Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

244

Advanced Programming Environments

Proceedings of an International Workshop
Trondheim, Norway, June 1986

Edited by
Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik

Springer-Verlag

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

244

Advanced Programming Environments

Proceedings of an International Workshop
Trondheim, Norway, June 16–18, 1986

Edited by
Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

Editors

Reidar Conradi
Tor M. Didriksen
Division of Computer Science, Norwegian Institute of Technology
N-7034 Trondheim-NTH, Norway

Dag H. Wanvik
RUNIT/SINTEF, Norwegian Institute of Technology
N-7034 Trondheim-NTH, Norway

International Workshop on Advanced Programming Environments

Organized by: IFIP Working Group 2.4 on Systems Programming Languages
in cooperation with ACM SIGPLAN/SIGSOFT

Sponsored by: Division of Computer Science (DCS), NTH
Computer Centre at the Univ. of Trondheim (RUNIT)
Kongsberg Våpenfabrikk, Branch Office Trondheim
Royal Norwegian Council for Technical and Scientific Research (NTNF), Oslo
Norsk Hydro, Oslo
Norwegian Teleadministration's Research Lab, Kjeller/Oslo
Integrert Databehandling A/S (IDA), Oslo
International Business Machines, Oslo

CR Subject Classification (1985): D.2.6, K.6.3

ISBN 3-540-17189-4 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-17189-4 Springer-Verlag New York Berlin Heidelberg

Library of Congress Cataloging-in-Publication Data. Advanced programming environments. (Lecture notes in computer science; 244) Papers presented at the International Workshop on Advanced Programming Environments, organized by IFIP's Working Group 2.4 on Systems Programming Languages in cooperation with ACM SIGPLAN/SIGSOFT, sponsored by Division of Computer Science (DCS), NTH and other organizations. 1. Electronic digital computers–Programming–Congresses. I. Conradi, Reidar. II. Didriksen, Tor M. III. Wanvik, Dag H. IV. International Workshop on Advanced Programming Environments (1986: Trondheim, Norway) V. International Federation for Information Processing. Working Group 2.4 on Systems Programming Languages. VII. ACM Special Interest Group in Programming Languages. VIII. ACM Sigsoft. IX. Norges tekniske høgskole. Division of Computer Science. X. Series.
QA76.6.A3327 1986 005 86-31536
ISBN 0-387-17189-4 (U.S.)

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

PREFACE

Many Programming Environment (PE) conferences have been arranged over the last years, and more will come. Why this one?

We feel that several of these have failed to bring industry and academia together in a fruitful dialogue, partly because of the topics covered. A traditional conference also leaves too little time for plenary discussions. Since IFIP's Working Group 2.4 on Systems Programming Languages (with 29 members) had planned a meeting on June 10-13 1986 in Trondheim, Norway, some Norwegians conceived the idea to arrange a conjunctive workshop. Trondheim hosts the largest university and research center in computers and electronics in Northern Europe, totalling over 500 scientists. Trondheim in mid-June also represents an attractive site for a combined arrangement.

IFIP WG 2.4 accepted this idea by mid-October 1985, but did not have the time or resources to arrange a traditional conference with invited and refereed papers. A true workshop cannot have a too large attendance, either. We therefore decided on 6-7 major subjects, 30-35 invited speakers, and an additional open, but screened audience of 70-75 persons. Members of the working group were given first priority to attend, in case of space shortage.

By November 15 1985 we had acquired sufficient external support for travel grants and had received indications from ACM SIGPLAN/SIGSOFT about "cooperation". A formal workshop committee was then established and we invited 50 selected persons to submit papers. By January 15th 1986, a preliminary workshop program was set up, based on 35 submitted abstracts - whereof 3 from WG 2.4 members. The authors were instructed to submit a draft paper by April 1st, and a full paper by May 25th. The draft versions have been commented by colleagues in Trondheim and Karlsruhe, at SEI/CMU and Genrad. Final versions of the papers were received by Aug. 1st.

