Formal Limits on Determining Reliabilities of Component-Based Software Systems

Martin Kappes
Fachbereich Informatik,
Johann Wolfgang Goethe-Universität*
D - 60054 Frankfurt am Main, Germany
kappes@psc.informatik.uni-frankfurt.de

Reinhard P. Klemm; Chandra M.R. Kintala
Lucent Technologies,
Bell Laboratories
Murray Hill, NJ 07974, USA
{klemm,cmk}@research.bell-labs.com

Abstract

We present a model for analyzing the reliability of component-based software systems. Each software component is described as a finite state machine whose transitions have failure probabilities that reflect the unreliable execution of elementary component operations. Alternatively, we model a component as a finite state machine with an associated reliability figure that expresses the overall reliability of the component. Using our model, we prove that there is no algorithm that can accurately calculate the reliability of a system of finite state machines with known or estimated overall reliabilities or transition failure probabilities. The same negative result holds even if we only want to approximate the system reliability.

Keywords: Communicating Finite State Machines, Component-Based Software, Reliability Approximation, Reliability Computation, Software Reliability.

1. Introduction

In recent history, software complexity has dramatically increased while at the same time hardware reliability has improved to the point where software faults are considered the major source of computer system failures [3]. Also, large software systems are increasingly built using commercial off-the-shelf (COTS) software components in order to speed up the development process and decrease development expenses. In this paper, we explore the feasibility of precisely or approximately calculating the reliability of component-based software systems. The problem was originally posed by [4]. We assume that we have complete insight into the structure of each component. We further make the assumption that the reliabilities or failure probabilities of elementary operations in each component or the overall reliabilities of components are known. Although not always realistic, these assumptions actually simplify and reinforce our central result described below. In another scenario, we view software components as black boxes with associated reliability figures. In our research, we consider only software systems with a fail-stop behavior, i.e., every system component failure translates into system termination that is distinguishable from normal system behavior.

We use networks of communicating finite state machines to model such software systems. Each transition or each component in those machines carries a reliability figure. Using this model, we show that determining the reliability of a component-based software system is inherently difficult even if

- we have complete insight into the inner workings of each component,

- we know or have an estimate of the failure probability of each elementary component operation or of each component as a whole, and

- each component has a relatively simple structure (we do not, however, impose restrictions on the complexity of the interaction between components).

More precisely, we prove that there is no algorithm that can precisely or approximately compute the reliability of a network of communicating finite state machines given the reliabilities of its transitions or components. Although it may be feasible to determine the reliability of a specific system (cf. [7] or [13]), our result indicates that the general problem of computing the reliability of an arbitrary software system is unsolvable.

We chose communicating finite state machines as an abstraction of the interactions between software components because they provide a simple and well-understood operational model for component-based software systems. For

---

*This work was done during a research visit of this author at Bell Laboratories.
†Please send all correspondence to this author.
these reasons, they have been extensively used in the special case of communication protocols (see, e.g., [1]) and more generally in modeling component interaction (cf. [12]). Finite state machines do not have the expressive power to reflect the entire behavior of an arbitrary algorithm. However, they can capture the essential features of most algorithms. In contrast to more powerful abstractions of general algorithms such as Turing machines, finite state machines can be easily analyzed, verified, and tested. In other words, they allow the construction of efficient algorithms for analysis, verification, and testing.

**Related Work:** Recent publications have raised the same question that we are trying to answer in our research, i.e., how to compute the overall reliability of a component-based software system with given component reliabilities and under the assumption that component failures will always translate into detectable system failures. In [16], the authors build a component dependency graph (CDG) with nodes representing components of the system and edges representing possible interactions between components. Reliabilities are assigned to the components and the interactions between components. This model obtains statistics about the system from actual system executions (called scenarios). These statistics are the average execution time of the system, average execution times of the components, probabilities of component interactions, and probabilities of input sets. The model uses these statistics in conjunction with the CDG to compute the reliability of system executions that mirror the sample executions. The paper also contains an extensive list of references to articles on the same topic. In contrast to [16], our paper is concerned with the feasibility of reliability computation and reliability approximation for all possible executions of a software system rather than showing an effective and efficient way of computing the overall system reliability for a set of sample scenarios. Another goal of our research was to show the surprising simplicity of a software system model that still allows us to prove the infeasibility of reliability computation and approximation.

