A Petri Net-based Distributed Debugger

An-Chi Liu
Department of Information Engineering
Feng Chia University
Taichung, Taiwan 40724, R.O.C.

Abstract
A distributed debugger based on the Petri net model is designed and implemented. The major functions supported are distributed breakpoints, step-by-step execution, and replay. The debugger consists of a preprocessor which inserts control functions into the source code, and a parser which generates a Petri net model of the distributed program for graphical monitoring and program simulation. The debugger also interfaces with existing sequential program debuggers to provide access to variables.

I. Introduction
Debugging is the most important filtering stage during the program development process. A debugger is designed to help the users to account for the data flow as well as control flow.

Among the functions that a debugger should provide are:

- replay of a previous execution of the process(es) under test
- behavior isolation of faulty software components
- access to data items (variables, pointers)
- creation of execution history
- isolation or retrieval of program states

A distributed debugger is further complicated by issues such as concurrency and interprocess communication.

Distributed program debuggers have been developed for various system configurations. Smith proposed a debugger for message-based communicating processes [1]. The debugger creates, manipulates, and monitor events, where an event is a combination of process states or messages. An event manager is responsible for the capture of events. Bates and Wileden presented an Event Definition Language (EDL) as a support for distributed system debugging [2]. EDL is a realization of Behavioral Abstraction (BA) approach for viewing distributed programs. BA is based upon considering the system activity as a stream of event occurrences. Garcia-Molina et al. proposed a two-phase procedure for distributed debugging, where the first phase uses the trace to identify the problem and if not successful, creates an artificial run-time environment based on the trace to locate the problem [3]. LeBlanc focused on the replay of a distributed program, as an essential function of a debugger [4]. A mechanism called instant replay is proposed to reproduce the behavior of parallel programs. During execution, instant replay saves the relative order (but not the associated data) of significant events. The use of shared objects is recorded in versions. A protocol then controls the access of shared objects to ensure an identical update at replay time. Brindle, Taylor and Martin proposed a debugger for Ada tasking [5]. This debugger simulates multiprocessing via a scheduler, and implements a software clock. This simulated clock provides time distortion for slow execution and breakpoints. Baiardi et al. presented a classification of message-passing call connections, based on which a list of possible communication bugs is derived [6]. The programmer is asked to write a program behavior specification. At run time, the actual program behavior is compared to its specification in order to detect errors. Buhr et al. presented a software CAD tool that includes features common to debuggers [7]. Their methodology is based on the iconic representation of software components. Based on the graphic representation, a source code framework is automatically generated.

In this paper we design and implement a debugger for distributed programs. We emphasize the graphical as well as textual debugging.

A graphic model of a distributed task provides a clear picture of the underlying program organization using a top-down approach. The graphic model serves not only as a representation tool, but also as an active graph, which monitors actively or passively the actual program execution. Petri nets were chosen as a model, because they represent concurrency naturally.
In what follows, Section II discusses our design philosophy; Section III describes the debugger architecture and our implementation; Section IV details the synchronization mechanism to facilitate Petri net monitoring; Section V concludes the paper.

II. Design Philosophy

2.1 Our Approach to Debugging

We consider a distributed program as a mesh of interconnected sequential segments of code. We are more concerned with validating the communication patterns of the distributed program than the behavior of individual segments. For the latter, an existing debugger (e.g., sdb and ctrace on Unix) can be used.

We present the user with a Petri net graph which is an image of the connections between the distributed modules of the program. The graph and the program run in parallel, being synchronized by the debugger central process. It is therefore possible to visualize at controlled speed the execution of the distributed program, and verify the proper sequencing of events.

