Bossa Nova: Introducing Modularity into the Bossa Domain-Specific Language

Julia L. Lawall,[1] Hervé Duchesne,[2] Gilles Muller,[2] Anne-Françoise Le Meur[3]

[1] DIKU, University of Copenhagen, Denmark
[2] OBASCO Group, École des Mines de Nantes-INRIA, LINA, France
[3] Jacquard Group, LIFL/INRIA, Université des Sciences et Technologies de Lille, France

Abstract. Domain-specific languages (DSLs) have been proposed as a solution to ease the development of programs within a program family. Sometimes, however, experience with the use of a DSL reveals the presence of subfamilies within the family targeted by the language. We are then faced with the question of how to capture these subfamilies in DSL abstractions. A solution should retain features of the original DSL to leverage existing expertise and support tools.
The Bossa DSL is a language targeted towards the development of kernel process scheduling policies. We have encountered the issue of program subfamilies in using this language to implement an encyclopedic, multi-OS library of scheduling policies. In this paper, we propose that introducing certain kinds of modularity into the language can furnish abstractions appropriate for implementing scheduling policy subfamilies. We present the design of our modular language, Bossa Nova, and assess the language quantitatively and qualitatively.

1 Introduction

Domain-specific languages (DSLs) have been proposed as a solution to ease the development of programs within a program family. A DSL is designed according to the results of a domain analysis, and provides high-level, domain-specific abstractions that facilitate programming in the domain and enable verification of domain-specific properties. Such languages have made programming accessible to non-experts in areas as varied as web-services [1, 4, 31] and animation [10].

Despite the success of DSLs, such languages are limited by the scope of the initial domain analysis. The ease of programming with a DSL may, however, lead programmers in unanticipated directions. In particular, experience with a DSL may reveal subfamilies within the family targeted by the language. We must then consider how to extend the DSL to provide appropriate abstractions for capturing these subfamilies. A solution should maintain the character of the DSL, to leverage the expertise and support tools developed around the language.

One approach that can enable a DSL to adapt to unanticipated needs is to embed the language in an existing richer general-purpose language [10, 14, 29]. The DSL can then inherit host-language features such as types, modules, and objects as the need arises. The inherited features, however, are determined by

what the host language provides, and not by domain needs. Accordingly, code can be difficult to understand, because information is not structured according to its role in the domain, and difficult to verify, because the general-purpose nature of the inherited features precludes introducing constraints to ease verification. We propose instead that DSL extensions should be individually designed according to domain requirements and in harmony with existing abstractions of the DSL. The language can then be embedded or directly compiled, as convenient. We illustrate our proposal in the context of the Bossa DSL for process scheduling [19, 20], which we extend with two forms of modularity.

The Bossa DSL. A process scheduler is the part of an operating system (OS) kernel that allocates the CPU to processes. Because the time at which a process gets access to the CPU affects the timing of all its subsequent actions, process scheduling has a profound effect on application behavior. From real-time systems to multimedia applications to energy-restricted embedded systems and beyond, applications have varied scheduling needs that cannot be met by a single scheduler. Not surprisingly, many scheduling policies have been proposed [2, 6, 9, 15, 24, 25, 28, 30, 33–35]. Still, few of these scheduling policies are available in commonly used OSes. Furthermore, implementing a scheduling policy in a legacy OS kernel is outside the expertise of most application developers.

To ease the implementation of scheduling policies in legacy OSes, we have developed the Bossa framework. Bossa extends a legacy OS with a documented scheduling interface [19] and provides a DSL for implementing scheduling policies [20]. It has been used to implement a variety of scheduling policies, including those for interactive, multimedia, and real-time applications. We have observed significant benefits in the understandability, conciseness, and safety of Bossa schedulers, as compared to direct coding at the OS level. These features have enabled undergraduate students with no previous kernel programming experience to implement schedulers in the Linux kernel without crashing the machine.

The Bossa DSL was designed to facilitate the implementation of one scheduling policy at a time. The ease of scheduler programming in Bossa, however, has lead us to begin implementing an encyclopedic multi-OS library of scheduling policies. This work has highlighted some properties of scheduling policies that were not taken into account in the original design of the language. We have observed that the policies found in the scheduling literature are often classified in families, and that the Bossa implementations of policies within a family have much in common. Furthermore, the code specific to a policy variant is intertwined with the code generic to the family, making it difficult to identify policy-specific features. These issues have called for the introduction of modularity in the Bossa DSL, to enable the separation of concerns and to enhance code reuse.