Another recent related article is [8]. The authors model software systems with component interaction graphs and use a Bayesian approach to compute the overall software system reliability from given component reliabilities. Rather than focusing on specific system execution scenarios as in [16], [8] restricts its focus to a subset of all software systems to make system reliability computation feasible. Our work differs from the research described in [8] in similar ways as it differs from [16].

We wish to emphasize that we use communicating finite state machines to model a single unchanging software system for computing its reliability. The goal and the analysis techniques we use are different from those in reliability growth models (cf. [13], [9]) which are used to estimate the changes in reliability as more testing is done and more faults are removed. Also, our work is different from that of Littlewood [10], Cheung [2] and Siegrist [14] who formulate Markov models to determine the reliability of a software system based on the reliabilities of the modules, time spent in the modules and inter-module transitions in that system.

This paper is organized as follows. Section 2 reviews some notations in automata and formal language theory. In Section 3, we introduce the model of a network of communicating finite state machines. Section 4 presents the main results of this paper for software systems with known reliabilities for elementary operations. Section 5 shows how these results can be modified to cover software systems with known component reliabilities. We summarize the paper in Section 6.

### 2. Automata Theory Preliminaries

In this section, we describe the notation that we use in our proofs. We assume that the reader is familiar with the common notions of formal language theory and the theory of computation as presented in, for example, [11].

Let $V^*$ denote the set of all words over an alphabet $V$. It includes the empty word $\epsilon$. For a set $X$, $P(X)$ or the power set of $X$, is the set of all subsets of $X$.

A (deterministic 1-tape) Turing machine $M$ is a tuple

$$M = (Q,V_I, V_S, \delta, q_0, q_F),$$

where $Q$ is a finite nonempty set of states, $V_S$ is a finite storage alphabet $V_I \subseteq V_S$ is the input alphabet, $\delta \in V_S - V_I$ is the blank symbol, $q_0 \in Q$ is the initial state, $q_F \in Q$ is the final state, $\delta : (Q - \{q_F\}) \times V_S \rightarrow Q \times (V_S - \{\\}) \times \{L, R\}$ is the transition function. Without loss of generality we assume $Q \cap V_S = \emptyset$. We also assume that the tape of a Turing machine contains an infinite number of blank symbols to the left as well as to the right of the used tape portion. The transition relation $\rightarrow_M$ on $V_I^* Q V_S^*$ is defined as follows: $a \rightarrow_M b$ if and only if

$$a = a_1 \sigma_1 p \sigma_2 a_2, \text{ where } a_1, a_2 \in V_I^*, \sigma_1, \sigma_2 \in V_S, p \in Q, \text{ and }$$

$$\beta = a_1 q q_0 \sigma_2 a_2 \text{ if } \delta(p, \sigma_2) = (q, \sigma_3, L) \text{ or }$$

$$\beta = a_1 q \sigma_3 q_0 a_2 \text{ if } \delta(p, \sigma_2) = (q, \sigma_3, R).$$

The language accepted by $M$ is defined as

$$T(M) = \{w \in V_I^* \mid q_0 w \rightarrow_M a_1 q_F a_2, a_1, a_2 \in V_S^*\},$$

where $a \rightarrow_M b$ denotes the reflexive transitive closure of $a \rightarrow_M b$.

We say that $M$ halts after a finite number of steps on input $w$ if and only if there is a natural number $k$ such that $\sup\{i \mid q_0 w \rightarrow_{M}^{i} a \text{ for some } a, b \text{ and } a \leq k\} < k$, where $a \rightarrow_{M}^{i} b$ denotes the $i$-fold product of the transition relation.
Notice that a Turing machine halts on an input $w$ if and only if $w \in T(M)$. The following theorem is crucial for our results. A proof of this theorem can be found in [11].

**Theorem 1** There is no algorithm for the following problem:

**Input:** A Turing machine $M$, an input $w \in V^*$.  
**Output:** “yes” if $M$ halts on input $w$ after a finite number of steps, i.e. $w \in T(M)$, “no” otherwise, i.e. $w \notin T(M)$.

Finite state machines can be considered as restricted Turing machines that are not allowed to rewrite or reread parts of their tape. Consequently, there are many algorithmic problems that can be solved by a Turing machine but not by a finite state machine. In contrast to Turing machines, however, virtually all interesting questions about finite state machines can be answered algorithmically. In fact, most properties of finite state machines can be computed efficiently, i.e., in polynomial time. The goal of this paper is to prove that the reliability of general software systems cannot be accurately or approximately determined by a program even if the software components are simple. Therefore, the expressive power of finite state machines is adequate for modeling software components in our research.

