\ E = \{e\},\ s$ and t are the start and termination nodes, respectively

$n = \langle C_i \rangle | \langle SS_m \rangle$

$C_i = \langle CN_i, RC_i, EC_i \rangle$

$SS_m = \langle SSN_m, RSS_m, ESS_m \rangle,\ SS_m\ has\ an\ SS\_CDG$

$SS\_CDG = \langle CDG, t_r \rangle$

$\mathrm{e} = \langle T_{ij}, RT_{ij}, PT_{ij} \rangle$

$T_{ij} = \langle n_i, n_j \rangle$

An example of a hierarchical CDG is shown in Fig. 9.

B. Constructing a CDG for Distributed Systems

First, we evaluate the parameters required to construct the CDG graphs for the system and subsystems, then we construct their component dependency graphs. Some of these parameters have been defined in Section IV. We describe them here with modifications suitable for the distributed system pradigm.

1) Scenario-Related Parameters: **A Scenario** "$\mathbf{S_k}$". S_k has the same definition as in Section IV-A-1. However, for hierarchical systems, a scenario could have a local or global scope. By global scope we mean interaction between components and subsystems at the system level. Local scope means interaction between components within a subsystem. Within a global scenario, one or more local scenarios could be triggered. We use the same symbol S_k for local and global scenarios because the construction of CDG for systems and subsystems is similar.

Probability of a Scenario "$\mathbf{PS_k}$". $\mathrm{PS_k}$ is the probability of execution of scenario k with respect to all other scenarios. For each subsystem, we define a scenario profile, a set of scenario execution probabilities for that particular subsystem.

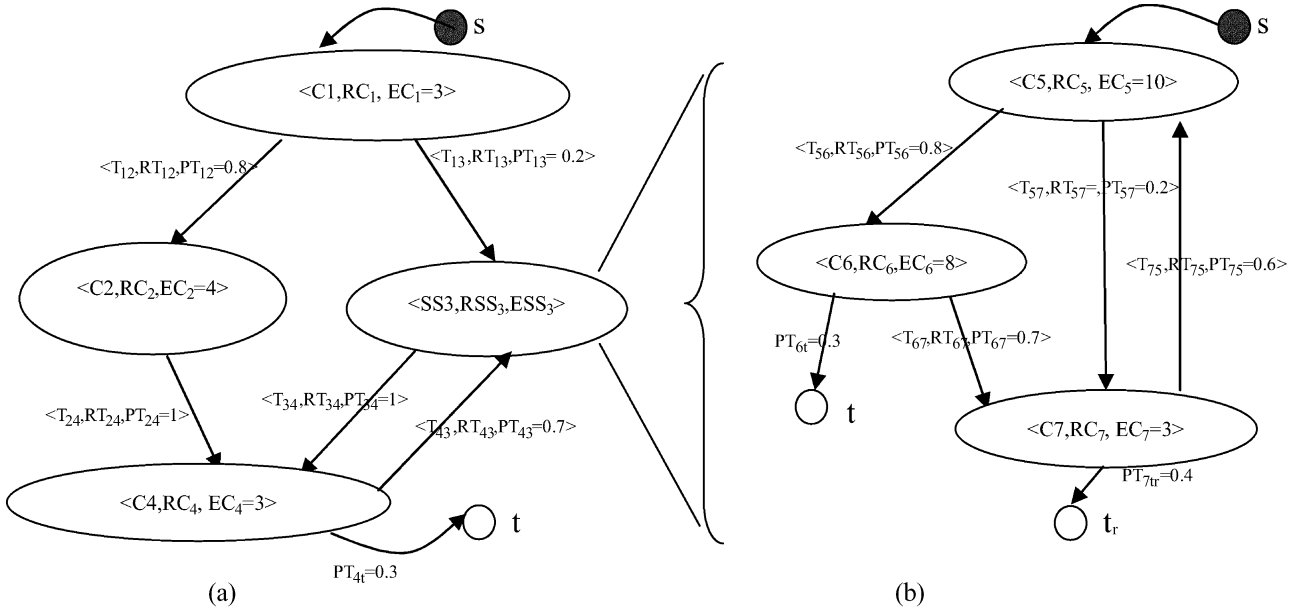

Fig. 9. (a) A system CDG, (b) an SS_CDG for SS3.

Average Execution Time of a Global Scenario "$\mathbf{AE_{appl}}$". AE_{appl} is defined by (1), Section IV-A-1, where S_k is a global scenario.

