Debugging a Smalltalk VM Assisted by Large Automated Reasoning

Boris Shingarov
LabWare, Monkton, Canada

Jan Vraný
LabWare, Dundee, UK

Abstract

We show how a full-scale automated-reasoning engine implemented in Smalltalk can be applied to assist in the programmer's cognitive task of traversing abstraction levels. This approach follows naturally from our definition of debugging as any activity aimed towards understanding a program. We introduce the notion of "dimensions of abstraction", give two examples ("stratum" and "mode"), and show how it is applied in debugging a native compiler backend.

2012 ACM Subject Classification Software and its engineering → Semantics; Software and its engineering → Runtime environments; Theory of computation → Operational semantics

Keywords and phrases Smalltalk, Virtual Machine, Automated Reasoning, Debugging, ISA Specification

Digital Object Identifier 10.4230/OASIcs.Programming.2025.4

Category Extended Abstract

> Why are some programmers so much better than others? What is the magical ingredient that makes it possible for some people to resonate with computers so well, and to reach new heights of performance? Many different skills are clearly involved. But after decades of observation I've come to believe that one particular talent stands out among the world-class programmers I've known – namely, an ability to move effortlessly between different levels of abstraction.
>
> D. E. Knuth

The authors' decades of practical experience engineering the industry's major Java and Smalltalk virtual machines has convincingly taught us the following lesson: implementing an optimized VM for modern weakly-deterministic multiprocessors, using traditional informal methods runs into many excruciating challenges. Of these, the increased difficulty to *move betwen different levels of abstraction* stands out. The question naturally arises: can the machine help?

In this talk we show how we apply a large-automated-reasoning system to assist in debugging the Smalltalk VM. In particular, the reasoning engine we use is our *MachineArithmetic* liquid type-based theorem prover, which we previously described in [10, 9]. The Smalltalk VM under debug is *Smalltalk-25* [9], in particular its compiler component *Tinyrossa* [13].

Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025).
Editors: Jonathan Edwards, Roly Perera, and Tomas Petricek; Article No. 4; pp. 4:1–4:6
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

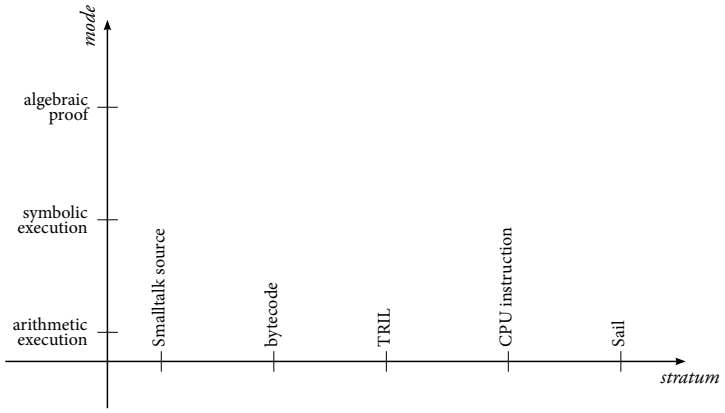

Figure 1 Levels-of-Abstraction can vary along several (here shown two) dimensions.

What are these *different levels of abstraction* that are so crucial to *move effortlessly between*? We found that in our daily work we constantly traverse two distinct dimensions.

- One obvious and usually-discussed meaning of "level of abstraction" is that which comes to mind when mentioning "high-level" vs "low-level programming languages", or the "strata" of JSR-45/JPDA [3, 7]. In a modern runtime, the most difficult bugs result from interplay between subtleties in the image, GC, shared-memory semantics etc.
- Another meaning involves the programmer's mode of thinking. We constantly switch between stepping through a concrete execution in the debugger trying to understand, "what is going on in this particular execution?", and squinting at the code asking ourselves, "will this work for *all* inputs? why, or why not?" In traditional informal programming, this activity is simply left as a subconscious, unstructured "thinking process" in the programmer's brain. On the other hand, existing formal methods go to the other extreme and even look down upon the very word "debugging" as derogatory; this is of little help when trying to solve a mysterious failure in a complex production system.

