Capturing OS Expertise in an Event Type System: the Bossa Experience

Julia L. Lawall
DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Gilles Muller
Ecole des Mines de Nantes
44307 Nantes Cedex 3, France
Gilles.Muller@inria.fr

Luciano Porto Barreto
COMPOSE group, INRIA/LaBRI
33402 Talence Cedex, France
barreto@labri.fr

Abstract

Emerging applications have increasingly specialized scheduling requirements. Changing the scheduling policy of an existing OS is, however, often difficult because scheduling code is typically deeply intertwined with the rest of the kernel. We have recently introduced the Bossa framework to facilitate the implementation and integration of new scheduling policies. While the use of Bossa simplifes the problem of implementing a new scheduler, knowledge of the control and data flow through the scheduling actions of the kernel is still needed to ensure that the behavior of the provided scheduling policy matches kernel expectations. In this paper, we propose a modular type system that provides a high-level characterization of the aspects of kernel behavior that affect the correctness of a scheduling policy. These types guide policy development and are linked with the compiler to enable static verification of correctness properties.

1. Introduction

The need for application-specific scheduling strategies is now well recognized in the OS community, as is demonstrated by the number of recent studies in this area [1, 6, 11, 14, 16, 18, 20]. Nevertheless, developing a new scheduling policy and integrating it into an existing OS is complex, because it requires understanding the (often implicit) conventions used by the OS implementation. One technique that can address this issue is the use of a framework to structure the implementation of a scheduler and make precise its interface with the kernel. Another technique is the use of a *domain-specific language* (DSL) for the development of new policies. A DSL is a programming language providing high-level abstractions appropriate to a given domain. Expertise captured in the language allows policies to be expressed in an intuitive and high-level manner, permits verification, and allows generation of efficient code that is automatically integrated in the target system. DSLs have demonstrated their interest in a variety of OS subsystems such as network protocols [19], memory coherency protocols [7] and drivers [13].

We have recently introduced the Bossa event-based framework for implementing schedulers [3]. The framework includes a DSL for writing scheduling policies. In the Bossa framework, each scheduling-related action in the kernel, referred to as a *scheduling point*, is replaced by the notification of a Bossa event. A scheduling policy is implemented as a collection of event handlers that are written in the Bossa DSL and translated to a C file by a dedicated compiler. The use of the high-level scheduling abstractions built into the Bossa DSL permits scheduling-specific verifications and optimizations.

Correctness of the Bossa implementation of a scheduling policy requires that the definition of each event handler be consistent with the expectations of the corresponding kernel scheduling points. Because kernel properties vary from OS to OS, knowledge of these expectations cannot be simply built into the Bossa compiler. In this paper, we propose to describe OS behavior using a collection of *event types* that describe the possible process states at each scheduling point and the corresponding expected effect of each event notification. Event types are written by an expert in the target OS. They are provided to the policy developer to guide policy design and implementation, and linked with the Bossa compiler to produce a compiler specialized to a particular OS. This specialized compiler uses knowledge of the DSL abstractions and the event types to check that the handlers of the implementation of a scheduling policy are consistent with kernel expectations.

The rest of this paper is organized as follows. In Section 2, we present the Bossa framework and DSL. Section 3 presents the use of event types to model OS behavior. Section 4 assesses the impact of event types on policy development and optimization. Finally, Section 5 considers related work and Section 6 concludes. We use the Linux 2.2 kernel and its scheduling policy as examples throughout the paper.

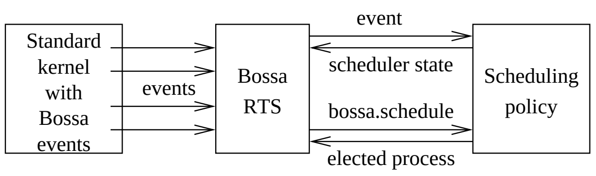

Figure 1. Bossa architecture

2. Bossa

We first present the Bossa framework as a whole, and then focus on the Bossa DSL, which is used to implement new scheduling policies.

