High-Confidence Operating Systems *

Radu Grosu, Erez Zadok, Scott A. Smolka, Rance Cleaveland, and Yanhong A. Liu
Stony Brook University

1 Introduction

Operating systems (OSs) are among the most sophisticated software systems in widespread use, and among the most expensive and time-consuming to develop and maintain. OS software must also be robust and dependable, since OS failures can result in system crashes that corrupt user data, endanger human lives (cf. embedded systems), or provide open avenues of attack for hackers or even cyber-terrorists.

OSs present their designers with enormous development challenges. On the one hand, many activities inside an OS happen concurrently: caches are flushed periodically; processes and threads are stopped and restarted; interrupts and other signals arrive at random times and must be handled promptly; data can be transferred through multiple channels (memory, DMA, I/O buses). Concurrent processing introduces well-known difficulties into the traditional code-test-debug paradigm, since errors can be difficult to repeat owing to race conditions between concurrent processes. On the other hand, debugging even sequential OS modules poses difficulties, since the OS's closeness to the actual computing hardware requires *in situ* testing.

It is also important that OS *system-call interfaces* be well documented so that they may serve as useful design guides for OS implementers and as interface definitions for application developers. Man pages, currently the primary source of documentation for system-call interfaces, are often incomplete, vague, ambiguous, and even incorrect. Another often under-appreciated aspect of OS software is the profusion of different OSs in use, particularly in the embedded-systems arena where OSs such as VxWorks or pSOS or (more often) proprietary ad hoc OSs are deployed.

The above observations highlight the great impact that improved OS development techniques would have on all enterprises that produce or use OS software. If man pages could be guaranteed to be correct and complete; if system calls could be certified to be free from deadlocks and memory leaks; if causes of system crashes could be quickly diagnosed; then the savings to the OS and application-development communities would be enormous. If this could also be accomplished while reducing OS development costs, then the impact is even greater. We refer to this ideal—better OSs at lower cost— as *High-Confidence Operating Systems* (HCOS).

In this paper, we present an overview of our work on bringing the HCOS concept to bear on the practice of OS development. Section 2 presents the overall methodology we are pursuing, a central component of which is the Concurrent Class Machines (CCM) modeling formalism. Section 3 describes how we are using CCMS to model system-call man pages. Section 4 discusses how we verify our CCM models against different kinds of requirements. Section 5 concludes with a status report.

In related work, efforts to validate OSs fall into three main categories: verification techniques [3], compilation techniques [4, 7], and external runtime testing [6]. The references given are a sampling. The HCOS approach focuses on the formal modeling of OS system calls and their interfaces, and utilizes newly developed techniques from all three categories.

2 Methodology

The organizing principle of our approach is that *an ounce of modeling is worth a pound of debugging*. In particular, we advocate the use of *formal operational modeling* as a methodology that can fundamentally and dramatically improve how OS software, and indeed any low-level system software, is developed. We envision these models

*Appears in proceedings of the Tenth ACM SIGOPS European Workshop (EW-2002).

being used throughout the OS development and deployment process, as active (i.e., executable) documentation for designers and application developers; as mechanically analyzable requirements and design artifacts; and as bases for reliable implementations.

The specific modeling formalism we are using, *concurrent class machines* (CCMs) [5], extends basic finite-state machines with features capturing a variety of object-oriented (OO) concepts, including classes and inheritance, objects and object creation, method invocation and exceptions, multithreading, guarded commands, and abstract collection types. The CCM model builds on our previous work in the formal modeling of hierarchic reactive systems, e.g. [1], and provides an intuitive, graphical notation for modeling system behavior at different levels of abstraction. In contrast with existing OO design notations, CCMs also possess a mathematically precise operational semantics that defines the execution steps that CCM models can engage in; this semantics makes CCM models candidates for a variety of different mechanical analyses. Figure 1 shows how CCMs provide a uniform basis for requirements analysis, verification, and code generation:

- **Executable and analyzable man pages**: CCMs model system-call interfaces and system properties, such as deadlock and livelock freedom. Unlike man pages, the resulting specifications are graphical, executable, precise and unambiguous.
- **Verifying models against requirements**: Verification techniques are used to check whether the CCMs derived (via compilation) from system-call implementations correctly implement man-page-derived system-call interface and requirements (required properties) that are also given as CCMs.
- **Models as system monitors**: Automatic code generation based on CCMs is used to produce efficient code for monitoring the runtime behavior of OS implementations in order to detect and, in some cases prevent, erroneous behavior.

A central idea in our approach is that of an *instrumented CCM*, where a CCM describing the implementation of an OS system call is combined with the CCM of the corresponding system-call interface or CCM of the requirements. Man-page CCMs can also be instrumented by combining them with requirements CCMs they must adhere to.

