Formal Methods Meet Domain Specific Languages

Jean-Paul Bodeveix[1], Mamoun Filali[1], Julia Lawall[2], and Gilles Muller[3]

[1] IRIT Université Paul Sabatier,
118 route de Narbonne, F-31062 Toulouse Cedex, France
{bodeveix, filali}@irit.fr
[2] DIKU University of Copenhagen, 2100 Copenhagen, Denmark
julia@diku.dk
[3] Ecole des Mines de Nantes INRIA, LINA, 44307 Nantes cedex 3, France
Gilles.Muller@emn.fr

Abstract. In this paper, we relate an experiment whose aim is to study how to combine two existing approaches for ensuring software correctness: Domain Specific Languages (DSLs) and formal methods. As examples, we consider the Bossa DSL and the B formal method. Bossa is dedicated to the development of process schedulers and has been used in the context of Linux and Chorus. B is a refinement based formal method which has especially been used in the domain of railway systems. In this paper, we use B to express the correctness of a Bossa specification. Furthermore, we show how B can be used as an alternative to the existing Bossa tools for the production of certified schedulers.

Keywords: DSL, scheduling, formal methods, refinements, decision procedure.

1 Introduction

During the last decade, the correctness of software has been a major issue. Several approaches have been proposed and tools supporting them have been implemented, some of which have been used in industry. One approach is the use of Domain Specific Languages (DSLs). A DSL contains domain-specific abstractions as well as domain-specific restrictions that enable verification of domain-specific properties. Another approach is the use of formal methods. Such methods associate mathematically rigorous proofs with each step in software design and development. In this paper, we consider how to combine these approaches, by showing how a general purpose proof environment based on the B formal method [1] can be used to express and verify some of the properties relevant to the DSL Bossa [9].

The Bossa DSL is dedicated to the development of kernel-level process schedulers and has been used in the context of Linux and the Chorus real-time operating system. Process scheduling is at the heart of all operating system (OS)

J. Romijn, G. Smith, and J. van de Pol (Eds.): IFM 2005, LNCS 3771, pp. 187–206, 2005.

behavior, making verification critical in this domain. Bossa has thus been designed with both programmability and verification in mind. It has a formal semantics and provides high-level scheduling-specific abstractions that simplify the programming of scheduling policies and make explicit information that is useful in scheduler verification. These features have, for example, enabled undergraduate students with no previous kernel programming experience to implement scheduling policies in the Linux kernel without crashing the machine.

B is a refinement-based formal method that has been used for the development of safety critical software, especially in the domain of railway systems [2, 4]. The main feature of a B development process is that it proves that the final code implements its formal specification. In this paper, we use B to express the correctness of a Bossa specification. Furthermore, we show how B can be used as an alternative to the existing Bossa tools for the production of certified schedulers.

The rest of this paper is organized as follows. Section 2 provides a review of Bossa with respect to its language and the verifications performed by its compiler. Section 3 gives a brief overview of the B method. Section 4 elaborates the B development of a Bossa specification. Section 5 describes how some of the proof obligations generated by the B development can be discharged automatically. Section 6 presents some related work. Section 7 draws some conclusions.

2 Bossa

This section introduces the Bossa DSL and the verifications performed by the Bossa compiler.

2.1 Bossa in a Nutshell

We introduce the Bossa DSL using excerpts of an implementation of a Rate Monotonic (RM) scheduling policy [5], shown in Figure 1. This policy manages a set of periodic processes, each of which is associated with a period attribute. Process election chooses the process that is ready to run and that has the shortest period. RM scheduling is useful in the context of general-purpose operating systems such as Linux for controlling multimedia applications and in the context of real-time operating systems such as Chorus [7] for managing periodic processes. The complete RM policy is implemented as 110 lines of Bossa code and is available at the Bossa web site, `http://www.emn.fr/x-info/bossa`. A grammar of the Bossa DSL is also available at this web site. Here, 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 (line 2) lists the policy-specific attributes associated with each process. For the RM policy, each process is associated with its period.

The `states` declaration (lines 4-11) lists the set of process states that are distinguished by the policy. Each state is associated with a state class (`RUNNING`,

```
1   scheduler RM = {
      process = { time period; ... }

      states = {
5       RUNNING running : process;
        READY ready : select queue;
        READY yield : process;
        BLOCKED blocked : queue;
        BLOCKED computation_ended : queue;
10      TERMINATED terminated;
      }

      ordering_criteria = { lowest period }

15    handler(event e) {
        On process.end { e.target => terminated; }

        On unblock.preemptive {
          if (e.target in blocked) {
20          if ((!empty(running)) && (e.target > running)) {
              running => ready;
            }
            e.target => ready;
          }
25      }

        ...
      }
    }
```

Fig. 1. Excerpts of the Bossa Rate Monotonic policy

`READY`, `BLOCKED`, or `TERMINATED`) describing the schedulability of processes in the state and an implementation as either a process variable (`process`) or a queue (`queue`). The names of the states of the RM policy are mostly intuitive. The `ready` state is designated as `select`, indicating that processes are elected from this state. The `computation_ended` state stores processes that have completed their computation within the current period.

