INTERACTIVE BLACKBOX DEBUGGING FOR CONCURRENT LANGUAGES

by

G.S. Goldszmidt, S. Katz and S. Yemeni

Technical Report #469

November 1987
Interactive Blackbox Debugging for Concurrent Languages

GERMAN S. GOLDSZMIDT, SHMUEL KATZ
Computer Science Department, Technion, Haifa
and
SHAULA YEMINI
IBM T.J. Watson Research Center, Yorktown Heights, New York

Abstract.

We describe a novel approach to the design of portable integrated debugging tools for concurrent languages. Our design partitions the tools set into two categories. The language specific tools take into account the particular features of a programming language for on-line experimenting with and monitoring of distributed programs. The language independent tools support off-line presentation and analysis of the monitored information. The separation of the language independent tools from the language specific tools allows adapting the tools to support debugging for a broad category of concurrent programming languages. The separation of interactive experimentation from off-line analysis provides for efficient exploitation of both user time and machine resources. We present an overview of our design and describe the implementation of a prototype debugging facility for OCCAM.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming;
D.2.5 [Software Engineering]: Testing and Debugging- Debugging Aids;

General Terms: Experimentation, Verification, Portability

Additional Key Words and Phrases: Distributed debugging, language independence, temporal logic
1. Introduction.

Programming concurrent systems is much more difficult than programming sequential ones. The additional complexity is due to the fact that there are multiple processes in a dynamically varying configuration, with multiple threads of control, interactions among these processes via message passing and/or by concurrent access to shared data, global and local nondeterministic control structures, and the possibility of deadlocks and starvation. Because of this added complexity, the need for debugging tools for concurrent systems and the capabilities required from such tools are much greater than for sequential systems.

Several works have addressed the design and/or implementation of debugging facilities for distributed systems. In Smith's [16] Spider system, the user can program daemon processes to monitor, modify and record inter-process events. It was designed as an extension of the Unix(†) kernel. Schiffenbauer [15] describes an interactive experimentation facility where the user can create, modify, delay or delete interprocess messages for a network of Alto computers over an Ethernet. In the debugger of Baiardi et al. [1] the programmer supplies a priori a description of the (ECSP) program’s expected behavior, using behavioral specifications. The execution time behavior of the program is then compared at runtime with the behavioral specifications, and predefined actions are performed whenever a difference between them is detected. Garcia Molina et al. [3] proposed the use of relational languages (e.g. SEQUEL) to query the traces of a distributed system which are organized as a distributed database. Robbins and LeBlanc’s [14, 10] Radar is a tool that displays a post-mortem replay of interprocess events in Pronet programs on a bit-mapped display as in a video movie, where processes are depicted as boxes and messages are represented by dots flowing between them. Joyce et al. [8] developed a monitoring system within the Jade programming environment. Their system provides user control of nondeterminism, textual and graphical presentation of the execution, statistics on interprocess interactions, a deadlock detector and a run time protocol checker.

Our approach is based on separating the design of a debugging facility into two different categories of tools: language specific, i.e., those which must be tailored to each particular language and language independent, i.e., those that can be used for a large class of languages. The language specific tools are those that provide functions for interactively experimenting with the program and recording the resulting runtime computation. The language independent tools do not directly interact with the implementation of the source language; their main purpose is to provide functions for displaying, querying and analysing the recorded computation. Since the relevant runtime behavior events are practically the same across a wide range of concurrent programming models, this separation makes it possible to port the analysis tools not only to all implementations of a particular language, but also to a wide range of other concurrent programming languages.

(†) Unix is a trademark of AT&T Bell laboratories
The main components of the language specific tool set are: 1) A source to source program transformer, that given a program and a specification of the set of events to be traced, generates: (i) a program that in addition to the original program's functionality, communicates the occurrence of the traced events to a monitor process; (ii) the monitor process itself, which records the communicated events in the database of the debugger; (iii) source-level information e.g., symbol tables. 2) a scheduler, which allows the user to determine the scheduling order of processes; and 3) a driver process that allows the user to simulate the program's external environment and to force particular choices in nondeterministic control constructs.

The language-independent tool set supports the display and analysis of the event information recorded in the database. It includes: 1) a temporal logic assertion checker which checks assertions on the recorded event sequences; 2) a replay tool that supports replaying event sequences, in order to help the user follow the evolution of the computation; and 3) a query tool which enables querying states in the computation.

Our debugging facility provides two modes of operation at execution time. In passive mode, it only records events performed by the program in its database, thus reducing the interference of the debugger with the execution of the program being debugged. In active mode it gives the user the capability of experimenting with the scheduling of processes, simulating external communication and selecting particular choices in nondeterministic constructs.

The choice of the name "blackbox debugging" comes from the similarity of our off-line analysis to the approach taken in aircraft monitoring. First all significant events are recorded in a "black box", in a way which interferes minimally with the normal execution. This blackbox recording could be initiated whenever problematic behavior is suspected, or left on indefinitely. The recorded information can later be analyzed to detect the source of errors. Such an approach also supports remote debugging. When released software malfunctions on site, the recording of events can be "switched on", and its output can be routed through a network to the maintenance site, where the programmers could apply the analysis tools to discover the cause of the error.

Our system most resembles [8] in that they also advocate separating monitoring from analysis. However our work differs from the Jade approach by supporting the monitoring of both interprocess and local events, and providing greater control over scheduling (e.g., by introducing priorities). Rather than checking assertions during execution, we provide a temporal logic assertion checker for the analysis of a program's behavior after execution has been interrupted. Our source level transformation approach also allows a greater independence from the programming environment, and automatically inserts code for the monitoring.

