Assume-Guarantee Supervisor for Concurrent Systems

Mohammad Zulkernine and Rudolph E. Seviora
Bell Canada Software Reliability Laboratory
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
[mzulker|seviora]@uwaterloo.ca

Abstract

Despite rigorous use of model checking, testing, and other technological innovations in software development there exists faults which elude those detection efforts and do not surface until the software is operational. These faults may lead to serious software failures (deviation of actual behavior from the desired one). Existing software failure detectors for concurrent systems are not compositional, and hence suffer from the state explosion problem. We present a compositional approach for automatic failure detection of concurrent programs specified as a collection of communicating finite state machines. The failure detector described in this paper is called assume-guarantee supervisor. The supervisor simultaneously observes the input/output and stable states of the target system, interprets the specification, and reports the discrepancies between these two as failures. We formalize an assume-guarantee paradigm for supervision, and provide a generic failure detection algorithm. We also describe the architecture and operation of a failure detection tool which employs the above model. This tool can be employed for online software failure detection in the operational stage of a system.

1 Introduction

With the rapid proliferation of software controlled systems in the applications where the quality of service is important, the need of reliable software is indisputable. The reliable operation of parallel and distributed systems is far more crucial and difficult than other softwares, because special mechanisms are required to handle the nondeterminism and concurrency issues involved in these systems. Many research efforts have addressed the need to improve the reliability of concurrent systems by applying model checking and testing techniques to the artifacts that appear throughout most of the software life cycle. Model checking generally subjects a model rather than an implementation, and decides whether the model satisfies *certain* properties. Software testing usually depends on an assumed input distribution, and reliability predictions based on software testing can be disqualified if the input distribution is inaccurate or the distribution changes during the operational stage of the software. Moreover, inherent nondeterminism invalidates most testing techniques.

In general, most research on software fault finding efforts have explored the first four stages of software life cycle (specification, design, implementation, and testing). Despite rigorous use of model checking, testing, and other technological innovations there exists faults which elude those detection efforts, and do not surface until the software is operational. Several studies have shown that no matter how much effort has been put into the early stages of the software development, building large fault-free software systems has proven nearly impossible in practice. Even professionally written programs may contain between one and ten independently detectable faults per thousand lines of codes [1], [2]. These faults may lead to serious software failures *(deviation of actual behavior from the desired one).* Given that, it is very important to have a tool which can be used for online monitoring of software systems in the operational stage. A software supervisor is such a tool. It reports the discrepancies between the behaviors prescribed by the specification model and the actually observed behavior as failures.

Compositional reasoning has been employed for the task of specification and verification in order to handle the complexity of large concurrent systems [3], [4], [5], [6], [7], [8]. It transfers the burden of proof from the global level to the local component level so that global properties can be inferred from independently verified component properties. However, compositional approach has not been applied in the operational stage of a software yet. Among all the compositional approaches assumption-commitment [6] (also called rely-guarantee [7]) is the most discussed technique. Nevertheless, we also agree with the view that it has been "more widely studied than actually used" [8]. According to this model, a component satisfies a guarantee G only if its environment satisfies an assumption A.

0-7695-0990-8/01/$10.00 (C) 2001 IEEE

In this paper, we propose a compositional approach to software supervision of parallel and distributed systems. Indeed, our motivation is spurred from the success of the compositionality principle used in the specification and verification of concurrent systems. The approach presented here incorporates the modularity and compositionality concepts of verification into the operational stage. We describe an application and adaptation of assume-guarantee style reasoning for automatic failure detection of concurrent programs. We formalize an assume-guarantee paradigm for supervision, and provide a generic failure detection algorithm. We also describe the architecture and operation of a failure detector which employs the above model. The target system is assumed to be specified as a collection of communicating finite state machines (CFSMs). The supervisor simultaneously observes the target system, interprets the specification, and reports the discrepancies between these two as failures. It can be employed for online failure detection without affecting the normal operation of the system.

Paper Organization. Thus far we have provided a general overview of the assume-guarantee supervisor by emphasizing the motivations behind this work. The remainder of the paper is organized as follows. Section 2 describes the use of a software supervisor. Section 3 explains the advantages of compositionality in the context of software supervision. In Section 4, we survey the most relevant research work. Section 5 first establishes the notations and definitions of the specification of concurrent systems we consider to supervise. Then it presents the proposed supervision technique illustrating the model, architecture, and operational algorithm. Section 6 is dedicated to demonstrate the operation of the supervisor using an application example. Finally, we conclude this paper with the summary of work done so far along with some future research directions.