The `ordering_criteria` (line 14) describes how to compare two processes in terms of a sequence of criteria based on the values of their attributes. The RM policy favors the process with the lowest period.

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. We show only the definitions of the handlers `process.end` and `unblock.preemptive`, which are used as examples in the B development.

Event handlers are parameterized by an event structure, `e`, that includes the *target process*, `e.target`, affected by the event. The event-handler syntax is based on that of a subset of C and 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 `process.end` event occurs when a process ends its execution. The corresponding handler (line 16) simply sets the state of the process to `terminated`. Because the `terminated` state is not associated with any data structure, this state change has the effect of removing the process from further consideration by the scheduler. An `unblock.preemptive` event occurs when a process unblocks. The corresponding handler (lines 18-25) checks whether the process is actually blocked, and if so sets the state of the target process to `ready` making it eligible for election. The handler also checks whether there is a running process (`!empty(running)`) and if so whether the target process has a higher priority than this running process (`e.target > running`). If both tests are satisfied, the state of the running process is set to `ready`, thus causing the process to be preempted.

2.2 Bossa Verification

The Bossa compiler verifies that a Bossa scheduling policy satisfies both standard safety properties, such as the absence of null-pointer dereferences, and safety properties derived from the scheduling requirements of the target OS. The latter properties are OS-specific and are described by a collection of *event types*. Event types are defined in terms of the state classes and specify the possible preconditions and corresponding required postconditions on process states at the time of invoking the various event handlers.

We present the event type notation using the types of the `process.end` and `unblock.preemptive` events when used with Linux 2.4. That of `process.end` is as follows:

```
[tgt in BLOCKED] -> [tgt in TERMINATED]
```

This type rule indicates that the target process of the event is initially in a state of the `BLOCKED` state class and that the handler must change the state of this process to a state of the `TERMINATED` state class. Because no other state classes are mentioned in the type, a `process.end` handler cannot perform any other state changes. The type for `unblock.preemptive` is as follows:

```
[tgt in BLOCKED] -> [tgt in READY]
[p in RUNNING, tgt in BLOCKED] -> [[p,tgt] in READY]
[tgt in BLOCKED] -> [tgt in BLOCKED]
[tgt in RUNNING] -> []
[tgt in READY] -> []
```

The first three type rules treat the case where the target process is in a state of the `BLOCKED` state class. Of these, the first two allow the handler to move the target process to a state of the `READY` state class, making the process eligible for election. The second rule additionally moves the running process to the `READY` state class, which causes it to be preempted. In the third rule, the target process remains in the `BLOCKED` state class, but is allowed to change state, *e.g.* to one representing a different kind of blocking. The remaining rules consider the cases where the target process is not actually blocked. In these cases, the event handler may not perform any state changes.

It is straightforward to show that the `process.end` and `unblock.preemptive` handlers presented above satisfy these types. The Bossa compiler includes a verifier that checks that a scheduling policy satisfies the event types. This verifier is based on abstract interpretation and uses the various high-level abstractions found in the Bossa language to infer the source and destination of state change operations [9].

3 A Brief Overview of the B Method

B is a state-oriented formalism that covers the complete life cycle of software development. It provides a uniform language, the Abstract Machine Notation, to specify, design, and implement systems. A typical development in B consists of an abstract specification, followed by some refinement steps. The final refinement corresponds to an implementation. The correctness of the construction is enforced by the verification of proof obligations associated with each step of the development.

A specification in B is composed of a set of modules called *(abstract) machines*. Each machine has an internal state, and provides services allowing an external user to access or modify its state. Syntactically, a machine consists of several clauses which determine the static and dynamic properties of the state.

Consider the following abstract machine, which specifies a simple system that stores a set with at most one element and provides various set operations:

```
MACHINE Singleton(ELEM)
VARIABLES elem, elems
INVARIANT
   elem ∈ ELEM
∧  elems ⊆ {elem}
INITIALISATION
   elem :∈ ELEM || elems := ∅
OPERATIONS
suppress ≙
  PRE elems ≠ ∅ THEN /* the precondition ensures that suppress
                        will be called with a nonempty set */
    elems := ∅
  END;
el ⟵ extract ≙ /*  extract returns el  */
  PRE elems ≠ ∅ THEN
```

```
    el := elem || elems := ∅
  END;

add(el) ≙ /* the precondition specifies the type of el
                and ensures that no elements will be overridden */
  PRE el ∈ ELEM ∧ elems = ∅ THEN
    elem := el || elems := {el}  /* B multi assignment */
  END;
bb ⟵ empty ≙
  IF elems = ∅ THEN bb := TRUE ELSE bb := FALSE END;
bb ⟵ nonempty ≙
  IF elems ≠ ∅ THEN bb := TRUE ELSE bb := FALSE END;
bb ⟵ contains(el) ≙
  PRE el ∈ ELEM THEN
    IF el ∈ elems THEN bb := TRUE ELSE bb := FALSE END
  END
END
```