2) Component-Related Parameters: Component-related parameters, EC_i and RC_i, are the same as defined in Section IV-A-2.

3) Transition Related Parameters: **Transition Probability** "$\mathbf{PT_{ij}}$". PT_{ij} is the probability of utilizing the transition from component/subsystem i to component/subsystem j. It is estimated from the number of interactions between components and subsystems in the analysis scenarios, as defined in Section IV-A-3. A transition probability PT_{ij} in the *system* graph is calculated from the set of *global* scenarios of interaction between elements of the system graph. A transition probability PT_{ij} in a *subsystem* graph is calculated using the set of *local* scenarios.

Transition Reliability "$\mathbf{RT_{ij}}$". In distributed systems, link reliabilities become a special concern. In addition to component interface mismatches, component distribution across a network adds more factors affecting link reliabilities. A message exchanged between components in a distributed environment is exposed to possible problems in the operating system calls, the underlying hardware technology, communication subsystems, and the physical network itself.

4) Subsystem-Related Parameters: **Execution Time of a Subsystem** "$\mathbf{ESS_m}$". $\mathrm{ESS_m}$ depends on its local scenarios, and the execution times of its constituent components. Depending on the nature of the subsystem and the services it provides, several approaches can be used to estimate ESS_m. For instance, for subsystems where the standard deviation of the execution time of its local scenarios is low, we can use the following point estimate for ESS_m:

$$ESS_{\mathrm{m}} = \sum_{\mathrm{k}=1}^{|\mathrm{S}|} \mathrm{PS_k} \times \mathrm{Time(S_k)} \tag{4}$$

where S_k is a local scenario for SS_m

For subsystems where the standard deviation of the execution time of scenarios is high, a probability distribution for ESS_m can be used. These often occur for subsystems where physical network elements are encapsulated, and there is a high level of uncertainty about network delays. In such cases, the CDG graph traversal algorithm, discussed later, consults a procedure to estimate the subsystem execution time, based on the probability distribution ESS_m each time the subsystem is traversed.

Subsystem Reliability "$\mathbf{RSS_m}$". RSS_m is a function of the reliabilities of subsystem's components, their link reliabilities, and local scenarios. The algorithm in Section VI-C can be used to give an estimate for the reliability of the subsystem. However, our objective is to study the sensitivity of the system reliability as a function of component and subsystem reliabilities, and to study the sensitivity of a subsystem reliability as function of the reliability of its components and link reliabilities. From system analysis, we can identify critical subsystems; while from subsystem analysis, we can identify critical components. An SS_CDG can also be traversed for point estimates of the subsystem reliability.

C. Extending Reliability Analysis Algorithm

We extend the graph traversal algorithm, defined in Section IV-C, to incorporate the traversal of subsystems.

By analyzing the algorithm in Fig. 10, we note that:

- n is used as a node in the CDG graph, representing a component or a subsystem.
- $GraphChildren(n)$ returns all the successors nodes of a node n.
- In case the algorithm is used to analyze the sensitivity of a subsystem to reliabilities of its children, the algorithm requires a point estimate of the subsystem's average execution time instead of AE_{appl}.
- R_{out} is the reliability of the system or subsystem based on the inputs to the algorithm (the CDG and AE_{appl} for the system, and SS_CDG and ESS_m for a subsystem).

```
Algorithm Scenario-Based Reliability Analysis for hierarchical systems
Parameters
        consumes CDG, AE_appl,
        produces  R_out
Initialization:
        R_out = 0, Time = 0, R_temp = 1

Algorithm
        push s, Time, R_temp          ; (s is the start node)
        while Stack not EMPTY do
                pop < n_i, Rn_i , En_i >, Time, R_temp
                if Time > AE_appl  or n = t    ; (t is a termination node)
                        R_out += R_temp        ; (an OR path)
                else
                        if Type(n_i) is Component
                                ∀ < n_j ,Rn_j , En_j > ∈ GraphChildren(n_i)
                                push (<n_j, Rn_j ,En_j>, Time += En_i ,  R_temp = R_temp*Rn_i*Rn_ij *Pn_ij )
                        else                          ; it is a subsystem
                                Time += EstimateSubsystemTime(n_i)
                                RSS = EstimateSubsystemReliability(n_i)
                                ∀ < n_j ,Rn_j , En_j > ∈ GraphChildren(n_i)
                                push (<n_j, Rn_j ,En_j>, Time,  R_temp = R_temp*RSS*Rn_ij *Pn_ij )
                        end
                end
        end while
```

Fig. 10. The extended CDG traversal algorithm.