2 Software Supervisor

A supervisor of a software system is a separate unit from the target software. It passively observes the inputs, outputs, and state information of the target system, and reports failure. In order to detect failures the supervisor must have the knowledge what the standard behavior is. Our research considers that the desired behavior of the target system is, or could be, specified in a formalism based on communicating finite state machines. The supervisor acquires the knowledge about the desired behavior of the target system by interpreting this requirement specification. It may be attached to either the entire system or a sub-system, provided that inputs, outputs, and stable state information are observable. Figure 1 shows the working environment of the supervisor.

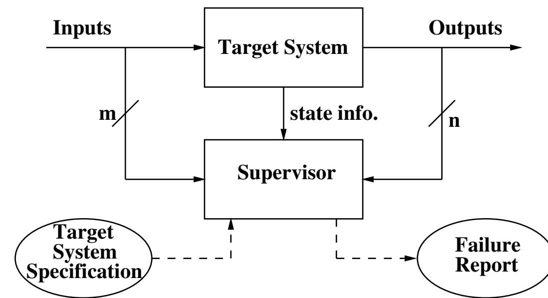

Figure 1. Working environment of the Supervisor.

There are several advantages of this type of online supervision. Early failure reporting allows the operator of the system to issue an advance warning, and attempt to remove the underlying faults before a more serious system failure may occur. A supervisor can be used to assist in testing the sub-systems of a large software system during the development by attaching it to each of the sub-systems. It helps to detect errors before they manifest themselves as externally observable failures.

Systems which exhibit nondeterminism are difficult to test because of the multiple outcomes for a single test case. In such cases, a supervisor can be used as a test oracle which is able to report relatively complete set of failures.

Furthermore, the supervisor can act as an instrument for measuring software reliability. Automatic reporting of failures provides an indication of the reliability of the executing software systems to its operators. History of mature technical disciplines shows the key role of precision instruments played in their advancement. Availability of a precision instrumentation for measurement and collection of software failures is likely to provide support to further evolution of software reliability engineering.

3 Why Compositionality in Supervision?

The “compositionality” property implies that a composite system satisfies its specification if all of its components satisfy their specifications [3], [4], [5]. Suppose a program P is composed of the parts $P_1, P_2, ...,$ and P_N by employing a set of operators (e.g., parallel). Let the specifications of $P_1, P_2, ...,$ and P_N be $M_1, M_2, ...,$ and M_N respectively. Then based on this property P satisfies M if each component P_i satisfies M_i for $i = 1, ..., N$.

To hide the complexity of real-time systems, and to provide more freedom in their implementation, most of the specification languages offer various types of nondeterminism in their syntax and semantics. In a concurrent system, if multiple transitions are triggered in different independent parallel processes, the execution order of those transitions is nondeterministic. As a consequence of this nondeterminism, a target system may show different, nevertheless legal

0-7695-0990-8/01/$10.00 (C) 2001 IEEE

external behaviors for the same set of stimuli from the environment. A supervisor monitoring such a nondeterministically specified system must be able to consider all possible legal behavioral alternatives to avoid erroneous failure reports. This is one of the major challenges of supervision as the number of behavioral alternatives required to be considered by the supervisor may be large resulting in a large supervisor time and space complexity. Compositional supervision ensures global system monitoring by supervising the components of a system, and then composing the independently verified components. In this way, it reduces the supervision of a complex system to supervising one component at a time. This is particularly useful to avoid the *state explosion problem* in case of parallel composition of interacting processes. Compositional supervision reduces the complexity of the specification from exponential to linear with respect to the number of the component specifications.

The supervisor needs to interpret the requirements specification accurately in order to derive the desired behavior of the target system. Compositionality implies a specification interpretation method, which requires to know what the basic semantic units are. In other words, compositionality forces one to proceed in a more systematic manner by defining the basic units properly, which leads to an error-free composition technique. So, this aspect of compositionality in a supervisor helps to figure out the weak points of a composite implementation, which has a high risk of being defective.

A system is called *open* if it interacts with an environment it does not control. The supervision of open systems is far more difficult than that for closed systems, because we need to verify the components of the system without knowing the environment behavior beforehand. Compositional supervision considers each component as an open system, and explicitly includes the assumptions on the environment in the component specifications. The component level supervision results may also provide valuable information for fault localization.

Compositionality in supervision facilitates constraint-based supervision, where a failure is reported when a constraint is violated. In comparison-based supervision, a supervisor waits for a target system output to compare it with the predicted output [9], [10]. Again, constraint-based supervisor can report not only the external failures but also the internal erroneous behaviors.

4 Related Work

In general, our work on supervision and that of others engaged in testing and model checking research differs in mainly three ways. First, in supervision the normal operational behavior of the target system is not interrupted, i.e., we do not control the input output originating from the environment. Second, the supervisor is capable of monitoring relatively complete set of behavior of the target system. Third, supervisor can be integrated in a real world environment when the system is running. In this section, we concentrate on the work related to the failure detection of software in its operational stage.