This machine specifies a family of systems all having the same abstract properties with respect to the parameter ELEM. By convention, a parameter starting with an uppercase letter is an abstract set. Otherwise, it must be given a type within the CONSTRAINTS clause. The clause VARIABLES defines the representation of the state of the machine. In this case, we only use the variables elem and elems. The clause INVARIANT constrains the domain of these variables. It states that elem is a member of ELEM and that elems is a subset of the singleton {elem}. Note that at this stage of the development the domain ELEM is abstract. We just assume that it is nonempty.[1] The initial state of the machine, which must satisfy the invariant, is specified in the INITIALISATION clause. In this example, the variable elem is initialized with any element of ELEM and elems is initialized to the empty set.

The services provided by a machine are specified in the clause OPERATIONS. In this case, we specify some standard set operations. To specify operations, B uses a mechanism of *generalized substitutions*. B defines six basic generalized substitutions: skip, multi-assignment (also called parallel-assignment), selection, bounded choice, unbounded choice, and preconditioned substitution. A generalized substitution acts as a predicate transformer. For example, the generalized substitution

PRE elems $\neq \emptyset$ **THEN** elems := $\emptyset$ **END**

corresponds to the predicate transformer

$$[\texttt{elems} \neq \emptyset \,|\texttt{elems} := \emptyset\,]$$

which is defined for any predicate P as follows:

$$[\texttt{elems} \neq \emptyset \,|\texttt{elems} := \emptyset\,]P \quad \Leftrightarrow \quad \texttt{elems} \neq \emptyset \wedge [\texttt{elems} := \emptyset]P$$

The soundness of a machine in B is given by *proof obligations* which verify that

[1] In B, abstract sets are nonempty and finite.

- The initial state satisfies the invariant.
- The invariant is preserved by the operations.
- The call of an operation must satisfy its precondition.

Some other clauses allow the introduction of constants (CONSTANTS) constrained by PROPERTIES.

An abstract specification can be materialized as an implementation by a mechanism of refinement. The abstract machine acts as the interface of the implementation: although the machine will be implemented by low level concrete variables, the user of a machine is always concerned by the variables and the operations defined at the abstract level. For example, in a real implementation of our system, we can implement the preceding singleton using a boolean variable, `full`, indicating whether the set is empty and a variable storing the singleton element managed by an instance of the `BASIC_ARRAY_VAR` machine. We refine the previous `Singleton` machine by the following `IMPLEMENTATION` machine:

```
IMPLEMENTATION Singleton_r(ELEM)
REFINES Singleton
CONCRETE_VARIABLES full
  IMPORTS BASIC_ARRAY_VAR(0..0,ELEM) /* generic array memory */
INVARIANT
       full ∈ BOOL
∧ (full = TRUE ⇒ (elems ≠ ∅ ∧ elem = arr_vrb(0)))
∧ (elems ≠ ∅ ⇒ full = TRUE)
INITIALISATION
  full := FALSE
OPERATIONS
add(el) ≙
  BEGIN
     STR_ARRAY(0,el); /* store el at index 0 */
     full := TRUE
  END;
suppress ≙ BEGIN full := FALSE END;
el ⟵ extract ≙ BEGIN el ⟵ VAL_ARRAY(0); full := FALSE END;
bb ⟵ empty ≙ IF full = TRUE THEN bb := FALSE ELSE bb := TRUE END;
bb ⟵ nonempty ≙ BEGIN bb := full END;
bb ⟵ contains(el) ≙
  VAR vv IN
     vv ⟵ VAL_ARRAY(0);
     IF full = TRUE ∧ el = vv THEN bb := TRUE ELSE bb := FALSE END
  END
END
```

The invariant of a refinement relates the abstract variables to the concrete ones and is called the "coupling invariant". From a user's point of view, operations provided by Singleton are also provided by Singleton_r; we cannot distinguish a call to a refined operation from a call to the abstract one.

The validity of a refinement is guaranteed by proof obligations: each concrete operation must be simulated by its abstract operation such that coupling invariant is preserved. Each abstract operation must be refined.

4 Expressing Bossa Specifications in B

This section describes how event types and scheduling policies specified in Bossa can be translated into B machines. The event types are translated into a B machine that models the abstract behavior of a scheduler. A Bossa specification is then translated into a refinement of this abstract scheduler. Thus, verifying the correctness of a Bossa specification amounts to verifying a refinement, which requires discharging a set of automatically generated proof obligations. We use the Rate Monotonic policy [5] presented in Section 2.1 to illustrate this approach.

In our approach, the information given by a Bossa scheduling policy is gradually taken into account at several levels of refinement. Figure 2 represents the architecture of the B project used in the conformance verification of the RM scheduling policy.

- The `scheduler` machine describes an abstract scheduler specified by Bossa event types.
- The `Classes` machine included by the `scheduler` machine defines classes of states and their transitions.
- The `rm` machine describes the rate monotonic policy as a refinement of the machine `scheduler`.
- The `RmTrans` machine and its refinements `RmTrans_r1` and `RmTrans_r2` describe transitions between rate monotonic policy states.
- The machines `Singleton`, `Queue`, `SelectQueue` and `VoidSet` describe the various collections of processes that can be used by a Bossa policy.