The objective of our research is to model and analyze the reliability of systems built with software components. Thus, our research requires some modifications to traditional finite state machines. We propose a model of a network of communicating finite state machines where each transition carries a failure probability (or transition reliability). Each component of a system is modeled as a finite state machine that can interact with other system components and the system environment. We present our model in the following section.

### 3. Networks of Communicating Finite State Machines

Suppose a software system $S$ is comprised of several components. Let us model the basic structure of each system component as a finite state machine with output. We can capture many forms of interaction between components by connecting the finite state machines with message queues to form a network of communicating finite state machines. Each machine in the network can receive input from the environment or from another finite state machine in $S$ through a message queue. The environment is an abstraction of the combination of users of the modeled software system, the underlying operating system, other programs, and other machines and devices that interact with $S$. Each finite state machine can send output to the environment or to other finite state machines in $S$ through message queues. The network is either synchronous, in which case all finite state machines execute a transition at the same time, or it is asynchronous, in which case there is a random order among machines to execute the next transition. For the sake of simplicity, we assume that only one machine can execute at any given time in an asynchronous network. In an asynchronous network with only two machines this implies that exactly one nondeterministically chosen finite state machine in $S$ can execute a transition at any given point in time. In contrast to existing approaches to computing the reliability of software systems, the communicating finite state machine-based modeling technique considers the low-level interaction among software components.

We assume that each transition in a finite state machine $M$ in $S$ has a given probability $p \in [0, 1]$ of working correctly (or a failure probability $1 - p$ of working incorrectly). If $M$ exhibits an erroneous behavior on a transition we assume a fail stop behavior that is detectable by an outside observer (e.g., $M$ halts and terminates the execution of the entire system). Thus, the faults under consideration cannot be masked by the system. This is, of course, not always the case in a distributed system, since the failure of a component could be counteracted by other system components either by design or by chance. In our model, the probability that an input $w$ is processed correctly by $M$ is defined as product of the probabilities of all transitions traversed during the computation on $w$.

Our model gives us the freedom of choice among different behaviors of the entire system in case of a component failure but retains several degrees of freedom that allow us to analyze the reliability of many different kinds of software systems. The following are some of the parameters that can be adjusted in this model:

- component interaction pattern (client/server, function/method call, message passing among parallel processes)
- component connection (explicit message passing via pipes, sockets, or other FIFO mechanisms, parameter passing between functions or methods)
- topology (ring, chain, star, tree, meshed, arbitrary)
- reliability definition

### 3.1. Formal Definition of a Special Case

We now present the formal definition of a very simple network of communicating finite state machines. Let us consider a system with only two finite state machines which are connected via queues. When attempting to read an input from the environment or a queue, each machine can detect the absence of an input. The unavailability of an input from the environment is conveyed to the machine with a special symbol $. If the queue is empty and the machine wants to read from the queue, it receives the special symbol $. Let
us denote the set of endmarkers \( \{ \$, \# \} \) by \( V_E \). Furthermore, let \([0, 1]\) be the set of all rational numbers between zero and one.

In our special case, we consider networks of two communicating finite state machines where only one machine receives input from the environment and sends output to the environment. In other words, only one of the two machines interacts with the environment. We call such a network a master/slave system. A master/slave system is a simple modification of the model presented in [6]. We can define a machine in a master/slave system as follows:

**Definition 2** A communicating finite state machine \( M \) is a tuple
\[
M = (Q, V_I, V_O, V_{msg}, V_E, \delta, q_0),
\]
where \( Q \) is a nonempty finite set of states, \( V_I \) is the input alphabet, \( V_O \) is the output alphabet, \( V_{msg} \) is the message alphabet, \( V_E = \{ \$, \# \} \) is the set of endmarkers, \( \delta : Q \times (V_I \cup V_{msg} \cup V_O \cup \{ \$ \} \to P(Q \times (V_O \cup V_{msg} \cup \{ \$ \} \times [0, 1]) \)

is the transition function (the rational number component is the transition reliability), and \( q_0 \in Q \) is the initial state. Without loss of generality we assume that \( V_I, V_O, V_{msg} \) and \( V_E \) are mutually disjoint.