This paper. In this paper, we address the needs identified in implementing an encyclopedic multi-OS library of scheduling policies by extending the Bossa DSL with two forms of modularity: *modules* and *transition aspects*. Modules separate scheduling concerns while transition aspects permit a module to adapt other modules in a controlled way. The extensions are designed according to a careful

analysis of the requirements of the scheduling domain. We assess the extensions on the implementation of a variety of scheduling policies. As compared to an embedded-language approach where the DSL inherits features from the host language, we find that our approach leads to policies that are more understandable, because information is structured according to the needs of the domain, and more verifiable, because we can constrain the module system in a way that eases verification. The resulting language, Bossa Nova, has been implemented by translation into the Bossa DSL, for which an implementation has previously been developed [19, 20]. Experiments with Bossa Nova have shown no performance overhead as compared to Bossa.

This work represents a case study in what happens when a DSL meets real programming needs. Specifically, we illustrate:

- Motivations for introducing new abstractions into a DSL.
- Goals that should be taken into account in designing these abstractions.
- The choice of specific features that the abstractions should provide to meet these goals.

While the design choices presented here are specific to Bossa, our contributions are to identify individual motivations, goals, and features that should be taken into account in extending DSLs and to illustrate the benefits that can be achieved by this approach.

The rest of this paper is structured as follows. Section 2 presents the Bossa DSL. Section 3 motivates the need for modularity, and presents our design choices. Section 4 assesses the resulting language, Bossa Nova, on numerous examples and compares the proposed forms of modularity to existing approaches. Finally, Section 5 describes related work and Section 6 concludes.

2 Bossa in a Nutshell

We introduce the Bossa DSL using excerpts of an implementation of an Earliest-Deadline First (EDF) scheduling policy [7, 22], shown in Figure 1. This policy manages a set of periodic processes, each of which is associated with a deadline within its current period. Process election chooses the process with the nearest deadline. The complete policy and a grammar of the Bossa DSL are available at the Bossa web site, `http://www.emn.fr/x-info/bossa`. We focus on the main features of the language: declarations and event handlers.

Declarations. The declarations of a scheduling policy define the process attributes, process states, and processes ordering used by the policy.

The `process` declaration (Figure 1, lines 2-3) lists the policy-specific attributes associated with each process. For the EDF policy, these are the period and the Worst-Case Execution Time (WCET) supplied by the process, a timer that is used to maintain the period, the offset of the deadline within each period, and the process's absolute deadline within the current period.

```
scheduler EDF = {                                                              1
  process = { time period; time wcet; timer period_timer;                      2
              time deadline; time absolute_deadline; }                         3
  states = { RUNNING running : process; READY ready : select queue;            4
             READY yield : process; BLOCKED blocked : queue;                   5
             BLOCKED period_yield : queue; TERMINATED terminated; }            6
  ordering_criteria = { lowest absolute_deadline }                             7
  handler (event e) {                                                          8
    On block.* { e.target => blocked; }                                        9
    On bossa.schedule {                                                        10
      if (empty(ready)) { yield => ready; }                                    11
      select() => running;                                                     12
      if (!empty(yield)) { yield => ready; }                                   13
    }                                                                          14
    On unblock.timer.period_timer {                                            15
      e.target.absolute_deadline = now() + e.target.deadline;                  16
      start_relative_timer(e.target.period_timer, e.target.period);           17
      switch e.target in {                                                     18
        case period_yield: {                                                   19
          e.target => ready;                                                   20
          if (!empty(running) && (e.target > running)) { running => ready; }   21
        }                                                                      22
        case running, ready: { e.target => ready; }                            23
        case READY, BLOCKED, TERMINATED: { }                                   24
      }                                                                        25
    }                                                                          26
    ...                                                                        27
  }                                                                            28
}                                                                              29
```

Fig. 1. An excerpt of the EDF scheduling policy

The `states` declaration (lines 4-6) lists the set of process states that are distinguished by the policy. Each state is associated with a state class (`RUNNING`, `READY`, `BLOCKED`, or `TERMINATED`) describing the schedulability of processes in the state. For example, the `ready` state is in the `READY` state class, meaning that it contains processes that are ready to run. A state is also associated with an implementation as either a process variable (`process`) or a queue (`queue`). Finally, the `ready` state is designated as `select`, indicating that processes are elected from this state.

The `ordering_criteria` (line 7) describes how to compare two processes in terms of a sequence of criteria based on the values of their attributes. Higher or lower values are favored using the keywords `highest` and `lowest`, respectively. The EDF policy favors the process with the lowest absolute deadline.