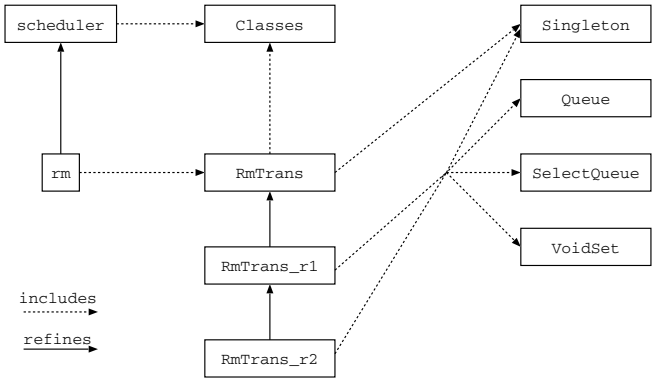

Fig. 2. Architecture of the B project

Remarks

- The preceding architecture does not depend on the scheduling policy. The machine `Classes` specifies state classes that are generic to the Bossa language. This machine can be used unchanged for all scheduling policies. The machines `Singleton`, `Queue`, `SelectQueue` and `VoidSet` specify various kinds of collections of processes that are similarly generic. The machine `scheduler` is specific to the event types for a given OS, but can be used with any policy designed for that OS. The remaining machines have generic roles, but policy-specific definitions. Of these, the machine `rm` specifies the considered policy (`r`ate `m`onotonic here), while the machine `RmTrans` and its refinements `RmTrans_r1` and `RmTrans_r2` specify the various states defined by the policy and the elementary transitions between them.
- The B source code of these machines could be generated automatically from the Bossa event types and from the Bossa specification of a policy.

4.1 Encoding the Event Types

The event types are defined in terms of a collection of abstract state classes. The B machine `Classes` associates each state class with the collection of processes that it contains. These collections are defined in terms of an abstract set of processes (Process), so that conformance proofs will not depend on the actual set of processes. Each state class is associated with a disjoint subset of Process. Because Bossa assumes that the target architecture has only one processor, the RUNNING state class can contain at most one process. This constraint is represented by creating a variable `running` to record the process in this state class and specifying that `Running` is either the empty set or a singleton set containing the value of this variable. The `Classes` machine also defines state transition operations. These operations either move a process from one class to another, e.g. the `CBlockedToTerminated` operation or allow an unbounded number of state changes between two given state classes, e.g. the `CReadyBlocked` operation.

MACHINE `Classes`
SETS
`Process`
VARIABLES
`Running, Ready, Blocked, Terminated, running`
INVARIANT
`Running` $\subseteq$ `Process`
& `Ready` $\subseteq$ `Process`
& `Blocked` $\subseteq$ `Process`
& `Terminated` $\subseteq$ `Process`
& `running` $\in$ `Process`
& `Running` $\cap$ `Ready` $= \emptyset$
& `Running` $\cap$ `Terminated` $= \emptyset$
& `Running` $\cap$ `Blocked` $= \emptyset$

```
&       Ready ∩ Terminated = ∅
&       Ready ∩ Blocked = ∅
&       Terminated ∩ Blocked = ∅
&       (Running ≠ ∅ ⇒ Running = {running})

INITIALISATION
      Running, Ready, Blocked, Terminated := ∅,∅,∅,∅
||    running :∈ Process /* running becomes an element of Process
OPERATIONS
  CBlockedToTerminated(tgt) ≙
   PRE tgt ∈ Blocked THEN
     Blocked :=Blocked - {tgt} || Terminated := Terminated ∪ {tgt}
   END;
  CReadyBlocked ≙
   ANY rr WHERE rr ⊆ Ready THEN
      Ready := Ready - rr || Blocked := Blocked ∪ rr
   END
  ...
END
```

The event types describe the state changes allowed between the state classes. They are expressed by the `scheduler` abstract machine, which includes the `Classes` machine defined above and an operation for each event. The system to be built is supposed open and preconditions of the events specify call conditions.

We illustrate the translation of a set of event types to a B machine using the rules for `process.end` and `unblock.preemptive` presented in Section 2.1. The event type for `process.end` is below. This rule indicates that when the event occurs, the targeted process (tgt) is blocked and the event handler must cause the process to become terminated.

```
MACHINE scheduler
INCLUDES Classes
OPERATIONS
...
/*
[tgt in BLOCKED] -> [tgt in TERMINATED]
*/
Process_end(tgt) ≙
  PRE tgt : Process & tgt ∈ Blocked THEN
    CBlockedToTerminated(tgt)
  END
...
END
```