Most operations are directed from the graph rather from a command line, as for instance, breakpoint manipulation, different modes for running the debugged program, and graphic operations. This yields a friendlier user interface and is closer to a natural representation of parallel processing. The user is totally relieved of any extra work besides the distributed program itself. Program to debugger communication is done by a preprocessor modifying the source code, and Petri net generation and drawing are also automatic. There is no modification in the OS kernel either, which makes our system more portable than EDL or the system described in [1]. The distributed program runs in the target system instead of a simulated distributed environment, as in [5]. In [6] the programmer writes a program behavior specification, which is simple enough to be clearly determined. Then the specification is usefully compared to the real program, where distributed events can be hidden by the complexity of the flow control. However, distributed events can take place within the layers of a communication protocol, which can create in depth a frequent exchange of messages. It is not obvious whether or not the specification can cope with a complex system. Our system offers levels of abstraction: local complex patterns of communication among a set of program modules can be clustered. In fact, the default graph offered to the user is already a clustered view. The programmer can choose which region deserves an expanded view, which will be laid out automatically. We support top-down debugging in another way also, by allowing the Petri net graph to be generated from incomplete code. Since a Petri net is executable, we can first test the validity of IPC calls extracted from a partially complete program. This early verification stage is considered important, since incorrect communication organization can cause major work toward the end of program completion.

2.2 Debugger Functions

The debugging process is conducted by combining the use of the following functions:

- Distributed Breakpoints.

A distributed breakpoint is a global state of the distributed program defined as a set of places on the Petri net model. These places inhibit the firing of all their output transitions. A distributed breakpoint affects only the selected subset of processes in the distributed program. If only one process is assigned firing inhibition, then the programmer simply simulates an abort condition on one of the processes.

- Visual Monitoring.

Visual monitoring offers real-time graphic display of the control flow. The use of Petri nets can further be applied for high-level debugging, e.g., testing the program-wide IPC. With more sophisticated Petri nets (timed, stochastic), such an operation allows to forecast the performance of the program, detect critical races and bottlenecks.

- Replay of a Previous Execution.

The replay occurs after an execution of the program, free from any interaction with the debugger.

- Step-by-Step Execution of the Program.

After the firing of each transition, the debugger waits for the user's input to proceed. This mode can alter the real execution if it is not used with replay.

Since the debugger operates on programs that are to run in a real computing environment, we let the debugger modify the program source files. The modification consists of the insertion of a function, called breakpoint control function (BCF), at strategic places in order to establish a communication link between the debugger and each individual process. The BCF is essential to support the replay and distributed breakpoints.

A typical debugging session would start by a program running free from interaction with the debugger. At that time, execution history of the program is recorded, under the form of a sequence of message exchanges for each process. Memory and processing overhead are minimized in this operation for allowing the program to run in real time conditions. Later, this free execution can be reproduced exactly using the history, in step-by-
step mode, or with distributed breakpoints. Communication between processes is closely monitored by installing a breakpoint both at the sender and the receiver. The breakpoint is placed immediately before or after the IPC call. Using the sequential program debugger, the contents of a message can be viewed. Experiments on the sensitivity of the distributed program to variable computing speed are performed with the Petri net running free of the program. Distributed breakpoints can still be defined to voluntarily slow down some processes.

III. Debugger Architecture

Our system assumes that all programs are written in C [8]. The initial distributed program is written to a single file, that includes some distributed programming directives, such as send/receive calls. The preprocessor divides the user file into as many source codes as independent processes. These automatically generated independent programs are the input to the debugger preprocessor. An alternate input can be a higher level task specification, translatable to C code (not necessarily to a complete program, but with enough statements to model the specification). Here we do not present the higher level specification, but only suggest an interface to our system. The sequence of operations involved in the treatment of a program by the debugger is described in Figure 1.

The three main components are: a program analyzer/parser, the Petri net modeling module, and the program/graph synchronization.

The program parser reads the program, and generates a program dependency tree and a modified source code. From the program dependency tree, the generation of the corresponding Petri net model is straightforward. Figures 2-5 show the complete evolution of a source code during the preprocessing stage.

These operations require three passes to complete:

Pass 1) identifies all the functions appearing in the code. These functions are marked depending on whether or not they participate to distributed events. When constructing the program model, we are only interested in the code that pertains to IPC. We therefore need to know beforehand which function identifiers are going to mark a statement (for or while loops, compound statement, expression, statements, etc.) as participating to IPC.

Pass 2) builds the program dependency tree and generates a modified source code by inserting BCFs at strategic locations in the source code, i.e. before each loop construct, function call, selection, compound and jump statements. Figure 4 illustrates the systematic placement of BCF calls. It is impossible for the second pass to know in advance if the complex statement that is currently being read is going to refer to IPC. We need to systematically perform the insertion. The marker _LCF_ in a function argument list indicates to the third pass that this particular argument is a token and may be removed, if it is found that the function call is not relevant to IPC.