Event handlers. Event handlers describe how a policy reacts to scheduling-related events that occur in the kernel. Examples of such events include process blocking and unblocking and the need to elect a new process.

The EDF policy defines 11 event handlers. Handlers are parameterized by an event structure, `e`, that includes the *target process*, `e.target`, affected by the event, if there is one. The event-handler syntax is based on that of a subset of C, to make the language easy to learn. The syntax provides specific constructs and primitives for manipulating processes and their attributes. These include constructs for testing the state of a process (*exp* `in` *state*), testing whether there is any process in a given state (`empty(`*state*`)`), testing the relative priority of two processes (exp_1 `>` exp_2), and changing the state of a process (*exp* `=>` *state*).

A `block.*` event occurs when a process blocks. The associated handler of the EDF scheduling policy (line 9) simply sets the state of the target process to `blocked`. A `bossa.schedule` event occurs when the kernel would like the policy to elect a new process. The associated handler (lines 10-14) uses the primitive `select()` to choose the highest priority process according to the ordering criteria from the state designated as `select`, *i.e.*, `ready`. It also manages any yielded process. Finally, an `unblock.timer.period_timer` event occurs when a process's `period_timer` timer expires, indicating the start of a new process period. The associated handler (lines 15-26) resets the absolute deadline of the target process (line 16), restarts the timer (line 17), and reschedules the target process for its computation in the new period (lines 18-25). If the process is in the `period_yield` state, meaning that it has completed its computation during its previous period, then its state is changed to `ready`, indicating that it is newly able to run (line 20). If the target process is in the `running` state or the `ready` state, then it is repositioned in the `ready` queue according to its new priority (line 23).

3 Modularity for Bossa

In the Bossa DSL presented above, a scheduling policy is implemented as a single unit, defining a complete set of event handlers. In our experience in developing a library of scheduling policies, we have observed that when scheduling policies are part of the same family, there is much commonality between their implementations. We first illustrate this commonality in the case of policies for managing periodic processes, such as EDF, and argue that this commonality motivates the need for modularity. We then propose a module system tailored to the needs of scheduling policies, which forms the basis of a modular variant of the Bossa DSL, named Bossa Nova. Finally, we briefly describe a second form of modularity, a variant of aspects, that we have also found useful in Bossa Nova.

3.1 The need for modules

Many scheduling policies have been developed for managing periodic processes, including Deadline Monotonic (DM) [7], Earliest-Deadline First (EDF) [7], Fixed Utilization Priority (FUP) [13], Least Compute Time (LCT) [13], Least-Laxity First (LLF) [7], Rate Monotonic (RM) [7], and Shortest Completion Time (SCT) [13]. The set of periodic policies thus amounts to a program subfamily. In implementing these policies in Bossa, we have observed that only the ordering criteria

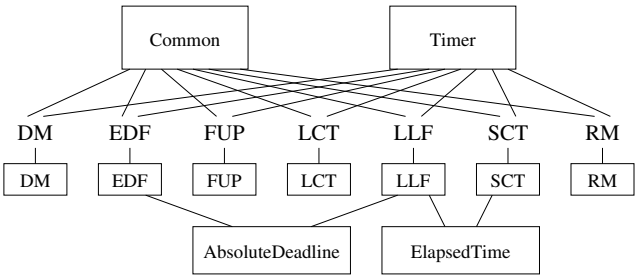

Fig. 2. Modular decomposition of a subset of the family of periodic policies (modules are boxed, policies are unboxed)

and the code calculating the values used in this criteria differ among them. Indeed, among these policies, the average size of the Bossa implementation is 123 lines, of which 100 are common to all of the policies.

One strategy in the face of this large amount of common code is to implement a scheduling policy by copying code from the Bossa implementation of another policy in the same subfamily. Nevertheless, we find that code that is common to the subfamily is mixed with code that is specific to a given policy, requiring careful rewriting of the copied code. This issue suggests the need for a modular programming strategy that separates these concerns, leading to a collection of standard modules that are useful in implementing policies of a given subfamily. Such a modular decomposition of the periodic policies is illustrated in Figure 2.

