System-Specific Interpreters Make Megasystems Friendlier

MATTHEW SOTOUDEH, Stanford University, USA

Modern operating systems, browsers, and office suites have become megasystems built on millions of lines of code. Their sheer size can intimidate even experienced users and programmers away from attempting to understand and modify the software running on their machines. This paper introduces system-specific interpreters (SSIs) as a tool to help users regain knowledge of and control over megasystems. SSIs directly execute individual modules of a megasystem in a gdb-like environment without forcing the user to build, run, and trace the entire system. A prototype framework to help write SSIs is described in this paper and available for download at https://github.com/matthewsot/ssi-live22.

1 INTRODUCTION

Ballooning user expectations have caused computer systems to grow in size and complexity over time. Operating systems, web browsers, office suites, compilers, and even websites are routinely hundreds of thousands or even millions of lines of code across a variety of programming languages. These megasystems are extremely useful, surprisingly dependable, and often free software. But their sheer size makes it increasingly difficult for users and hobbyist programmers to understand and modify them. Even compiling some megasystems, such as the Linux kernel, can be a surprisingly challenging task. Such barriers represent an unfortunate bottleneck through which even reasonably motivated and skilled users are prevented from understanding and exercising control over the software that has become necessary for their daily lives.

This paper proposes the development and use of system-specific interpreters (SSIs) to help users understand and modify portions of such megasystems. The key insight is that, from the perspective of any given module of the larger system, the rest of the system exposes a domain-specific language in which the module is written. Directly building an interpreter (the SSI) for that language allows the user to run individual modules, files, and functions without having to wait for the system to build, construct inputs triggering the code in question, or insert tracing code.

In fact, programmers already build such SSIs informally in their heads when attempting to understand megasystems. We usually start at some interesting part of the code, e.g., a particular device driver, and read it without knowing exactly how it is called by the overall megasystem or what all of the functions it calls do. Over time, an informal mental model of this interface between the module and megasystem is formed to enable such underconstrained reasoning.

Our key idea is to formalize and mechanize this mental model of the interface between the module and the rest of the system. By giving users the tools to formally specify (as an SSI) their previously informal model of the system–module interface, the cognitive burden of remembering a large model is reduced. Executing code against this model via the SSI helps users find inconsistencies or inaccuracies in their mental model of the system.

We describe a framework that makes writing SSIs easier. The key idea is to separate implementation of language-level syntax and semantics from system-level syntax and semantics. Once written, SSIs simulate execution of a module in isolation, similar to a unit test. The user interacts with the SSI’s gdb-style interface to trace and modify intermediate states of the execution. After modifying the code, the SSI can be re-run to check if the modification had the desired effect. By directly interpreting the specific code that the programmer cares about, SSIs are a first step towards bringing the power of live programming and REPL environments to bear on the unique challenges of lower-level systems code.

Author’s address: Matthew Sotoudeh, Stanford University, Stanford, CA, USA, sotoudeh@stanford.edu.
Consider a curious Jane Doe. One day, she is using a Raspberry Pi to control a light strip and becomes interested in how the kernel’s GPIO driver works. She pulls up the Linux source code, and finds the relevant pinctrl-bcm2835 driver code excerpted in Figure 2a. Reading the code, she notices that the broader Linux kernel provides essentially a domain-specific language (DSL) to the driver code. This DSL includes, for example, commands to remap (devm_ioremap_resource) or write to (writel) memory-mapped input-output (MMIO) addresses. The DSL exposed by the kernel actually spans multiple real languages, e.g., MMIO addresses are implicitly read from a device tree source include (DTSI) file that gets compiled into the kernel at build time.

Jane quickly recalls why playing with megasystem codebases is a daunting task. Traditional debugging tools like gdb require her to build and run the code, but Jane has only ever used a pre-built kernel. Even if she could build the kernel or driver, Jane may not always have immediate access to a Raspberry Pi to run the code on. Even if all of those issues were worked out, Jane would have to spend time writing a test program that correctly invokes the right combination of syscalls to trigger the driver code she is trying to understand, and then somehow figure out what the driver code is actually doing to the hardware, i.e., what MMIO addresses are actually being written to at each point in the program and with what values. Unless Jane wants to try and understand the code entirely in her head, without the benefit of actually running it like code to see what it does, she must solve all of these problems before even beginning to understand the code.