2.1. The Bossa framework

The goal of the Bossa framework is to separate the implementation of the scheduler from the rest of the kernel, so that the scheduling policy can easily be changed at the time of kernel recompilation. The Bossa framework replaces scheduling actions in the kernel, such as the modifying of a process state or the electing of a new process, by Bossa event notifications. Events notifications are processed by the Bossa run-time system (RTS), which invokes the appropriate handler defined by the scheduling policy. Bossa events are organized into a hierarchy, according to the function and source of the event. A policy can provide a single event handler for all events related to a particular function, such as a `block.*` handler that applies to all blocking events, or handlers for specific events, such as blocking when waiting for a file or memory page. The selected handler returns the state of the scheduler, indicating whether there is a running process, or there are ready processes available. This information is used at the next `bossa.schedule` event notification: The RTS only invokes the `bossa.schedule` handler of the policy if the current scheduler state indicates that there is no running process and a ready process is available. Otherwise if there is a running process, the RTS allows it to continue, and if there are no running or ready processes, the RTS runs the kernel idle loop.

The RTS is fixed and provided by the framework. The scheduling policy is provided by the policy developer, and implemented using the Bossa DSL. Both the RTS and the compiled scheduling policy are integrated into the kernel. Our performance results, reported in detail elsewhere [4], typically show no penalty for the use of Bossa, on both standard OS benchmarks and realistic applications such as compilation of the Linux kernel.

2.2. The Bossa DSL

A Bossa scheduling policy declares: (i) a collection of scheduling-related structures to be used by the policy, (ii) a set of event handlers, and (iii) a set of interface functions, allowing users to interact with the scheduler. Figure 2 shows some of the declarations made by the Bossa implementation of the Linux 2.2 policy. The `process` declaration lists the policy-specific attributes associated with each process. As reflected by the `policy` field, the Linux 2.2 scheduling policy manages FIFO and round-robin real-time processes, as well as non-real-time processes. The other fields of the process structure are used to determine the current priority of the associated process. The `states` declaration lists the sets of states that are manipulated by the scheduling policy: `running`, `ready`, `yield`, `blocked`, and `terminated`. Each state is described by a class, which indicates the schedulability of processes in the state. The state in the `RUNNING` class contains the currently running process. The states in the `READY` class contain processes that are eligible to be elected at the next context switch. The states in the `BLOCKED` class contain processes that are not eligible to be elected at the next context switch. The states in the `TERMINATED` class represent terminating, and thus no longer schedulable, processes. Each state is also associated with a data structure: either a process variable (`process`) or a queue (`queue`).[1] One `READY` state is designated as `sorted`; only processes in this state can be elected. Finally, the `ordering_criteria` declaration specifies how the relative priority of processes is computed.

Figure 3 shows the definitions of several event handlers of the Linux 2.2 policy: `system.clocktick`, `block.*`, and `unblock.*`. The effect of the `system.clocktick` handler depends on whether the target process is a real-time process, and if so, on the process's real-time policy. A clock tick has no effect on a FIFO real-time process. If a non-FIFO process has used up its ticks, it is moved to either the `ready` queue or the `expired` queue. Otherwise, the handler decrements the number of ticks remaining. The `block.*` handler moves the target process to the `blocked` queue. The `unblock.*` handler moves the target process from the `blocked` queue to the `ready` queue, if the target process is indeed blocked.

2.3. Assessment

The use of the Bossa framework isolates all policy-specific scheduling actions into a single file implementing a well-defined interface, thus simplifying understanding of the policy as a whole. The use of the Bossa DSL further permits the implementation of the policy to be expressed in

[1] No data structure is associated with `terminated`, as terminating processes need not be accounted for by the scheduler.

```
type policy_t =
  enum {SCHED_FIFO, SCHED_RR, SCHED_OTHER};

process = {
  policy_t policy;
  int      rt_priority;
  time     priority;
  time     ticks;
  system struct ctx  mm;
}

states = {
  RUNNING running : process,
            previous old_running;
  READY ready     : sorted queue;
  READY expired   : queue;
  BLOCKED yield   : process;
  BLOCKED blocked : queue;
  TERMINATED terminated;
}

ordering_criteria = {
  highest rt_priority, highest ticks,
  highest ((mm == old_running.mm) ? 1:0)
}
```

Figure 2. Declarations of the Linux 2.2 scheduling policy

```
On system.clocktick {
  if (running.policy != SCHED_FIFO) {
    if (running.ticks == 0) {
      running.ticks = running.priority;
      if (running.policy == SCHED_RR) {
        running => ready;
      }
      else {
        running => expired;
      }
    }
    else {
      running.ticks--;
    }
  }
}

On block.* {
  e.target => blocked;
}

On unblock.* {
  if (e.target in blocked) {
    e.target => ready;
  }
}
```

Figure 3. Selected event handlers of the Linux 2.2 policy

a concise and high-level manner. The approach facilitates policy evolution, because the Bossa DSL compiler ensures the consistency of the various parts of the scheduling policy. Nevertheless, the framework and the DSL alone are not sufficient to ensure that the behavior of a given scheduling policy matches kernel expectations.