- $EstimateSubsystemTime(n_i)$ returns the estimate of the execution time of the subsystem. For subsystems with point estimates, this value is a constant. For subsystems with execution-time probability distribution function, a procedure is used to get an estimate from the distribution.
- $EstimateSubsystemReliability(n_i)$ is used to return an estimate of the reliability of the subsystem. This procedure can use the same algorithm to traverse the SS_CDG of the subsystem recursively.

The proposed algorithm extends the algorithm in Section IV-C to allow traversal of hierarchical CDG, where a node can be a subsystem defined by an SS_CDG. For a subsystem node, the algorithm estimates its execution time and reliability, and can traverse the subsystem graph. The algorithm can also be used to analyze the reliability for subsystems as a function of its constituting components, and transition reliabilities.

VII. Example 2: A Distributed Medical Informatics System

We use the example of a distributed medical informatics system to illustrate the proposed reliability analysis approach. The *Digital Imaging and Communication in Medicine* standard (DICOM) [3] defines the communication messages and application services between distributed medical systems. It is not the objective of this section to assess the reliability of applications developed in compliance with DICOM. Instead, we use few scenarios to illustrate how CDG, and the reliability analysis algorithm could be applied. Fig. 11(a) illustrates a high-level structure of the system as a set of client/server application entities (*AE Client* and *AE Server* subsystems) connected via a network (*Network* subsystem). DICOM specifies the transport and presentation layer for a network protocol as DICOM Upper Layer (*DICOM UL Client* and *DICOM UL Server* subsystems). Fig. 11(b) shows a simplified version of the next level decomposition of subsystem *AE Client*. *AE Client* consists of another subsystem which manages *Information Object Definitions (IOD)*, such as patients, visits, studies, image, overlays, etc. (defined in Part 3 of the DICOM standard [3]). *DICOM Message Service Elements (DIMSE)* subsystem defines the set of possible message services which a client could use (defined in Part 7,8 of [3]). The *AE Client Manager* subsystem is the interface for the application which passes requests to *Service Class User (SCU)*. An *SCU* is the core of the client's side which manages the creation of *IOD*, and *DIMSE* Primitive Messages according to the required service class. The services which an *SCU* can request are defined as a pair of *IOD* and *DIMSE* (defined in Part 4 of [3]). The *DIMSE Protocol Machine* subsystem is responsible for sending and receiving messages from the lower-level layers (*DICOM UL Client* subsystem). The *Message Factory* subsystem is responsible for translating internal messages (primitives) to external messages, to be sent to the server, and vice versa.

We identified three simplified scenarios to construct the system level *CDG*. The three scenarios are *Association Establishment* using *ASSOCIATE* messages, *Connection Verification* service using *C_ECHO* messages, and *Retrieval service* using *N_GET* messages. DICOM specifies many more services and messages; these three provide an illustration.

The *Association Establishment* scenario is executed whenever a link to the server is required. Once a link is established, the *Verification scenario* is periodically executed. We assume that the execution probabilities of the three scenarios are 0.25, 0.5, and 0.25, respectively. Using these probabilities, the CDG parameters identified in Section VI-A, and the sample application scenarios, we construct a CDG for the system, as shown in Fig. 12.

To illustrate the analysis of the sensitivity of the system reliability to subsystem, component or link reliabilities, we apply the extended SBRA algorithm (Section VI-C) to the system CDG from Fig. 12. We assume perfect subsystem transitions, and 40%