3 Operational Modeling of Man Pages

The process of determining the exact behavior of a system call begins with a careful reading of its documentation, including an inspection of the arguments that are passed to the system call, its return values, and their types. Man pages typically specify valid inputs and expected return values. The latter are divided into values that indicate success and values that indicate failures, or *exceptions*. Based on this information, an initial mock-up of a CCM can be developed.

Figure 2 shows an example, the modeling of the `creat` system call as a method of the `FileSystem` CCM. Let us first describe the visual notation. A class machine is a named rounded box that has several compartments: one for attributes and one for each method. A method has an entry point (hollow circle), several exit points (filled circles) and several exception points (filled diamonds). Exit and exception points may be marked with an expression denoting the return value. The entry point is connected to the exit (and exception) points by transitions and method invocations. A transition (shown as a labeled arrow) is an atomic guarded assignment. A method invocation (shown as a rounded box) has an entry point, an exit point and several exception points. Exit and exception points may be marked with variables to hold the return values. Exceptions propagate by default to the enclosing levels. A method may contain local variable declarations. As in UML, an attribute or method marked with + is public.

The `creat` method takes as arguments a pathname *pn* and a mode *m*, splits the first into a path *p* and a name n, creates a new file with path p, name n and mode m and returns its file descriptor *d*. If a file already exists at pn, `creat` truncates it to zero bytes. However, the call only works for regular files, not directories (for which the user should use `mkdir`). We reflect this condition in the CCM as follows: if the file name to be created already exists and the type of that file is `dir`, then exit this system call with the error condition `EISDIR`; otherwise continue to the next step in the CCM.

In cases where the man pages are insufficiently detailed or known to be inaccurate, we inspect the actual kernel sources at or near the entry point of that given system call into the kernel. For example, the manual page for the `creat` system call (on Red Hat Linux 7.1) does not specify that it will return an `ENOQUOTA` (quota exceeded) error code if the user's quota was exceeded when trying to add the new file; or that it will return `EIO` (I/O error) if a hardware failure occurred while trying to add the new file's entry to the on-disk directory. We found these conditions by inspecting the kernel sources for a

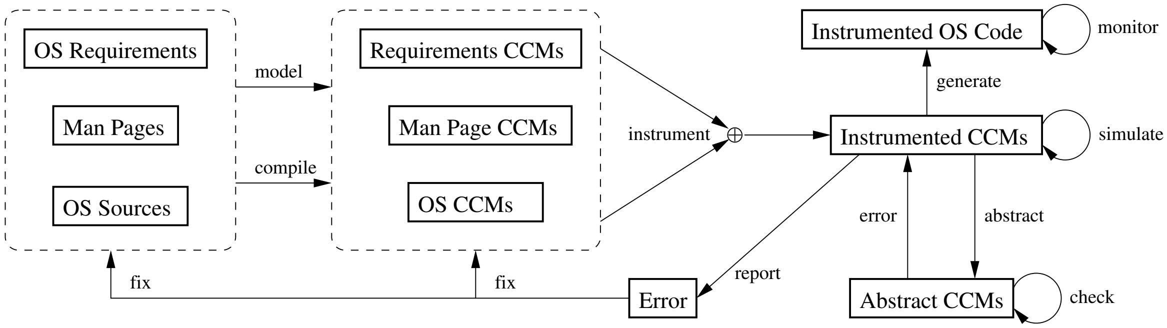

Figure 1: HCOS methodology overview.

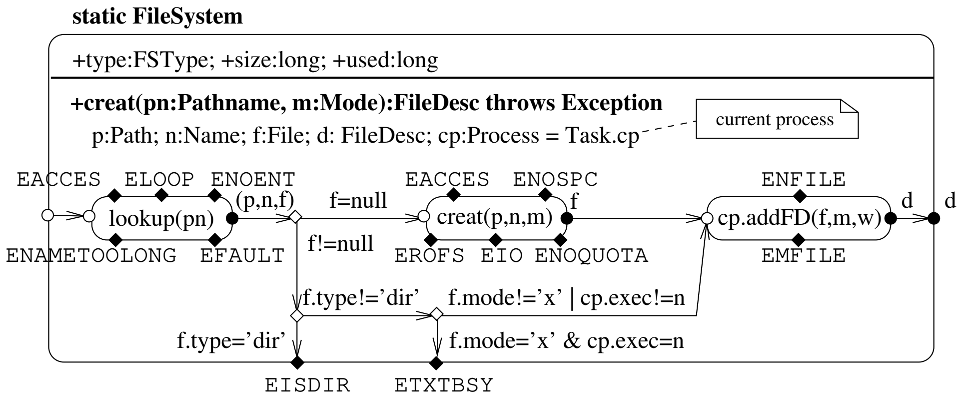

Figure 2: The CCM for the `creat` system call.

running version of Linux. Incorporating this behavior into the CCM above is straightforward, and the result is a more complete accounting of the behavior of `creat` than is available from its man pages. The description is also much more concise; instead of the several pages of text used to document `creat`, the diagram fits onto less than half a page.