In N-version programming and the recovery block method, several versions of software must be developed by separate teams which lead to high development cost [13]. Even when these versions are developed separately, it has been observed that they tend to have similar faults. As a result, the fault tolerance of the system using these two approaches is not as good as expected. Moreover, they cannot handle the system which produces output in a nondeterministic manner.

The Observer approach [14] for online validation of distributed system observes all the messages exchanged between the system components rather than the external input/output only. It reports failure by comparing the observed behavior of the system to a formal specification. However, to build the observational model the Observer relies on a single global model constructed by composing the individual process specifications. A variant of the Observer described in [15] investigated the advantages of using equivalent multiple binary (two-state) observers instead of a single large one. However, this one also requires the global state graph of the system under observation to derive these small observers. So, both of the above approaches are non-compositional, and neither has explicitly considered the effect of specification nondeterminism.

The belief-based supervisors [9], [10] maintain a collection of hypotheses called consistent belief sets, which are basically the sets of possible global states. Even though they do not make the parallel composition of individual machines into a single one statically, the number of beliefs or global states may grow exponentially as the nondeterminism increases. However, they observe the target system as a *complete* black box, i.e., only the external inputs and outputs are visible to the failure detector.

Passive testing research [16], [17] aimed at detecting failures in the same setting as the supervisor, where the inputs to the system cannot be controlled. They also take into account of specification nondeterminism. However, in the case of collection of communicating finite state machines they construct a composite machine to employ the same algorithms used for stand alone finite state machine.

Test oracles [20] are capable of evaluating the outputs of test cases. The difficulty of automatically detecting the failures is reduced by the control over the inputs to the program in testing. Specification nondeterminism is a consideration but it is generally avoided by judicious selection of inputs.

The work in runtime result checking [18], [19], and software audits [21] dealt with more constrained set of prob-

lems. Runtime result checking is used for the verification of the results of computation of mathematical functions. A software audit checks a data structure of a program periodically, and may detect a limited set of errors by comparing the data structure with a duplicate or by checking its consistency. It is an intrusive failure detector, which must have access to the data structure of the main program. In such work, specification nondeterminism and concurrency issues were out of their scopes.

5 Compositional Supervisor

The fundamental challenge of *compositional supervision* is to show that the whole implementation behaves correctly (according to its specification) if all of its components behave correctly in isolation (according to the component specifications). The implementation under supervision or the target system is specified using communicating finite state machines. The environment of each component is assumed to be the other components, and is usually left implicit in the specification of individual processes in a set of CFSMs. The supervisor obtains the stable state information from the implementation and transforms the individual CFSM specifications in the form of assume-guarantee style specifications. It also derives the overall system assumptions and guarantees by observing the external behavior of the system. Finally, the supervisor tries to discharge the assumptions using assume-guarantee rules. If all the assumptions cannot be discharged then it is concluded that a failure has occurred.

5.1 Target System Model

The supervisor model or the target system specification is expressed in a set of communicating finite state machines. The behavior of each process is represented by a separate finite state machine (FSM). An FSM is a directed graph (V, E), where V is a set of states, and E is the set of edges or possible state transitions. In addition, a process uses implicit queues to represent channels for communication. Each process has only one input queue for all the incoming messages (we call it incoming channel of a process). Any process which wishes to send a message to a particular process will send the message through the incoming channel of the receiver process. No assumption can be made on the time that a process can stay in a state. In other words, the processes run in parallel with nondeterministic speeds. A process can be regarded as a sequence of states, and formally a process state is defined as follows:

Definition 1 Process State. *For a process, P_j, a state is a tuple (s_j, c_j), where s_j represents the current symbolic state of P_j, and c_j is the sequence representing the contents of the P_j's input queue.*

A transition or execution of a system is a sequence of global states, where each global state consists of the current states of all the processes, and the contents of the input channels. Suppose a system consists of N communicating processes: $P_1, P_2, ...$, and P_N, then the global state is defined as follows:

Definition 2 Global State. *A global state S is a $2N$-tuple $(s_1, s_2, ..., s_N, c_1, c_2, ..., c_N)$, where each s_j represents the current symbolic state of the process P_j, and each c_j is the sequence of messages sent from all other processes to P_j at state s_j (i.e., contents of the channel c_j at s_j).*

In this work, we assume that only the *stable* global states of the system are available for the use of the supervisor. In these states, the system waits for an external input i.e. input from the environment of the whole system. In other words, no further progress can be made without any external input. In this setting, in a stable global state there is no *enabling internal transition* since the input ports are empty. In the global states other than the stable ones, at least one of the component machines has internal input/output operations ready to be executed. Note that the initial global state is always a stable one; all the processes are in their initial states and all the channels are empty.