Pass 3) reads the modified source code generated by Pass 2 and removes the redundant BCFs'. Pass 2 was indeed unable to know if the construct that was about to be identified by the parser would participate to IPC. By default, a BCF call was placed. For instance, if the lexer returns the keyword "if", we know what type of construct is coming up, but we do not know its contents yet until it is fully read. The third pass refers to both the program tree and the symbol table. Figure 5 shows the result of the third pass.
main()
{
    int data[C_SIZE], index, i, sort_count;
    GetProcName("PROC1", &index);
    strcpy(RemoteProcess, "PROC2");
    Receive((char *) data);
    StoreNode(0);
    if (quicksort(data, C_SIZE, C_SIZE / 2, MESSAGE) == OUT) {
        sort_count = FINAL;
        monitor();
        Receive(data, 0);
        for (Receive(&sort_count);
            sort_count != FINAL;
            Send((char *) data, C_SIZE, "", 0);)
        return;
    }
    monitor(FINAL);
    Receive((char *) data);
    quicksort(data, C_SIZE, C_SIZE / 2, NO_MESSAGE);
    RetrieveNode(0);
    Send((char *) data, C_SIZE, "", 0);
}

Fig. 2 Original Source Code

PROC0
    Receive
    if (quicksort())
        for
        Send
        main()
        Receive
        Send

PROC1
    Receive
    quicksort()
    Send
    monitor()

PROC2
    quicksort()

Fig. 3 Second Pass: Program Dependency Tree

The simplest form for statements in C is called expression statement. These, together with expressions found in the conditional parts of for/while-loops or if statements, often contain function calls. It is not possible to frame such constructs with a BCF call and keep the program syntax unchanged (and even valid). For all

main()
{
    int index, i;
    GetName("PROC1");
    strcpy(RemoteProcess, "PROC2");
    _Receive(_LCF_1, (char *) data);
    StoreNode(0);
    {
        _BCF_2(0);
        if (quicksort(data, C_SIZE, C_SIZE / 2, MESSAGE) == OUT) {
            _BCF_4(0);
            sort_count[0] = SortCount;
            _Send(_LCF_5, (char *) sort_count, 4, RemoteProcess, 0);
            _BCF_8(0);
            for (Receive(_LCF_7, (char *) sort_count);
                sort_count[0] != FINAL;
                _Receive(_LCF_8, (char *) sort_count)) {
                _BCF_9(0);
                sort_count[0] = SortCount;
                _Send(_LCF_10, (char *) sort_count, 4, RemoteProcess, 0);
                _BCF_0(0);
            }
            _BCF_11(0);
        }
        _BCF_12(0, data, C_SIZE, "", 0);
        _BCF_13(5); return;
    }
    _BCF_4(0);
    _BCF_14(0);
}

monitor(_LCF_15, FINAL);
_receive(_LCF_16, (char *) data);
quicksort(_LCF_17, data, C_SIZE, C_SIZE / 2, NO_MESSAGE);
RetrieveNode(0);
_send(_LCF_18, (char *) data, C_SIZE, "", 0);
{
    _BCF_19(6); return;
}

Fig. 4 Second Pass: Intermediate Modified Source Code

function calls, we add another argument to the argument list of the IPC function, a token number, preceded during the second pass by the _LCF_ marker, to indicate the origin of this argument. In the function definition, we insert as the first statement of the function a BCF call, corresponding to the halt before execution of any statement in the function. Also, before each return statement, or after the last statement of the function if there is no return statement, we insert the second part of the breakpoint control, the one that corresponds to the halt after the execution of the statement calling the function.

Another scheme that would reduce three passes into only two is based on building the tree structure of the complete program already in pass 1. Then, by marking in pass 1 all possible positions for the BCF function, the walk of the syntax tree between pass 1 and 2 would determine which marked positions are to be kept, for their participation to IPC. This approach would reduce processing time because it would save on I/O, but it obliges to build a much larger syntax tree. The trade-off is between memory and performance.