We have studied the applicability of our design for two different concurrent programming
languages, OCCAM(1) [7] and NIL [6], which together represent a wide spectrum of concurrent programming languages. A prototype system for OCCAM was implemented. Its complete description can be found in [4].

The rest of this paper is organized as follows: Section 2 describes the design goals, Section 3 includes an overview of the debugger, and Section 4 defines events and describes the debugger's database. The language specific parts of the debugger are described in Section 5, while Section 6 presents the language independent tools. Section 7 evaluates the suitability of our design for two different concurrent programming paradigms. Section 8 contains concluding remarks and a brief description of a prototype debugging facility for OCCAM.

2. Design Goals.

Four major design goals were established for our debugging environment: abstraction, transparency, interactivity, and portability.

Abstraction. The ability to abstract from the detailed state of a program is a fundamental capability of any debugging facility. The person debugging a program needs information representing the state of the program at the source language level, i.e., symbolic debugging, and even at a higher, specification-related level. However, many debugging systems provide only low-level details of the state of the program, leaving to the user the mapping of these details to higher level abstractions. For large programs the user also needs aids to focus attention on designated parts of the program such as the internal state of a process and inter-process communications, and also means to select the appropriate level of detail of the information which the debugger presents for user examination.

Our debugger provides information both at the source language level and at the more abstract level of temporal logic behavioral specifications. It also provides means to select relevant state information from the recorded execution and present it at various levels of detail. It conceals from the user low-level machine dependent information such as contents of memory locations, registers, etc.

Transparency. A debugger for a concurrent language is transparent when the set of computations potentially observed during program execution are identical both in the presence and in the absence of the debugger. This means that the debugger itself does not affect the computation of the program being debugged. Debuggers of sequential programs can insert checks at any point in the computation without perturbing it. This is because the behavior of a sequential program is not affected by the delay between sequential instructions. However, in distributed applications the presence of debugging checks can affect the behavior of the program, so that certain possible computations (interleavings) may not occur during debugging sessions. For example, when a process has to choose among a set of process communications in a nondeterministic construct, if any of the potential communicating partners are delayed by debugging checks, then the set of alternatives considered by the implementation will probably be affected. In this

(1) OCCAM is a trade mark of the INMOS Group of Companies
example the debugger can interfere with the "natural" behavior of the processes, by providing unfair opportunity to some of them. If the debugger always delays the same subset of processes, it may be masking an execution sequence which can contain an error.

In both active and passive modes our debugger supports transparency. In active mode, the user has interactive control over the interference of the debugger, since the system allows the user to force particular choices in nondeterministic control constructs and particular schedulings, thus allowing the user to determine the desired interleaving. In passive mode, the debugger only records the events in the order in which they occur, thus minimizing the interference of the debugger with the execution of the program being debugged.

**Interactivity.** Because of the complex interactions of processes in a program with one another and with the external environment, and because of the inherent nondeterminism of concurrent systems, it is essential that a debugger provide facilities for the user to interact with the system during execution.

In our debugger, the user can interact with the system to simulate the external environment with which a process or a set of processes interacts by supplying inputs on input channels and examining the outputs on output channels. This also makes it possible to perform unit testing of a process before the rest of the system with which it is to interact has been implemented. The user can also interactively influence the ordering of otherwise nonordered events such as process scheduling and selections in nondeterministic control constructs. The scheduling of the processes can be simulated, thus allowing a simulation of different hardware configurations.

**Portability.** A major goal of the project was to design debugging facilities which are portable, i.e., independent of any particular implementation of the source programming language. Most debugging facilities are strongly embedded within the implementation of the runtime environment, usually just on top of the operating system. Many debugging facilities are even dependent on a particular machine instruction set, making them non-portable for new hardware architectures. Related to portability is the concept of reusability. The design should be general enough to allow the reuse of debugging tools for many programming languages. It should also be modular, in a way that permits both modification and extension of its features.

Our interactive debugging tools are written in the same languages as the programs being debugged, and use the communication facilities provided by these languages to interact with the programs. This use of the languages' high-level communication facilities eliminates the need for dependencies on particular language implementations. Our off-line analysis tools are implemented in a portable widely available high-level language, Prolog [2].

3. Overview of the Debugging Facilities.

Figure 1 shows a schematic view of the debugger's architecture. It shows the relationships between the modules of the debugger: Transformer, Monitor, Scheduler, Driver, Timestamp, Database, Replay,
Figure 1. The architecture of the debugger.
Queries and Assertion Checker, and the standard modules provided by the programming environment: Editor, Compiler, Application Execution Environment. This architecture is intended for designing a debugging environment for any concurrent language. The services provided by each of these modules will be explained in the following sections.

The debugger provides facilities for gathering execution time information, experimenting with the system, presenting the information and analyzing it. The facilities which must be tailored to the particular language, i.e., the language-specific facilities are: 1) generating code for automatic tracing of events (transformer); 2) experimenting with the scheduling priorities of the processes (scheduler); 3) suspending an execution (monitor); 4) simulating the environment of individual processes (driver); and 5) forcing particular choices in nondeterministic constructs (driver). Those facilities that display and analyze the recorded sequence of events and are therefore language independent are: 1) simulating a replay of (all or only specific kinds of) events (replay); 2) checking temporal logic assertions on the recorded event sequence (assertion checker); and 3) querying the state of the program at any place in the computation (queries).