Event types can also be non-deterministic. For example, the type for `unblock.preemptive`, reproduced below, allows three different behaviors if the target process is blocked, and specifies additional behaviors if the target process is

running or ready. In the B translation, `SELECT` is used to identify the current state classes of relevant processes and `CHOICE` expresses the non-determinism.

```
/*
[tgt in BLOCKED] -> [tgt in READY]
[p in RUNNING, tgt in BLOCKED] -> [[p,tgt] in READY]
[tgt in BLOCKED] -> [tgt in BLOCKED]
[tgt in RUNNING] -> []
[tgt in READY] -> []
*/
Unblock_preemptive(tgt) ≙
 PRE tgt : Process ∧ tgt ∈ (Running ∪ Ready ∪ Blocked) THEN
   SELECT tgt ∈ Blocked ∧ Running ≠ ∅ THEN
     CHOICE CRunningBlockedToReadyReady(tgt)
       OR CBlockedToReady(tgt)
     END
   WHEN tgt ∈ Blocked ∧ Running = ∅ THEN CBlockedToReady(tgt)
   WHEN PTRUE THEN skip
   END
 END
```

Remark. In the `scheduler` machine, we have only specified the transitions that can be performed between state classes. The Bossa event types also specify when transitions are allowed within a state class as is represented by the rule `[tgt in BLOCKED] -> [tgt in BLOCKED]` of the `unblock.preemptive` event type. While this transition could be expressed in B by refining the specification of the state classes, we have not done so to maintain readability. It follows that in our B model, state changes within a class are always allowed.

4.2 Encoding a Scheduling Policy

A scheduling policy is introduced as a refinement of the abstract scheduler. It redefines the scheduling events using its own states, which refine the previously introduced state classes. The management of policy-specific states is introduced gradually in order to factorize some of the proof obligations.

- The first refinement level introduces the representation of states in terms of collections of processes. In order to establish the link between policy states and state classes, the machine `Classes` is included. Elementary state transitions are defined and apply both to policy states and state classes.
- The next refinement level drops the state classes, which are not used in the implementation. However, this machine inherits the link between states and state classes established by the first level.
- The last refinement level introduces the implementation of state membership.

Data Representation. The data structures used at the abstract level ensure the correctness of state changes while preserving some properties, e.g. a process cannot be lost, cardinality constraints are enforced. The preservation of these properties is established by verifying proof obligations. At the abstract level, states are represented by sets and checking the state of a process amounts to testing set membership. In order to simplify the proof of the conformity of scheduling policies, abstract machines defining sets of processes are provided. Generally, they provide insertion and extraction operations. We have developed a library of such machines with their efficient implementations.

- The `Singleton` machine (see Section 3) is used when there is at most one process in a given state. Its insertion operation is preconditioned so that processes cannot be overridden and its invariant ensures the cardinality constraint. This machine supports Bossa states declared as `process` (`running` and `yield` for RM).
- The `Queue` machine can contain any number of processes. This machine supports Bossa states declared as `queue` (`blocked` and `computation_ended` for RM).
- The `SelectQueue` machine is used when a state can contain any number of processes and processes in this state can be accessed in sorted order using the Bossa operator `select`. This machine support Bossa states declared as `select queue` (`ready` for RM).
- The `VoidSet` is used when a state does not record any processes. This machine supports Bossa states for which no implementation is specified (`terminated` for RM). The machine does not provide observation operations so that its implementation does not store any process.

State Transitions. The machine `RmTrans` establishes the link between policy states and states classes. Once established, this invariant is reused by machines including or refining `RmTrans`. To establish the link, `RmTrans` includes both the `Classes` machine and machines for each kind of state.[2] The invariant of `RmTrans` specifies how states classes are split into disjoint concrete states. In order to preserve this invariant, operations are defined as acting both on concrete states and on state classes. For example, `RMRunning2Yield` applies if running is non empty and yield is empty. The running process is deleted from the running state and added to the yield state. This operation is in parallel performed on state classes: `CRunning2Ready` is also called, as `Ready` is the state class of `yield`.

```
MACHINE RmTrans
INCLUDES
  Classes,
  ru.Singleton(Process),      /* running state */
  rd.SelectQueue(Process,period),   /* ready state */
```

[2] The notation INCLUDES `pr.m` includes the machine `m` and adds the prefix `pr` to the identifiers of `m` in order to avoid any conflict.

```
  yl.Singleton(Process),     /* yield state */
  bl.Queue(Process),         /* blocked state */
  ce.Queue(Process),         /* computation_ended state */
  tm.VoidSet(Process)        /* terminated state */
INVARIANT
 bl.elems ∩ ce.elems = ∅
∧ rd.elems ∩ yl.elems = ∅
∧ Running = ru.elems
∧ Ready = rd.elems ∪ yl.elems
∧ Blocked = bl.elems ∪ ce.elems
∧ Terminated = tm.elems
OPERATIONS
 RMRunning2Yield ≙
   PRE ru.elems ≠ ∅ ∧ yl.elems = ∅ THEN
     yl.add(ru.elem) || ru.suppress || CRunning2Ready
   END;
  ...
END
```