We call a machine deterministic if and only if for all \( q \in Q \), \( \sigma_1 \in V_I \cup \{ \$ \} \) and \( \sigma_2 \in V_{msg} \cup \{ \$ \} \) we have
\[
|\delta(q, \sigma_1)| + |\delta(q, \sigma_2)| + |\delta(q, \$)| \leq 1.
\]

We now proceed to the definition of a master/slave system.

**Definition 3** A master/slave system \( S \) is a special case of a network of two communicating finite state machines. It is defined as a tuple
\[
S = (M_1, M_2, V_I, V_O, V_{msg}, V_E),
\]
where \( M_1 = (Q_1, V_I, V_O, V_{msg}, V_E, \delta_1, q_0^1) \) is a communicating finite state machine (master), and \( M_2 = (Q_2, \emptyset, \emptyset, V_{msg}, V_E, \delta_2, q_0^2) \) is a communicating finite state machine (slave).

Such a system is shown in Figure 1. We call a master/slave system deterministic if and only if both the master and the slave are deterministic.

A configuration of \( S \) is a tuple \( (w, p, q, x, y, z, r) \), where \( w \in (V_I \cup \{ \$ \})^*, p \in Q_1, q \in Q_2, x, y \in (V_{msg})^*, z \in V_O, r \in [0, 1] \). The set of all configurations \( c(S) \) is defined as
\[
c(S) = (V_I \cup \{ \$ \})^* \times Q_1 \times Q_2 \times (V_{msg})^* \times V_O \times [0, 1].
\]

Informally speaking, the master \( M_1 \) can read an input symbol from the environment or from the queue of messages from \( M_2 \) to \( M_1 \) (its incoming queue). It then changes its state as a function of the input symbol and its previous state. It can also write an output symbol to the environment or to the queue connecting \( M_1 \) to \( M_2 \) (its outgoing queue). The behavior of \( M_2 \) differs from that of \( M_1 \) in that \( M_2 \) can only read input from its incoming queue and write output to its outgoing queue. Both machines are allowed to perform \( \varepsilon \)-moves instead of reading input symbols, i.e., to select a new state and potentially an output symbol depending only on the current state.

In asynchronous operation mode either \( M_1 \) or \( M_2 \) performs a computation step according to their transition function. Formally, the asynchronous transition relation \( \vdash_{aS} \) on \( c(S) \) is defined by \( \alpha \vdash_{aS} \beta \) if and only if
\[
- \text{either } \alpha = (aw, p, q, bx, y, z, r), a \in V_I \cup \{ \$ \}, b \in V_{msg} \cup \{ \$ \}, \beta = (w, p', q, x, ye, zd, r \cdot s), c \in V_{msg} \cup \{ \$ \}, d \in V_O \cup \{ \$ \}, \text{and we have } (p', \tau, s) \in \delta_1(p, \sigma) \text{ such that } \sigma = \alpha b \text{ and } \tau = cd f \text{ or } \sigma = \delta_1, a\alpha x = e, \text{ and } \tau = cd f.
\]

- or \( \alpha = (w, p, q, a, ye, z, r), a \in V_{msg} \cup \{ \$ \}, \beta = (w, p', q', xc, ye, z, r \cdot s), c \in V_{msg} \cup \{ \$ \}, \text{and we have } (q', c, s) \in \delta_2(q, \sigma) \text{ such that } \sigma = a \text{ or } \sigma = \delta_2, a\beta x = e, \text{ and } ay = e.
\]

In synchronous operation mode, both automata execute a transition at the same time. Formally, the synchronous transition relation \( \vdash_{sS} \) on \( c(S) \) is defined by
\[
(aw, p, q, bx, cy, z, r) \vdash_{sS} (w', p', q', ax, ye, zd, f, r \cdot s_1 \cdot s_2),
\]
\( a \in V_I \cup \{ \$ \}, b, c \in V_{msg} \cup \{ \$ \}, d, e \in V_{msg} \cup \{ \$ \}, f \in V_O \cup \{ \$ \}, \text{ if and only if } (p', \tau, s_1) \in \delta_1(p, \sigma), (q', d, s_2) \in \delta_2(q, \sigma') \text{ such that } \tau = ef \text{ and } \sigma = \alpha b \text{ or } \sigma = \delta_1 \text{ and } a\alpha x = e; \sigma' = c \text{ or } \sigma' = \delta_2 \text{ and } ay = e.
\]

In this paper, we will focus on the asynchronous operation mode. Therefore, we will write \( \vdash \) instead of \( \vdash_{aS} \). All results presented here can also be obtained for the synchronous operation mode by minor modifications to the constructions presented in the next section.

