A Domain-Specific Language for Device Drivers

Christopher Conway

30 October 2002

1 Introduction

Device drivers have been noted as a major source of faults in operating system code [2]. Largely for efficiency, device drivers and other systems code have historically been written in low-level languages like C. Unfortunately, these languages do not provide the type safety and robustness one would expect in critical systems code. Work has been done to augment the type safety of low-level languages [3, 6], but the efficacy of this work is limited by both fundamental and practical concerns.

In this paper, I will describe a domain-specific language and compiler for network interface device drivers. The language has been used to implement a driver for NE2000 network cards, a widely available class of inexpensive LAN adaptors. The language includes direct support for the operational semantics of device drivers and provides a high level of type safety. Concurrency semantics are included for the description of devices with multiple independent operational units.

Though the language has been built for and tested on network drivers, it is flexible enough to describe a wider class of drivers. The compiler is also designed to be readily ported to a wider class of operating systems.

2 Related Work

A variety of approaches have been suggested to improve the reliability of low-level software and device driver software in particular. Crary and Morrisett propose a typed assembly language (TAL) as a compiler target for preserving type information from higher-level languages [3]. Unfortunately, the most common systems programming language, C, is not much more strongly typed than a traditional assembly language and there is little the compiler can do to improve the type safety of C code.

Deline and Fähndrich use a similar typing system in the C-like programming language, VAULT [4]. The use of variables is controlled through type guards which describe when an operation on a variable is valid. In order for the compile to accept the program, it must respect the type guards' access specifications and types must match at program join points. VAULT is not a domain-specific language, but its restrictive type system means it is not fully general either—the use of alias types results in a loss of type safety. The difficulty and lack of flexibility in programming VAULT may prevent its wide adoption in systems programming.

A more practical approach is static analysis of traditional C systems code. Ball and Rajamani developed a system, SLAM, that is currently in use in the Microsoft Windows group [1]. SLAM operates on a specification for correct behavior developed separately from the driver code. The result is very good error detection at compile time for the properties captured by the specification. However, the analysis can be slow and may take many iterations to complete. In addition, the types of errors that may be detected are restricted in principle, and limited as well by the correctness of the behavior specification.

A group at the University of Rennes has done work on domain-specific languages for device drivers. Thibault, et al., developed GAL, a domain-specific language for X Windows video drivers [7]. The project combines a partial evaluation framework with a language tailored to video driver operations to produce driver code that is nearly 90% smaller than the equivalent C code and just as fast. This work is promising, but the methodology may not be applicable to device drivers as a whole.

Mérillon, et al., also of the University of Rennes, designed a more general solution for device driver development: the Devil interface definition language [5]. A Devil specification describes entities exposed for interaction with a hardware device (e.g., I/O ports, memory-mapped registers). The specification is compiled into a C module for manipulating the device, allowing the driver programmer to write to a clean API and avoid writing low-level code. This approach prevents certain common low-level programming errors, but it does not fully specify the protocol for using the

device, and it does not provide the type safety of a higher-level solution.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002. ACM SIGPLAN-SIGACT.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Alberta, Canada, October 2001. ACM.

[3] K. Crary and G. Morrisett. Type structure for low-level programming languages. In *International Colloquium on Automata, Languages, and Programming 1999*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, Prague, Czech Republic, July 1999. Springer Verlag.

[4] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001. ACM SIGPLAN.

[5] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, pages 17–30, San Diego, California, October 2000. USENIX.

[6] G. Morrisett. Type checking systems code. In *European Symposium on Programming*, volume 2305 of *Lecture Notes on Computer Science*, pages 1–5, Grenoble, France, April 2002. Springer Verlag.

[7] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation–application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May-June 1999.