Elimination of Abstract Data. The refinement step `RmTrans_r1` is used to redefine operations without managing state classes. The `Classes` machine is no longer included and operations only act on policy states.

```
REFINEMENT RmTrans_r1 REFINES RmTrans
INCLUDES
  ru.Singleton(Process), rd.SelectQueue(Process,period),
  yl.Singleton(Process), bl.Queue(Process),
  ce.Queue(Process), tm.VoidSet(Process)
OPERATIONS
 RMRunning2Yield ≙
   BEGIN yl.add(ru.elem) || ru.suppress END;
  ...
END
```

State Membership. The data-representation machines provide an abstract variable (`elems`) containing the set of processes in the corresponding state. In Bossa, the implementation of state membership relies on an attribute attached to each process. It is represented in B by the variable `state: Process → State` which is introduced in a new refinement. Its declaration is split into INVARIANT and ASSERTIONS. The ASSERTIONS clause is proved once as being implied by the invariant. Then, the preservation of the assertion predicates is ensured provided the invariant is preserved.

```
REFINEMENT RmTrans_r2 REFINES RmTrans_r1
SETS
```

```
  State = {RmNowhere, RmRunning, RmReady, RmBlocked, RmCompEnded,
           RmYield, RmTerminated}
INCLUDES
  bl.Queue,
  ...
VARIABLES
  state
INVARIANT
  /* state is a relation between Process and State */
  state : Process ↔ State
∧ state = Process × {RmNowhere} <+ /* definition of the state */
    (rd.elems × {RmReady} ∪         /* relation */
     bl.elems × {RmBlocked} ∪
     ce.elems × {RmCompEnded} ∪
     yl.elems × {RmYield} ∪
     ru.elems × {RmRunning} ∪
     tm.elems × {RmTerminated})
ASSERTIONS
  state :  Process → State /* the relation is functional */
```

The introduction of the `state` variable avoids referencing abstract sets of states for testing state membership. The refinement `RmTrans_r2` thus uses the operations of the data-collection abstract machines instead of their abstract variables.

The Algorithm. The scheduling policy is defined as a refinement of the abstract scheduler. Its B code is translated from the Bossa specification. As compared to the abstract scheduler, some tests are added in order to get the current concrete state of a process and to call the correct state transition operation.

For example, the handler for the `unblock.preemptive` event is specified in Bossa as follows:

```
On unblock.preemptive {
  if (e.target in blocked) {
    if ((!empty(running)) && (e.target > running)) {
      running => ready;
    }
    e.target => ready;
  }
}
```

The translation of this handler to B is immediate. Note that the process comparison `p1 > p2` is translated into `period(p1) < period(p2)`, thus inlining the ordering criteria defined in Section 2.1. Furthermore, policy specific variables are introduced. The Rate Monotonic policy described in Bossa defines a counter (`missed_deadlines`) and a timer variable.

```
REFINEMENT rm REFINES scheduler
INCLUDES RmTrans  /* state transition machine */
VARIABLES
   missed_deadlines, timer  /* policy specific variables */
INVARIANT
   missed_deadlines : Process --> NATURAL
&  timer : Process --> INTEGER
INITIALISATION
   missed_deadlines := Process * {0} || timer := Process * {0}
OPERATIONS
  Unblock_preemptive(tgt) ≙
    VAR isbk IN
      isbk <-- RMisBlocked(tgt);
      IF isbk = TRUE THEN
        VAR hru IN
          hru <-- RMhasRunning;
          IF hru = TRUE ∧ period(tgt) < period(running) THEN
            RMRunning2Ready
          END;
          RMBlocked2Ready(tgt)
        END
      END
    END
  END
```

Proof obligations generated for this machine express that it is a refinement of the abstract scheduler. They are the main properties that must be checked in order to ensure that the scheduling policy complies with the event types associated with the underlying kernel.

5 Proof Automation

The proof obligations generated for the preceding Bossa/B development are not automatically proved by the provers available with Atelier B. Although, it should be possible to add some tactics for discharging some of the remaining proofs, this section instead introduces a decidable logic fragment that supports the expression of proof obligations. It follows that their verification is automatic. We believe that identifying logic fragments for automating the proof process is essential for the scalability of a proof based approach.

5.1 An Overview of Monadic Second Order Logic and Mona

Definition 1 (S1S and WS1S logics). *Let $\{x_1, \ldots, x_n\}$ be a set of first order variables and $\{X_1, \ldots, X_n\}$ a set of monadic second order variables. A minimal grammar for these logics is defined as follows:*

– *A term t is inductively defined by:*

$$t ::= 0 \mid x_i \mid s(t) \textit{s is the successor symbol}$$

– *A formula f is inductively defined by:*

$$\begin{aligned} f ::= & t \in X_i \quad \textit{set membership} \\ & \mid \neg f \mid f \wedge f \\ & \mid \exists_1 x_i.\ f \quad \textit{first order quantification} \\ & \mid \exists_2 X_i.\ f \quad \textit{second order (set) quantification} \end{aligned}$$

This syntax is extended as usual by first order operators and quantifiers ($\vee, \Rightarrow, \forall \ldots$) and some arithmetic relations. For example, $a \leq b$ is defined by $\forall_2 X.\ X(a) \wedge (\forall_1 x.\ X(x) \Rightarrow X(s(x))) \Rightarrow X(b)$.