Consider a computation \( (w \cdot \$, q_0, q_0^1, e, e, e, 1) \vdash^* \)
\( (w', p, q, x, y, z, r) \). In this computation, \( r \) is the reliability that this computation will be performed to completion (without a failure) in the system. For an input \( w \) we define the reliability \( \text{rel}(S, w) \) of \( S \) on input \( w \) as the infimum over all computations for that input \( w \). Formally
\[
\text{rel}(S, w) = \inf \{ r : (w \cdot \$, q_0, q_0^1, e, e, e, 1) \vdash^* \}
\]
\( (w', p, q, x, y, z, r) \).

\footnote{Since \( \delta_1 \) is a mapping \( \delta_1 : Q \times (V_I \cup V_{msg} \cup V_O \cup \{ \$ \} \to P(Q \times (V_O \cup V_{msg} \cup \{ \$ \} \times [0, 1]) \), these conditions imply that \( |\delta_1| \leq 1 \) and \( |\delta_2| \leq 1 \), thus \( a = c \text{ or } b = e \text{ and } c = e \text{ or } d = e \).}
4. Systems With Known Reliabilities of Its Operations

If implemented in software, a master/slave system $S$ is a very simple case of a software system comprised of two components. The transitions of $S$ form the elementary operations of a system. Suppose we know the reliabilities of the elementary operations of $S$, either by estimating the reliabilities or based on previous observations. The main question we address in this paper is how the overall reliability of $S$ can be computed as a function of the reliabilities of the elementary operations. A related question is how to compute a good approximation for the overall reliability of $S$.

It turns out that neither the precise nor the approximate computation of the overall reliability of $S$ is possible in an algorithmic fashion. We assume here that an approximation that deviates by more than $1/2$ from the actual reliability cannot be considered satisfactory. These results are presented below. They are based on the construction of a master/slave network of communicating finite state machines $S$ for a given Turing machine $M$ in such a way that the reliability of $S$ on input $w$ is higher than $1/2$ if and only if $w$ is not accepted by $M$.

Informally, we construct a master/slave network $S$ of communicating finite state machines with components $M_1$ and $M_2$ where the queues are used to store the configuration of a Turing machine. Since a computation step in the Turing machine only changes the state and the symbols immediately located to the left and to the right of the current state, such a step can be simulated by first buffering these symbols in the states of $M_1$ and then rewriting these symbols according to the transition function of the Turing machine. The slave automaton $M_2$ only has to forward each symbol in its incoming queue to its outgoing queue. In more detail, $M_1$’s operations can be divided into three phases. In the initialization or scan phase, $M_1$ consumes its complete input $w$ and writes the string $q_0w$ to its output queue. After the initialization, $M_1$ only consumes symbols from its incoming queue. The automaton switches to the read phase where it buffers three symbols from its incoming queue. If the state of the Turing machine is in the middle of the buffered symbols, $M_1$ simulates the next computation step of the Turing machine and proceeds to the write phase. Otherwise, the buffer shifts one position to the right by forwarding the leftmost symbol of the buffer to the outgoing queue and adding a symbol to the right from the incoming queue. If $M_1$ is in a write state, it will write the leftmost symbol in the buffer contained in its state to the outgoing queue. When all symbols are written to the queue, $M_1$ enters the read phase again. By reiterating this process, the master/slave network simulates the computation of the Turing machine. Furthermore, if the symbol to be stored during the write phase is the accepting state $q_f$ of the Turing machine, $M_1$ changes with reliability $1/2$ to the only final state $f$. This is the only transition with reliability less than 1.

Therefore, we find that $rel(S, w) = 1/2$ if and only if $w \in T(M)$, and $rel(S, w) = 1$ if and only if $w \notin T(M)$. Hence, the following lemma holds (the detailed formal proof can be found in Appendix A).

**Lemma 4** Let $M = (Q, V_F, V_S, \delta, q_0, q_F)$ be a Turing machine. We can effectively construct a deterministic master/slave network of communicating finite state machines $S = (M_1, M_2, V_I, V_O, V_msg, V_E)$ such that for all $w \in V_I$

- $rel(S, w) = 1/2$ if and only if $w \in T(M)$, and
- $rel(S, w) = 1$ if and only if $w \notin T(M)$.

We will use this lemma to prove that there is no algorithm to compute the reliability of a master/slave system $S$.
for a given input \( w \).