We sent out invitations to about 2000 persons, based on mailing lists from previous ACM conferences and other available lists. The workshop was advertised in ACM SIGPLAN Notices, in ACM Operating Systems Review, and on electronic bulletin boards (SW-ENG). We received 150 sign-ups by mid-April, twice as many as we could accomodate. See the enclosed attendance list.

The workshop committee would like to thank the people who made this arrangement possible: speakers, participants, referees, sponsors, WG 2.4 members and observers, local staff, and NTH's conference secretariat. The editors would also like to thank Sharon Berg and Øystein Valle for proofreading and typing of the discussion transcripts.

Trondheim, November 1986

The editors:

Reidar Conradi Tor M. Didriksen Dag H. Wanvik

Workshop committee

William Waite	Univ. Colo., Boulder	workshop chair, WG 2.4 chair
Mary Shaw	CMU, Pittsburgh	program chair
John Nestor	SEI, Pittsburgh	ass. program chair
Lynn R. Carter	Genrad, Phoenix	WG 2.4 secr.
Reidar Conradi	DCS/NTH, T.heim	organizing chair, editor
Tor M. Didriksen	DCS/NTH, T.heim	ass. organizing chair, co-editor
Dag H. Wanvik	RUNIT, T.heim	ass. organizing chair, co-editor
Else J. Svorkås	DCS/NTH, T.heim	treasurer
Mari Sæterbakk	NTH, T.heim	local arrangements

TABLE OF CONTENTS

An asterisk (*) marks 10 min. panel papers, otherwise 25 min.

PROGRAMMING-IN-THE-SMALL

PROGRAMMING-IN-THE-LARGE

CONFIGURATION/VERSION CONTROL

TOOL INTEGRATION

SOFTWARE ENGINEERING DATABASES

PROGRAM REUSE AND TRANSFORMATIONS

KNOWLEDGE-BASED AND FUTURE PROGRAMMING ENVIRONMENTS

SOURCE LEVEL DEBUGGERS: EXPERIENCE FROM THE DESIGN AND IMPLEMENTATION OF CHILLSCOPE

Svein O. Hallsteinsen

RUNIT
The Computing Center of the University of Trondheim
N-7034 Trondheim-NTH Norway

Abstract

This paper is based on experience from the design and implementation of an interactive source level debugger as part of a programming enviroment for CHILL. The debugger is based on a variant of the event action breakpoint. By combining the ability to detect a rich repertoire of events, including events concerned with the interaction between concurrent processes, and a command language including the source language, a very powerful tool has been obtained. The paper describes the main features of the debugger and discusses some design decisions.

1 INTRODUCTION

1.1 Background

CHILL is the programming language recommended by the CCITT for programming SPC telephone exchanges. It is a Systems Implementation Language (SIL) in the Algol tradition and includes features like user defined types, modules a la Modula 2, programmable exception handling and concurrent processing. CHILL was developed in roughly the same period as Ada and is a language of the same sort.

A programming support environment for CHILL, called CHIPSY, is being developed at RUNIT. CHIPSY is currently hosted on VAX and ND-100 minicomputers and produces target code for the INTEL-86 family of

microprocessors. It is intended for the development of professional real time software. This paper is based on experience from the design and implementation of CHILLscope, which is CHIPSY tool for testing and debugging CHILL programs at the source level. Here testing means executing the program with the intent to demonstrate the absence of bugs, while debugging means finding the exact cause of a bug that has been revealed during testing, and designing a correction that will remove it. Both are concerned with observation and analysis of program execution. As this is exactly what debuggers supports, they are useful both for testing and debugging. Testing and debugging are also strongly interleaved, so in the sequel we shall use debugging loosely to denote this mixed activity.