Validity of a formula. A closed formula is valid in S1S or WS1S if it is valid in the interpretation on the set $\mathbb{N}$ of natural numbers, where s is the successor function, first order variables relate to the natural numbers and second order variables to the subsets (finite in the case of WS1S) of $\mathbb{N}$. These two logics are decidable [11]. WS1S is concerned with finite sets only while S1S is also concerned with infinite sets. The Mona tool [6] implements a decision procedure for WS1S.

With respect to our study, we can use Mona to decide some of our proof obligations, specifically those that concern individual processes and sets of processes (regardless of the effective number of processes).

5.2 A Translation Example

As an example of the use of Mona in discharging proof obligations, we consider the proof obligations associated with the refinement of the `ReadyToRunning` operation. According to the B method, the proof obligation for a refinement of an abstract preconditioned operation op_a by a concrete operation op_r is:

$$\begin{aligned} & \mathrm{Inv}_a(st_a) \\ \wedge\ & \mathrm{Inv}_r(st_a, st_r) \wedge \mathrm{Pre\_op}(st_a) \wedge op_r(st_r, st'_r) \\ \Rightarrow\ & \exists st'_a :\quad op_a(st_a, st'_a) \\ & \qquad\qquad \wedge\ \mathrm{Inv}_r(st'_a, st'_r) \end{aligned}$$

where:

– $\mathrm{Inv}_a(st_a)$(resp. Inv_r) is the invariant of the abstract (resp. of the concrete) machine,[3]
– Pre_op is the precondition of the abstract operation,
– $op_a(st_a, st'_a)$(resp. $op_r(st_r, st'_r)$) is the before-after predicate of the abstract (resp. concrete) operation.

[3] The invariant of the refinement is expressed over the product of the abstract and concrete states in order to express the so called "coupling" invariant between the abstraction and the concretization.

Informally, the preceding formula says that each concrete step op_r can be simulated by an abstract step op_a so that the coupling invariant Inv_r is preserved.

Given the abstract and refined states introduced in Section 4.2, we can express the preceding proof obligation within Mona as follows (the complete Mona text is given in the appendix A):

```
all2 Ready, Running, Terminated, Blocked:
all1 running,running':
all2 rdelems, ruelems, ylelems,
      rdelems', ruelems', ylelems':
all1 rdelem, ruelem, ylelem,
      rdelem', ruelem', ylelem':
        Inv_a(Ready, Running, Terminated, Blocked, running)
     &  Inv_r(Ready, Running, Terminated, Blocked, running,
              rdelems,rdelem,ruelems,ruelem,ylelems,ylelem)
     &  pre_RunningToReady_a(Ready, Running, Terminated, Blocked, running)
     &  RunningToReady_r(rdelems,rdelem,ruelems,ruelem,ylelems,ylelem,
           rdelems',rdelem',ruelems',ruelem',ylelems',ylelem')
   => ex2  Ready',Running',Terminated',Blocked': ex1 running':
       RunningToReady_a(Ready, Running, Terminated, Blocked, running,
                        Ready',Running',Terminated',Blocked',running')
     & Inv_r(Ready', Running', Terminated', Blocked',running',
             rdelems',rdelem',ruelems',ruelem',ylelems',ylelem');
```

Thanks to its decision procedure, the Mona tool establishes that the preceding predicate is valid. It outputs the following result:

```
AUTOMATON CONSTRUCTION
100% completed
Time: 00:00:00.01

Automaton has 1 state and 1 BDD-node

ANALYSIS
Formula is valid
```

6 Related Work

A large number of DSLs have been developed for a wide range of domains [12]. Many of these DSLs provide no verification, and those that do typically either rely on verification provided by a general-purpose host language [10] or use ad hoc analyzers, as was originally done for Bossa. The former approach is, however, limited to the facilities of the host language, which are rarely adequate for expressing and checking domain-specific properties, while the latter puts a huge burden on the DSL developer.

The DSLs Promela++ [3] and ESP [8] both provide both standard code generators and translators to code suitable for use with the SPIN model checker.

While these approaches are in the spirit of the work presented here, the state explosion problem implies that these languages use model checking for bug finding, but not complete verification. Furthermore, these approaches require specifying properties in the general-purpose specification language of SPIN, while the Bossa event types are domain-specific. Indeed, the high-level of the event type specification is crucial to enable our refinement-based approach.