3. Modeling OS behavior

To ensure that the event handlers of a scheduling policy are consistent with the kernel behavior, the DSL compiler must be aware of the control and data flow through the scheduling points in the kernel. To provide this information, our description of OS behavior consists of two parts: a collection of *event sequences* that describe possible sequences of Bossa events, and a collection of *event types* that describe possible inputs and corresponding outputs for each scheduling event handler. These are interdependent, in that the provided event types must describe a behavior for all inputs that can occur along control paths allowed by the event sequences. We first present event sequences and event types, and then show how this information is used in the Bossa DSL compiler.

3.1. Event sequences

Linux 2.2 kernel code exhibits a form of pseudo-parallelism, in that execution of a system call by an application can be interrupted at any point by execution of interrupt handlers. We use automata to describe the sequences of events explicitly notified during the treatment of a system call and during the treatment of an interrupt, and then consider the possible interactions between the sequences described by these automata.

In the Linux 2.2 kernel, several common patterns of scheduling events occur in the implementation of system calls. These patterns are described by the automaton shown in Figure 4 (double-circles in the automaton represent the point at which control returns from the system call to the application). The top two branches describe the behavior of system calls that perform at most one scheduling event, such as a request to yield the processor, possibly followed by a `bossa.schedule` event. The remaining branches describe event sequences used by system calls that repetitively yield the processor while waiting for access to a resource.

As in many Unix systems, the treatment of an interrupt in the Linux 2.2 kernel is divided into a very short interrupt

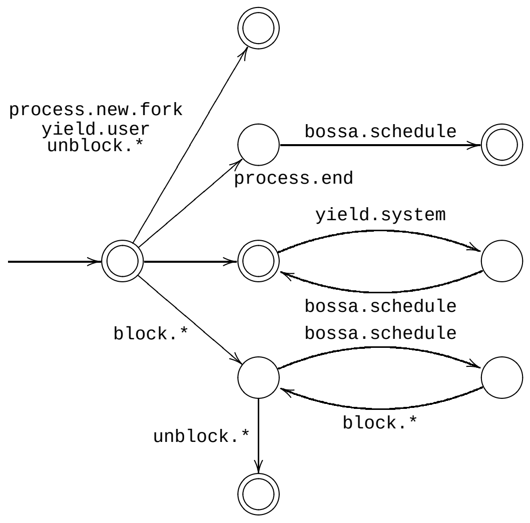

Figure 4. Automaton of Bossa events occurring in system calls

handler, which runs with interrupts disabled, followed by more complex *bottom-half* code, which runs with interrupts enabled. Scheduling points only occur in bottom-half code. The automaton shown in Figure 5 describes the possible sequences of Bossa events used at these scheduling points:

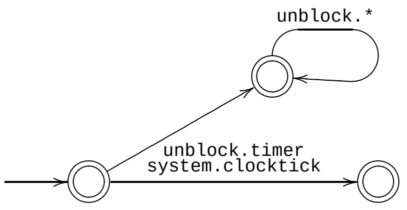

Figure 5. Automaton of Bossa events occurring in interrupts

The complete set of event sequences consists of any sequence allowed by any interleaving of any number of instances of the automata. For example, given the two automata above, `system.clocktick` → `block.*` → `unblock.*` → `bossa.schedule` → `block.*` → `unblock.*` is a valid event sequence.

3.2. Event types

Event sequences describe the possible control flow through the event handlers of a scheduling policy. Complementarily, event types describe the possible inputs of each event handler, and the expected result for each such possible input. Event types are classified as those that describe OS behavior local to a single system call or a single interrupt handler, and those that describe the expected event handler behavior taking into account the effects of event sequences on process states.