2.1 Using a System-Specific Interpreter

Suppose instead that Jane had a system-specific interpreter (SSI) for this device driver, as shown in Figure 2b. The SSI loads the driver source code, then provides a gdb-like REPL interface to a simulated instance of the driver. Jane can ask it to load the driver and run different driver operations in a simulated environment. She can trace, set breakpoints, and step-through the driver code. Figure 2b shows how Jane can use the SSI to print out the arguments of every call to writeln, and set a breakpoint to inspect the values of local variables.

Because the SSI is directly interpreting the driver code itself, not the entire Linux kernel, it starts and executes quickly. If Jane later modifies the code, she can immediately re-run the SSI to verify it produces the desired behavior. Only once Jane has understood the driver code and made any desired modifications might she need to worry about building and loading the driver. The SSI allows her to separate concerns: understanding and modifying an individual module in the megasystem without first understanding the entire megasystem and associated build system.
From pinctrl-bcm2835.c:

```c
void bcm2835_gpio_wr(struct bcm2835_pinctrl *pc, uint32 val)
```

From bcm283x.dtsi:

```c
gpio@0x7e200000 { compatible = "bcm,bcm2835-gpio"; reg = <0x7e200000 0xb4>;
```

(a) Snippets from the pinctrl-bcm2835 driver  
(b) REPL interface to the driver’s SSI

---

2.2 Writing a System-Specific Interpreter

More likely, nobody has yet written an SSI for kernel drivers, but Jane has heard of the concept and thinks it may help her task. She starts with a pre-written SSI framework, which has a parser for and implementation of the base C language. Because the semantics of the underlying language are handled by the SSI framework, Jane only needs to write the user-facing interface (what should be run when the user requests to enable interrupts for a pin?) and handlers for methods that call out to the larger megasystem API (what should of_address_to_resource return?).

By default, the SSI framework will execute even in the presence of unknown methods by tracking their return values as symbolic unknowns. If these values do not affect the behavior Jane is interested in, she never needs to model those methods. If they do affect some value Jane asks the SSI to print, the SSI will helpfully point her towards exactly what missing API method was missing (Figure 3b). This highlights an important feature of SSIs: the model of the surrounding megasystem that Jane writes does not need to exactly match that of the original system; it just needs to be operationally indistinguishable with respect to the questions that Jane wants to ask about this individual module.

In this way, Jane implements the SSI in an iterative, interactive fashion (Figure 3). She starts off with the raw SSI framework, which informs her when certain methods must be modeled to trace the behavior she is concerned with. She then reads the code and documentation, forming a mental model of those methods and mechanizing that model within the SSI. She repeats this process, iteratively growing and mechanizing her mental model of the system-module interface with the assistance of the SSI framework.
Fig. 3. Users implement SSIs iteratively, refining their model of the system-module interface until the questions they want to ask can be answered. Jane’s initial model of the `of_address_to_resource(np, which_resource, ptr_to_iomem)` sets `*ptr_to_iomem` to a fresh symbolic value and returns zero (Figure 3a). But when she attempts to log memory writes during driver execution, the SSI informs her that the location and value of some of those writes depend on the value assigned to `*c` in the method (Figure 3b). She refines the model (Figure 3c) until the SSI can provide the concrete information she is searching for (Figure 3d).