CCM models have several benefits. In addition to their clarity and conciseness, they are *executable*, which enables application and OS developers to experiment with system calls on different inputs to see how they behave. CCMs also permit inter-system-call analyses in general, and those involving concurrency in particular, to be rigorously studied. For example, suppose one process tries to read a (shared-mapping) memory-mapped file, while another process tries to write to that same file asynchronously not using the mmap-interface. The OS must carefully coordinate the transfer of (possibly cached) data, lock pages and files, to ensure data coherency. Moreover, this seldom-used combination of Unix features should never result in a kernel crash. Whereas a small file system tool named FSX [8], recently released by Apple, can detect a few such anomalies, such tools have to be written by hand. Our method allows such anomalies to be systematically uncovered using *fully automatic* techniques that thoroughly search the state space of a CCM.

4 Verifying Models Against Requirements

In Section 3 we discussed how to use CCMs to model system-call interfaces. These models may be seen as detailed specifications that implementations should adhere to. In addition, other, simpler system requirements (e.g. "a file is locked before it is accessed") may be encoded as CCMs that act as monitors, entering bad states when an undesired system state is entered. Finally, detailed code-level CCMs can be used faithfully to model the actual behavior of a system call implementation, and can be obtained via compilation.

Given these different levels of models, one would like to check that they are in agreement, namely, that man-page models agree with requirements, that code models satisfy requirements, and that code models match man-page models. A traditional approach to handling this question involves the use of a refinement relation to check whether a given CCM refines (is faithful to) another CCM. Our methodology uses a novel, albeit mathematically equivalent, alternative that relies on the use of *instrumented CCMs*.

The basic idea behind this approach is the following. Given a high-level CCM (e.g., man-page model) and a lower-level one (e.g., code-level model), we use the former to track the execution of the latter. The state space of the resulting instrumented CCM is then explored to see if the wrapper ever enters a bad state (i.e., raises an unintended exception); if so, an error trace leading from the start state of the instrumented CCM is reported to the user for debugging purposes. Essentially, the instrumented CCM checks that the CCM in question *refines* its specification. The modularity property of refinement checking allows such checks to be performed at the level of component CCMs.

In order for the instrumented-CCM approach to verification to be practical, the *state-explosion* problem must be overcome: the number of states to in the instrumented CCM is likely to be intractably large. One approach to coping with state explosion is to use *abstractions* to eliminate distinctions between data values and thus reduce the number of distinct states. We are investigating using a combination of data and predicate abstraction [2] to obtain good abstractions.

A well-known drawback of abstraction-based techniques is the *false-positive* problem: a path to an error state may exist in the abstracted system that is not possible in the concrete system, owing to the loss of too much information in the abstracted conditional statements. To combat this problem our method uses counter-examples generated during reachability analysis to successively refine the data abstractions used: see the directed edge from "Instrumented CCMs" to "Error" in Figure 1. If an error trace is detected in the abstracted CCM, the trace is replayed on the unabstracted CCM to see if it is feasible. If not, conditions on transitions are modified to refine the abstracted CCM, increasing the size of its state space but eliminating the possibility of the spurious trace. We are developing an efficient algorithm for symbolically checking the feasibility of an error path in the instrumented CCM and returning the corresponding abstraction predicates if the path is not feasible.

We are also investigating the problem of constructing a *minimal* instrumented CCM for a given specification CCM and a corresponding implementation CCM. The smaller the wrapper (instrumentation), the smaller the overhead incurred during monitoring and verification. Once a property has been verified, its wrapper can be removed from the instrumented CCM and code, also leading to reduced overhead.

5 Status

We are currently developing tool support for the CCM modeling formalism with the goal of applying HCOS techniques to several variants of Linux, including SMP, Beowulf, and embedded Linux. To date, a prototype has been implemented that consists of a visual front-end for interactive specification using CCMs, and automatic generation of Java code for most CCM features. Other tools developed for the analysis of non-CCM operational models are also being retargeted to CCMs.

References

[1] R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Computer Aided Verification, 12th International Conference*, LNCS 1855, pages 280–295. Springer, 2000.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001, SIGPLAN Notices 36(5)*, pages 203–213, 2001.

[3] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*, Chateau Lake Louise, Banff, Canada, October 2001. ACM SIGOPS.

[5] R. Grosu, Y.A. Liu, S.A. Smolka, S.D. Stoller, and J. Yan. Automated software engineering using concurrent class machines. In *Proceedings of ASE'01, the 16th IEEE International Conference on Automated Software Engineering*. IEEE, 2001.

[6] A. Kolawa and A. Hicken. Insure++: A Tool to Support Total Quality Software. http://www.parasoft.com/insure/papers/tech.htm, March 2001.

[7] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. ERASER: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[8] A. Tevanian and C. Minshall. File System Exerciser. http://www.codemonkey.org.uk/cruft/fsx-linux.c, 1991.