**Theorem 5** There is no algorithm for the following problem:

**Input**: A master/slave system \( S \), an input \( w \in V_I^* \) and a given reliability level \( c \in [0, 1] \).

**Output**: “yes” if \( \text{rel}(S, w) > c \), “no” otherwise.

**Proof**. Suppose there were an algorithm \( \mathcal{A} \) that outputs “yes” if and only if \( \text{rel}(S, w) > c \). We can construct an algorithm \( \mathcal{A}' \) as follows: \( \mathcal{A}' \) receives as input a Turing machine \( M \) and an input \( w \) for \( M \). Using the construction in the proof of Lemma 4, \( \mathcal{A}' \) builds a network of communicating finite state machines \( S \) based on \( M \). It then uses \( \mathcal{A} \) as subroutine on input \((S, w, 1/2)\). If \( \mathcal{A} \) outputs “no” (“yes”), then \( \mathcal{A}' \) outputs “no” (“yes”). However, the output of Algorithm \( \mathcal{A} \) is “no” if and only if \( \text{rel}(S, w) \leq 1/2 \). According to the above lemma, this is the case if and only if \( w \in T(M) \). Therefore, \( \mathcal{A}' \) solves the halting problem, which is a contradiction. \( \square \)

Theorems 5 and 6 refer to specific input words \( w \) for a master/slave network. In practice, we would like to know whether a master/slave network computes all input words with a certain minimum reliability. As the reader may expect by now, there is no algorithm that can answer this question either.

**Theorem 7** There is no algorithm for the following problem:

**Input**: A master/slave system \( S \), a given reliability level \( c \in [0, 1] \).

**Output**: “yes” if for all \( w \in V_I^* \) \( \text{rel}(S, w) > c \), “no” otherwise.

**Proof**. Suppose there were an algorithm \( \mathcal{A} \) that outputs “yes” if and only if \( \text{rel}(S, w) > c \) for all \( w \in V_I^* \). We can construct an algorithm \( \mathcal{A}' \) which receives a Turing machine \( M \) as input, similar to the algorithm in the proof of Theorem 5. Using the construction in the proof of Lemma 4, \( \mathcal{A}' \) builds a network of communicating finite state machines \( S \) based on \( M \). It then uses \( \mathcal{A} \) as subroutine on input \((S, w, 1/2)\). If \( \mathcal{A} \) outputs “yes” (“no”), then \( \mathcal{A}' \) outputs “yes” (“no”). However, the output of Algorithm \( \mathcal{A} \) for \( c = 1/2 \) is “yes” if and only if \( \text{rel}(S, w) > 1/2 \) for all \( w \in V_I^* \). According to the above lemma, this is the case if and only if \( w \not\in T(M) \). Thus, algorithm \( \mathcal{A}' \), answers “yes” if and only if \( T(M) = \emptyset \) for the Turing machine \( M \). However, the question of whether \( T(M) = \emptyset \) for a given Turing machine \( M \) is not recursively enumerable (cf. [11]). This results in a contradiction and therefore proves the claim. \( \square \)

An analogous result holds in the case of reliability approximations.

**Theorem 8** For any \( 0 \leq \epsilon < 1/2 \) there is no algorithm for the following problem:

**Input**: A master/slave system \( S \).

**Output**: A reliability approximation \( \text{approx}(S) \) such that

\[
\left| \inf\{\text{rel}(S, w) | w \in V_I^* \} - \text{approx}(S) \right| \leq \epsilon.
\]

**Proof**. We can reuse the construction in the proof of Theorem 6. We obtain \( \text{approx}(S) > 1/2 \) if and only if \( T(M) = \emptyset \) for the Turing machine \( M \). However, the question of whether \( T(M) = \emptyset \) for a given Turing machine is not recursively enumerable (cf. [11]). This results in a contradiction and proves our claim. \( \square \)
5. Systems With Known Reliabilities of Its Components

In the model presented in the previous section, reliabilities are associated with the elementary operations of software modules. We assumed that we have complete insight into the inner workings of each component, and that we know or have an estimate of the reliability of each elementary component operation. In many applications, however, we do not have reliabilities associated with the elementary operations of the modules. Instead, we have a single reliability figure for each module, characterizing its overall reliability. Therefore, we want to look at the following changes to the model that allow us to determine the reliability of a software system from the reliabilities of its constituent components.

- Instead of associating reliability figures to the transitions of the modules, we assign one reliability figure to each module.