1.2 Interactive Debugging Techniques

The basic technique employed by most debuggers is user controlled breakpoints. This technique consists of permitting the user to specify points in the program where he wants execution to be interrupted and control transferred to the debugger. Typically the debugger will allow the user to inspect the state of the computation, modify the state of the program, or even modify the program (patching), before continuing execution.

Source level debuggers for high level languages employ basically the same technique but supports access to the program being debugged in source level terms. Ideally a source level debugger should releave the programmer completely from having to know about the machine level details, but the degree of source levelness varies from the ability to use symbolic names instead of machine addresser to full transparency.

Since the technique was first introduced in the FLIT debugger /Stoc60/, it has been considerably refined. A conditional breakpoint has a condition attached and is only effective when the condition evaluates to true. An event - action breakpoint has a sequence of commands attached which will be executed when program execution hits the breakpoint. Also,rather than placing the breakpoint at a point in the program, it may be attaced to an event that may occur during program execution. These may be simple events, e.g. assignment of a particular location, or more complicated ones like the generalized path expressions described in /Brue83/ or the behavioural

abstractions described in /Bate83/ which supports detection of patterns of events expressed in a formalism resembling grammars.

Also the debugger should support viewing the program in terms of the abstractions introduced by the programmer by means of the abstraction mechanisms of the source language.

1.3 Programming the Debugging Activity

The most interesting quantities for analysing program behaviour is not always directly available as attributes of the objects of the program, but have to be derived by complicated computations. Also they may depend on observations at different points in time. E.g. complicated data structures may require extensive extraction and formatting before a suitable view can be displayed. Likewise lots of tedious inspection may be required before the interesting case occurs. This calls for considerable expressive power in the debugger command language. One approach is to include the source language in the command language of the debugger. This has been done successfully for languages like LISP, PASCAL and C and is also the approach taken in CHILLscope.

2 CHILLSCOPE COMMAND LANGUAGE

2.1 Traps

CHILLscope is based on a variation of the event - action breakpoint which we call a trap . The event specifications are of the form operation performed on object; e.g. executing a particular action statement, calling a given procedure or delaying a given process.

For specifying particular objects, the full object access capabilities of CHILL is supported. Both for objects and operations "any" specifiers are available, such that events like "delaying any process" or "any operation performed on a given process" may be expressed.

The action part of the trap is a sequence of commands. The available commands are a subset of CHILL plus the following:

DISPLAY: display given attributes of an object, e.g. contents of a location, type of a procedure

BREAK: suspend execution of action part and execute commands entered interarctively from the user terminal

RUN: resume execution of suspended action part or resume execution of program from a given point

ENABLE/ DISABLE: enables/disables a given trap

The CHILL subset includes assignment, procedure call, conditional statement, concurrent processing and declaration of variables. The delineation of the subset was mainly motivated by budget limitations of the current implementation and may easily be extended. The following examples show trap definitions that might be used for debugging the sample CHILL program included in appendix.

```
TRAP tracePolling =                 !a trap tracing state
WHEN linePoller:poll                !changes on the
DO                                  !subscriber lines
    DISPLAY lineTable(1);
OD;

TRAP checkLineTab =                 !a trap detecting
WHEN linePoller:poll                !inconsistencies in
IF lineTable(1).curState /= hookOn  !the lineTable
AND lineTable(1).server = NULL
THEN
    DISPLAY lineTable(1),1;
    BREAK;
FI;
```

We believe that the ability to call precompiled procedures, paired with the ability for precompiled procedures to execute debugger commands, is a very powerful concept that will stimulate the growth of libraries of debug utilities, both general ones and application specific ones. E.g. a procedure for displaying a linked list may look as follows:

```
displayList:PROC(list REF listElem);
   DO WHILE list /= NULL;
      CHILLscope('DISPLAY list->.value');
      list := list->.next;
   OD;
   END displayList;
```