The program dependency tree generated by the parser is limited to the control flow structure of the program. No data representation is present. The nodes of the tree
correspond to the program statements, expressions, functions and distributed modules (i.e. sequential programs). The descendents of a node are the statements or expressions that are encapsulated in the node construct. Expressions (logical, arithmetic expressions) are made of a list of referenced identifiers, themselves function names only.

IV. Petri Net Monitoring

4.1 Petri Net Model

We consider 5 main classes of program statements or components: functions or procedures, loops, conditional statements, jump statements, and interprocess events (process creation or IPC). Every Petri net representation of these components includes one or more of both starting places and ending places, which are used to connect Petri net patterns to each other. For example, a for-loop is modeled as shown in Figure 6.

Fig. 5 Third Pass: Final Modified Source Code

the connection of this Petri net to other model patterns. Tr1 represents a selection between two firing paths, either continuing the loop or exiting it.

Functions that have a recursive structure require some special attention. Using recursion in a program allows to extend in depth the program structure. However, recursion makes the program analysis more difficult, and program representation almost impossible. We attempt to give a convenient Petri net representation of a function calling itself one or more times.

Let us consider a function \( f() \), which can call itself up to \( n \) times sequentially (Figure 7). \( N \) is taken as a maximum, the function will necessary be invoked at least once with no recursion at all. Let us divide \( f() \) into \( n+1 \) code segments, each segment being between two calls to \( f() \). Figure 7 illustrates such a structure, with \( n = 2 \). Calls to \( f() \) are labeled \( c_i \) (i from 1 to \( n \)) and code segments between calls are labeled \( s_i \). The topmost place is the entry point to \( f() \). Each call \( c_i \) is represented by a transition, (whether or not the call will be performed) one output place of the transition going back to the function entry place, the other going to statement \( s_{i+1} \). The “return” is represented by a transition also, which directs the flow back to the statements following the \( c_i \) transitions.

Fig. 6 Modeling of the For-Loop
resume the execution of a former function invocation. However, this order can be determined in two other ways, depending on how we use the Petri net model: (1) If the net is monitoring a running program, returns and branching are queried by the net from the program itself, or (2) If the net is used for program simulation, the $c_t$ transitions are assigned branching probabilities, and the return transitions are correlated to the $c_t$.

![Diagram](image)

**Fig. 7** Recursion Modeling

We illustrate the modeling of the program of Figure 2 with Figure 8.

### 4.2 The Breakpoint Control Function

The operations of the BCF function depend on the selected debugging mode: (1) program history, (2) replay, or (3) distributed breakpoints.

1) **Program History**: for each \( \text{Receive}() \) call, extract and store the token identifying the sending process/statement. The token was appended to the message by the sender during the \( \text{Send}() \) call.

2) **Replay**: for each \( \text{Receive}() \), the debugger reads the next pair of tokens from the program history, and allows the corresponding \( \text{Send}() \) statement to proceed.

3) **Distributed Breakpoints**: Each statement sends a first token to the debugger, updating the state of the model. This token is issued from the BCF that immediately precedes the statement. The debugger lets the Petri net model fire, up to the transition that represents the statement. Depending on whether the program is arrived at a breakpoint, the token will be sent back by the debugger to the

With the first token, we graphically determine the state to which the program arrived, and with the second token, we can tell if the IPC call succeeded (especially important for receive calls).

Once the BCFs' are inserted in the source code, we have an important issue to solve: how does a BCF know to which statement it corresponds? To answer this question requires the BCF to know which is the node of the program tree it maps to. When the BCF is inserted, its argument includes the unique identification number assigned to a statement when building the program tree. The token identification number is unique throughout the entire distributed program. A BCF assigned to a function requires more processing, since the BCF call will be part of the function code and therefore, it is not possible to know what the current function invocation is. The solution is to modify every invocation to the function, and add one more argument to the argument list, i.e. the token number, that in turn will be passed to IPC calls, first and last statements of the function (see section III).