Definition 3 Stable Global State. *A global state $(s_1, s_2, ..., s_N, c_1, c_2, ..., c_N)$, is called stable iff all the channels of the processes are empty in that state, i.e., $c_1 = c_2 = ... = c_N = nill$.*

Visibility of Stable Global States. The widespread usage of checkpointing and message logging strategies to increase the level of software-based fault tolerance in distributed systems suggests the feasibility of observing the stable global states. In general, during the failure-free execution a snapshot of the entire state of a software process is saved periodically (checkpointing), while messages sent or received by the process are logged (message logging) between check points. If a failure is occurred, these saved states and messages are utilized to restart the system from a previous checkpoint [13]. In this work, only the symbolic states of the individual processes are extracted when the target system reaches in a stable global state. These states are a subset of the system global states. Visibility of these states facilitates compositional approach of supervision and reduces the complexity of the supervisor. It also helps to detect internal erroneous behaviors of a system.

5.2 A/G Model for Supervision

In the assume-guarantee model [6], [7], one part of the specification defines the guaranteed behavior of a component, and the other part of the specification defines the assumed behavior of the system in which the component is

0-7695-0990-8/01/$10.00 (C) 2001 IEEE

Figure 2. Architecture of the Assume-Guarantee Supervisor.

interacting. This implies conditional specification for a process, usually written as $\{A_j\}P_j\{G_j\}$, where A_j is called the assumption and the G_j is called the guarantee of the process P_j. If the assumed behavior (A_j) about the environment is satisfied by the environment, the component will satisfy the behavior guaranteed (G_j) by itself. By combining the set of assume-guarantee behavior pairs (A_j, G_j) in an appropriate manner, it is possible to prove the correctness of the whole system.

In the literature, there are several forms of rules proposed for the assume-guarantee model in the context of different settings [4], [5], [6], [7]. However, they belong to the same principle. In many cases they have been studied as an extension of Hoare Logic [6]. In this view, the specification is a tuple (p, A, G, q), where p and q are precondition and postcondition respectively, while A and G are assumptions and guarantees respectively describing the interactions between the component and its environment. Here, we use the similar forms of assume-guarantee rules by extending the rules with different interpretations of p and q. In our case, the predicate p and q are assumed to be true for a process if they are initially at state p and then transfers to state q. More specifically, p and q are state information of the process instead of predicates on the state of a process.

To give a formal interpretation of the specification form used in this paper, we define the following. A process P_j can be regarded as a finite or infinite sequence of states $s_j = s_{j0}, s_{j1}, ..., s_{ji}, ...$, where s_{ji} denotes the $(i+1)$-th state of the process P_j. The expression $P_j \models (p_j, A_j, G_j, q_j)$ means that the process P_j satisfies a specification (p_j, A_j, G_j, q_j), and this can be interpreted as follows:

- *for any sequence of states and for any i, if $p_j = s_{j(i-1)}$, $q_j = s_{ji}$, and at $s_{j(i-1)}$ the environment of $P_j \models A_j$ then $P_j \models G_j$ at s_{ji}.*

Now we formally present the complete set of compositionality rules of the assume-guarantee model employed for the supervision:

$$\frac{\begin{array}{ll} P_1 \models (p_1, A_1, G_1, q_1) & A \wedge G_1 \rightarrow A_2 \\ P_2 \models (p_2, A_2, G_2, q_2) & A \wedge G_2 \rightarrow A_1 \\ \multicolumn{2}{c}{G_1 \wedge G_2 \rightarrow G} \end{array}}{(P_1 || P_2) \models (p_1 \wedge p_2, A, G, q_1 \wedge q_2)}$$

In words, in order to show that the composite system $P_1||P_2$ satisfies $A \rightarrow G$, we must prove the following premises:

- Each P_j satisfies $A_j \rightarrow G_j$. This means that if the environment of jth component satisfies A_j then the component itself satisfies G_j.
- The individual component assumptions A_1 and A_2 must be satisfied by the conjunction of the overall system assumption A and the individual guarantees G_1 and G_2.
- The overall guarantee G must be implied by the individual guarantees G_1 and G_2.

5.3 Supervisor Architecture

This section describes the high-level architecture or the static characteristics of the *Assume-Guarantee Supervisor* shown in Figure 2. It consists of the following components.

Implementation Under Supervision (IUS). This is the target system we want to supervise. Its stable states, and the Input/Output events exchanged between the system and its environment are available to the supervisor.