The expressive power of the event specifications is inherently good,

although it suffers in the current implementation because detection of access to arbitrary variables is not supported. This is due to lack of hardware support on the ND100 and we expect to be able to remove this restriction when porting to the VAX.
In /Brue83/ and /Bate83/ the detection of patterns of events is emphasized, but this is not explicitely supported in CHILLscope. This may however be outweighed by the expressive power of the action language. This is illustrated in the example below where checking an event pattern is implemented by combining traps detecting the individual events and a set of precompiled procedures implementing pattern recognition. The patterns are specified in a formalism resembling EDL /Brue83/.

```
CALL defPat('call','e1'((e2'e4)/e3/e4)'(e5?e6)');
TRAP e0 = WHEN START aServer DO CALL startPat('call',a); OD;
TRAP e1 = WHEN START bServer IF inPat(a,'e1') NOT THEN BREAK; FI;
TRAP e2 = WHEN SEND accepted IF inPat(a,'e2') NOT THEN BREAK; FI;
TRAP e3 = WHEN SEND noAnswer IF inPat(a,'e3') NOT THEN BREAK; FI;
TRAP e4 = WHEN SEND quit     IF inPat(a,'e4') NOT THEN BREAK; FI;
TRAP e5 = WHEN STOP aServer  IF inPat(a,'e5') NOT THEN BREAK; FI;
TRAP e6 = WHEN STOP bServer  IF inPat(a,'e6') NOT THEN BREAK; FI;
```

2.2 Identifier Binding

Throughout a CHILL program, the same identifier may be used in different meanings. The visibility rules of CHILL ensures that at any point in the program, the visible identifiers have a unique meaning. The debugger uses the current focus of execution as the default context to bind identifiers.

However, the debugger must also support access to objects whose identifier is not visible in the current context or is visible with another meaning. To cater for this a notation for indicating the context explicitely has been adopted. The context indication has a static part and a dynamic part. The static part specifics the static scope while the dynamic part selects the process instance and procedure invocation.

E.g. if **i** is a location defined in a recursive procedure **p** defined in a module **m**, and **q** denotes a process having just called **p**, then **[q]m:p#2:i** denotes the **i** in the 2nd recursive incarnation of **p** evoked by **q**.

This notation makes it possible to specify non existing objects. If that occurs during the evaluation of an action, the offending command is ignored. Another possibility is that the event specification of a trap refers to an object that does not exist. In that case our approach is that an event does not occur while its specification refers to non existing objects.

3 DEBUGGING CONCURRENT PROGRAMS

The CHILL language supports programming of concurrent processes and the applications CHILL is intended for are highly concurrent systems. Therefore the design of CHILLscope has emphasized support for debugging such programs.

It is common to distinguish between the debugging of each process as a sequential program, and debugging of the cooperation between the processes. Supplied with the ability to detect relevant events, the trap mechanism proved to be an adequate tool also for the inter-process activities. These events are starting, stopping, delaying and continuing processes and operations on synchronisation objects. The DISPLAY command is capable of displaying the state of processes and synchronisation objects.

In /Smit85/ a debugger designed solely for debugging inter process activity is described. This debugger has a concept called a demon which is very similar to the trap.

The role of the debugger in the system of concurrent processes requires some comment. It is essential that it is well defined which process executes the action part of a trap, and also who executes the commands entered interactively after a break command. In our approach, the effect is as if the action attaced to the trap was inserted in the trapped process at the point were the event occurred. The other processes may or may not continue execution depending on the scheduling algoritm. In the current implementation they will not, but the trapped process may cause another process to continue while the first one is still trapped by executing a forced scheduling action.

An alternative approach used in several other debuggers is

implementing the debugger as a separate process. Our choise was motivated by the following:

- In order to have full control over the inter process communication and the state of the processes and the synchronization objects in the debugger, it is necessary to be able to execute syncronization actions appearing in the action parts of the traps as if they were executed by given processes. Another possibility would have been to provide special commands for manipulating the data structures of the execution support system, but this was discarded because it was felt to be in conflict with the source level orientation.