A typical debugging scenario using these facilities might be:

(a) The user annotates the program with: 1) program variables and labels to be traced, 2) breakpoints, and 3) scheduling priorities for the processes.

(b) When the debugger is invoked, it first calls the source to source program transformer, to transform the program's source code to a program that has the same functionality as the original, but with an additional monitor process and additional communication statements for reporting events to the monitor process.

(c) The new code is compiled and then executed.

(d) During execution all recorded events are reported to the monitor, and recorded in the debugger's database. The user can interactively modify the scheduling priorities of the application's processes, by pressing a key to start an interactive dialogue with the scheduler.

(e) From time to time execution may be suspended, either because a preset breakpoint is reached, or because the user explicitly requests suspension by pressing a key.

(f) Whenever execution is suspended or after normal termination, the user can analyze the computation by: 1) displaying the event history by invoking the replay tool; 2) querying the state of the computation by invoking the queries tool; or 3) providing temporal assertions to be checked by the assertion checker. After suspension the system can either be aborted or resumed.

(g) After discovering an erroneous process(es), the user can focus the debugging activity on it (them). The user can execute the process with an artificial interactive environment that simulates its (their) environment and allows to force selections of particular choices in nondeterministic constructs, by invoking the driver.
The above scenario describes a top-down debugging strategy in which the entire program is
debugged. Our tools can also assist in applying a bottom-up strategy, in which processes are first
debugged individually, as proposed in [3].

4. Events and the Debugger's Database.

We assume a concurrent system consisting of a dynamically varying collection of processes which
communicate by message passing, where message passing is either synchronous or queued, or by remote
procedure calls. We consider the state of such a system to consist of the collection of internal process
states (IPS), and the global system state (GSS). The IPS for each process consists of the values of its
local variables and its program counter. The GSS consists of a status variable for each process in the sys­
tem, indicating whether that process is running, waiting, aborted or terminated, and a list of ongoing pro­
cess interactions, e.g., sending or receiving a message or making a remote call.

Events are actions that modify the system state, such as instantiating / terminating a process, modi­
fying a variable, reaching a label, sending / receiving a message or making a call. Events serve as the
interface between the language specific and language independent sets of tools. We distinguish between
the events which affect an internal state (IPS events) and those which affect the global state (GSS events).
This separation provides a framework for a higher level of abstraction of the program's activities.

All GSS events must be tracked since they are obviously needed to analyze and understand the com­
putation of a concurrent program. Thus, our debugger automatically records their full history, i.e., the
complete sequence of changes to the GSS. In contrast, for IPS events it suffices to track only those of
relevance to a particular debugging session. Thus, we allow the events in the history of the IPS to be
recorded at the discretion of the user. The user specifies the variables and labels that should be traced,
and the debugger automatically generates code to record each modification of a traced variable and each
time that control reaches a traced label.

Each occurrence of a significant event during the execution of the program is recorded in the
debugger's database (the blackbox). This record will later be examined using the presentation and
analysis tools to analyze the program's behavior in that particular computation.

Our prototype debugger [4] is based on four types of events: 1) assignments to variables; 2) reach­
ing a label; 3) interprocess communication; and 4) process instantiation or termination. Obviously, each
programming language would have its particular set of events. However, it appears that the above four
categories are both universal, and are a minimum set from which a description of the complete state of the
program can be recreated at any point in its computation, using the record of the events and the source
code.

Each recorded event has the following information associated with it: 1) Event Type, which is one
of the predetermined kinds of events for the language; 2) Process Id, which identifies the instance of the
process that performed the event; 3) Source Code Line, which determines the line of the source code that
Interactive Blackbox Debugging for Concurrent Languages

corresponds to the event; and 4) *Timestamp*, which gives a consistent total ordering to the recorded events. The timestamp algorithm used in the prototype is based on reordering the monitor's output stream of events to reflect causality. Its details can be found in [4]. Depending on the type of the event, its description may include additional parameters, e.g., the name of the variable or label being traced, the name of the channel used for communications, etc. In the prototype, events are recorded in compressed format, to reduce both the amount of storage needed for the database and the time needed for I/O. The database includes symbol tables for mapping the source program's identifiers to their internal indexes.

The user's ability to select the IPS objects to be traced allows abstracting the relevant events from the detailed behavior of the program. It also helps to improve the performance of the debugger, by reducing both the runtime overhead and the amount of storage needed for recording events into the database. We have chosen to have our tools automatically generate the code needed for tracing the events, rather than let the user manually insert the corresponding output statements. Manually inserting these statements would be tedious, inefficient and even error-prone, since the user may forget to insert statements at critical points.

The following two sections describe the tools of our debugger. A small example of a simple OCCAM program will be used to demonstrate some of the facilities that the tools provide. A skeleton of the program appears in the appendix. The program has four processes executing in parallel: Two producers, a merger and a consumer. After an (unspecified) process called "initialize" completes its computation, the four processes are instantiated in parallel. Both producers send values to the merger which adds them and passes the result to the consumer. The merger process (addmerge) waits in a nondeterministic construct (ALT) for input from either of the producers. A bug was introduced to demonstrate the use of our tools. The merger expects the producers to send messages alternatively. Due to different timing conditions producer I will send two consecutive messages. The first one will be lost and the second one will be merged with the last message received from producer2. Note that the symbol '!' is OCCAM's send statement and '?' is the receive statement. PAR denotes the instantiation, in parallel, of the following processes, while SEQ denotes a serial execution of the following statements.