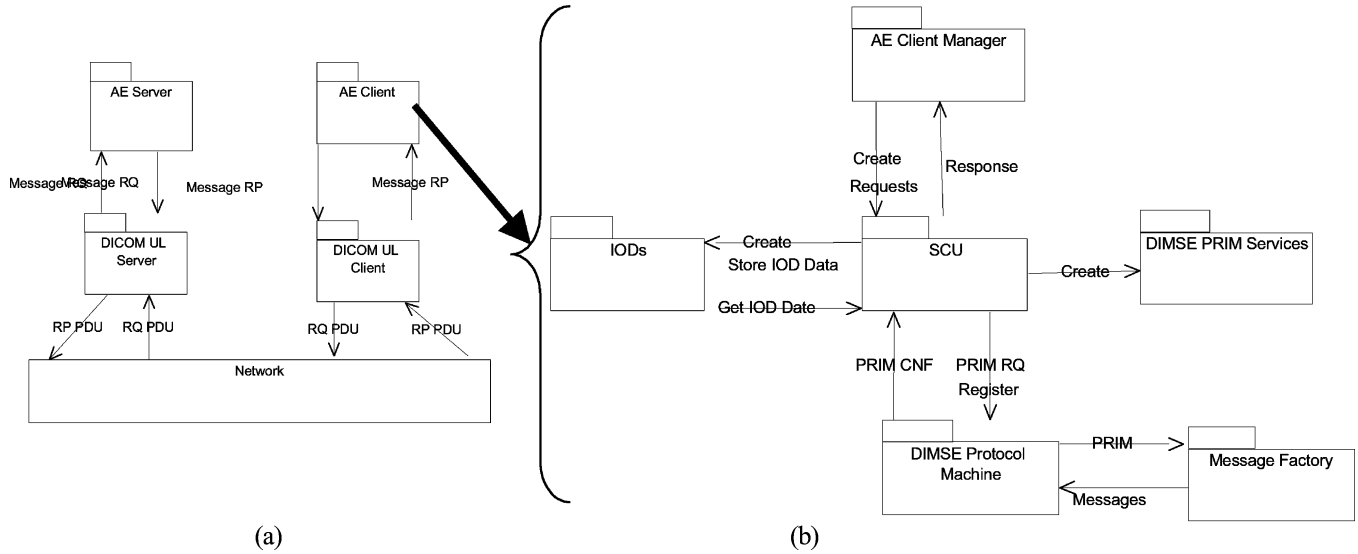

Fig. 11. (a) A DICOM system structure (b) structure of DICOM subsystem AE Client.

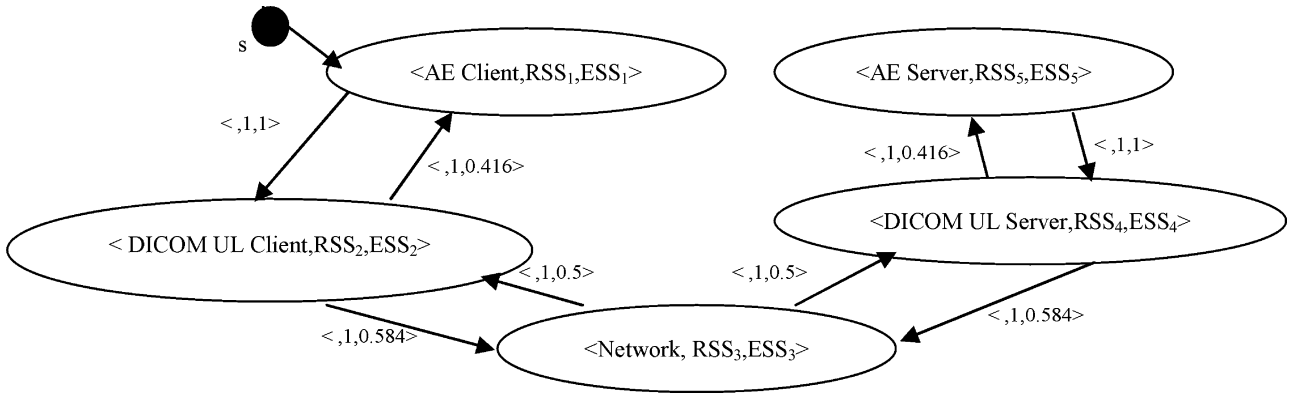

Fig. 12. CDG for the medical informatics system.

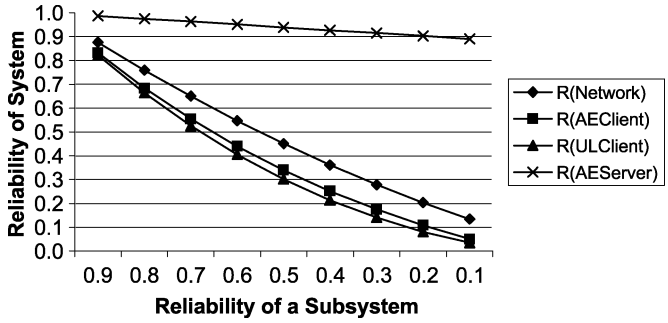

Fig. 13. Sensitivity of system reliability to component reliabilities.

of the execution time being spent in the network subsystem, 10% for each of the *DICOM UL* subsystems, and 20% for each of the *AE* subsystems. Fig. 13 shows sample results.

