

Modularity for the Bossa process-scheduling language

Julia L. Lawall

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

Anne-Françoise Le Meur

DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
lemeur@diku.dk

Gilles Muller

Obasco Group, EMN/INRIA
44307 Nantes Cedex 3, France
gilles.muller@emn.fr

Abstract Bossa is a framework for implementing scheduling policies in standard operating systems. This framework provides a domain-specific language that eases the specification of scheduling policies and enables verification of critical safety properties. In previous work we have specified the semantics of the Bossa language and the verification process. In this work, we propose a modular version of the language, to take advantage of the many opportunities for code reuse and modularity provided by scheduling policies.

1 Introduction

Although process scheduling is an old problem, there is still no perfect scheduler for all applications. From real-time systems to multimedia applications to energy-restricted embedded systems and beyond, applications have varied scheduling needs. Still, few scheduling policies are available in standard operating systems (OSes). We have developed the Bossa framework to enable programmers to easily and safely implement scheduling policies [3, 4]. This framework provides a domain-specific language (DSL) for programming policies and an event-based run-time system that integrates policies with a standard OS kernel. Using this approach, schedulers can be implemented safely and concisely, typically in under 200 lines of code.

The syntax of the Bossa DSL is based on C, as this language is familiar to systems programmers. Using this syntax, a scheduling policy is specified as a collection of event handlers and other supporting declarations. This monolithic approach, however, does not take advantage of the many opportunities for code reuse and modularity. Scheduling policies come in families, such as policies for periodic processes or policies that address processor overload [1], and policies in these families share many commonalities. Furthermore, a scheduling policy is often composed of a set of orthogonal concerns, such as blocking and unblocking, yielding, and timer management, that can be shared across multiple families. Indeed, a library

of implementations of such concerns could facilitate experimentation with new policy variants. We thus propose to explore a modular approach to specifying scheduling policies in which a scheduler is expressed as the composition of a set of modules, each addressing a single scheduling concern.

2 Overview of the Approach

A Bossa scheduling policy is defined in terms of a set of features comprising states, global variables, process attributes, priority criteria, event handlers, and interface functions. The states, global variables, and process attributes store information describing the eligibility and priority of processes for access to the CPU. The priority criteria determine the relative priority of processes. The event handlers and interface functions use all of this information to react to scheduling-related kernel events and user-level requests, respectively. A scheduling policy must define a complete set of event handlers, as specified by the associated run-time system for the target OS.

Isolating orthogonal scheduling concerns in separate modules implies the need to partition the various features of a scheduling policy according to these concerns. Figure 1 shows a scheduler `MyScheduler` that is defined in terms of the modules `Simple`, `Yield` and `Clock`, which address the blocking, yielding, and timer-management concerns, respectively. Essentially, a scheduler amounts to the union of the features defined in the constituent modules. In Figure 1, only the event handlers are shown. Some handlers are defined in only one module, while others are defined in multiple modules and composed by the scheduler.

Because process scheduling is such an important part of the overall functioning of a system, it is essential that the policy designer be able to maintain a global view of the policy behavior, even when using a modular approach. Policy features may be either centralized in the scheduler or localized in the modules. In the design of a modular version of Bossa, both strategies are taken, in different cases, with the

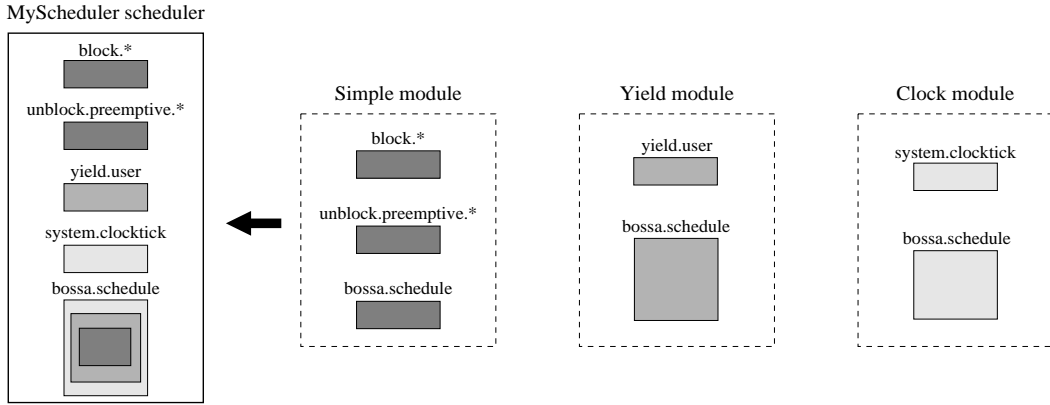


Figure 1: Composition of the scheduler MyScheduler

goal of balancing the need for a global view with the need to define separate reusable units.