3.2 Modules for process scheduling

The design of our module system is guided by the requirements of the scheduling domain. As a scheduling policy is a critical component of an OS, its implementation must be understandable and verifiable. Our experience in implementing a library of scheduling policies has further shown the need for code reuse. Accordingly, we structure the module system according to the following principles. To enhance understandability, the module system organizes a scheduling policy as a centralized *scheduler*, giving a global view of the policy behavior, and a collection of modules, each implementing a single scheduling functionality. To enhance verifiability, the module system provides fine-grained control over external access to module elements, making it clear where it is valid to reason in terms of properties local to a module. Finally, to enhance reusability, modules do not refer directly to the other modules making up a given policy. Instead, information about the relationships between modules is localized in the scheduler.

In the rest of this section, we describe how the requirements of understandability, verifiability, and reusability influence the design of the interaction between the module system and the main features of Bossa: process states, process attributes, and event handlers. In each case, the various constraints identified

```
scheduler EDFSched = {                                                          1
  states = { RUNNING running : process; READY ready : select queue;             2
             READY yield : process; BLOCKED period_yield : queue;               3
             BLOCKED blocked : queue; TERMINATED terminated; }                  4
  modules { EDF(),                                                              5
            AbsoluteDeadline(),                                                 6
            Timer (running, ready, period_yield),                               7
            Common (running, ready, yield, blocked, terminated) }               8
  process { EDF.period reads Timer.period,                                      9
            EDF.wcet reads Timer.wcet,                                          10
            EDF.absolute_deadline reads AbsoluteDeadline.absolute_deadline,     11
            AbsoluteDeadline.period_timer reads Timer.period_timer }            12
  ordering_criteria { EDF }                                                     13
  handler { unblock.timer.period_timer : AbsoluteDeadline, Timer; }             14
}                                                                               15
```

Fig. 3. The scheduler used in the Bossa Nova implementation of the EDF policy

are checked by the Bossa Nova compiler. We use the Bossa Nova implementation of the EDF scheduling policy shown in Figures 3 and 4 as an example.

Process states. A main activity of a scheduling policy is to adjust process states, taking into account both OS requirements and the strategy of the policy. Thus, the set of process states manipulated by a policy gives a sense of the scope of the policy's scheduling strategies. To provide a global view of the policy, we define the states centrally in the scheduler, as shown in lines 2-4 of Figure 3. States are then passed to the modules as needed, as shown in lines 5-8 of Figure 3.

A module may only explicitly refer to the states among its parameters. A state change operation *exp* `=>` *state*, however, implicitly references the current state of the process *exp*. We allow this state to be any state defined by the scheduler. This strategy implies that the module does not have to be aware of the complete set of states defined by the policy, and thus facilitates code reuse, but limits the ability to reason about state contents across the handlers of a module. When a module needs to be sure that only it can affect the set of processes in a given state, it can annotate the associated parameter as `unshared`. Such a parameter must be instantiated by the scheduler to a state that is not passed to any other module and no other module can remove processes from the state. An example of such a parameter is `period_yield` (Figure 4, line 16), in which the Timer module stores processes that have completed their computation within a given period. The state used by the scheduler to instantiate this parameter (Figure 3, line 7) is only passed to this module. A global analysis of the scheduling policy shows that no other module removes processes from this state.

Process attributes. Process attributes record process information that persists across successive events. Because this information is typically specific to a sin-

```
module EDF() {                                                          1
  process = { requires time period; requires time wcet;                 2
              requires time absolute_deadline; }                        3
  ordering_criteria = { lowest absolute_deadline }                      4
}                                                                       5
module AbsoluteDeadline() {                                             6
  process = { time deadline; time absolute_deadline; requires timer period_timer; }  7
  handler (event e) {                                                   8
    On unblock.timer.period_timer {                                     9
      e.target.absolute_deadline = now() + e.target.deadline;           10
      next();                                                           11
    }                                                                   12
  }                                                                     13
}                                                                       14
module Timer(RUNNING process running, READY select queue ready,        15
             BLOCKED unshared queue period_yield) {                     16
  process = { time period; time wcet; timer period_timer; }             17
  handler (event e) {                                                   18
    ...                                                                 19
    On unblock.timer.period_timer {                                     20
      start_relative_timer(e.target.period_timer, e.target.period);     21
      switch e.target in {                                              22
        case period_yield: {                                            23
          e.target => ready;                                            24
          if (!empty(running) && e.target > running) { running => ready; }  25
        }                                                               26
        case running, ready: { e.target => ready; }                     27
        case READY, BLOCKED, TERMINATED: { }                            28
      }                                                                 29
    }                                                                   30
  }                                                                     31
}                                                                       32
module Common(...) { ... }                                              33
```

Fig. 4. The modules used in the Bossa Nova implementation of the EDF policy

gle scheduling functionality, process attributes are declared locally to the module defining the functionality. To facilitate communication between modules, all process attributes are implicitly exported for read access. To enable reasoning about the behavior of a module across its various handlers, however, write access is only allowed in the defining module. Finally, to enhance reusability, a module imports an attribute without mentioning the name of the defining module, by simply annotating the attribute declaration with `requires`. The link between exported and imported attributes is made in the scheduler.