5. The Language Specific Tools.

In this section we describe the parts of the debugger which must be tailored to each language for which a debugger is to be implemented. These include: 1) The source to source *Transformer*, which adds to the source program i) communication statements that report the occurrence of the traced events to a monitor process, and ii) code for the monitor process itself; 2) The *Scheduler* which allows the user to interactively modify the scheduling of processes, thereby experimenting with different interleavings of a concurrent program, and 3) The *Driver*, which allows the user to simulate the external environment interacting with a process (or a set of processes) being debugged, and to force decisions on nondeterministic constructs.
5.1 The Transformer.

The transformer is a preprocessing tool. It takes as input the user source program (USP), and produces an augmented program (AP). AP has the same functionality as USP, but in addition it includes output statements that report the occurrence of the traced events to a monitor process, and statements that implement breakpoints and scheduling instructions. The instructions to the transformer are presented as comments inserted in the source code, so that debugging can be turned on or off without having to modify the program.

The debugging instructions supported in the prototype version of the debugger appear as comments on separate lines: 1) The line \--TRACE indicates that the variables declared on the following line are to be traced; 2) \--LABEL inserts labels(t) at the specified location in the source code and reports whenever such a label is reached during execution 3) \--PRIORITY p assigns static scheduling priority p to the process containing this statement (the interpretation of p is explained in the next section); and 4) \--BREAK suspends execution when control reaches the designated point in the program. The transformer also produces the symbol tables required for interpreting the recorded event information, as mentioned in the previous section.

The monitor is a process which executes concurrently with the augmented program, and whose main function is to receive reports of events from the program and then record them in the debugger's database. The reporting of an event blocks the reporting process, i.e., the process cannot continue executing until the report has been received by the monitor. Thus, by refusing to accept messages from the program, the monitor can suspend the program's normal operation, simulating a breakpoint and giving the user the opportunity to examine and analyze the program's state. The monitor suspends an execution either in response to the user pressing the key designated as the break key or the program reaching a preset breakpoint. Note that while this does not implement an instantaneous halt of the distributed system, it does not allow any process to progress beyond its next traced event. After such a break has occurred and the analysis has been completed, the monitor (and thus the program) can resume normal operation upon the user's request. The monitor is implemented as a generic process. In our OCCAM debugger prototype, the generic part is an array of channels. The monitor is linked to the augmented program and instantiated together with it.

An example of a source program (USP) and the corresponding transformed code (AP) for a single process are given in Figures 2a and 2b respectively. Italic letters denote the instructions to the transformer supplied by the user and inserted as comments. Bold letters denote the resulting additions to the code made by the transformer. Pid is the process instance unique identifier. The static scheduling priority of the process is 5 as seen on line 37. The reports of events are given to the monitor through a channel identified as monitor[pid]. These are messages with the following fields: Event type ('s' for start,
'a' for assignment, 'i' for input, 'o' for output, 'l' for reaching a label and 't' for termination). Source code line, and event-type dependent fields for identifiers (given as the corresponding index of the symbol table) and values.

\begin{verbatim}
37. --PRIORITY 5
38. PROC addmerge =
39. --TRACE
40. VAR x, y, ready:
41. SEQ
42. ready := 0
43. WHILE x <> End_of_stream
44. SEQ
45. ALT
46. c1 ? x
47. SKIP
48. c2 ? y
49. SKIP
50. ready := ready + 1
51. IF
52. ready = 2
53. SEQ
54. c3 ! x + y
55. --LABEL labelA
56. ready := 0
57. --BREAK
58. SKIP:
\end{verbatim}

Figure 2a. Transformer’s Input

\begin{verbatim}
38.1 PROC addmerge (VALUE pid) =
38.2 SEQ
38.3 scheduler.priority[pid] ! 5
38.4 monitor[pid] ! 's'; 38; 4
40. VAR x, y, ready:
41. SEQ
42. ready := 0
42.1 monitor[pid] ! 'a'; 42; 3; ready
43. WHILE x <> End_of_stream
44. SEQ
45. ALT
46. c1 ? x
46.1 SEQ
46.2 monitor[pid] ! 'i'; 46; 3; 1; x
47. SKIP
48. c2 ? y
48.1 SEQ
48.2 monitor[pid] ! 'i'; 48; 4; 2; y
49. SKIP
50. ready := ready + 1
50.1 monitor[pid] ! 'a'; 50; 3; ready
51. IF
52. ready = 2
53. SEQ
54. c3 ! x + y
54.1 monitor[pid] ! 'o'; 54; 5; x + y
55.1 monitor[pid] ! 'l'; 55; 1
56. ready := 0
56.1 monitor[pid] ! 'a'; 56; 3; ready
57.1 Break[pid] ! ANY
58.1 monitor[pid] ! 't'; 58; 4
58.2 SKIP:
\end{verbatim}

Figure 2b. Transformer’s Output

5.2 The Scheduler.

The scheduler is a process that gets scheduling input from the user and accordingly instructs the monitor which process to serve next at each point. Scheduling input is in the form of: 1) static
Assignments of priorities to processes; or 2) dynamic modification of priorities; or 3) specific instructions from the user as to which process should be scheduled next.

The scheduler permits the user to experiment with different schedulings by assigning different relative speeds to the processes in the program. It is based on the fact that the monitor polls the processes in round robin fashion to receive reports of events. Since processes get blocked waiting to report their events, the service frequency of the monitor determines their scheduling. This frequency can be modified by the user using the scheduler, to give more or less frequent service to each of the application processes.