Event types describing local behavior. An event type specifies the following properties: (i) the emptiness or non-emptiness of the states associated with each class, (ii) the classes associated with the states of the source and target processes of the event, and (iii) the permitted and required movement of processes between classes during the handler. An event can be associated with multiple types, reflecting the various possible states of processes on entry to the event handler and the various effects that the event handler is allowed to produce. For example, `system.clocktick` should intuitively either leave the running process in its current state, if the process has ticks remaining, or preempt the process if the process has exhausted its time slice; `block.*` should move the running process to a `BLOCKED` state; `unblock.*` should move the target process from a `BLOCKED` state to a `READY` state. These behaviors are described by the following event types:

`system.clocktick:`
⟨target ∈ `RUNNING`⟩ → ⟨target ∈ `RUNNING`⟩
⟨target ∈ `RUNNING`⟩ → ⟨target ∈ `READY`⟩
⟨target ∈ `RUNNING`⟩ → ⟨target ∈ `BLOCKED`⟩

`block.*:`
⟨target ∈ `RUNNING`⟩ → ⟨target ∈ `BLOCKED`⟩

`unblock.*:`
⟨target ∈ `BLOCKED`⟩ → ⟨target ∈ `READY`⟩

In addition to describing OS properties, event types can also be used to express constraints of the Bossa framework. In particular, the Bossa framework ensures that the `system.clocktick` handler is only called when there is a running process, and the `bossa.schedule` handler is only called when there is no running process. While these constraints could be built into the Bossa compiler, expressing them at the level of event types makes all constraints on handler behavior accessible to the policy developer within a uniform framework.

Event types describing event sequence behavior. The Bossa state of a process reflects the schedulability of a process, rather than whether or not the process is currently running. For example, a `block.*` event places the target process in a `BLOCKED` state, indicating that the process is not currently schedulable, but it is not until the next `bossa.schedule` event that the process is actually preempted. Because state changes can occur in both system calls and interrupts, and the actions of system calls and interrupts are unknown to each other, the Bossa state of a process can be different than the state expected by the context in which the process is used. Event types must thus take into account all possible states of the source and target processes that can result from event sequences.

We first consider how a process can have an unexpected Bossa state in the treatment of a system call. In the Linux 2.2 kernel it is not possible to switch to a new process during bottom half code. Consequently, an interrupt bottom half that changes the state of the running process to indicate preemption does not actually affect the running process until the next `bossa.schedule` event, which triggers a context switch. Other events that can be performed by the application before this `bossa.schedule` event must thus take into account the possibility that the currently running process is not actually in the `RUNNING` state. The Linux 2.2 kernel avoids this issue because the operation of updating a process state is independent of the current state of the process. This approach, however, is difficult to understand and error-prone, because it relies on implicit conventions about the possible state of the affected process at the point of the state change. Bossa event types characterize both the source and target state of each allowed transition. This approach improves readability and safety of a policy, but implies that multiple, sometimes unintuitive, cases must be explicitly accounted for.

As an example of the interaction between interrupts and system calls, consider an event sequence beginning with a `system.clocktick` event in an interrupt bottom half followed by a `block.*` event in a system call. The type of `system.clocktick` allows the handler to move the process in the `RUNNING` state to a `RUNNING`, `READY`, or `BLOCKED` state. The event type of `block.*` must thus describe the expected behavior when the target of the event is in a state associated with any of these classes. The complete set of event types for `block.*` is thus:

$$\begin{array}{l}
\langle \text{target} \in \texttt{RUNNING} \rangle \rightarrow \langle \text{target} \in \texttt{BLOCKED} \rangle \\
\langle [\,] = \texttt{RUNNING}, \text{target} \in \texttt{READY} \rangle \\
\quad \rightarrow \langle \text{target} \in \texttt{BLOCKED} \rangle \\
\langle [\,] = \texttt{RUNNING}, \text{target} \in \texttt{BLOCKED} \rangle \\
\quad \rightarrow \langle \text{target} \in \texttt{BLOCKED} \rangle
\end{array}$$