3 IMPLEMENTATION OF PROTOTYPE INTERPRETER FRAMEWORK

We have implemented a prototype framework for building SSIs. The base framework is roughly 1,000 lines of Python code. Implementing the driver-specific SSI on top of this framework shown in Figure 2 takes approximately another 100 lines of Python code. The first key idea is to separate interpretation of the language-level syntax and semantics from the system-level syntax and semantics. This allows the SSI developer to focus on understanding and implementing the system-level constructs without having to worry about the core language-level constructs (such as addition, pointer dereference, etc.) that are shared by all programs. The second key idea is to be as resilient to missing semantics as possible. This allows the SSI developer to implement only the minimum system-level semantics needed for the behavior they are interested in. System-level semantics that do not affect that behavior should be safely ignored by the interpreter. The rest of this section describes how our prototype framework implements these key ideas.

3.1 Parsing

SSIs must be able to reasonably parse the code even if the user does not know how to build it, e.g., we must handle missing files and headers. In this context, C is known to be ambiguous to parse, e.g., the parsing of `T * x;` depends on whether `T` is a typedef or a global variable. The same issues arise when system-specific macros are used or when domain-specific languages, such as DTSI, are compiled into the system during the build phase. More broadly, we would like the parser to be flexible enough to allow the interpreter to assist even if the program is being edited and not currently in a valid syntactic state.

Our two key parsing mottos are: parse only what you need and parse compositionally. We start parsing exactly where the user asked us to run. We never parse code that is irrelevant to what the user has asked us to run. When parsing, we do not use a single, monolithic grammar. Instead, we have a hierarchy of rules that can match, e.g., an `if` statement rule or a `while` statement rule. When these rules match, they (almost) never recurse into other rules. Instead, like island grammars and
microgrammars [2, 17], we parse with holes. A simplified if statement rule looks for the keyword if, followed by a balanced set of parenthesis, followed by a balanced set of curly brackets. The contents of the parentheses and brackets need not be valid C code, just balanced. Then, upon executing the if statement, we re-execute the parser on the contents of the condition and bracketed code. By keeping the parsers extremely compositional in this form, the user can insert their own custom parsing rules into the grammar hierarchy without fear of messing up assumptions made by a monolithic, integrated grammar. Similarly, by parsing compositionally with holes, if we do not end up taking the positive branch of the if statement, we completely avoid having to deal with any syntax errors within the branch body.

3.2 Interpretation
Whereas in the parsing stage we needed to handle missing, invalid, or system-specific syntactic constructs, in the interpretation stage we need to handle missing, invalid, or system-specific semantic constructs.

The simplest approach we provide to address this is to allow the programmer to hook in to the interpretation phase to provide system-specific semantics. For example, the user can write an interpreter-level function that is called whenever of_address_to_resource is encountered in the code, making the correct changes to program state.

The second approach we provide is to simply ignore certain values by treating them as symbolic. By default, all memory values are symbolic. As operations are performed on these objects, new symbolic values are formed via term-like compositions. For example, executing int x = a + b; would create new memory locations for x, a, b, with symbolic values v_x, v_a, v_b, and a constraint v_x = v_a + v_b. If the concrete value of a variable is identified, e.g., from a constant assignment or taking a branch comparing against a constant, we propagate the constant greedily. For example, if we first executed int a = 1, b = 0; then we would indeed have set v_x = 1.

Why perform such a semi-symbolic execution, if the user ultimately cares about the concrete values? The key insight is that the user usually only cares about a subset of the values, e.g., what values are written to which MMIO memory locations. Many values during a program execution do not affect those that the user is interested in. For example, the bcm2835 driver interlaces spinlock, memory-management, and error-logging operations with the main code, but these do not affect the actual values written to the GPIO-control MMIO addresses. Leaving such values symbolic allows the user to implement only those system-module interface methods that actually impact the subset of behavior they care about.

Because the code is interpreted, a number of interesting analyses can be quickly implemented on top of an SSI. For example, every time a new symbolic or concrete value is created during execution, the SSI tracks the values it was created from along with the line being executed when it was created. At any point, the user can request the interpreter produce a trace explaining the program steps that led to this value. Such analyses are relatively straight-forward to implement in SSIs built with our framework and can significantly improve the debugging experience.