- The reliability of a computation is defined as the product of the reliabilities of all modules traversed in this computation.

By applying some modifications to the proofs presented in the last section, we can show that neither the precise nor the approximate computation of the overall reliability of $S$ based on component reliabilities is possible in an algorithmic fashion. Therefore, considering overall reliabilities associated with entire components instead of reliabilities associated with elementary operations of modules does not simplify the problem of determining the system reliability. For a complete definition of component reliabilities and a detailed proof of the results mentioned in this section, we refer the reader to [5].

6. Interpretation of Our Theorems and Practical Implications

We showed that there is no single algorithm to compute, either precisely or approximately, the reliability of an arbitrary software system given the reliabilities of the components in that system or the reliabilities of the operations in the components. Much like many of the results on undecidability in theoretical computer science, our theorems put formal/mathematical limits on our ability to find algorithms to compute system reliabilities from the constituent component reliabilities. Our findings, however, do not imply that the reliability of a specific software system cannot be determined. What they do imply is that for every method that attempts to compute or approximate the overall software system reliability from given operation or component reliabilities, there will be software systems for which this method fails. It may very well be possible to find a method that determines the overall software reliability for a large subclass of software systems but this method will not cover all software systems.

Our theoretical results imply that the process of computing the reliability of a component-based software system from reliabilities of elementary operations or components in that system cannot be completely automated. Therefore, software reliability engineers will have to depend on heuristics, expertise, and creativity to arrive at methods that would obtain such reliability figures for their specialized class of software systems. One clear example of such a creative method is to construct probabilistic models of software systems and to incorporate execution scenarios of those systems, as described in [16] (cf. Section 1). Other examples are the use of hierarchical models [8] (cf. Section 1), fault propagation analysis [15] and test information and test cases [7] to estimate the reliabilities of component-based software systems.

References

A. Proof of Lemma 4

The formal proof for Lemma 4 is as follows: Let $M = (Q, V_s, V_o, \delta, q_0, q_F)$ be a Turing machine. We construct a master/slave network of communicating finite state machines $S$ where the queues are used to simulate the working tape of the Turing machine. Formally, $S = (M_1, M_2, V_I, V_O, V_{msg}, V_E)$, where

$$
V_O = \{Y\}
$$

$$
V_{msg} = V_s' \cup (\oplus) \cup Q
$$

$$
M_1 = (Q_s, V_s, V_O, V_{msg}, \delta_1, [scan, 1])
$$

$$
M_2 = ([q_0], \emptyset, V_{msg}, \delta_2, q_0^{*}).
$$

To distinguish the symbols of the input alphabet from those of the message alphabet, we need a disjoint copy $V_s'$ of $V_s$. In the remainder of this proof, we will not maintain the distinction between $V_s'$ and $V_s$ because it is clear from the context which alphabet we are referring to. Without loss of generality we furthermore assume that $Q \cap V_s' = \emptyset$ and $\forall \varphi \in Q \cup V_s'$. Unless otherwise noted, the reliability of each transition is 1.

The transition function of $M_2$ is defined as follows:

$$
\delta_2(q_0^*, \sigma) = \{(q_0^*, \sigma, 1)\} \text{ for all } \sigma \in V_{msg}.
$$

In other words, $M_2$ immediately forwards each symbol in its incoming queue to its outgoing queue. The state set $Q_s$ of $M_1$ consists of three types of states, $Q_s, Q_r, Q_w$, and a final state $f$. The set $Q_s$ is the set of $\text{scan}$ states and is defined as $Q_s = \{[\text{scan}, 1],[\text{scan}, 2],[\text{scan}, 3]\}$. In these states, $M_1$ forwards all input symbols to its outgoing queue and encloses the input in $\oplus$-symbols. In its initial state, $M_1$ reads no input, writes a single $\oplus$-symbol to its outgoing queue and moves to state $[\text{scan}, 2]$:

$$
\delta_1([\text{scan}, 1], e) = \{(\text{[scan}, 2], \oplus, 1)\}.
$$

In $[\text{scan}, 2]$, $M_1$ writes the initial state of the Turing machine to its outgoing queue and proceeds to state $[\text{scan}, 3]$:

$$
\delta_1([\text{scan}, 2], e) = \{([\text{scan}, 3], q_0, 1)\}.
$$

In $[\text{scan}, 3]$ it forwards the symbols in the input to its outgoing queue:

$$
\delta_1([\text{scan}, 3], \sigma) = \{([\text{scan}, 3], \sigma, 1)\} \text{ for all } \sigma \in V_f.
$$