Environment Behavior Repository (EBR). The *EBR* receives the externally observable I/O of the target system. These are used to generate the overall assumptions and

0-7695-0990-8/01/$10.00 (C) 2001 IEEE

guarantees of the target system, and are stored in the *System A/G Buffer.*

System $\mathbf{A/G}$ Buffer. This buffer temporarily stores the overall assumptions and guarantees of the target system *(IUS).*

Target State Repository (TSR). This unit stores the sequence of stable global states of the system. This state information are used by the *A_j/G_j Generator* to generate individual process assumptions and guarantees.

Supervisor Model. This is the formal requirement specification of the *IUS*. This contains the legitimate behaviors of the target system. In our case, we consider that the supervisor model is specified as a collection of communicating finite state machines which has been described in detail in Section 5.1.

$\mathbf{A_j/G_j}$ Generator. This unit interprets the supervisor model by going through the individual FSM specification, and derives assumptions and guarantees in terms of message events. The generated assumptions/guarantees are sent to the *A_j/G_j Buffer.*

$\mathbf{A_j/G_j}$ Buffer. This buffer temporarily stores the individual process assumptions and guarantees which are eventually fed to the *Assumption Discharger* unit in order to decide about any failure.

Assumption Discharger. This unit receives the system's overall assumptions/guarantees from the *System A/G Buffer*, and the individual process assumptions/guarantees from *A_j/G_j Buffer* unit. It verifies the logical implications discussed in Section 5.2, and reports failure (if there is any).

5.4 Operation of the A/G Supervisor

The operation or the dynamic characteristics of the A/G supervisor can be described as follows. The target system (IUS) sends external Input/Output to the *EBR,* and stable state information to the *TSR*. Then the *EBR* derives the assumptions and guarantees of the whole system, and stores them in the *System A/G Buffer.* The *A_j/G_j Generator* receives the external Input/Output from the *EBR*, and stable state information from the *TSR*. On receiving this information, the *A_j/G_j Generator* derives individual process assumptions and guarantees between two consecutive stable states of the process by interpreting the target system process specifications. The generated individual process assumptions and guarantees are stored in the *A_j/G_j Buffer.* Finally, all the buffered assumptions and guarantees stored in both buffers are fed to the *Assumption Discharger* for the verification of the assumptions. The *Assumption Discharger* verifies whether the assumptions made by each component are satisfied by the other components. If all the assumptions cannot be discharged based on the rules described in Section 5.2 then it reports a failure. Here, by "failure" we have meant not only the violation of external behavior but also the violation of internal constraints.

In a nutshell, the complete operation cycle of the supervisor may be divided into three main steps (c.f., Algorithm 1): generation of overall assumptions and guarantees (line 05-09), derivation of individual process assumptions and guarantees (line 14-21), and finally verification of the assumptions whether they can be discharged or not (line 23-29). In this algorithm, s_{ji} is the $(i+1)$-th stable state of the process P_j. $< A_j >^i_{i-1}$ are P_j's assumptions on the other processes and the environment. $< G_j >^i_{i-1}$ are the guarantees made by P_j on its transition from the i-th stable state to the $(i+1)$-th stable state.

Algorithm 1: ***Detection of Software Failures.***
Input: *A specification of N CFSMs and a trace of the behavior (external input/output) of an implementation along with its stable global states.*
Output: *Whether the IUS system's Input/Output belongs to the specification.*

```
01. /* initial stable global state */
02. S_0 = [s_10, s_20, ....., s_N0];
03. for i = 1 to ∞
04. /* while target system is operating */
05.   observe external I/O until
06    next stable global state;
07.   /* compute overall system
08.   assumption and guarantee */
09.   derive A/G from I/O;
10.   observe current stable global state:
11.   S_i = [s_1i, s_2i, ....., s_Ni];
12.   /* compute individual process
13.   assumptions and guarantees */
14.   for j = 1 to N
15.   /* for all component processes */
16.     if (s_ji is reachable from s_j(i-1))
17.       store assumptions (< A_j >^i_{i-1})
18.       and guarantees (< G_j >^i_{i-1});
19.     endif
20.     else return ``failure" and exit;
21.   endfor
22.   /* prove the premises */
23.   for j = 1 to N
24.   /* for all component processes */
25.     if (< A_j >^i_{i-1} is not discharged
26.     by < G_{k,k≠j} >^i_{i-1}s and A)
27.       return ``failure" and exit;
28.     endif
29.   endfor
30. endfor
```

We consider the assumptions and guarantees as traces. A trace is a sequence of send and receive events. A receive