4 RELATED WORK
SSIs bring megasystem-editing in languages like C closer to the live programming environment dream [21, 22]. They empower the programmer to run incomplete code and evaluate the effect of modifications much more quickly than a full build-run-debug cycle. Differences with traditional live programming include highlighting the use of system-specific information; presenting a gdb-style interface rather than full editor; and focusing on the benefits that programmers get by simply writing such an SSI in the first place, even before using it.
SSIs are also similar to unit tests that mock methods external to the module [4]. SSIs essentially allow the unit tests to be written entirely separate from the code and do not require the user wait for a full system build. Further, because SSIs are generally written in a higher-level language, mocks can be written using higher-level language features. The interpreter approach also allows the user to more easily inspect and modify program execution.

The Alloy [13] and Redex [14] projects show the power of lightweight and partial mechanization of mental models of software for understanding and debugging those models. Running parts of existing code in isolation has been studied in the context of microexecutions for binary testing [10] and under-constrained symbolic execution for kernel bug finding [19], inspiring the semi-symbolic interpretation in Section 3.2.

Language server protocols allow certain questions about software to be answered, such as the type of a variable or what methods may be called at any given point in the program [11].

C interpreters [20, 23] and frameworks for building interpreters [5] exist but, to our knowledge, are not designed to incorporate the sort of system-specific information and flexible, failure-tolerant interpretation described in Section 3.

The “systems-as-languages” way of thinking about systems is by now folklore in the software community [15, 18] and embraced by the language-oriented programming language Racket [9]. We aim to bring this paradigm to megasystems written in imperative languages like C. Given the ubiquitousness of this way of thinking about systems as domain-specific languages, it is surprising that no prior work appears to have attempted to directly interpret that implicitly-defined DSL. The closest work we are aware of is a line of research incorporating system-specific information into bug-checkers and compiler optimizations [6–8, 12].

Alternative approaches to analyzing incomplete code either apply neural networks [1] or attempt to repair the code, automatically inserting a minimal amount of new typedefs and method stubs to allow the code to build with a standard compiler [3, 16]. In theory, this avoids the need of the SSI framework to re-implement the base language semantics. We initially attempted to use the tool from [16] on the C driver code, but found that it was unable to correct the errors, especially when macros were used. We encountered similar difficulties with our own attempts at implementing such an approach. This experience motivated the flexible parsing and interpretation approach described in Section 3, where the interpreter can simply ignore any parts of the code it does not actually attempt to run.

5 CONCLUSION, LIMITATIONS, AND FUTURE WORK

The complexity of modern megasystems poses challenges to user freedom and control over their computing. We propose system-specific interpreters (SSIs) to address some of these challenges. SSIs provide users with more immediate feedback than a full build-run-trace cycle, helping them better understand and modify megasystem source code. Our prototype framework simplifies the implementation of SSIs by decoupling the syntax and semantics of the language and system.

SSIs are not perfect. Someone must write an SSI for the system, and keep it up-to-date as the system evolves. Base languages with more complicated semantics, such as C++ and Rust, will be more difficult to support. Although semi-symbolic execution helps mitigate some of these issues, it introduces new challenges like deciding on control flow when branching on symbolic values.

Once SSIs are available, new use cases for them may be found. Direct integration with IDEs may simplify the user experience. Similarly, SSIs could be used as a semantic code search tool, or to evaluate reasonableness of suggestions from autocompletion tools in the context of this specific system. Finally, they may be used by to extract preconditions for code, enabling better static analysis and profile-guided optimizations.
ACKNOWLEDGMENTS

The author is grateful for financial support from the National Science Foundation under grants 1918056 and DGE-1656518. The author would also like to thank the anonymous reviewers for feedback that improved the quality of this paper. This paper benefited greatly from separate discussions the author had with Aditya V. Thakur and Zachary Yedidia regarding the problems posed by megasystems, as well as Dawson Engler regarding system-specific analyses.

Icons in Figure 1 were designed by OpenMoji, the open-source emoji and icon project, and licensed under CC BY-SA 4.0. The OpenMoji project is available on the Web at https://openmoji.org/ and the licence can be read at https://creativecommons.org/licenses/by-sa/4.0/.