As shown in lines 2-3 of Figure 1, the Bossa implementation of the EDF policy declares five attributes, relating to the management of the process pe-

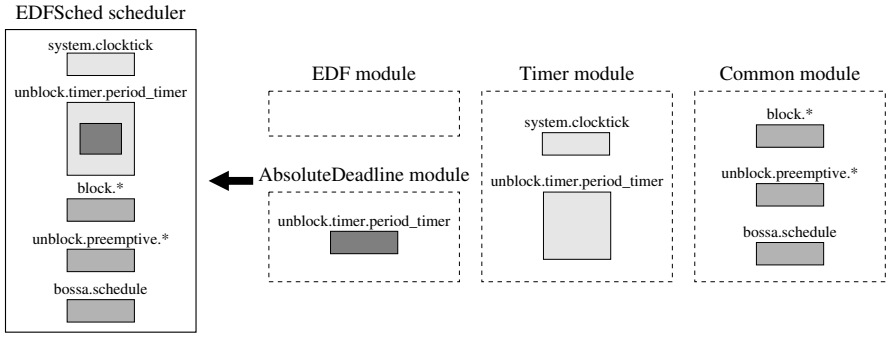

Fig. 5. Composition of handlers in the EDF scheduling policy

riod (`period`, `wcet`, and `period_timer`) and the process deadline (`deadline` and `absolute_deadline`). In the Bossa Nova implementation (Figure 4), the former are localized in the Timer module and the latter are localized in the AbsoluteDeadline module. The EDF module imports the `period`, `wcet`, and `absolute_deadline` attributes, which it declares using `requires`, as shown in lines 2-3 of Figure 4. These attributes are instantiated in the `process` declaration of the scheduler (Figure 3, lines 9-11), which declares that the EDF `period` and `wcet` attributes read the corresponding attributes of the Timer module and the EDF `absolute_deadline` attribute reads that of the AbsoluteDeadline module.

Event handlers. Event handlers react to OS events. Because multiple scheduling functionalities may need to react to the same event, multiple modules may define a handler for a given event. In this case, the scheduler lists the names of the modules defining a given handler in the order in which the definitions are to be composed. Execution begins with the handler defined by the first module in the list. A handler uses `next()` to invoke the next handler in the composition.

Figure 5 illustrates the composition of some of the handlers in the EDF policy. Both the Timer module and the AbsoluteDeadline module define an `unblock.timer.period_timer` handler. That of the Timer module (Figure 4, lines 20-30) represents a complete implementation of a minimal handling of the event: it restarts the timer and reschedules the target process. This handler is thus at the end of any composition sequence. The handler of the AbsoluteDeadline module (Figure 4, lines 9-12) extends this behavior by updating the locally defined `absolute_deadline` attribute and then invoking the next handler.

There are some constraints on the use of `next()` to ensure the integrity of the individual modules and of the composition. A handler typically updates process attributes and changes process states. When these operations are essential to the implemented scheduling functionality and are not idempotent, it is essential that the handler be invoked exactly once. To meet this requirement, a handler that appears before the end of a composition sequence must use `next()` exactly once along every control-flow path. In exceptional cases, a module may simply

provide a default definition for a handler, but not require that this definition be used. Such a handler can be declared to be `overrideable`. If all subsequent handlers in a composition are declared as `overrideable`, then `next()` may be omitted along some or all control-flow paths. Finally, handlers used at the end of a composition sequence may not invoke `next()`. This constraint ensures that the module writer intended the handler to be used in this position.

3.3 Aspects for process scheduling

Modules isolate the data and code associated with a given scheduling functionality. We have found, however, that the data associated with a scheduling functionality may need to be updated in response to actions, such as state changes, that take place in other modules. For example, the module ElapsedTime, used by several periodic policies (see Figure 2), maintains the running time of each process. Obtaining this value requires storing the current time whenever the process enters the `RUNNING` state and recording the difference between the current time and the stored time whenever the process leaves this state. Such state changes can occur in any module. As it is not desirable to let other modules make arbitrary side effects to the process attributes of ElapsedTime, the ElapsedTime module itself must be able to adapt other modules with the appropriate computations.