2.1 Defining a scheduler

A scheduler definition combines a set of modules to form the implementation of a complete scheduling policy. Figure 2 defines the scheduler MyScheduler, illustrated previously in Figure 1.

The definition of a scheduler begins with the declaration of the process states. Process states are at the heart of a scheduling policy, as they determine process eligibility. Thus, the declaration of process states is centralized in the scheduler, and process states are passed as needed to the individual modules. Each state declaration includes a state class attribute and an implementation attribute. The state class attribute, `RUNNING`, `READY`, `BLOCKED`, or `TERMINATED`, describes the schedulability of processes in the state. The implementation attribute, `process` or `queue`, describes the data structure used to record the set of processes in the state.

Module combination is described using the construct `uses modules with constraints`, where `modules` is a list of module names with their state arguments and `constraints` describes constraints on how features of these modules are composed. The set of features defined by a scheduler is essentially the union of the set of features defined by the listed modules. Process attributes that are declared in one module and referenced in another must be connected explicitly using the `process` declaration, as described in Section 3.3. A similar mechanism is provided for variables. The `ordering_criteria` describes the relative priority of processes. As for the states, this information is centralized in the scheduler to provide a global view of the policy behavior. The `handler` declaration

```

scheduler MyScheduler = {
  states = {
    RUNNING running : process;
    READY  ready  : sorted queue;
    READY  expired : queue;
    READY  yield  : process;
    BLOCKED blocked : queue;
    TERMINATED terminated;
  }

  uses { Simple (running, ready, blocked),
        Yield (yield, ready),
        Clock (expired, ready, running) }

  with {
    process { Clock.quantum = Simple.priority }
    ordering_criteria = { highest Clock.ticks }
    handler {
      bossa.schedule =
        Clock.bossa.schedule extends
        Yield.bossa.schedule extends
        Simple.bossa.schedule;
    }
  }
}

```

Figure 2: Definition of the scheduler MyScheduler

specifies how multiple handlers for a given event are composed. All handlers for the event must be included in the composition. The order of composition is independent of the order in which the modules are mentioned in the `uses` declaration, and can differ for every event. Interface functions are treated similarly.

Based on the included modules and the composition constraints, the Bossa compiler checks that the set of handlers defined by the composed scheduler is complete and that all of the definitions made by the modules are used and correctly instantiated.

```

module Simple (RUNNING process running,
              READY sorted queue ready,
              BLOCKED queue blocked) {
  process { int priority; }

  handler (event e) {
    On block.* { e.target => blocked; }
    On unblock.* {
      if (e.target in blocked) {
        e.target => ready;
      }
    }
    On bossa.schedule { select() => running; }
  }

  interface {
    set_priority(process p, int priority) {
      p.priority = priority;
    }
  }
}

```

Figure 3: Definition of the module `Simple`

2.2 Defining a module

A module definition declares the features relevant to a scheduling concern. Figure 3 defines the module `Simple` that addresses the process blocking and unblocking concern. This module is parameterized by the states `running`, `ready`, and `blocked`. It declares a `priority` process attribute and an associated interface function `set_priority`, as well as three event handlers: `block.*`, `unblock.*`, and `bossa.schedule`. The handlers `block.*` and `unblock.*` react to process blocking and unblocking events, respectively. The handler `bossa.schedule` elects a process for access to the CPU.

In the handlers, `e.target` refers to the process affected by the event, the binary operator `in` tests whether a process is in a given state, the binary operator `=>` changes the state of a process, and the function `select()` identifies the highest priority process that is eligible for execution.