From Fig. 13, we recognize that the reliability of the system is severely affected by the degradation in the reliability of the *Network* and *UL Client* subsystems. Due to the selected set of scenarios, the *Network* subsystem is heavily utilized, while the *Server* subsystem is lightly involved. From a more detailed set of scenarios, we would be able to more precisely identify subsystems which critically affect the reliability of the system.

For each subsystem, we can construct an SS_CDG. For example, for the *AE Client* subsystem, shown in Fig. 11(b), and using just one scenario for simplicity, we construct the SS_CDG for *AE Client* subsystem, as shown in Fig. 14.

To illustrate the type of analysis we can conduct from an SS_CDG, we apply the extended SBRA algorithm (of Section VI-C) to the SS_CDG of the *AE Client* subsystem (Fig. 14). The goal is to analyze the subsystem reliability as a function of the reliabilities of various components. Here, we assume perfect component transitions. Fig. 15 shows the sensitivity of the application reliability to component reliabilities.

From Fig. 15, we recognize that the reliability of the *AE Client* subsystem is severely affected by the degradation in the reliability of *Protocol Machine* and *SCU* components, and peripherally influenced by the reliabilities of *DIMSE or IOD* components.

Another type of analysis we can conduct using the SBRA algorithm and CDG is the sensitivity of the system or subsystem reliabilities to the transition reliabilities. For instance, using the SS_CDG in Fig. 14, and assuming perfectly reliable components for the *AE Client* subsystem, we can analyze subsystem reliability as a function of transition reliabilities between its components (Fig. 16).

From Fig. 16, we recognize that the *AE Client* subsystem reliability is more sensitive to the transition reliability between the *Protocol Machine* and the *Message Factory*, and less sensitive to transition reliability between the *SCU* and *DIMSE* components.

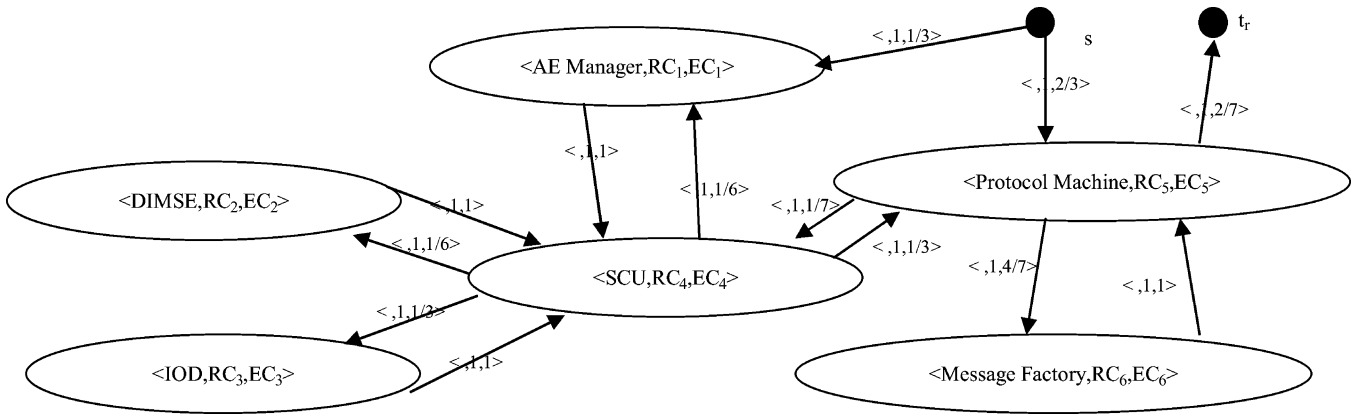

Fig. 14. SS_CDG for the AE Client subsystem.

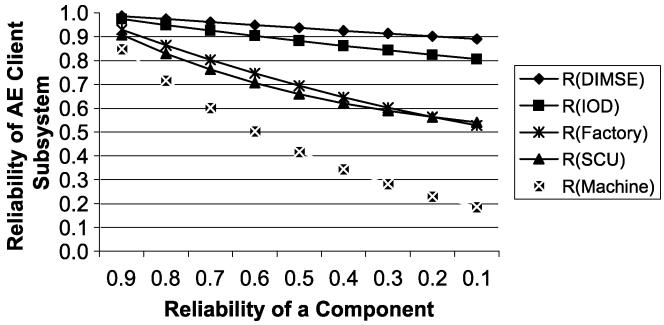

Fig. 15. Sensitivity of the AE Client subsystem reliability to component reliabilities.