The need to update an attribute value when executing a particular kind of code elsewhere in the policy implementation amounts to a crosscutting concern. Accordingly, we look to Aspect-Oriented Programming [18] for inspiration, and add a form of aspects to Bossa Nova. We refer these aspects as *transition aspects*, because they account for some kind of transition in the system. A transition aspect implementing the behavior required by ElapsedTime on state transitions is as follows, where the state class names refer to the associated sets of states:

```
transition(process p) = {
  On READY => running { p.start_time = now(); }
  On running => READY, BLOCKED { p.elapsed_time += now()−p.start_time; }
}
```

Transition aspects are similar to aspects in languages such as AspectJ [17], but are restricted to the updating of attributes in response to current conditions. Accordingly, an aspect can only be attached to a state change or attribute reference, and cannot itself perform state changes. When multiple aspects apply to a single construct, they are ordered such that an aspect that defines an attribute appears before all aspects that reference the attribute; mutually recursive references are rejected by the Bossa Nova compiler. This ordering ensures that each attribute reference obtains an up-to-date value.

4 Evaluation

We now evaluate the forms of modularity provided by Bossa Nova. We first consider the benefits of modularity in programming scheduling policies and then

Family	Periodic				Round Robin	Proportion
Module	Common	Timer	AbsoluteDeadline	ElapsedTime	RR	Proportion
Lines of code	68	47	28	45	35	29

		Policy-specific module	Scheduler	Modular	Monolithic
Periodic	DM	23	22	160	109
(sharing illustrated in Figure 2)	EDF	26	34	203	123
	FUP	20	27	162	110
	LCT	9	26	150	106
	LLF	45	39	272	161
	SCT	42	35	237	147
	RM	9	26	150	106
	Family total			503	862
Round Robin	Basic round robin	15	28	146	96
(sharing Common and RR)	Best [3]	74	30	207	158
	Family total			182	254
Proportion	Basic proportion	48	32	177	124
(sharing Common and Proportion)	Move-to-rear [6]	41	29	167	123
	Family total			179	247

Fig. 6. A comparison of the lines of code used in the modular and monolithic implementations of various scheduling policies. "Family total" is explained in the text.

compare the dedicated approach used in Bossa Nova to the approaches to modularity found in general-purpose languages.

4.1 Benefits of modularity in implementing scheduling policies

We evaluate Bossa Nova with respect to a selection of policies from our on-going development of an encyclopedic, multi-OS library of scheduling policies. All of these policies are available at the Bossa web site.

Code sharing. We use the families of periodic, round-robin, and proportional scheduling policies to illustrate the effect of modularity on the amount of code that must be written to implement a new policy in a given family. Figure 6 shows the number of lines of code in the shared modules, and in a variety of scheduling policies in these families. All the policies use modules; the SCT, LLF, and Best policies also use transition aspects. In each case, the modular implementation is around 50% larger than the monolithic implementation. This increase is due to the introduction of the scheduler and the repetition of keywords (`process`, `handler`, *etc.*) between modules. In general, the extra code is eliminated by the Bossa Nova compiler, which generates a monolithic Bossa DSL implementation.

To amortize the cost of creating generic modules, they must be used by many different policies. The "Family total" entry associated with each family in Figure 6 shows the total number of lines in the policy-specific modules and schedulers added to the number of lines in one instance of each of the generic modules used by the family. This value excludes the Common module, which we may assume to be sufficiently widely used that its amortized cost is negligible. For the periodic family, the total number of lines that must be implemented in the modular case is 58% of the total number of lines required in the monolithic case.

For the round-robin and proportional families, the ratio is 72%, reflecting the fact that fewer policies have been implemented in these families.

Separation of concerns. Even when a scheduling policy is not part of a family, modularity can be useful to separate concerns. The Borrowed Virtual Time policy provides scheduling for both real-time and interactive processes [9]. This policy uses three main process attributes: the actual virtual time (AVT), the effective virtual time (EVT), and the warp. These attributes depend on several other process attributes, and the relevant calculations appear in multiple event handlers, making the monolithic implementation long (almost 300 lines) and difficult to understand. In the Bossa Nova implementation, each of the AVT, EVT, and warp is managed by a separate module. These modules highlight how the attributes are computed and the relationships between them. For the AVT, a twelve-line computation is required to compute the value whenever a process becomes newly able to run. This code is isolated in a transition aspect that delimits the computation and makes explicit the conditions under which it applies.