The inconsistency between the expected state of a process and its Bossa state can also occur in the treatment of an interrupt. To obtain access to a resource, one strategy used in the Linux 2.2 kernel is for a process first to add itself to a wait queue, then to test the availability of the resource, and finally to block if the resource is not available. In this case, the interrupt that unblocks waiting processes when the resource becomes available may actually find that a waiting process has not yet blocked, and thus is still running.[2] Thus, the event type for `unblock.*` must account for the case where the target process is in the `RUNNING` state. Furthermore, because `system.clocktick` followed by `unblock.*` is a valid event sequence, the event type for `unblock.*` must account for the case where the target process is placed by `system.clocktick` in a `READY` or `BLOCKED` state as well. The complete set of event types for `unblock.*` is thus:

$$\begin{array}{l}
\langle \text{target} \in \texttt{BLOCKED} \rangle \rightarrow \langle \text{target} \in \texttt{READY} \rangle \\
\langle \text{target} \in \texttt{RUNNING} \rangle \rightarrow \langle \text{target} \in \texttt{RUNNING} \rangle \\
\langle \text{target} \in \texttt{READY} \rangle \rightarrow \langle \text{target} \in \texttt{READY} \rangle \\
\langle \text{target} \in \texttt{BLOCKED} \rangle \rightarrow \langle \text{target} \in \texttt{BLOCKED} \rangle
\end{array}$$

The set of event types induced by the event sequences for `unblock.*` illustrates a case where the event sequences require adding a type for an input configuration for which a type based on local behavior is already available. The `system.clocktick` event can place the target process in a `BLOCKED` state, but this `BLOCKED` state is not necessarily one that represents processes waiting for a resource, and thus to be moved to a `READY` state by `unblock.*`. The new type $\langle \text{target} \in \texttt{BLOCKED} \rangle \rightarrow \langle \text{target} \in \texttt{BLOCKED} \rangle$ allows the `unblock.*` handler to leave such processes in their current state. This example illustrates the trade-off between the goal of a precise description of OS behavior and the goal of a description that is applicable to diverse scheduling policies.

The complete set of event types for the Linux 2.2 kernel is shown in Appendix A.

3.3. Event sequences and event types in the Bossa compiler

Bossa provides an event-type compiler to be used by the OS expert and a policy compiler to be used by the policy designer, as illustrated by Figure 6. The event-type compiler checks that the event sequences and the event types are mutually consistent. The policy compiler uses the event types and event sequences to verify and optimize the policy according to the kernel behavior and to produce the corresponding C code.

The event-type compiler first checks that each of the automata is consistent with the event types describing the corresponding local behavior (*i.e.*, in each sequence described by an automaton, the output of one event must be an acceptable input for the next event). The compiler then checks the completeness of the provided types. For this purpose it propagates the input configuration of each provided type

[2]This case is also checked for by the Linux 2.2 kernel, which treats an unblock event differently depending on whether the target is blocked or already in the runqueue.

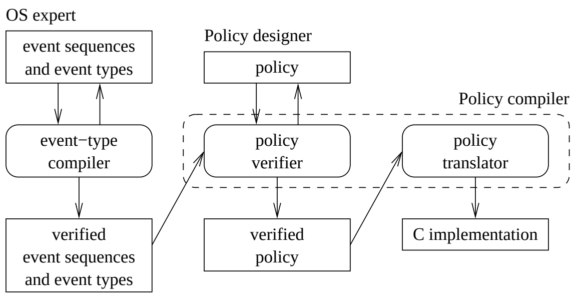

Figure 6. Structure of the Bossa DSL compiler

through the intervening possible event sequences to identify all possible states of the source and target processes at the point at which the event is actually called, and checks that an event type is provided for each possible case. If this step fails, the event-type compiler returns to the OS expert a list of the input configurations for which no event type is provided, as well as a possible event sequence leading to each such configuration. When verification of the event sequences and event types succeeds, the event type compiler produces a representation of this information suitable for linking with the policy compiler.

Once the event sequences and event types are accepted by the event-type compiler, they are used to create an instance of the compiler for the targeted OS. For a given scheduling policy, the Bossa compiler performs an interprocedural analysis of the specified event handlers, guided by the event sequences, and checks that each event handler implements the behavior required by its event types on all reachable inputs. The policy compiler then translates the policy into C code. Information about process states derived from the event types is used to eliminate unnecessary tests in the generated code.