The second problem is related to the fact that the program can be partially modeled, as was mentioned in section II. Therefore some BCF synchronization calls between the running program and the Petri net model...
will have to be ignored, since they would correspond to none of the transitions in the model. How do we determine which tokens a program should expect, and which ones it should skip if the synchronization point is not being modeled? The solution consists of listing first all the tokens to receive, i.e., defining tokens for all the nodes of the program tree. Then from the Petri net model, we list all the tokens that are deliverable by the Petri net, called the list of synchronization tokens. We subtract the elements of the second list from the first list. The third list we obtain is the list of tokens that the program can skip, called the list of permanent tokens. Typically, when a synchronization point is reached in the program, a BCF call takes place, and a certain token is expected. The list of permanent tokens is first scanned. If the token is not found, then we scan the list of synchronization tokens. The program proceeds if the token is there; otherwise, the program halts until the token arrives. Synchronization and permanent tokens are put in separate queues. Once referenced, a permanent token remains in its queue. When a BCF acknowledges a synchronization token, the token is deleted from its queue.

4.3 Synchronization Algorithm
We outline here the algorithm that synchronizes the concurrent executions of the debugger process, monitored program, and Petri net model.

The Debugger

Step 1) Define the current token set.
Step 2) Determine the list of transitions to fire. If the list is empty, stop.
Step 3) Receive the next token from a program statement (say s).
Step 4) Advance the firing up to the statement s. Send to the graphic interface a list of transitions to fire, and a list of transitions not to fire (to allow the modeling of selection statements).
Step 5) Check on the firing authorization of statement s.
Step 6) If this is the first token from statement s, i.e., s has not been executed yet, the token is sent back to the program to resume execution. If it is the second token from statement s, i.e., the statement has been executed, then fire the corresponding transition and send the transition name to the graphic interface.

Graphic Interface

Step 1) Wait for a list of transitions to fire and a list of transitions not to fire. If an empty list is received, this indicates the end of the firing process.

Step 2) Provide visual display on transitions to fire, and movement of tokens from places to places. Transitions not to fire have their input places deprived from tokens.

The Program

Step 1) Reach a statement s controlled by the BCF, entering the opening BCF.
Step 2) Read the permanent token queue. If the token is found, execute s and repeat from Step 1; otherwise, send a token to the debugger and wait for its response.
Step 3) Receive the first token from the debugger and now exit the BCF.
Step 4) Execute s.
Step 5) Enter the closing BCF at the end of s.
Step 6) Send the second token to the debugger and wait until it comes back.
Step 7) Exit the closing BCF function, and repeat from Step 1.

A conditional statement can be presented for execution in the program and possibly in the Petri net model. Since there is no determination of the token flow according to data value in a Petri net, the future firing will be determined by which path the program happens to follow. First, reaching for either a synchronization token or a permanent token will tell if the model cares for the answer back from the program as to what branch of the program was followed. This question depends on whether or not the conditional statement appears in the model. If the query will not be issued from the model, then the synchronization is the same as before. The token is simply read out of the permanent token queue, and the program proceeds. If, however, the model represents the conditional statement, then the firing of the transitions of the statements potentially following the conditional statement will be dependent on a token issued by the program to the model. Execution of step 4) of the debugger part in the synchronization algorithm ensures that the Petri net firing is updated to a state that immediately follows the flow path selection.

V. Conclusion

In this paper, we present the functions and implementation guidelines of a distributed debugger. Several key features are:

- Graphic user interface,
- Automatic generation of the Petri net model of the program,
- Graphical monitoring of program execution,
- No modification of the operating system kernel,
- Interface to the existing sequential debuggers,
- Portable to a heterogeneous environment.
The superposition of the distributed debugger on top of a sequential program debugger allows us to decouple sequential programming from distributed program behavior.

Future work concerns the development as well as improvement of functions in the debugger, for example, use of a program core dump allows a selective rollback with complete data recovery; or adopt an iconic system to overlay the Petri net model to represent the process-to-data relationship. We can also incorporate tools such as deadlock detection for Petri net analysis within the debugger. This analysis coupled to an intelligent processing of the program execution history can help the user determine the cause of a bug, in particular if it came from the system (loss of a message) or from a wrong synchronization scheme.

The distributed debugger serves as the first step to achieve our ultimate goal—establishing a distributed program development environment.

REFERENCES