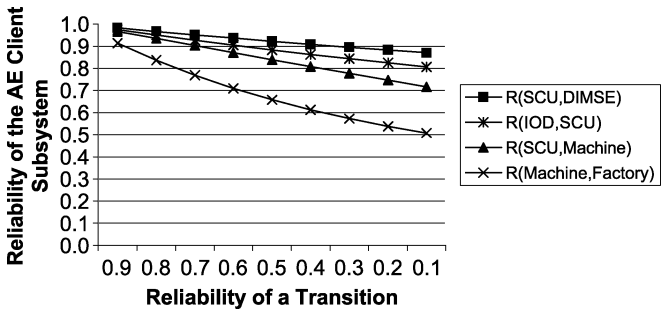

Fig. 16. Sensitivity of the AE Client subsystem reliability to transition reliabilities.

Similarly, we can construct SS_CDG for each subsystem based on a complete set of scenarios to further analyze the sensitivity of system and subsystem reliabilities to the reliability of their constituents.

VIII. Discussion

This paper defines a new approach to analyze the reliability of component-based, and distributed software systems. The proposed SBRA algorithm is suitable for systems whose analysis is based on usage scenarios. The execution profile of these scenarios is assumed to be available. *Component Dependency Graphs* serve as reliability models for component-based systems. They are extended to suit complex distributed systems by incorporating the effects of physical component distribution, and the hierarchy of subsystems. The proposed SBRA is a stack-based algorithm suitable for analyzing the sensitivity of system level reliability as a function of component, link, and subsystem reliabilities. The same algorithm is applicable to analyze the sensitivity of subsystem reliability as a function of component reliabilities, and link reliabilities. Algorithm application results in identifying critical components, subsystems, and links which require increased attention in testing, verification, and validation. The algorithm can also be used to impose reliability constraints on acquired off-the-shelf components. Two case studies, *Waiting queues simulator* and *Distributed medical informatics system*, illustrate the application of the approach. SBRA is based on scenarios, execution times, and component usage patterns. Using the proposed CDG model, we incorporate the effect of frequently executed components, interfaces, and links; hence, we can direct more testing and development effort to critical artifacts. The approach is applicable early in the development phase because it is based on scenarios involving reusable components, which could be objects or patterns. Using scenarios to derive CDG allows us to incorporate uncertainties in the component's usage patterns in reliability analysis. Applications using the same components, but with different usage scenarios, are likely to exhibit different sensitivity to the component reliabilities. Our reliability models can be used as the basis for other types of analysis. For instance, in [33], Yacoub and Ammar describe a reliability-based risk analysis approach which builds on top of the CDG models defined in this paper. Reliability estimates, as well as impact and severity factors, are used for risk analysis.

The proposed approach has some limitations. Currently, the algorithm does not consider the overall application reliability growth as a function of time. Further, some scenarios may be more critical than others, but they are seldom executed. The model could be expanded to consider the impact of the failure severity of critical scenarios in the analysis. The algorithm does not consider failure dependencies among components either. Future work could investigate how to treat COTS components, and their specific requirements, in CDG, and in the SBRA algorithm. Evaluation and empirical assessment of the methodology is also a topic of future investigation. Currently the CDG model does not take into consideration parallel execution of components. More research effort is required to extend the model, and the algorithm, for reliability analysis of parallel systems.

REFERENCES

[1] E. Addy *et al.*, "A controlled experiment in software reuse," *Software Qual. J.*, vol. 8, no. 3, pp. 169–195, Nov. 1999.

[2] M. Delamaro *et al.*, "Integration testing using interface mutations," in *Proc. 7th Int. Symp. Software Reliability Engineering (ISSRE'96)*, White Plains, New York, Oct. 30–Nov. 2, 1996, pp. 112–121.

[3] National Electrical Manufacturers Association (NEMA), *Digital Imaging and Communication in Medicine (DICOM) Standard*. Washington, DC, USA: NEMA Office of Publication.

[4] W. Everett, "Software component reliability analysis," in *Proc. IEEE Symp. Application-Specific Systems and Software Engineering & Technology (ASSET'99)*, Richardson, Texas, Mar. 24–27, 1999, pp. 204–211.