4. Verification and optimization

The main benefit of event types is to enable error detection. In our experience, the interactions captured by the event sequences are easily overlooked, and the verifications performed by the Bossa compiler based on event types detected several missing cases in our initial implementation of the Linux 2.2 policy. The event types also enable the compiler to eliminate some unnecessary tests. In the Linux 2.2 policy, these primarily affect the calculation of the scheduler state. As shown in Appendix A, in several of the event types derived from the event sequences the `RUNNING` class is known to be empty. In these cases, the compiler can statically determine that the scheduler state cannot be `RUNNING`. The use of the event sequences by the compiler allows the compiler to avoid analysis with respect to input configurations that are allowed by the event types but are not reachable in a given policy. For example, the `yield.user` handler of the Linux 2.2 policy, shown below, can place a process in the `yield` process variable. This operation is only valid if `yield` is empty, but the event type of `yield.user` does not give any information about `BLOCKED`, which is the class of the `yield` variable. The event sequences enable the compiler to determine that `yield` is always empty when the code `running => yield` is executed.

```
On yield.user {
  if (!empty(READY) &&
        e.target in running) {
    if (running.policy == SCHED_OTHER) {
      running => yield;
    }
    else {
      running => ready;
    }
  }
}
```

5. Related Work

Recently, there has been growing interest in new approaches to verification of operating systems code, including the introduction and checking of assertions [10], verification that common sequences of operations are used in an expected way [9], as well as more complex techniques such as model checking [5, 15] and theorem proving [12]. Nevertheless, all of these approaches are limited by the lack of domain-specific knowledge, and thus exhaustive static verification of important properties is often infeasible.

The work on Composable Execution Environments (CEE) includes a logic for specifying scheduling-specific properties [17]. The goal of this logic is to check whether undesirable interactions, such as deadlocks, can occur when a given combination of existing schedulers manage a particular set of tasks. Our goal is orthogonal: to describe the interface between an existing Bossa-ready scheduler and new scheduling policies.

The integration of a Bossa policy into an OS is analogous to the weaving performed by an aspect language [2]. The techniques for describing the behavior of the target system proposed here could be useful to verify the correctness of systems constructed using aspects, in particular aspect-oriented OS components [8].

6. Conclusion

In this paper, we have presented a modular type system describing OS properties for use with the Bossa DSL. The information contained in these types significantly increases the amount of verification that can be performed at compile time, thus providing confidence in the correctness of the policy. Some compiler optimizations are also enabled. The separation of these types from the language implementation should facilitate the porting of Bossa to other operating systems. A port to Linux 2.4 is currently underway.

References

[1] A. Atlas and A. Bestavros. Design and implementation of statistical rate monotonic scheduling in KURT Linux. In *IEEE Real-Time Systems Symposium*, pages 272–276, 1999.

[2] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming OS schedulers with domain-specific languages and aspects: New approaches for OS kernel engineering. Int. Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD, April 2002.

[3] L.P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, March 2002.

[4] L.P. Barreto, G. Muller, J.L. Lawall, and K. Kono. A framework for simplifying the development of kernel schedulers: design and performance evaluation. Technical Report 02/8/INFO, Ecole des Mines de Nantes, 2002.

[5] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A language-based approach to protocol construction. In *Proceedings of the ACM SIGPLAN Workshop on Domain Specific Languages*, Paris, France, January 1997.

[6] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportional fair scheduling in multiprocessor servers arbitrary deadlines. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.

[7] S. Chandra, B. Richards, and J.R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 237–248, 1996.

[8] Y. Coady, G. Kiczales, J.S. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can aspect-oriented programming help? In *Proceedings of the Tenth ACM SIGOPS European Workshop (EW'2002)*, Saint-Emilion, France, September 2002. To appear.

[9] A. Engler, D. Yu, 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 Systems Principles*, pages 57–72, Banff, Canada, October 2001.

[10] T. Jim, Morrisett G, Grossman D, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[11] I. Kouvelas and V. Hardman. Overcoming workstation scheduling problems in a real-time audio tool. In *Proceedings of the USENIX 1997 Conference*, pages 235–242, Anaheim, CA, January 1997.

[12] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

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

[14] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 184–197, October 1997.

[15] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 488–497, Limerick, Ireland, June 2000.

[16] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, pages 89–102, October 2001.

[17] J. Regehr, A. Reid, K. Webb, and J. Lepreau. Composable execution environments. Flux group technical note 2002-02, University of Utah, May 2002.

[18] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI-99)*, pages 145–158, February 1999.

[19] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 234–243, Austin, Texas, May 1999. IEEE Computer Society Press.

[20] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, 5(4):475–488, August 1997.

A. Event types for the Linux 2.2 kernel

Event types are classified as being derived from local behavior or being derived from interactions due to event sequences. The type of a hierarchical event applies to all more specific events for which no other information in that section is provided. For example, the types based on event sequences for `yield.*` also apply to the `yield.user`, `yield.system.pause`, and `yield.system.immediate` events.