The priority of a process indicates the number of times that the process will not be served by the monitor. Thus a process with priority of 5 would mean that the process is served once every six rounds. The default policy is that all processes have the same priority. An example interaction with the scheduler is given in Figure 3.

Since all significant events are reported, and since the monitor can block at each event report, the scheduling can be modified arbitrarily at each event report. Thus, the granularity of "context switching" is defined by the distance between traced events. Since only a subset of the traced events i.e. those that modify the GSS, actually influence the program's degree of interleaving in any meaningful way, this granularity is sufficient to obtain any possible interleaving. The delay caused by the monitor will never make it impossible to reveal an error, since the user is given the possibility of forcing the application to perform any legal interleaving of the processes, either by assigning scheduling priorities or by specifying the scheduling himself.

Of course, it is not guaranteed that an execution which reveals a particular error will be chosen. By giving the user control over the scheduling of the processes, the debugger provides effective control over its own effect on the application. In other words, the user is able to control the Heisenberg effect, i.e.,
that the measuring tool affects the subject being measured, which is inherent to debugging facilities for distributed systems [1, 8, 14, 15].

Testing a program under different schedulings can be very useful for applications which are developed in one particular hardware/software configuration and then deployed in another. The programmer will need to test the behavior of the program under different (simulated) timing conditions which are related to the target configuration.

5.3 The Driver.

The driver tool provides an artificial environment to run a single process (or a set of processes), without the rest of their environment. This allows testing a process in isolation, even before the rest of their environment has been implemented, thus supporting a bottom-up debugging strategy [3].

The process' real environment is simulated by the user. The driver prompts the user for the input expected by the current process, i.e., on which channels the process wants to receive messages, so that the user can supply the required inputs. All the process' output messages are passed to the terminal screen, and the user supplies all of the input messages that are expected from the environment.

The driver also provides the ability to force a process to make particular decisions in nondeterministic constructs. The user is presented with a description of the possible entries of the construct and the

```
-> DRIVER addmerge
ddmerge is waiting for input at:
1. c1 ? x or 2. c2 ? y
Select entry:
-> 2
Enter input for c2:
-> 12
addmerge is waiting for input at:
1. c1 ? x or 2. c2 ? y
Select entry:
-> 1
Enter input for c1:
-> 8
addmerge:
c3 ! 20
::
```

Figure 4. Interacting with the driver
values of the local guards (if any), and is requested to decide which path the process should take. Non-deterministic selections which depend on interprocess communications are a known source of errors, particularly timing errors, therefore it is very important that debugging facilities provide options for experimenting with them.

Figure 4 shows an example of interaction with the driver. Note that the user is simulating the environment of process addmerge. Thus, the user is prompted to select the channel on which addmerge is to receive input. In this example, c2 is selected; the user is then prompted for the value of this input, and the user chooses 12. The output sent from addmerge on channel c3, i.e., the value 20, is then displayed to the user on the screen.

The driver is generated from the source program and a generic template during the preprocessing phase. The driver more than any other of our tools is heavily dependent on the particular semantics of the language for which the debugger is designed. In particular the semantics of communication, e.g., whether communication requires simultaneous agreement between both communicating partners or whether messages are queued, has a major impact on the design of the driver.

6. The Language Independent Tools.

Based on the information recorded in the database, the language independent tools help the user review and analyze the behavior of the program. They include: 1) the assertion checker, a tool to analyze the behavior of a computation using assertions formulated in temporal logic on the sequences of events recorded on the database; 2) the replay tool to present the recorded computation, displaying its evolution event by event; and 3) the queries tool which permits the user to formulate queries on the current state of the system at any point in the recorded computation.

6.1 Checker of Temporal Logic Assertions.

The temporal logic assertions checker allows expressing assertions to be checked on the system’s history of execution, i.e., the sequence of events recorded by the monitor in the debugger’s database. The assertions are formulated in linear time temporal logic [12], which is a logic for assertions about sequences and is well suited to specify the behavior of distributed programs. Since the recorded events are always a finite sequence, the checking procedure will always terminate.

The checker recognizes two predefined time variables, which represent the upper time bound of the event sequence and the "present time", i.e., the place in the sequence where assertions are checked. These two variables define the range of the sequence of events to be considered, also called the time interval of interest. The values of these variables can be modified, so that the interval checked can be reduced or enlarged. For convenience, assertions can be given names and then be used to build more complex assertions.

The assertions include simple assertions on the system’s state using regular logic connectors, and
assertions with any combination of temporal logic operators. The tool provides builtin boolean functions which aid in expressing desired properties of the sequence. Since the particular events distinguished by the debugger and their corresponding primitive analysis functions depend on the programming language for which the debugger is intended, we describe here those appropriate for the primitive events described in Section 4.

The predicate $\text{running}(P)$ is true iff at the specified time, process $P$ is running. The predicate $\text{terminated}(P)$ is true iff at the specified time, $P$ has already terminated. A builtin function $\#B$ returns the number of states in the computation up to the present time, which satisfy the predicate $B$. Some predicates relate directly to the previously executed event. Thus $\text{receive}(P, C, V)$ is true when a message has just been received at process $P$ on channel $C$ with value $V$, and false otherwise; when $V$ is irrelevant, the third parameter may be omitted. The predicate $\text{at}(P, L)$ is true iff the last event recorded from process $P$ implies reaching the label $L$. "Var $\leftarrow$ val" is a binary predicate which is true if Var was assigned the value val in the previous event. The complete list of predefined predicates is given in [4]. The standard relational operators ($=, \neq, >, \geq, <, \leq$) can be used in formulating assertions. In addition, temporal assertions can use the operators: "$ \Rightarrow "$ (Implies), "$ \leftrightarrow "$ (If and only if), "$ \land "$ (And), "$ \lor "$ (Or), "$ \neg "$ (Not), "$ \Box "$ (Always), "$ \Diamond "$ (Eventually), "$ \ominus "$ (Next) and "$ U "$ (Until).