3 Sharing between Modules

Although each module should represent a separate concern, it is necessary to allow some sharing of information between modules. Three kinds of information can be shared: (i) handlers and interface functions, (ii) states, and (iii) variables and process attributes. In designing the semantics of this sharing, our goal has been to minimize the possibility of unexpected

```

module Yield (READY unshared process yield,
             READY sorted queue ready) {
  handler (event e) {
    On yield.* { e.target => yield; }
    On bossa.schedule {
      if (empty(ready)) { yield => ready; }
      super();
      if (!empty(yield)) { yield => ready; }
    }
  }
}

```

Figure 4: Definition of the module `Yield`

interactions between modules.

3.1 Handlers and interface functions

It is common that multiple modules need to react to the same scheduling event. For example, all of the modules shown in Figure 1 define a `bossa.schedule` handler, each contributing concern-specific information to the strategy for electing a new process. When a handler is defined in more than one module, the scheduler definition makes explicit the order in which the handlers should be composed, using the `extends` declaration (see Figure 2). In such a composition, each handler except the last one must use the construct `super()` to indicate the point at which the next handler in the composition should be invoked. To ensure that no behavior is duplicated or discarded, `super()` must be invoked exactly once on every control-flow path through such a handler.

The `Yield` module, shown in Figure 4, illustrates the use of `super()`. This module allows a process to defer the rest of its time slice to any other eligible process. The yielding process is stored in the state `yield` and is to be considered eligible for election only if there is no other eligible process. This strategy is implemented by the `bossa.schedule` handler, which puts the process in the `ready` state (the state of eligible processes) only if there is no other process in this state. The handler then uses `super()` to invoke the next `bossa.schedule` handler in the composition. In the composition defined by `MyScheduler` (Figure 2), this use of `super()` invokes the `bossa.schedule` handler of the `Simple` module, which elects a new process.

3.2 States

Normally, a state can be used freely in multiple modules, in particular, in any module to which it is

passed as an argument. For example, the scheduler `MyScheduler` (Figure 2) passes the state `ready` to all three of its constituent modules. In some cases, however, it is important for the integrity of a module that only the handlers of that module can manipulate a given state. For example, correct functioning of the `Yield` module requires that the process referred to by `yield` in `bossa.schedule` is the same as the process most recently placed in this state by the `yield.*` handler. Thus, the `yield` state should not be accessed by any other module. Such a state can be declared as `unshared`, as shown in the declaration of `yield` in Figure 4. The Bossa compiler checks that the state bound to a parameter declared as `unshared` is not passed to any other module.

The `unshared` declaration can affect the set of handlers that must be defined by a module. Specifically, the Bossa language requires that any process in a state declared as `READY` (thus representing processes that are able to run) must be considered for election, at least if no other process is eligible. Thus, a module that declares an `unshared READY` state, such as `yield`, must define a `bossa.schedule` handler that takes this `unshared` state into account. The Bossa compiler checks that `bossa.schedule` is defined in any such module.

3.3 Variables and process attributes

A module can import variables and process attributes from other modules by using the keyword `requires`. To avoid unexpected interactions between modules, imported variables and attributes are read-only. The source of each imported variable or attribute is determined by the scheduler that uses the module.

The `Clock` module, shown in Figure 5, illustrates the importing of process attributes. This module implements a scheduling strategy that allocates a quantum of CPU time to each process. The module keeps track of the remaining portion of the quantum of each process in a `ticks` attribute (initialized to the quantum in the handler `process.new`, which is not shown). On each `system.clocktick` event, *i.e.*, on each `system.clocktick` event, the handler decrements the value of the `ticks` attribute of the running process. When this value reaches 0, `ticks` is reset to the quantum and the running process is moved to an `expired` state. The handler `bossa.schedule` returns expired processes to the `ready` state when there are no other `ready` processes, at which point they again become eligible for election. This strategy is independent of how the quantum is defined, and thus `quantum` is declared as an imported process attribute.

```

module Clock (READY unshared queue expired,
              READY sorted queue ready,
              RUNNING process running) {
  process = {
    int ticks;
    requires int quantum;
  }

  handler (event e) {
    On system.clocktick {
      running.ticks--;
      if (running.ticks <= 0) {
        running.ticks = running.quantum;
        running => expired;
      }
    }
  }

  On bossa.schedule {
    if (empty(ready)) {
      foreach (p in expired) { p => ready; }
    }
    super();
  }
}

```

Figure 5: Definition of the module `Clock`

The connection between imported process attributes and their definition is made in the scheduler definition. The definition of `MyScheduler` contains:

```
process { Clock.quantum = Simple.priority }
```

This declaration indicates that the quantum process attribute of the `Clock` module refers to the `priority` process attribute of the `Simple` module. Imported variables are treated similarly.