Isolation of OS-specific behavior. Often the details of the interaction with the target OS are orthogonal to the concerns of a given scheduling policy. In this case, we can use a module to isolate OS-specific behavior, thus simplifying the policy implementation and making it easy to use a scheduling policy with multiple OSes. In the examples in this paper, the Common module encapsulates the interaction with the OS (see Figure 2). We have implemented this module for use with Linux 2.4. In this implementation, the treatment of unblocking and yielding is specific to this OS, while other operations are generic.

4.2 Comparison to the approaches of general-purpose languages

We have designed module and aspect systems specific to the problem of implementing scheduling policies, rather than reusing existing approaches. To justify our choice, we compare our module and aspect systems to existing general-purpose approaches, in terms of the understandability, verifiability, and reusability of scheduling code.

Understandability. Key to the understandability of a Bossa Nova scheduling policy is the scheduler, which gives an overview of the set of modules used by the policy, the information that is defined by each module, and the information that is shared between modules. Module systems that can provide such a global view include Units [11] and a variant of CLOS mixins [5], in which a module either defines new behaviors or describes how to combine the information provided by other modules. Both of these approaches, however, allow a combining module to be itself combined with other modules, and thus there may be no single unit that gives a complete view of the program. Furthermore, in both approaches, a combining module cannot declare data, as we require for the declaration of process states. Other approaches, such those found in Object-Oriented languages or ML [23], allow modular units to be created and used throughout the program

and thus provide no global view. While one could organize programs written in such languages so that a single module creates the connections between all of the other modules, there would be no guarantee that this style is respected.

Verifiability. The verifiability of a Bossa Nova scheduling policy is enhanced by the constraints on access to process states, process attributes, and event handlers, that make it possible to reason about module behavior across a sequence of events. General-purpose module systems either forbid external access to module information, *e.g.* using Java's `private` modifier, or allow unlimited external access, *e.g.* using Java's `public` modifier. There is, however, no built-in way to provide read and write access in the defining module but only read access in other modules or to constrain the number of invocations of a given function.

Reusability. The reusability of the modules of a Bossa Nova scheduling policy is enhanced by the property that a module does not explicitly mention the names of other modules. In Object-Oriented languages, relationships between classes are expressed using inheritance, which requires naming the superclass. Traits [27], CLOS mixins, and Units allow defining modules that are externally combined, and thus do not mention the names of other modules. Traits, however, does not allow modules to define local state, and CLOS mixins and Units do not provide a unique global view.

Aspects. Aspect ordering is a major problem in understanding, verifying, and reusing aspects defined using traditional approaches. For example, AspectJ relies on a combination of defaults based on the order in which aspects are declared and explicit directives [17]. These approaches are fragile, burden the programmer, and do not take the semantics of the aspects into account. In our approach, aspect ordering is determined by def-use relationships reflecting the semantics of the aspects and the needs of the domain.

In summary, while some general-purpose approaches to modularity provide some of the features that we require for Bossa Nova, none provides either the domain-specific distinctions between different kinds of values or the fine-grained control over the use of program entities that we require.

5 Related Work

To illustrate the strategies taken to incorporate advanced language features in DSLs, we consider some other DSLs that provide module systems.

Some embedded DSLs inherit the module system of the host language. Leijen and Meijer embed a language for constructing database queries in Haskell [21]. They argue that it is possible to exploit the Haskell module system, but their use of this module system does not exhibit any domain-specific properties. Elliot takes a similar approach in a DSL for animation [10]. Other embedded DSLs provide domain-specific module systems. Verischemelog is a hardware description language embedded in Scheme [16]. As this language targets Verilog users,

it explicitly provides a module system based on that of Verilog, rather than using that of the host Scheme implementation. Thus, Verilog's use of modules is compatible with our approach: language abstractions are designed according to domain needs rather than relying on what is provided by a host language.

Module systems have also been developed from scratch for DSLs, as we have done for Bossa Nova. Risla is a compiled DSL for use in banking applications [32]. After several years of use, the abstraction facilities of the language were found to be insufficient and the language was extended with a module system. The task of implementing the extension was facilitated by the use of language specification tools. The nesC DSL for networked embedded systems was designed from the start around the use of components, implemented using a dedicated module system [12]. NesC applications have been found to require little new code, instead relying on a large number of small components, suggesting the appropriateness of the module system and the overall language design to the targeted domain.