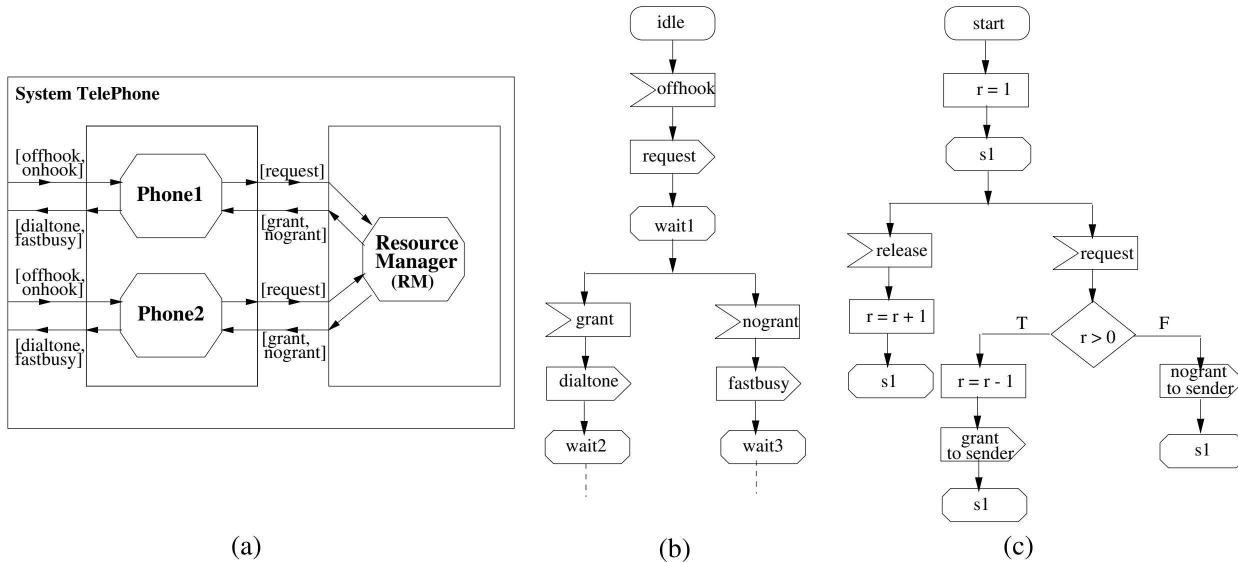

Figure 3. An SDL specified Telephone System. (a) Block diagram (b) Phone process (c) RM process.

event in a process is treated as an assumption on its environment, while a send event is regarded as a guarantee of itself. It is important to note that in the case of the complete system and its environment, this is treated in an opposite way. A logical argument about an assumption is valid if all the events in the assumptions have the corresponding complementary events in the premises of the argument. Two events are complementary if they are the "send" and "receive" events of the same message. This is stated based on the fact that a message cannot be received unless it was sent before. For example, if a process P_1 sends a message to process P_2, then $!a$ event at P_1 (denoted by $P_2!a$) is a complementary event of the $?a$ event at P_2 (denoted by $P_1?a$). (A "!" symbol indicates the transmission of a message, and a "?" symbol represents a reception.) Here, $!a$ event is a guarantee of P_1, while $?a$ is considered as an assumption for P_2 about its environment. So, $?a$ assumption can be discharged if there is a $!a$ event in the premise of the logical implication used to discharge this assumption.

6 Application Example

We describe the supervision of a very simplified concurrent system to explain the operation of the proposed technique. The concepts and results presented in this paper will be applicable in general to the systems which are specified in a formalism based on communicating finite state machines. However, for the sake of notational convenience, and its widespread applications, we use the SDL (Specification and Description Language) - a standard of the International Telecommunication Union (ITU) for specifying communication protocols [11]. We separate the assumptions and guarantees expressed in the SDL, and represent these as sequences of message events using the notations similar to those used in Hoare's CSP [12].

Let us consider a concurrent system of a 2-user telephone system specified using the SDL. The block diagram shown in Fig. 3(a) shows the communication among the 2 instances of a *Phone* process, a *Resource Manager (RM)* process, and the environment (Env). A *Phone* can receive an `offhook` or an `onhook` from the environment, and it sends a `dialtone` or `fastbusy` back to the environment. It communicates with the *RM* by sending `request` or `release`, and by receiving `grant` or `nogrant`. Fig. 3(b) and (c) show a part of the process diagrams of *Phone* and *RM* respectively. A *Phone* sends a `request` to *RM* upon receiving an `offhook` from the environment. The *RM* responds back with a `grant` or a `nogrant` depending on the value of the variable `r`. Then the *Phone* sends to the environment a `dialtone` if it receives a `grant`, or it sends a `fastbusy` if it receives a `nogrant` from the *RM*.

Suppose both *Phone1* and *Phone2* receive `offhook` signals from the environment at about the same time. Then each of them will send a `request` message to the *RM*. Since, the individual processes run with nondeterministic speeds (i.e., SDL channels have nondeterministic delays), the order of requests received by the *RM* is nondeterministic. More specifically, the *RM* could receive `request` message from the *Phone1* first and then from the *Phone2* or vice versa. Since there is only one resource available, according

to the specification, one of the users will receive dialtone, and the other will receive fastbusy signal. The specification does not deterministically dictate which one will get what signal. So, the supervisor should be able to accept (as correct) both of the following externally observable sequence of events occurred between the system and the environment: {<offhook1, offhook2, dialtone1, fastbusy2>, <offhook1, offhook2, dialtone2, fastbusy1>}. (The number at the end of a signal indicates the corresponding phone.)