Figure 5 shows six assertions related to the merger in our example program. The first three implicitly refer to the state at the "present" time, while the last three relate to the time interval of interest, i.e.,

\begin{align*}
\text{at}(\text{addmerge}, \text{labelA}) & \Rightarrow (\text{ready} = 2) \\
\text{send}(\text{producer1}, c1, 3) & \land \text{receive}(\text{addmerge}, c1, 3) \\
& \{ \text{3 has just been transmitted from producer1 to addmerge} \}
\end{align*}

\begin{align*}
(\text{running(producer1)} & \lor \text{running(producer2)}) \Rightarrow \text{terminated(initialize)} \\
& \{ \text{If producer1 or producer2 are running then initialize has terminated} \}
\end{align*}

\begin{align*}
\Diamond (\text{terminated(\text{addmerge})} \land (x = \text{End_of_stream})) \\
& \{ \text{Eventually addmerge terminates and } x = \text{End_of_stream} \}
\end{align*}

\begin{align*}
\Box ((\text{ready} \leftarrow 2) \Rightarrow \Diamond \text{send(\text{addmerge}, c3, x + y)}) \\
& \{ \text{Always if ready is assigned 2 then eventually addmerge will send } x + y \text{ on channel c3} \}
\end{align*}

\begin{align*}
\Box \text{at}(\text{addmerge}, \text{labelA}) & \Rightarrow (\#\text{receive(\text{addmerge}, c1)} = \#\text{receive(\text{addmerge}, c2)})) \\
& \{ \text{Always if addmerge is at labelA then the number of messages received from c1 and c2 are equal} \}
\end{align*}

Figure 5. Assertions.
6.3 Queries.

The queries tool allows the user to formulate queries about any state of the system. Queries involve the values of variables and the states of processes. Our debugger supports querying of the following: 1) The value of a variable; 2) The execution state of a process, e.g., whether it is running or terminated; 3) The value communicated on a particular channel; 4) The Internal Process State of a process; 5) The Global System State.

The queries tool eliminates the need to manually search through the recorded computation to extract the state at any particular point of interest. The points in the computation at which the user requests to query the state can be given either explicitly by a timestamp or they can be specified implicitly, using assertions. In the latter case the query will be checked at the first place in the event sequence where the assertion holds. For example, the user may request the IPS of a process (its last event and the values of its traced variables) when a variable gets some value, when the process terminates, etc.

---

GSS when time=108

<table>
<thead>
<tr>
<th>(Pid)</th>
<th>Process</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>initialize</td>
<td>terminated</td>
</tr>
<tr>
<td>(2)</td>
<td>producer1</td>
<td>running</td>
</tr>
<tr>
<td>(3)</td>
<td>producer2</td>
<td>running</td>
</tr>
<tr>
<td>(4)</td>
<td>addmerge</td>
<td>running</td>
</tr>
<tr>
<td>(5)</td>
<td>consumer</td>
<td>not started</td>
</tr>
</tbody>
</table>

IPS(addmerge) when x + y = 11

<table>
<thead>
<tr>
<th>Time</th>
<th>Line</th>
<th>(Pid)</th>
<th>Process</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>115</td>
<td>46</td>
<td>(4)</td>
<td>addmerge</td>
<td>c1 ? x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
</tr>
<tr>
<td>ready</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6. Display of Queries tool.
Figure 6 displays two queries. The first one requests the Global System State at a given timestamp (108). The second one presents the Internal Process State of addmerge when an assertion \(x+y = 11\) becomes true.

6.2 Replay.

The Replay tool displays on the terminal screen a simulated replay of the events recorded during execution in the debugger's database to aid the user in visualizing the evolution of the computation. The tool displays the line of the source code of the process where the event occurred, the process instance identifier, the process name and a description of the event. The recorded events are displayed on the