6 Conclusion

In this paper we have presented the Bossa Nova language, which provides modules and transition aspects for implementing scheduling policies. These forms of modularity enable substantial code reuse when implementing multiple scheduling policies within a single family, allow separation of concerns in complex policies, and separate policy-specific code from OS-specific details. Furthermore, the use of forms of modularity dedicated to the scheduling domain improves the understandability and verifiability of scheduling code as compared to the use of approaches found in general-purpose languages. These observations suggest that rather than inheriting language features, as done in an embedded language, it is more fruitful to construct DSL extensions directly, based on an analysis of the needs of the domain.

We have used the language to extend our library of scheduling policies with policies from a range of policy families, including both classical policies and policies developed in recent research. In on-going work, we are adding to the set of policies and families represented. As a practical example, we are currently applying Bossa Nova and our library of scheduling policies to the use of a standard PC as a Personal Video Recorder (PVR).[4] A PVR must provide a variety of video services, such as encoding, decoding, and picture-in-picture. These services need to maintain a specific rate, but may have unpredictable computation requirements. Existing PVR software does not provide any quality of service guarantees and indeed less is known about how a scheduling policy can provide such guarantees for processes with unpredictable computation requirements than for processes with strict computation bounds. The ease of generating new policy variants and combinations in Bossa Nova can aid in evaluating existing policies and new variants in this setting.

[4] `http://www.emn.fr/x-info/bossa/bossabox`

References

1. D. L. Atkins, T. Ball, G. Bruns, and K. C. Cox. Mawl: A domain-specific language for form-based services. *IEEE Trans. Software Eng.*, 25(3):334–346, 1999.
2. 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, Phoenix, AZ, Dec. 1999.
3. S. A. Banachowski and S. A. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. In *Multimedia Computing and Networking*, volume 4673, San Jose, CA, Jan. 2002.
4. C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Trans. Internet Tech.*, 2(2):79–114, 2002.
5. G. Bracha and W. R. Cook. Mixin-based inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, Oct. 1990.
6. J. L. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia*, pages 63–73, Seattle, WA, Nov. 1997.
7. F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, West Sussex, England, 2002.
8. *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99)*, Austin, TX, Oct. 1999.
9. K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 261–276, Kiawah Island Resort, SC, Dec. 1999.
10. C. Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Trans. Software Eng.*, 25(3):291–308, 1999.
11. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, Canada, June 1998.
12. D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 1–11, San Diego, CA, June 2003.
13. C. Guss. Lecture notes: ECSE-421 embedded systems, 2004. `http://http://www.ece.mcgill.ca/~info421/`.
14. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), Dec. 1996.
15. K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, Dec. 1998.
16. J. Jennings and E. Beuscher. Verischemelog: Verilog embedded in Scheme. In DSL99 [8], pages 123–134.
17. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001.
18. 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*, LNCS 1241, pages 220–242, Jyväskylä, Finland, June 1997.

19. J. L. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Third International Conference on Generative Programming and Component Engineering*, LNCS 3286, pages 436–455, Vancouver, Canada, Oct. 2004.
20. J. L. Lawall, G. Muller, and H. Duchesne. Language design for implementing process scheduling hierarchies (invited paper). In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation - PEPM'04*, pages 80–91, Verona, Italy, Aug. 2004.
21. D. Leijen and E. Meijer. Domain specific embedded compilers. In DSL99 [8], pages 109–122.
22. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.
23. D. B. MacQueen. An implementation of Standard ML modules. In *LISP and Functional Programming*, pages 212–223, Snowbird, Utah, July 1988.
24. 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*, pages 184–197, Saint-Malo, France, Oct. 1997.
25. J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In RTAS'2001 [26], pages 141–148.
26. *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.
27. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, LNCS 2743, pages 248–274, Darmstadt, Germany, July 2003.
28. Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference*, pages 134–139, New Orleans, LA, June 1999.
29. O. Shivers. A universal scripting framework, or Lambda: the ultimate "little language". In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, LNCS 1179, pages 254–265, Singapore, Dec. 1996.
30. 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 Third USENIX Symposium on Operating Systems Design and Implementation*, pages 145–158, New Orleans, LA, Feb. 1999.
31. P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium*, LNCS 2257, pages 192–208, Portland, OR, Jan. 2002.
32. M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van der Meulen. Industrial applications of ASF+SDF. In *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96*, LNCS 1101, pages 9–18, Munich, Germany, July 1996.
33. D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Trans. Netw.*, 5(4):475–488, Aug. 1997.
34. W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 149–163, Bolton Landing, NY, Oct. 2003.
35. W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In RTAS'2001 [26], pages 149–156.