Case 1. *User1 gets dialtone, User2 gets fastbusy.*
Based on the observed external events the *EBR* unit of the supervisor derives the following overall assumption/guarantee:
A = <Phone1!offhook, Phone2!offhook>
G = <Phone1?dialtone, Phone2?fastbusy>.
The *TSR* unit collects the current stable state information, which informs the A_j/G_j *Generator* that the target has moved from the previous stable global state $(idle, idle, s1)$ to the current stable global state $(wait2, wait3, s1)$. Then the A_j/G_j *Generator* derives the following individual process assumptions and guarantees by interpreting the corresponding process specifications:
Phone1: A_1 = <Env?offhook, RM?grant>
and G_1 = <RM!request, Env!dialtone>.
Phone2: A_2 = <Env?offhook, RM?nogrant>
and G_2 = <RM!request, Env!fastbusy>.
RM:
(A_3 = <Phone1?request, Phone2?request>
and G_3 = <Phone1!grant, Phone2!nogrant>),
(A_3 = <Phone1?request, Phone2?request>
and G_3 = <Phone1!nogrant, Phone2!grant>),
(A_3 = <Phone1?release, Phone2?release>
and G_3 = <>).
Now, all the generated assumptions and guarantees are sent to the *Assumption Discharger* for verification using the following logical implications: $A \wedge G_2 \wedge G_3 \rightarrow A_1$; $A \wedge G_1 \wedge G_3 \rightarrow A_2$; $A \wedge G_1 \wedge G_2 \rightarrow A_3$; $G_1 \wedge G_2 \wedge G_3 \rightarrow G$. Note that, the same pair of A_3/G_3 should be used to prove all the logical implications.
Failure Report: All of the above implications can be proved correct by using the first A_3/G_3 set of the *RM*. So, no failure is reported.

Case 2. *User1 gets fastbusy, User2 gets dialtone.*
This can also be proved in the same manner as the Case 1, and no failure is reported by the supervisor.

Case 3. *Both the users get dialtone.*
Here, the overall assumption/guarantee are:
A = <Phone1!offhook, Phone2!offhook>
G = <Phone1?dialtone, Phone2?dialtone>.
The stable state information we have $S_i = (idle, idle, s1)$ and $S_{i+1} = (wait2, wait2, s1)$. Again the individual process assumptions/guarantees are:
Phone1: A_1 = <Env?offhook, RM?grant>
and G_1 = <RM!request, Env!dialtone>.
Phone2: A_2 = <Env?offhook, RM?grant>
and G_2 = <RM!request, Env!dialtone>.
RM: The same A_3/G_3 sets as listed in the Case 1.
Failure Report: Here, if we consider the first A_3/G_3 set of the *RM,* the assumption A_2 cannot be discharged by the logical implication $A \wedge G_1 \wedge G_3 \rightarrow A_2$. For the second set of A_3/G_3, A_1 cannot be discharged by $A \wedge G_2 \wedge G_3 \rightarrow A_1$. Also the third set does not help to prove all the implications. Since all the implications cannot be proved using any of the A_3/G_3 pairs, a failure is reported.

7 Conclusion and Future Work

In this paper, we have described a novel technique for automatic detection of failures of concurrent programs. Our technique is compositional, i.e., behavioral correctness of the whole system is proved without constructing the global state graph. The compositionality is achieved by separating the transitions of individual state machines into message receptions and transmissions. We have formalized an assume-guarantee paradigm for software supervision, and provided an algorithm for online failure detection by employing the assume-guarantee rules. We have also described the design principles of the failure detector in detail. Our application example endorses the effectiveness of this technique in handling specification nondeterminism. The proposed compositional supervisor reduces the complexity of failure detection from exponential to linear with respect to the number of concurrent processes. This approach is very suitable for open system supervision. It facilitates constraint-based supervision, and helps for partial fault localization. The supervisor is also capable of detecting the internal erroneous behaviors even though they do not appear at some observable output level. This has been possible due to the visibility of the stable states of the target system.

The foundation of assume-guarantee principle is not new and composition principles of such specifications have been proposed for a number of formalisms [6], [7]. What is new in this paper is that we advocate the use of compositional principles in the operational stage of softwares in order to monitor the actual behavior of the end product. We propose a disciplined adaptation and application of assumption-commitment or rely-guarantee techniques for automatic software failure detection, and formalize the principles in that context. We are confident that a more careful study of other compositional model checking principles (e.g., lazy compositional approach [8]) would lead to more applications of these techniques in monitoring concurrent programs.