<table>
<thead>
<tr>
<th>Time</th>
<th>Line</th>
<th>(Pid)</th>
<th>Process</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>16</td>
<td>(2)</td>
<td>producer1</td>
<td>Started</td>
</tr>
<tr>
<td>102</td>
<td>38</td>
<td>(4)</td>
<td>addmerge</td>
<td>Started</td>
</tr>
<tr>
<td>103</td>
<td>42</td>
<td>(4)</td>
<td>addmerge</td>
<td>ready := 0</td>
</tr>
<tr>
<td>104</td>
<td>18</td>
<td>(2)</td>
<td>producer1</td>
<td>c1 := 3</td>
</tr>
<tr>
<td>104</td>
<td>46</td>
<td>(4)</td>
<td>addmerge</td>
<td>c1 := x</td>
</tr>
<tr>
<td>105</td>
<td>50</td>
<td>(4)</td>
<td>addmerge</td>
<td>ready := 1</td>
</tr>
<tr>
<td>106</td>
<td>76</td>
<td>(3)</td>
<td>producer2</td>
<td>Started</td>
</tr>
<tr>
<td>107</td>
<td>82</td>
<td>(3)</td>
<td>producer2</td>
<td>c2 := 4</td>
</tr>
<tr>
<td>107</td>
<td>48</td>
<td>(4)</td>
<td>addmerge</td>
<td>c2 := y</td>
</tr>
<tr>
<td>108</td>
<td>90</td>
<td>(5)</td>
<td>consumer</td>
<td>Started</td>
</tr>
<tr>
<td>109</td>
<td>50</td>
<td>(4)</td>
<td>addmerge</td>
<td>ready := 2</td>
</tr>
<tr>
<td>110</td>
<td>54</td>
<td>(4)</td>
<td>addmerge</td>
<td>c3 := 7</td>
</tr>
<tr>
<td>110</td>
<td>96</td>
<td>(5)</td>
<td>consumer</td>
<td>c3 := z</td>
</tr>
<tr>
<td>111</td>
<td>56</td>
<td>(4)</td>
<td>addmerge</td>
<td>at labelA</td>
</tr>
<tr>
<td>112</td>
<td>56</td>
<td>(4)</td>
<td>addmerge</td>
<td>ready := 0</td>
</tr>
<tr>
<td>113</td>
<td>18</td>
<td>(2)</td>
<td>producer1</td>
<td>c1 := 5</td>
</tr>
<tr>
<td>113</td>
<td>46</td>
<td>(4)</td>
<td>addmerge</td>
<td>c1 := x</td>
</tr>
<tr>
<td>114</td>
<td>50</td>
<td>(4)</td>
<td>addmerge</td>
<td>ready := 1</td>
</tr>
<tr>
<td>115</td>
<td>18</td>
<td>(2)</td>
<td>producer1</td>
<td>c1 := 7</td>
</tr>
<tr>
<td>115</td>
<td>46</td>
<td>(4)</td>
<td>addmerge</td>
<td>c1 := x</td>
</tr>
<tr>
<td>116</td>
<td>50</td>
<td>(4)</td>
<td>addmerge</td>
<td>ready := 2</td>
</tr>
<tr>
<td>117</td>
<td>54</td>
<td>(4)</td>
<td>addmerge</td>
<td>c3 := 11</td>
</tr>
<tr>
<td>117</td>
<td>96</td>
<td>(5)</td>
<td>consumer</td>
<td>c3 := z</td>
</tr>
</tbody>
</table>

Figure 7. Display of Replay tool.
screen according to their timestamp order. Recall that timestamps are assigned to events by the timestamp module in a way which maintains a consistent global order preserving causality [4].

The replay tool provides the following options: 1) singlestepping 2) multistepping 3) skipping and 4) filtering. In singlestep mode the user is prompted for further commands after the display of each individual event. Multistep mode provides a display of one "screenful" of events at a time, where the user can modify the screen size. Skipping forwards or backwards in the event sequence, can be done either explicitly by giving the specific timestamps to be displayed, or implicitly, as the place where a given assertion first becomes true. By filtering events from the event sequence, the user of the tool can focus on the actions performed by the computation on particular events of interest, while excluding other events from the display. For example, the user may want to see the changes in value of some variables, the events performed by some processes, communications exchanged between a particular pair of processes, etc. Figure 7 displays part of the events performed by the four processes of the program. Filters have been set on the processes to exclude other events from the display. The user can clearly see that addmerge received two consecutive messages from channel c1 (113-115).

7. Suitability for Multiple Concurrent Programming Models.

In designing a debugger for concurrent languages, we considered two somewhat different concurrent programming models. Our conjecture was that if our design were found to be suitable for both models, it would accommodate a wide range of concurrent programming languages. The two models we considered are those of NIL and OCCAM. A detailed evaluation of the suitability of our design for each language appears in [4]. Here, we will provide only a brief description of each language to point out the main differences between them. We then explain how despite these differences, the design of our debugger is suitable for both, and therefore we believe it provides a basis for debuggers for other concurrent languages.

NIL is a very high level language for programming large, long-lived distributed systems. Processes are used as the unit of modularity, as well as the unit of concurrency. Processes are serial programs with local data (there is no global or shared data), which communicate solely by message passing. Communication is queued and both no-wait send and remote procedure calls are supported. The configuration of a NIL system may evolve dynamically: processes may be dynamically instantiated and linked into a running system; new communication channels may be established by communicating port connections in messages, and processes may themselves change ownership by being sent in messages to other processes. NIL provides exception handling facilities. In addition to type checking, NIL compilers support typestate checking [17], providing compile-time detection of a large class of semantic errors that in other languages, could be detected only during execution.

OCCAM is a parallel programming language, designed as a calculus for describing concurrent algorithms. Because of the different emphasis, OCCAM is a smaller language than NIL, with few constructs and features. It provides primitive untyped facilities for data structuring, and does not support exception
Interactive Blackbox Debugging for Concurrent Languages

Processes are the unit of concurrency, but the process configuration is determined statically. OCCAM’s process instantiation primitives are embedded within the block structure model. Processes spawned within a parallel clause have a common lifetime: they all start at the same time, and the clause itself terminates when all component processes have terminated. Processes communicate by synchronous unbuffered communication over untyped channels. The binding of communication channel is statically determined by the matching of port names within the scope of a common enclosing parallel clause.