4 Assessment

In our experience, the set of event handlers required by the Bossa framework can easily be categorized into separate concerns, leading to a straightforward decomposition of a scheduling policy into modules. Table 1 lists the modules used in implementing a collection of standard schedulers. `RR` is a round-robin scheduling policy. `Linux` is the scheduling policy of the Linux 2.4 OS. `RM` is the Rate Monotonic scheduling policy [1]. `EDF` is the Earliest-Deadline First scheduling policy [1]. In each case, the scheduler implementation uses 6 or 7 modules. Some of these modules are very generic, such as `Block` (blocking, unblocking and process election) which is used in multiple policies, or are generic to certain kinds of imple-

Module	Scheduler				LOC
	RR	Linux	RM	EDF	
Block	x	x	x	x	25
PreemptiveUnblock		x	x	x	14
Preempt		x	x	x	7
Interrupted	x				15
Fork	x		x	x	12
FIFOTimeslice	x				29
NoTimeslice			x	x	5
LinuxTimeslice		x			107
Yield			x	x	25
FIFOYield	x				9
LinuxYield		x			36
RRPriority	x				23
RunningActiveMM		x			24
RMTimer			x		60
EDFTimer				x	78
LOC (lines of code)	31	36	33	33	

Table 1: The use of modules by various schedulers

mentation strategies, such as `FIFOYield` which is appropriate for policies using a FIFO ready queue. Others, such as the last four modules listed in Table 1, are policy-specific. Each of the policies shares some modules with the others. The RM and EDF schedulers, which are both targeted towards the management of periodic real-time processes, share almost all of their constituent modules. These experiments suggest the interest of a library of modules, exhibiting various degrees of genericity, to be used in implementing both new scheduling policies and variants of existing ones.

5 Conclusion and Future Work

We have presented a new approach to describing a scheduling policy as a set of modules, each addressing a separate scheduling concern. These modules can provide a high degree of reusability within a given family of scheduling algorithms and ease experimentation with new variants.

Our approach was originally inspired by *traits* [5], which are lightweight composable units of behavior, designed to address the problems of multiple inheritance in object-oriented languages. From traits, we have adopted the idea of a commutative composition of units. Nevertheless, other strategies taken by traits did not seem appropriate in the context of Bossa. Trait composition discards the definitions of names that are defined in multiple traits. Because of the critical nature of scheduling code, we felt that discarding any provided behavior would be too risky, and thus the Bossa syntax requires the programmer to specify how the definitions associated with such names should be composed. Traits are stateless, to avoid problems that arise when inheriting a single state object indirectly via multiple inheritance paths. Again to enhance safety, we preferred to define all

state locally in the relevant modules. Because the Bossa syntax only allows one level of inheritance, state information can only be inherited via one inheritance path.

The current Bossa DSL is structured around the handling of kernel-specified events, and we have maintained this structure here. Some concerns apply, however, not to events, but to particular kinds of state changes, *e.g.*, all changes from a state in the `RUNNING` state class to a state in the `READY` state class, which represent preemption. Such concerns can be said to be *crosscutting* [2]. An example is the monitoring of the elapsed execution time. Introducing such code by hand in a Bossa policy is error-prone, because the source state is not explicit in a Bossa state-change operation. We are currently considering how to associate behaviors with specific state changes in Bossa.

Acknowledgments We thank Yvonne Coady for comments on a previous draft of this paper.

References

- [1] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mameri. *Scheduling in Real-Time Systems*. John Wiley & Sons, Ltd., 2002.
- [2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997.
- [3] J. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *ACM SIGOPS European Workshop 2002 (EW'2002)*, pages 54–61, Saint-Emilion, France, Sept. 2002.
- [4] J. Lawall, G. Muller, and A.-F. L. Meur. On the design of a domain-specific language for OS process-scheduling extensions. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, Vancouver, Canada, Oct. 2004. To appear.
- [5] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, number 2743 in Lecture Notes in Computer Science, pages 248–274, Darmstadt, Germany, July 2003.