An Explicitly Modelled Action Language Debugger

Simon Van Mierlo Yentl Van Tendeloo

April 5, 2017

Abstract

This document describes how a debugger was created for action language in the Modelverse.

1 Introduction

Developers spend a significant amount of their time debugging the code they write. Since the dawn of programming, researchers and practitioners have developed and improved debugging techniques and tools. The code of the program is instrumented to observe and modify its state and control flow. Instrumentation can be manual, for example through log statements, or automatic, by exposing the program's symbols and allowing for the program to be paused (manually, or automatically, using breakpoints). The goal of debugging is to find the source of an observable error. While a defect might be observed at a particular time (through a failed test, a program crash, erroneous output), the offending code that causes the defect might have been executed long before that. A standard debugging session consists of the developer formulating hypotheses, testing these hypotheses by (partly) executing the program. Execution is often done step-by-step, while state changes are observed and interpreted. By refuting and refining hypotheses, the developer ultimately reaches the offending statement.

Zeller [6] perfects this approach and terms it "scientific debugging". More advanced approaches include static and dynamic program slicing, omniscient (or backwards-in-time) debugging, which partly automate the debugging tasks. The focus in these approaches lies on analysis (static and/or dynamic) to uncover statement dependences (slicing), which may lead to "code smells". If the full trace of a crashed program is available, it can be debugged post-mortem, and the developer cannot only traverse it forwards, but also backwards (omniscient debugging).

While these techniques work well for sequential, deterministic programs, they often fail when concepts such as concurrency, non-determinism, spatial distribution and real-time behaviour are introduced. Typical programming languages are not equipped to represent these concepts natively, and such behaviour is often an (unwanted) by-product of complex code. Model Driven Engineering promises to alleviate these problems by offering

specialized modelling languages that can express these concepts natively. And, in special cases, domain-specific languages can be developed whose semantics are closely tied to an application domain. This plethora of languages has an undesirable property: each language needs its own tooling, including (visual) editors, simulators, code generators, test frameworks, and debuggers. These come with a massive development effort, and the research community has focused on providing (semi-)automated support to generate these artefacts. For debuggers, multiple works address how debugging support can be added to existing modelling languages and their tooling [4, 5, 2, 1]

More recently, researchers are exploring how languages can be combined in a (syntactically and semantically) meaningful way, forming a hybrid language [3]. If both languages have debugging support, the debuggers for each language are merged to create a hybrid debugger. These techniques are valid only if the debuggers are explicitly modelled. A significant amount of modelling languages (for example, Statecharts and model transformations) use action language to model imperative code constructs. To implement debugging for these languages meaningfully and in a repeatable way, it is important the action code debugger is explicitly modelled. In this report, we explain how we added debugging support to an explicitly modelled action language in the Modelverse, a model repository and management system. The debugger is explicitly modelled using Statecharts.

Section 2 explores program debugging in the context of gdb, a popular debugging tool for C and C++. Section 3 explains how the behaviour of debuggers can be explicitly modelled using Statecharts. Section 4 explains how the Modelverse executes programs, and the limited infrastructure present to debug them. Section 5 contains the contribution of this report and explains how the Modelverse was extended with debugging support for action language. Section 6 concludes the report.

2 Program Language Debuggers

A well-established, advanced and actively developed debugger is the GNU debugger (gdb) [1]. It allows developers to debug programs written in C or C++, and has partial support for a number of other languages. This section presents an overview of its capabilities, as it is a representative example of a code debugger. We do not list all of gbd's features, since that would occupy too much space. Instead, we focus on its operations for taking control over the program's control flow, how state can be inspected, how state can be changed, and its record/replay capabilities.

2.1 Taking Control

A program is placed under control of gdb by executing the command *gdb program*. Alternatively, gdb can be attached to an already running process. When gdb is started, a shell opens in which the user can type commands.

[1] https://www.gnu.org/software/gdb/

Breakpoints Breakpoints are automatic pauses on a state condition. They are set with the *break* command. If no parameters are passed, a breakpoint is placed at the next instruction to be executed in the current stack frame. A *location* (function name, line number, or an address of an instruction) can be passed as parameter. Optionally, a condition can be passed. In that case, the condition must evaluate to true in order for the breakpoint to trigger. A *tbreak* is a special kind of breakpoint which is removed once it triggers. It is possible to list all breakpoints, and to delete or disable them.

Watchpoints automatically pause the program when the value of some variable changes. Typically, a user monitors whether a particular variable's value changes by executing *watch foo*. The *rwatch* command allows to break the program when a variable is read. The *awatch* command allows to break the program when a variable is read or written. It is possible to list all breakpoints, and to delete or disable them.

Catchpoints automatically pause the program when certain types of events occur. This includes syscalls, exceptions, and the loading of shared libraries.

Continuing If a program is paused, execution can be resumed in several ways.