Related Work

Following the Moldable Debugger of Chiş, Gîrba and Nierstrasz [4], Shingarov [8] demonstrates an application of his "Universally-Live Debugger" to debug OpenSmalltalk VM [6] running on an actual GDB-controlled target. In addition to the convenience of debugging the production VM as if it were running in simulation, the ULD of [8] extended the Smalltalk debugger by adding three simultaneous perspectives of the target VM:

- Smalltalk perspective: What Smalltalk code is the remote VM executing?
- VM perspective: Stepping through the native CPU code in the Smalltalk debugger UI;
- Postmortem JIT or "time travel" perspective: What was the Cog JIT thinking when it emitted this native code?

Even though the ULD of [8] made some limited use of *MachineArithmetic* for internal representation of machine instructions, its only notion of "execution" was based on the fully-concrete (aka "purely-arithmetic") GDB. Because of this, it could only look at a particular fully-concrete execution, and made no attempt at reasoning over the code's semantics in order to assist in connecting the programmer's modes of thinking.

Present Contribution

We show a large-automated-reasoning based debugger that attempts to aid the debugging of our *Tinyrossa* compiler by providing tool support for the programmer's transition across *strata* and across *modes-of-thinking*. In this context the word "large" in "large reasoning" means that now – six decades after Automath – our standard algorithms scale up to handling *complete* and *authoritative* definitions of real ISAs (such as RISC-V and AArch64)[1] as opposed to toy axiomatizations.

The debugger works by proceeding in two directions. Bottom-up, it obtains the ISA semantics by transforming the normative Sail [2] definition of RISC-V into a core-ML program and reflecting that program into its refinement type via a process described by Vazou *et al.* [12]. In the opposite direction, the programmer can step through the interpreted execution of the TRIL input to Tinyrossa, or at any point descend down the stratum hierarchy by stepping into the native CPU code, and then further down into the Sail definition of an instruction.

A choice of interpreters is available, per each "mode of thinking":

- *Fully concrete*: this is the notion of "debugging" that most programmers are used to. The state of the machine – all memory locations, registers *etc* – are bound to arithmetic values.
- *Symbolic*: similar to the `angr` binary analysis engine of Shoshitaishvili *et al.* [11]. For illustration, let's imagine a RISC-V processor in a state in which the `x1` register contains the symbolic value a. Then, we execute the instruction `addi x1,x1,1`. Now `x1` contains the symbolic value $a + 1$. One similarity between `angr` and our symbolic interpreter is that both use the Z3 prover [5] directly, so for example those a and $a+1$ expressions above are Z3 AST nodes. One difference between them, is that `angr` obtains the ISA semantics from Valgrind VEX IR, while our interpreter indirects through a frontend allowing it to process authoritative ISA specifications in Sail, ASL, "PowerPC Green-Cloth pseudocode" *etc.*
- *Algebraic*: the native small-step operational semantics is compiled into a dependently-typed core-ML-like IR, which is typechecked by the MachineArithmetic prover. This allows to check big-step semantic properties, including of looping programs, such as loop termination, or find self-similarity of recursive execution trace. More importantly, we hope the explicit definition of the weak behavioral envelopes in the authoritative specifications will allow us to apply the tool in the most interesting use case: finding and debugging race conditions in a concurrent VM running on a weakly-consistent machine.

The APIs of these interpreters are polymorphic with our Smalltalk API to GDB, and we implemented functions for transforming state between modes; this allows arbitrary mixing of modes. For example, the program can run for a billion instructions on a real machine and the next instruction's Sail definition might be stepped into. Or execution may proceed in lockstep on a real machine and on the Sail simulator.

How would it be useful to a compiler writer who is not a CPU developer, to go below the instruction level and step into the Sail source? We see a number of ways.

In day-to-day work, a compiler writer / VM developer may meet with edge cases of CPU behaviour about which the prose ISA manual does not provide perfect clarity. For one example, when the execution of an instruction aborts due to a page fault, how much of the

[1] We use the RISC-V ISA for the experiments we show at the Workshop; the method is directly applicable to any Sail-defined ISA semantics. Section 2.2.5 of our *Smalltalk-25* paper [9] discusses generalizations to non-Sail definitions.

state is rolled back? OpenSmalltalk VM [6] uses SEGV fault trapping as the fundamental basis for debugging, and when we were porting the trap handling from the simulated to the real CPU [8], we discovered that the simulator and the silicon answer this question differently, causing mysterious failures. Is this a VM bug? or a simulator bug? or perhaps even a silicon bug? Only the authoritative sub-instruction spec provides the ultimate answer.