REFERENCES

[1] Toufique Ahmed, Premkumar T. Devanbu, and Vincent J. Hellendoorn. 2021. Learning lenient parsing & typing via indirect supervision. Empir. Softw. Eng. 26, 2 (2021), 29. https://doi.org/10.1007/s10664-021-09942-y
[2] Fraser Brown, Deian Stefan, and Dawson R. Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 199–216. https://www.usenix.org/conference/usenixsecurity20/presentation/brown
[3] Barthélémy Dagenais and Laurie J. Hendren. 2008. Enabling static analysis for partial java programs. In Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA, Gail E. Harris (Ed.). ACM, 313–328. https://doi.org/10.1145/1449764.1449790
[4] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014. IEEE Computer Society, 201–211. https://doi.org/10.1109/ISSRE.2014.11
[5] Alan M. Durham, Edson Sussumu, and Arlindo Flávio da Conceição. 2003. A framework for building language interpreters. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 191–196. https://doi.org/10.1145/949344.949398
[6] Dawson R. Engler. 1997. Incorporating Application Semantics and Control into Compilation. In Proceedings of the Conference on Domain-Specific Languages, DSL ’97, Santa Barbara, California, USA, October 15-17, 1997, Chris Ramming (Ed.). USENIX. http://www.usenix.org/publications/library/proceedings/dsl97/engler.html
[7] Dawson R. Engler. 1999. Interface Compilation: Steps Toward Compiling Program Interfaces as Languages. IEEE Trans. Software Eng. 25, 3 (1999), 387–400. https://doi.org/10.1109/32.798327
[8] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In 4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000, Michael B. Jones and M. Frans Kaashoek (Eds.). USENIX Association, 1–16. http://dl.acm.org/citation.cfm?id=1251230
[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In 1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-5, 2015, Asilomar, California, USA (LIPIcs, Vol. 32), Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113
[10] Patrice Godefroid. 2014. Micro execution. In 36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014, Pankaj Jaitole, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 539–549. https://doi.org/10.1145/2568225.2568273
[11] Nadeeshan Gunasinghe and Nipuna Marcus. 2022. Language Server Protocol and Implementation.
[12] Seth Hallem, Benjamin Chelf, Yichun Xie, and Dawson R. Engler. 2002. A System and Language for Building System-Specific, Static Analyses. In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 69–82. https://doi.org/10.1145/512529.512539
[13] Daniel Jackson. 2019. Alloy: a Language and Tool for Exploring Software Designs. Communications of the ACM 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843
[14] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,
[15] Butler W. Lampson. 1983. Hints for Computer System Design. In Proceedings of the Ninth ACM Symposium on Operating System Principles, SOSP 1983, Bretton Woods, New Hampshire, USA, October 10–13, 1983, Jerome H. Saltzer, Roy Levin, and David D. Redell (Eds.). ACM, 33–48. https://doi.org/10.1145/800217.806614

[16] Leandro T. C. Melo, Rodrigo Geraldo Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. 2018. Inference of static semantics for incomplete C programs. Proc. ACM Program. Lang. 2, POPL (2018), 29:1–29:28. https://doi.org/10.1145/3158117

[17] Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin. 2008. Practical Scope Recovery Using Bridge Parsing. In Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5452), Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk (Eds.). Springer, 95–113. https://doi.org/10.1007/978-3-642-00434-6_7

[18] Peter Norvig. 1992. Paradigms of artificial intelligence programming: case studies in Common LISP. Morgan Kaufmann.

[19] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 49–64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[20] Alexei Svitkine. 2009. Interactive Console for the C Programming Language.

[21] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. J. Vis. Lang. Comput. 1, 2 (1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

[22] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013, Brian Burg, Adrian Kuhn, and Chris Parnin (Eds.). IEEE Computer Society, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[23] Vasil Vasilev, Ph. Canal, Axel Naumann, and Paul Russo. 2012. Cling--the new interactive interpreter for ROOT 6. In Journal of Physics: Conference Series, Vol. 396. IOP Publishing, 052071.