Clearly, such different languages will differ both in the types of events they recognize and in their requirements from a debugger. Because NIL supports exception handling, the set of events recognized by a NIL debugger would include the raising of exceptions. It would appear that in general, a language with more features would have a richer set of events. The fact that very little information is available from a static examination of an OCCAM program text makes it necessary for its debugger to help the user in detecting types of errors that in NIL could be detected automatically by a compiler.

Still, in spite of the difference in emphasis and in computation model between the two languages, their common aspects make it possible to design a debugger framework well suited to both. From the point of view of programming in the large, both languages are process oriented, i.e., the unit of concurrency is the process, and interprocess communication is by message passing. From the point of view of programming in the small, both languages have standard Algol-like sequential constructs, such as assignments, conditionals, loops, etc. In both languages nondeterministic structures are guarded by local expressions and input communications. Thus while their set of events would not be identical, the major events of starting and terminating processes, assigning values to variables, reaching a label or sending or receiving a message are common to both.

In support of our claim that our model is well-suited to other process-oriented languages, note that the Ada process model shares features of both NIL and OCCAM. In Ada, tasks declared in a common block are jointly instantiated, as in OCCAM, while the task access variables behave like the dynamically spawned NIL processes. The binding of ports between processes is determined statically by block structured scope rules as in OCCAM. In spite of the existence of many differences between Ada and NIL and between Ada and OCCAM, its process orientation as well as its being based on the Algol model would seem to make our debugger design well-suited to debuggers for Ada multitasking.

8. Summary and Conclusions.

In contrast to most other debugger designs [1, 14, 15, 16], our design is totally independent of the particular implementation of a language and the operating system on which it is running. Thus, a debugger for a particular language is portable across all implementations of the same language.

© Ada is a registered trade mark of the U.S. Department of Defense.
Interactive Blackbox Debugging for Concurrent Languages

Our tools allow a user to interactively experiment with different executions of concurrent programs by simulating system decisions such as scheduling and nondeterministic selections, as well as simulating other processes with which the program being debugged communicates. The tools also provide powerful high level capabilities to analyze the observed behavior of such programs.

We consider the separation of the activities of gathering execution time information from the analysis of this information as one of the major advantages of our approach. From the point of view of software reuse, this separation makes it possible to use the same analysis tools in debuggers for a wide range of concurrent languages. While the languages vary in their event sets, the analysis to be performed will typically be identical. From the point of view of optimal utilization of both computer and human resources, this separation has several advantages. It allows us to run a single execution trace against an open-ended set of tests, using the assertion checker, the query processor and the replay tool. Since analysis is done off-line, it cannot interfere with the gathering of information, therefore increasing transparency. Finally, this approach allows analysis to take place at a separate site from the one where the malfunction was detected, i.e., it supports remote debugging.

The alternative of analyzing a computation during its execution requires a priori formulation of a description of the intended behavior. The main problem with this approach is the difficulty in formulating specifications in advance in a way which will expose the errors. That this is difficult has been demonstrated in the work on program verification. The user may often find that specifications formulated before the execution of the program did not help to find the cause of an observed error. In this case, the program will have to be reexecuted, possibly from its starting point, with newly formulated specifications. Reproducing the erroneous computation is not trivial since the system executed a particular interleaving, which was influenced by the runtime checks that the debugger performed. If there is no mechanism to reproduce the computation, the programmer will probably see timing errors [5, 15], i.e., errors that appear only for particular interleavings, which will require many executions until they occur again and their causes are clearly understood.

The main disadvantage of off-line analysis is its requirements for large amounts of storage. Still, the space requirements can be reduced by using data compression techniques [4]. In spite of this drawback, our view is that the proposed design provides for better utilization of the programmer's time, which we consider the most important measure of efficiency.

A prototype debugger along the lines we have outlined was implemented for OCCAM. The transformer was implemented in C [9], using a lexical analyzer generator, Lex [11]. The monitor, scheduler and driver are OCCAM processes. The replay tool, the queries tool and the checker of temporal assertions are implemented in Prolog [2], while the debugger's database is in fact a Prolog database. Prolog proved to be an excellent choice for the rapid prototyping of the tools, though the efficiency of our Prolog implementation leaves much to be desired.

The user interface in our prototype is window based and provides simultaneous views of different
Interactive Blackbox Debugging for Concurrent Languages

parts of the programming and debugging environment. The environment includes windows for: 1) the editor with the text of the source code of the program; 2) communicating with the application; and 3) the language independent analysis tools. All the tools are menu-driven.

References:


Appendix.

CHAN c1, c2, c3:

PROC initialize=
::
::

--PRIORITY 4
PROC producer1=
::
v := 3
WHILE v < last
SEQ
  c1 ! v
  v := v + 4
  c1 ! 0
  c1 ! End_of_stream
::

--PRIORITY 8
PROC producer2=
::
u := 4
WHILE u < max
SEQ
  c2 ! u
  u := u + 10
  c2 ! 0
::

--PRIORITY 3
PROC consumer=
::
WHILE z <> 0
SEQ
  c3 ? z
::
::

--PRIORITY 5
PROC addmerge=
--TRACE
VAR x, y, ready:
SEQ
  ready := 0
WHILE x <> End_of_stream
SEQ
  ALT
    c1 ? x
    SKIP
    c2 ? y
    SKIP
    ready := ready + 1
  IF
    ready = 2
  SEQ
    c3 ! x + y
    --LABEL labelA
    ready := 0
    --BREAK.
    SKIP:
::

--MAIN PROGRAM
SEQ
initialize
PAR
  producer1
  producer2
  addmerge
  consumer
::
::

--END OF MAIN PROGRAM