A.1 Event types describing local behavior

```
process.new.* :
 [tgt in NOWHERE] -> {[tgt in READY], [tgt in BLOCKED]}

process.new.fork :
 [src in RUNNING, tgt in NOWHERE] -> {[src in RUNNING, tgt in READY],
                                      [src in RUNNING, tgt in BLOCKED]}

process.new.initial_process :
 [[] = RUNNING, tgt in NOWHERE] -> {[[] = RUNNING, tgt in READY],
                                    [[] = RUNNING, tgt in BLOCKED]}

process.end : {[tgt in BLOCKED], [tgt in READY]} -> [tgt in TERMINATED]

process.end : [tgt in RUNNING] -> [tgt in TERMINATED]

yield.* :
 [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

yield.system.* :
 [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

yield.system.pause :
 [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

yield.system.immediate :
 [tgt in RUNNING] -> { [tgt in READY], [tgt in RUNNING] }

yield.user :
 [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

system.clocktick :
 [tgt in RUNNING] -> { [tgt in READY], [tgt in BLOCKED], [tgt in RUNNING] }

block.* : [tgt in RUNNING] -> [tgt in BLOCKED]

unblock.* : [tgt in RUNNING] -> [tgt in RUNNING]

unblock.* : [p in RUNNING, tgt in BLOCKED]
            -> { [[p,tgt] in READY],
                 [p in RUNNING, tgt in READY],
                 [p in RUNNING, tgt in BLOCKED] }

unblock.* : [[] = RUNNING, tgt in BLOCKED]
            -> { [[] = RUNNING, tgt in READY],
                 [[] = RUNNING, tgt in BLOCKED] }

unblock.synchronous_wake_up_process : [tgt in RUNNING] -> [tgt in RUNNING]

unblock.synchronous_wake_up_process :
 [tgt in BLOCKED] -> { [tgt in READY], [tgt in BLOCKED] }

unblock.wake_up_process : [tgt in RUNNING] -> [tgt in RUNNING]

unblock.wake_up_process : [p in RUNNING, tgt in BLOCKED]
                          -> { [[p,tgt] in READY],
                               [p in RUNNING, tgt in READY],
                               [p in RUNNING, tgt in BLOCKED] }

unblock.wake_up_process : [[] = RUNNING, tgt in BLOCKED]
                          -> { [[] = RUNNING, tgt in READY],
                               [[] = RUNNING, tgt in BLOCKED] }

unblock.timer : [tgt in RUNNING] -> [tgt in READY]

unblock.timer : [tgt in READY] -> [tgt in READY]

unblock.timer : [p in RUNNING, tgt in BLOCKED] ->
                     { [p in RUNNING, tgt in BLOCKED],
                       [p in RUNNING, tgt in READY],
                       [[p,tgt] in READY] }

unblock.timer : [[] = RUNNING, tgt in BLOCKED]
                -> { [[] = RUNNING, [tgt] in READY],
                     [[] = RUNNING, [tgt] in BLOCKED] }

bossa.schedule : [[] = RUNNING, q in READY] -> [q in RUNNING, READY!]
}
```

A.2 Event types describing event sequence behavior

```
unblock.timer :
[tgt in TERMINATED] -> [tgt in TERMINATED]

unblock.* :
[tgt in TERMINATED] -> [tgt in TERMINATED]

unblock.* :
[tgt in READY] -> [tgt in READY]

yield.* :
[[] = RUNNING, tgt in BLOCKED] -> [tgt in BLOCKED]

yield.* :
[[] = RUNNING, tgt in READY] -> { [tgt in READY], [tgt in BLOCKED] }

unblock.wake_up_process :
[tgt in READY] -> [tgt in READY]

unblock.synchronous_wake_up_process :
[tgt in READY] -> [tgt in READY]

process.new.fork :
[tgt in NOWHERE, [] = RUNNING, src in BLOCKED] ->
  {[src in BLOCKED, tgt in READY], [[src,tgt] in BLOCKED]}

process.new.fork :
[tgt in NOWHERE, [] = RUNNING, src in READY] ->
  {[[src,tgt] in READY], [src in READY, tgt in BLOCKED]}

block.* :
[[] = RUNNING, tgt in BLOCKED] -> [tgt in BLOCKED]

block.* :
[[] = RUNNING, tgt in READY] -> [tgt in BLOCKED]
```