- Admittedly, it is also very useful to have special debugging processes that observe the processes being debugged. For example, this is a common technique to reduce the probe effect when debugging in real time. However this may be programmed explicitely in the command language.

4 PIECEWISE DEBUGGING

Independent program modules is a valuable programming paradigm that should be supported also at debug time. Finding bugs during early testing of individual modules may prove conciderably more efficient than finding the same bug in test runs involving the whole program. The basic facility needed is some mechanism for simulating the absent part of the program.

In CHIPSY the linker builds stubs representing the objects imported by the module being tested from the absent part of the program. In the stubs for procedures and processes the body has been replaced by a breakpoint, such that their behaviour can be simulated by debugger commands. Debugger commands are also used to stimulate the object being debugged.

5 IMPLEMENTATION ISSUES

There are essentially two ways to implement a debugger of the kind described above, by interpretive execution testing for the events to be trapped in the inner loop of the interpreter, or by executing the compiled code with break instructions inserted to allow detection of the events. The break instructions may be inserted permanently by the compiler at regular intervals, e.g. between each statement, or dynamically by the debugger as needed to detect the events introduced in trap definitions.

In CHILLscope the latter appproach was chosen, mainly because of its superior performance. It is typically one to two decades better than interpretive execution.

A drawback of this approach, is that it restricts the types of events that are supported, because the events has to be detectable by break instructions inserted at a limited set of locations. It may be claimed that it is only a matter of inserting enough break instructions, or resorting to single stepping, but then the performance advantage is sacrificed.

How serious this drawback is depends on what hardware support is available. E.g. in the current implementation of CHILLscope, detecting access to arbitrary locations is not supported, while on computers supporting guards on arbitrary storage areas, this would be no problem. The repertoire of detectable events for concurrent processing is fairly complete, because they can be detected by break instructions inserted in the runtime support system.

Another drawback of this approach is difficulties with maintaining the source level correspondence if the code has been optimized by the compiler. However, work has been published showing practical solutions to these problems for a number of common optimizations /Henn82/ /Zell83/. The problem has not yet been addressed in the CHILLscope project, because the current compiler does no optimization that causes problems for the debugger.

The debugger works on the ordinary generated code. I.e. no additional or special code is generated to support the debugger. Although convenient, this is not so important in the current implementation

were only debugging in the host environment is supported. However, CHILLscope was designed to be adaptable for debugging in the target environment also, and there it is important that the operational code is identical to the code that has been debugged. One may even want to debug code in operation.

In real time systems, the probe effect is a problem. That is, the precence of the debugger disturbs the execution in such a way as to mask bugs /Gait84/. To minimize the disturbance we envisage an implementation of the debugger, where the program being debugged and the machine near parts of the debugger runs in the target computer, while the rest of the debugger runs in the host environment. This will have the intended effect only if detecting events and executing the associated actions is possible without communicating with the host computer. This is achieved by translating the traps at definition time to an intermediate form where all relevant information is available, which is stored in the target computer. The break and display commands will still necessitate communication with the host computer, so these should be avoided in the time critical processes. The DICE system /Frit83/ employs a similar approach, but there the traps are compiled into machine code and inserted into the program by an incremental compiler. This gives more efficient execution of the trap action and thus less disturbance.

6 FUTURE WORK

CHILLscope has just been released to CHIPSY users, so the only experience with using it is from acceptance testing.

From this limited experience we can conclude that a windowed user interface is badly needed. To debug a program with concurrent processes with only a single stream oriented user interface is hard.

More flexible access to program data is another important area for improvement. Program information is now mainly accessed by name. In many situations, however, there is a need to pose queries more in the form of database queries.

Such access is supported in a limited form for dynamic process data. E.g. '**DISPLAY ALL ACTIVE aServer INSTANCES SIGNAL**' means display the signal queue of all active instances of the process definition