[5] S. Gokhale and K. Trivedi, "Dependency characterization in path-based approaches to architecture-based software reliability prediction," in *Proc. IEEE Workshop on Application Specific Software Engineering and Technology (ASSET'98)*, Richardson, Texas, Mar. 26–28, 1998, pp. 86–89.

[6] S. Gokhale *et al.*, "Reliability simulation of component-based software systems," in *Proc. 9th Int. Symp. Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany, Nov. 1998, pp. 192–201.

[7] M. Heimdahl *et al.*, "Specification and analysis of intercomponent communication," *IEEE Comput.*, pp. 47–54, Apr. 1998.

[8] J. Horgan and A. Mathur, "Software testing and reliability," in *Handbook of Software Reliability Engineering*, M. R. Lye, Ed. New York: McGraw-Hill, 1996, ch. 13, pp. 531–566.

[9] International Telecommunication Union (ITU-T) Recommendation Z.120 (10/96) for Message Sequence Charts (MSC). [Online]http://www.itu.int/itudoc/itu-t/rec/z/z120.html

[10] T. Khoshgoftaar *et al.*, "Identifying modules which do not propagate errors," in *Proc. IEEE Symp. Application-Specific Systems and Software Engineering & Technology (ASSET'99)*, Richardson, Texas, Mar. 24–27, 1999, pp. 185–193.

[11] S. Krishnamurthy and A. P. Mathur, "On the estimation of reliability of a software system using reliabilities of its components," in *Proc. 8th Int. Symp. Software Reliability Engineering (ISSRE'97)*, Albuquerque, New Mexico, Nov. 1997, pp. 146–155.

[12] J. Laprie and K. Kanoun, "Software reliability and system reliability," in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. New York: McGraw-Hill, 1996, ch. 2, pp. 27–69.

[13] D. Mason and D. Woit, "Problems with software reliability composition," in *9th Int. Symp. Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany, Nov. 1998, Fast Abstracts, pp. 41–42.

[14] B. Meyer *et al.*, "Building trusted components to the industry," *IEEE Comput.*, pp. 104–105, May 1998.

[15] J. Musa *et al.*, *Software Reliability: Measurement, Prediction and Application*. New York: McGraw Hill, 1987.

[16] J. Musa *et al.*, "The operational profile," in *Handbook of Software Reliability Engineering*, M. R. Lyu *et al.*, Ed. New York: McGraw-Hill, 1996, ch. 5, pp. 167–216.

[17] R. Pressman, *Software Engineering: A Practitioner's Approach*, 4th ed: McGraw Hill, Inc., 1997.

[18] S. Sanyal *et al.*, "Framework of a software reliability engineering tool," in *Proc. IEEE High-Assurance Systems Engineering Workshop (HASE'97)*, Washington, DC, 1997, pp. 114–119.

[19] V. Shah and S. Bhattacharya, "Fault propagation analysis based variable length checkpoint placement for fault tolerant parallel and distributed system," in *Proc. 21st Annu. Int. Computer Software and Applications Conf. (COMPSAC'97)*, Bethesda, Maryland, Aug. 1997.

[20] Unified Modeling Language Resource Center. Rational Rose Inc.. [Online]http://www.rational.com/uml/documentation.html

[21] J. Voas *et al.*, "Tolerant software interfaces: can COTS-based systems be trusted without them?," in *Proc. 15th Int. Conf. Computer Safety, Reliability, and Security (SAFECOMP'96)*, Vienna, Oct. 1996, pp. 126–135.

[22] J. Voas, "Error propagation analysis for COTS systems," *IEEE Comput. Control Eng. J.*, vol. 8, no. 6, pp. 269–272, Dec. 1997.

[23] ——, "Certifying off-the-shelf software components," *IEEE Comput.*, pp. 53–59, June 1998.

[24] K. Weidenhaupt *et al.*, "Scenarios in system development: current practice," *IEEE Software*, pp. 34–45, Mar./Apr. 1998.

[25] D. Woit and D. Mason, "Component independence for software system reliability," in *2nd Int. Software Quality Week Europe (QWE'98)*, Brussels, Belgium, Nov. 9–13, 1998.

[26] A. Wesslen *et al.*, "Assessing the sensitivity of usage profile changes in test planning," in *Proc. 11th Int. Symp. Software Reliability Engineering (ISSRE 2000)*, San Jose, California, Oct. 2000, pp. 317–326.