When $M_1$ encounters the end-of-input marker $\delta$, it writes another $\oplus$ to its outgoing queue and proceeds to the first $\text{read}$ state in the set $Q_r = \{[\text{read}, w] \mid w \in (V_{msg})^*, |w| \leq 3\}$:

$$
\delta_1([\text{scan}, 3], \delta) = \{([\text{read}, e], \oplus, 1)\}.
$$

By moving through read-states, $M_1$ buffers three symbols from its incoming queue:

$$
\delta_1([\text{read}, x], \sigma) = \{([\text{read}, x\sigma], e, 1)\}
$$

for all $x \in (V_{msg})^*, |x| < 3, \sigma \in V_{msg}$.

There are two possible cases once three symbols have been buffered. If the state of the Turing machine is in the middle of the three symbols, $M_1$ simulates the next computational step of the Turing machine. Otherwise, the buffer shifts one position to the right by forwarding the leftmost symbol of the buffer to the outgoing queue and adding a symbol to the right from the incoming queue:

$$
\delta_1([\text{read}, \tau_1 q \tau_2], \sigma) = \{([\text{read}, \tau_1 \tau_2 \tau_3], \tau_1, 1)\}
$$

for all $\sigma, \tau_1, \tau_2 \in V_{msg}, \tau_2 \in V_{msg} - Q$.

If a state of the Turing machine is in the middle of the three buffered states, $M_1$ simulates the next computational step of the Turing machine. It then proceeds to one of the write states in the set $Q_w = \{[\text{write}, w] \mid w \in (V_{msg})^*, |w| \leq 5\}$. We have to give special consideration to the case where the Turing machine has arrived at either end of the tape:

$$
\delta_1([\text{read}, \tau_1 \tau_2], \sigma) = \{([\text{write}, \beta], e, 1)\}
$$

for all $\tau_1, \tau_2 \in V_{msg}, \sigma \in Q$, where

$$
\tau_1 \tau_2 \rightarrow [M] \beta \text{ if } \tau_1, \tau_2 \neq \delta.
$$

$$
\tau_1 \rightarrow [M] \gamma \text{ if } \tau_1 = \delta, \tau_2 \neq \delta, \beta = \gamma \gamma.
$$

$$
\tau_1 q \rightarrow [M] \gamma \text{ if } \tau_1 = \neq, \tau_2 = \delta, \beta = \gamma \gamma.
$$

Notice that the length of $\beta$ is between 3 and 5 symbols. If $M_1$ is in a write state, it will write the leftmost symbol in the buffer contained in its state to the outgoing queue. If this symbol is the accepting state $q_F$ of the Turing machine, $M_1$ changes with reliability $1/2$ to state $f$. This is the only transition with reliability less than 1:

$$
\delta_1([\text{write}, \sigma \tau], e) = \{([\text{write}, \tau], \sigma, 1)\} \text{ if } \sigma \neq q_F
$$

$$
\{([f, Y, 1/2]) \text{ otherwise}\}
$$
for all \( \sigma \in V_{msg} \), \( \tau \in (V_{msg})^* \), \(|\tau| \leq 4 \). This process continues until the buffer in \( M_1 \)'s state is empty. At that time, \( M_1 \) switches to \([\text{read}, e]\) and the aforementioned process repeats itself:

\[
\delta_1([\text{write}, e], e) = \{ ([\text{read}, e], e, 1) \}.
\]

By induction we can prove that for all \( q \in Q - \{ q_f \} \) and \( \alpha, \beta \in V_2^* \) we have

\[
(w \#_1 [\text{scan}, 1], q_0^2, e, e, e, 1) \overset{\bullet}{\rightarrow} (e, [\text{read}, e], q_0^2, \Diamond \alpha q \beta \Diamond, e, e, 1)
\]

if and only if \( q_0 w \overset{\bullet}{\rightarrow} q_0 \beta \). Using this fact it is not difficult to show that

\[
(w \cdot \$, [\text{scan}, 1], q_0^2, e, e, e, 1) \overset{\bullet}{\rightarrow} (e, f, q_0^2, \alpha, \beta, Y, 1/2)
\]

if and only if \( q_0 w \overset{\bullet}{\rightarrow} q_0 \beta \). This proves our claim because in all other cases the reliability of a computation of \( S \) is 1.