7 Conclusion

DSLs provide a high-level means of implementing solutions to complex problems within a given domain. When the domain has critical safety or security requirements, verification of these implementations is essential. In this paper, we have shown a systematic means of using the B formal method to verify a process scheduling policy implemented using the Bossa DSL. This verification covers within a single framework both verification of the scheduler structure, as also provided by existing Bossa verification tools, and verification of part of the implementation strategy (i.e., the use of the `state` function to optimize state membership tests), which is not covered by the Bossa verifier. In the development presented here, most of the work can be reused directly for verification of other scheduling policies, except for the proofs related to the event handler definitions themselves (i.e., the second part of Section 4.2). However, using a dedicated decision procedure such as Mona should help in automating the verification of most of the proof obligations. In future work, we plan to generalize this part of the development as well, to produce an executable code and hence a certified Bossa compiler. We will also consider how this approach can be applied to other DSLs.

Acknowledgement

This work was supported in part by the CORSS: "Composition et raffinement de systèmes sûrs" project of program "ACI: Sécurité Informatique" supported by the French Ministry of Research and New Technologies.

References

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.
2. F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 2215 of *Lecture Notes in Computer Science*, pages 298–315. Springer-Verlag, Guildford, UK, april 2005.
3. 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, Jan. 1997.

4. P. Behm, P. Desforges, and J.-M. Meynadier. Météor : An industrial success in formal development. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier*, volume 1393 of *Lecture Notes in Computer Science*, page 26. Springer-Verlag, 1998.
5. F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, West Sussex, England, 2002.
6. J. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, http://www.brics.dk/∼mona, pages 58–73, Aarhus, May 1995.
7. Jaluna. Jaluna Osware. http://www.jaluna.com.
8. S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: a language for programmable devices. In *Proceedings of the ACM SIGPLAN'01 conference on Programming Language Design and Implementation*, pages 309–320, Snowbird, UT, June 2001.
9. J. 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 (GPCE'04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 436–455, Vancouver, October 2004. Springer-Verlag.
10. D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99)*, pages 109–122, Austin, TX, Oct. 1999.
11. W. Thomas. Automata on infinite objects. In J. Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–192. MIT Press, 1990.
12. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

A Mona Expression of a Proof Obligation

```
pred ajouter(var2 elems, var1 el, var2 elems') =
  elems' = elems union {el}
;
pred atmostSingleton(var2 S, var1 e) =
/* S1S expression that a set containts at most 1 element */
  S sub {e}
;
pred Inv_a(var2 Ready, Running, Terminated, Blocked, var1 running) =
  Running inter Ready = {} & Running inter Terminated = {}
& Running inter Blocked = {} & Terminated inter Blocked = {}
& Ready inter Terminated = {} & Ready inter Blocked = {}
& atmostSingleton(Running,running)
;
pred pre_RunningToReady_a(var2 Ready, Running, Terminated, Blocked,
                          var1 running) =
Running ~= {}
;
/* Running2Ready =
   PRE HasRunning THEN Ready := Ready \/ Running || Running := {} END;
*/
```

```
pred RunningToReady_a(
  var2 Ready, Running, Terminated, Blocked, var1 running,
  var2 Ready', Running', Terminated', Blocked', var1 running') =
  Ready' = Ready union Running & Running' = {}
& Terminated' = Terminated & Blocked' = Blocked & running' = running
;
/* Running2Ready = BEGIN rd.ajouter(ru.elem) || ru.supprimer END; */

pred RunningToReady_r(
  var2 rdelems, var1 rdelem, var2 ruelems,
    var1 ruelem, var2 ylelems, var1 ylelem,
  var2 rdelems', var1 rdelem', var2 ruelems',
    var1 ruelem', var2 ylelems', var1 ylelem') =
  ajouter(rdelems,ruelem,rdelems')
& ruelems' = {} & ylelems' = ylelems & ylelem' = ylelem
;
pred Inv_r(
  var2 Ready, Running, Terminated, Blocked, var1 running,
  var2 rdelems, var1 rdelem, var2 ruelems,
  var1 ruelem, var2 ylelems, var1 ylelem) =
  rdelems = Ready \ ylelems
& Running = ruelems & ylelems sub Ready
& atmostSingleton(ylelems,ylelem) & atmostSingleton(ruelems,ruelem)
;
/* refinement proof obligation */

all2  Ready, Running, Terminated, Blocked: all1 running,running':
all2 rdelems, ruelems, ylelems, rdelems', ruelems', ylelems':
all1 rdelem, ruelem, ylelem, rdelem', ruelem', ylelem':
    Inv_a(Ready, Running, Terminated, Blocked, running)
   & Inv_r(Ready, Running, Terminated, Blocked,running,
           rdelems,rdelem,ruelems,ruelem,ylelems,ylelem)
   & pre_RunningToReady_a(Ready, Running, Terminated, Blocked, running)
   & RunningToReady_r(rdelems,rdelem,ruelems,ruelem,ylelems,ylelem,
       rdelems',rdelem',ruelems',ruelem',ylelems',ylelem')
 => ex2  Ready',Running',Terminated',Blocked': ex1 running':
   RunningToReady_a(Ready, Running, Terminated, Blocked, running,
                    Ready',Running',Terminated',Blocked',running')
   & Inv_r(Ready', Running', Terminated', Blocked',running',
           rdelems',rdelem',ruelems',ruelem',ylelems',ylelem');
```