[27] N. Scheidewind, "Software reliability engineering for client-server systems," in *Proc. 7th Int. Symp. Software Reliability Engineering (ISSRE 1996)*, White Plains, New York, Oct. 1996, pp. 226–235.

[28] A. Whittaker and M. Thomason, "A Markov chain model for statistical software testing," *IEEE Trans. Software Eng.*, vol. 20, no. 10, pp. 812–824, Oct. 1994.

[29] A. Whittaker, K. Rekab, and M. Thomason, "A Markov chain model for predicting the reliability of multi-build software," *J. Inform. Software Technol.*, vol. 42, no. 12, pp. 889–894, Sept. 2000.

[30] J. Poore *et al.*, "Planning and certifying software system reliability," *IEEE Software*, pp. 88–99, Jan. 1993.

[31] T. Firley *et al.*, "Timed sequence diagrams and tool-based analysis—a case study," in *The 2nd Int. Conf. The Unified Modeling Language, Beyond the Standard (UML'99)*, vol. 1723, Lecture Notes in Computer Science, Springer-Verlag, Oct. 1999, pp. 645–660.

[32] K. Goseva-Popstojanova and K. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation, an International Journal*, vol. 45, pp. 179–204, 2001.

[33] S. Yacoub and H. Ammar, "A methodology for architectural-level risk analysis," *IEEE Trans. Software Eng.*, vol. 28, pp. 529–547, June 2002.

[34] V. Cortellessa *et al.*, "Early reliability assessment of UML based software models," in *3rd Int. Workshop on Software Performance*, Rome, Italy, July 2002, pp. 302–309.

[35] H. Singh *et al.*, "A Bayesian approach to reliability prediction and assessment of component based systems," in *12th IEEE Int. Symp. Software Reliability Engineering (ISSRE '01)*, Hong Kong, Nov. 2001, pp. 12–21.

[36] H. Jin and P. Santhanam, "An approach to higher reliability using software components," in *12th IEEE Int. Symp. Software Reliability Engineering (ISSRE '01)*, Hong Kong, Nov. 2001, pp. 1–11.

[37] D. Hamlet *et al.*, "Theory of software reliability based on components," in *23rd Int. Conf. Software Engineering*, Toronto, Canada, May 2001.

Sherif Yacoub is a member of the Research Staff at Hewlett-Packard Laboratories, Palo Alto, California. Dr. Yacoub earned a Ph.D. in Computer Engineering from West Virginia University in 1999, where he worked as a Research Assistant Professor until July 2000. He received an M.Sc. in Electrical Engineering in 1997 and a B.Sc. in Computer Engineering in 1994 from Cairo University, Egypt, where he also lectured in software engineering. Dr. Yacoub is a member of the IEEE and the ACM professional organizations. His research interests include software design, design patterns, software reuse at the architectural level, content understanding, interactive voice response systems, design quality, risk analysis, and software reliability.

Bojan Cukic is an Associate Professor at the Lane Department of Computer Science and Electrical Engineering, West Virginia University. He received the Dipl. Ing. degree from the University of Ljubljana, Slovenia, and M.S. and Ph.D. in Computer Science from the University of Houston, TX. He investigates testing, analysis, and software reliability assessment methodologies for safety critical and intelligent flight control systems. Dr. Cukic is the co-director of the Center for Identification Technology Research (CITeR), an NSF IUCRC (Industry University Cooperative Research Center) concentrating on biometrics. Dr. Cukic served as the program co-chair of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003) and the 8th IEEE International High Assurance Systems Symposium (HASE 2004). He is a member of the IEEE, IEEE Computer Society, and IEEE Reliability Society.

Hany H. Ammar is a Professor of Computer Engineering in the Lane Department of Computer Science and Electrical Engineering at West Virginia University. He joined the faculty at WVU in 1990 after five years of service on the faculty at Clarkson University. Dr. Ammar has published over 98 articles in prestigious journals and conference proceedings such as the IEEE TRANSACTION ON SOFTWARE ENGINEERING, and the IEEE TRANSACTIONS ON RELIABILITY. Dr. Ammar has served on the Program Committees of several Prestigious conferences, served as the program co-chair of the 2002 International Symposium of Software Reliability Engineering (ISSRE 2002), and is now serving as the steering committee chair of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA). He is a member of the IEEE and the ACM professional organizations.