A deeper cause is the automated reasoning: the only [complete and authoritative] source of ISA semantics available to the tools, is the Sail spec. But this is also very convenient: when a proof fails, the counterexample is mapped back to a Sail execution trace easily understood by VM engineers who are not necessarily expert semanticists or proof theorists.

The final and – we hope – most interesting cause shall become clear when we implement integration with the Cat [1] weak memory models: visualizing a race condition as an execution trace immediately meaningful to a VM engineer.

An Example

Consider the function $abs(x)$, computing the absolute value of its (machine signed-integer) argument:

$$abs(x) = \begin{cases} x, x \geq 0 \\ 0 - x, x < 0 \end{cases}$$

Consider a TRIL representation for `abs`:

```
N000    bbstart <entry>
N016    iflcmple BB_002
N014      lload x
N015      lconst 0
N001    bbend </entry>  <!-- pass through to BB_001 -->
N002    bbstart <BB_001>
N005    lreturn
N004      lload x
N003    bbend </BB_001>
N006    bbstart <BB_002>
N011    lreturn
N010      lsub
N008        lconst 0
N009        lload x
N007    bbend </BB_002>
```

For the RISC-V (RV64) target, Tinyrossa emits the following machine code:

```
+00 bge zero, a0, .+0x8
+04 ret
+08 sub a0, zero, a0
+0C ret
```

Does this seemingly elementary program always output the absolute value, for *any* two's complement 64-bit signed integer? Let's ask the typechecker to check a simple property: is the output always nonnegative? The typechecker gives the answer `UNSAFE` with the counterexample

```
x = 0x8000000000000000
```

What is this `0x8000000000000000`? Did we just discover a bug in our RISC-V program? is this a bug in Tinyrossa's RISC-V backend? did we start from a buggy TRIL program in the first place? With a little stepping in the debugger, we easily see we are looking at an overflow when computing `0-MININT` due to a fundamental quirk in two's complement arithmetic: the 64-bit `abs(MININT)` simply doesn't fit in 64 bits.[2] Thus the problem is in the contract of `abs` itself, whose domain must be

```
BitVec64 | [ :x | x != MININT ],
```

not just `BitVec64`.

Artifact Availability

Like all the other subsystems in Smalltalk-25, the presented debugger is implemented in Smalltalk and available from GitHub[3] under the MIT license.

References

1 Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016. doi:10.48550/arXiv.1608.07531.

2 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, volume 3, pages 71:1–71:31, 2019. Proc. ACM Program. Lang. 3, POPL, Article 71. doi:10.1145/3290384.

3 Gilad Bracha and Arthur Ryman. JSR 45: Debugging Support for Other Languages. URL: https://jcp.org/en/jsr/detail?id=45.

4 Andrei Chis, Tudor Gîrba, and Oscar Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 102–121. Springer, 2014. doi:10.1007/978-3-319-11245-9_6.

5 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-78800-3_24.

6 Daniel Ingalls, Eliot Miranda, Clément Béra, and Elisa Gonzalez Boix. Two decades of live coding and debugging of virtual machines through simulation. *Softw. Pract. Exp.*, 50(9):1629–1650, 2020. doi:10.1002/SPE.2841.

7 Java Platform Debugger Architecture (JPDA). URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/.

[2] In other words, there are more negative machine integers than positive ones, therefore there isn't an abs function total on BitVec64.

[3] https://github.com/shingarov, https://github.com/janvrany

8 Boris Shingarov. What is "debugging" anyway? Experiences with target-agnostic Cog, 2020. URL: https://www.youtube.com/watch?v=kFni50djp9k.

9 Boris Shingarov and Jan Vraný. Towards a Dynabook for verified VM construction. *J. Comput. Lang.*, 80:101275, 2024. doi:10.1016/J.COLA.2024.101275.

10 Boris G Shingarov. Live proof-by-induction. In *FAST Workshop 2022 on Smalltalk Related Technologies*, 2022. URL: https://openreview.net/forum?id=AaMvINlc7d.

11 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016. doi:10.1109/SP.2016.17.

12 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158141.

13 Jan Vraný and Boris Shingarov. Tinyrossa: A compiler framework for vertical, verified construction of Smalltalk VMs. In Emma Söderberg and Luke Church, editors, *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming, Programming Companion 2024, Lund, Sweden, March 11-15, 2024*. ACM, 2024. doi:10.1145/3660829.3660838.