Of course, what we have provided here is just a scientific kernel of the method. Several research issues that require

0-7695-0990-8/01/$10.00 (C) 2001 IEEE

immediate attention are a better understanding of the trade-offs among the power of failure detection, degree of state observbility, and the complexity of proof rules. Again, the main objective of the use of compositional approach is to reduce belief explosion [9], [10] or state explosion. However, nothing comes for free, and we may run into the problem of assumption explosion. The complete answers to these research questions can only be determined by additional experimentation. The technique discussed in this paper is being implemented in order to monitor the control program of a small telephone exchange. We have omitted the issues regarding the implementation of this tool in this abridged conference version of the paper.

References

[1] C.A.R. Hoare, "How did software get so reliable without proof," *Lecture Notes in Computer Science,* Vol. 1051, 1996, Springer-Verlag.

[2] R. Glass, "Persistent Software Errors," *IEEE Transactions on Software Engineering.,* 7(2), pp. 162-168, March 1981.

[3] W. P. de Roever, "The Need for Compositional Proof Systems: A Survey," In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference (COMPOS '97),* Vol. 1536 of *Lecture Notes in Computer Science,* pp. 1-22, Bad Malente, Germany, 1997, Springer-Verlag.

[4] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Transactions on Programming Languages and Systems,* 17(3):507-534, May 1995.

[5] M. Abadi and L. Lamport "Composing specifications," *ACM Transactions on Programming Languages and Systems,* 15(1):73-132, Jan. 1993.

[6] J. Misra and K. M. Chandy, "Proofs of Networks of Processes," *IEEE Transactions on Software Engineering,* 7(4), pp. 417-426, July 1981.

[7] C. B. Jones, "Tentative steps towards a development method for interfering programs," *ACM Transactions on Programming Languages and Systems,* 5(4):596-619, 1983.

[8] N. Shankar, "Lazy Compositional Verification," *Lecture Notes in Computer Science,* Vol. 1536, Springer-Verlag, 1997.

[9] T. Savor and R. Seviora, "Toward Automatic Detection of Software Failures," *IEEE Computer,* Vol. 21, No. 8, pp. 68-74, August 1998.

[10] J. Li and R. Seviora, "Automatic Failure Detection with Conditional Belief Supervisors," *Proc. of the 7th International Symposium on Software Reliability Engineering,* IEEE CS Press, pp. 4-13, Oct. 1996.

[11] ITU-T, *Recommendation Z.100, Specification and Description Language, SDL,* ITU-Telecommunication Standardization Sector, Geneva, Switzerland, 1992.

[12] C.A.R Hoare, *Communicating Sequential Processes,* Prentice-Hall International, 1985.

[13] P. Jalote, *Fault Tolerance in Distributed Systems,* Prentice Hall, New Jersy, 1994.

[14] M. Diaz, G. Juanole, and J.P. Courtiat, "Observer - a concept for formal on-line validation of distributed systems," *IEEE Transactions on Software Engineering,* Vol. 20, No. 12, pp. 900-913, Dec. 1994.

[15] C. Wang and M. Schwartz, "Fault Detection with multiple observers," *IEEE/ACM Transactions on Networking*, Vol. 1, No. 1, pp. 48-55, February 1993.

[16] D. Lee, A Netravali, K. Sabnani, B. Sugla, and A. John, "Passive Testing and Applications to Network Management," *Proc. of the IEEE International Conference on Network Protocols,* pp. 113-122, Oct. 1997.

[17] M. Tabourier, Ana Cavalli, and Melania Ionescu, "A GSM-MAP Protocol Experiment Using Passive Testing," *Lecture Notes in Computer Science,* Vol. 1708, pp. 915-934, Springer-Verlag, 1999.

[18] S. Sankar and M. Mandal, "Concurrent runtime monitoring of formally specified programs," *IEEE Computer,* pp. 32-41, March 1993.

[19] M. Blum and H. Wasserman, "Reflections on the pentium division bug," *IEEE Transactions on Computers,* Vol. 45, No. 4, pp. 385-393, April 1996.

[20] D.B. Brown, A. Porter, C. Puchol, J. C. Ramming, and L. Votta, "An automated oracle for software testing," *IEEE Transactions on Reliability,* Vol. 41, No. 2, pp. 272-279, June 1992.

[21] J. R. Connet, Edward J. Pasternak, and Brude D. Wagner, "Software defenses in real-time control systems," *Fault-Tolerant Computing,* pp. 94-99, June 1972.

0-7695-0990-8/01/$10.00 (C) 2001 IEEE