- The *continue* command resumes continuous execution.
- The *step* command resumes execution until the program reaches a different source line. Optionally, the user can specify how many steps are to be executed.
- The *next* command resumes execution until the program reaches to a different source line in the current stack frame. Optionally, the user can specify how many steps are to be executed.
- The *finish* command resumes execution until the currently executing function returns.
- The *until* command resumes execution until a source line past the current line in the current stack frame is reached. Optionally, the user can specify the location of the line at which to stop execution.
- The *advance* command resumes execution up to a given location (not necessarily in the same stack frame).
- The *stepi* command executes one machine instruction. Optionally, the user can specify how many steps are to be executed.
- The *nexti* command executes one machine instruction, but if it is a function call, it proceeds until the function returns. Optionally, the user can specify how many steps are to be executed.
- The *reverse-continue* starts executing the program in reverse.
- The *reverse-step* command executes the program in reverse until the program reaches a different source line. Optionally, the user can specify how many steps are to be executed.
- The *reverse-next* command runs backward to the beginning of the previous line executed in the current stack frame. Optionally, the user can specify how many steps are to be executed.

- The *reverse-finish* command resumes execution until the point at which the currently executing function was called.
- The *reverse-stepi* command executes one machine instruction in reverse. Optionally, the user can specify how many steps are to be executed.
- The *reverse-nexti* command executes one machine instruction in reverse, but if it is a return from a function, it proceeds until the point at which the function was called. Optionally, the user can specify how many steps are to be executed.

2.2 Inspecting the State

The state of the system is made up of the variable's values (data) and the execution state (frames). Each time a function call is made, a new stack frame is constructed with the location of the call in your program, the arguments of the call, and the local variables of the function being called. To find out how a program got where it is, gdb provides the *backtrace* operation, which prints out one line per frame for all frames on the stack, starting from the current frame. Optionally, the *full* option can be passed to also print values of local variables. A frame can be selected with the *frame* command. With the *info* command, the user can print out a detailed description of the frame, including:

- The address of the frame.
- The address of the next frame down (called by this frame).
- The address of the next frame up (caller of this frame).
- The language in which the source code corresponding to this frame is written.
- The address of the frames arguments.
- The address of the frames local variables.
- The program counter saved in it (the address of execution in the caller frame).
- Which registers were saved in the frame.

The *info args* and *info locals* commands print information on all arguments and local variables of the selected frame, respectively.

Using the *print* command, an expression is evaluated and its value printed in a format appropriate to its data type.

2.3 Altering the State

To change the value of a variable, gdb provides the *set var* command. For example, *set var foo=5* would assign the value 5 to the variable foo (only if foo exists and 5 is a value that can be assigned to it).

Manually changing the instruction pointer is done with the *jump* command. It allows to resume execution at a specific location of the program.

State can be altered indirectly by using the *signal* command which sends a particular signal (specified by name or number) to the program.

To return manually from a function, the *return* command can be used, optionally specifying a return value.

2.4 Altering the Program

Programs run by gdb can be altered in several ways.

With the *set write on* command, the user can write to the loaded binary and patch it. This can be useful for emergency repairs.

Newer version of gdb also support on-demand compilation and injection of code. This is achieved using the *compile code* command, whose argument is a valid line of source code. The compiled code is executed immediately and removed when finished.

2.5 Record/Replay

With the record/reply functionality of gdb, users can record the execution of a program and later replay it. Central to this process is the *record* command, which starts the records and replay process. By default, it fully captures the execution trace and allows replaying and reverse execution. While in this process, if the execution log includes the record for the next instruction, gdb will not really execute the instruction, but instead take all events that would happen normally from the execution log. If the execution log does not include a record for the next instruction, it is executed normally and its result recorded for future replay. The execution log can be traversed freely, and allows more control than the usual stepping functions. The *record goto* command allows to go to a specific location in the execution log. The *record delete* command deletes the subsequent execution log (in replay mode) and starts recording a new one. Record/replay is stopped with the *record stop* command.

3 Explicitly Modelling Debuggers

4 Existing Infrastructure

The Modelverse Kernel (MvK) is responsible for executing user code, contained in tasks. These tasks are modelled as action code, and compiled to graph-like structures in the Modelverse, which are interpreted by the kernel. Multiple tasks are interleaved (with some fairness). When the task is compiled, it consists of execution primitives, such as if-statements, while-loops, variable accesses, ... Each primitive is executed by a number of *rules* in the Modelverse: each application of a rule corresponds to a *phase* of that primitive.

Algorithm 1 The main loop of the Modelverse kernel.

```
function InterpretLoop
    while True do
        for all t ∈ getTasks() do
            executeRule(t)
        end for
    end while
end function
```

5 Extensions to the Modelverse

6 Conclusion

References

[1] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry. Supporting efficient and advanced omniscient debugging for xdsmls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 137–148, New York, NY, USA, 2015. ACM.

[2] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *11th European Conference on Modelling Foundations and Applications (ECMFA)*, L'Aquila, Italy, July 2015.

[3] Sadaf Mustafiz, Bruno Barroca, Claudio Gomes, and Hans Vangheluwe. Towards modular language design using language fragments: The hybrid systems case study. In *Information Technology: New Generations*, pages 785–797. Springer International Publishing, 2016.

[4] Simon Van Mierlo. Explicitly modelling model debugging environments. In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*, pages 24–29, 2015.

[5] Simon Van Mierlo, Yentl Van Tendeloo, and Hans Vangheluwe. Debugging Parallel DEVS. *SIMULATION*, 93(4):285–306, 2017.

[6] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.