Applying the B Formal Method to the Bossa Domain-Specific Language

Jean-Paul Bodeveix
Mamoun Filali
IRIT Université Paul Sabatier
{bodeveix,filali}@irit.fr

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

Gilles Muller
OBASCO Group, École des Mines de Nantes-INRIA, LINA
gilles.muller@emn.fr

1. Introduction

Domain-specific languages (DSLs) are used in both industry and research, in complex areas as varied as data mining [5], graphics [7], and device driver development [10]. A DSL provides high-level domain-specific abstractions that encapsulate domain expertise, thus making programming easier and less error-prone. Such languages also promise to be more amenable to verification, as the set of language abstractions can be designed to be easy to relate to desired properties and can be constrained to avoid problematic constructs such as unbounded loops. Nevertheless, many DSLs provide no verification, and those that do typically either rely on verification provided by a general-purpose host language or use ad hoc analyzers. 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. We observe that many powerful verification systems have been developed in the formal methods community. Our goal is to realize the potential for verification of DSLs by harnessing these resources in designing and implementing DSL verifiers.

To begin to bridge the gap between existing approaches to program verification and DSLs, we are applying the B formal method [1] to the Bossa DSL [9]. 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, 3]. The main feature of a B development process is that it proves that the final code implements its formal specification. Bossa is a DSL for specifying operating system (OS) kernel process scheduling policies. It has been implemented in the OSes Linux and Chorus. It has a formal semantics, and an ad hoc verifier that checks that a Bossa scheduling policy conforms to a model of the scheduling requirements of the target OS. This verifier is, however, hand-coded and complex, and thus hard to maintain as the language or set of desired properties evolves. Furthermore, it only checks the interaction between the scheduling policy and the OS, and is thus not suitable for checking properties of the implemented algorithm, such as liveness and fairness. In this paper, we show how B can be used to replace the ad hoc verifier of Bossa. We are currently working on extending this formal development to account for a wider range of properties.

The rest of this paper is organized as follows. Section 2 provides an overview of Bossa. Section 3 gives a brief overview of the B method. Section 4 elaborates the B development of a Bossa specification and describes how some of the proof obligations generated by the B development can be discharged automatically. Section 5 draws some conclusions.

```
1  scheduler RM = {
    process = { time period; ... }
    states = {
     RUNNING running : process;
5    READY ready : select queue;
     READY yield : process;
     BLOCKED blocked : queue;
     BLOCKED computation_ended : queue;
     TERMINATED terminated;
10   }
    ordering_criteria = { lowest period }

    handler(event e) {
     On block.* { e.target => blocked; }
15
     On unblock.preemptive {
      if (e.target in blocked) {
       if ((!empty(running)) && (e.target>running)) {
        running => ready;
       }
20     e.target => ready;
      }
     }
    ...
25  }
   }
```

Figure 1. Excerpts of the Bossa Rate Monotonic policy

2. The Bossa DSL

We introduce the Bossa DSL using excerpts of an implementation of a Rate Monotonic (RM) scheduling policy [6], shown in Figure 1. This policy manages a set of periodic processes. Process election chooses the runnable process that has the shortest period. 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 (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`, `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 OS kernel. Examples of such events include process blocking and unblocking and the need to elect a new process. Figure 1 shows only the handlers `block.*` and `unblock.preemptive`.

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 `block.*` event occurs when a process blocks. The corresponding handler (line 16) simply sets the state of the process to `blocked`. 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 running process to be preempted.

Verification Verification for Bossa focuses on checking that a Bossa scheduling policy satisfies the scheduling requirements of the target OS. These requirements are OS-specific and are described by a collection of *event types* [9]. Event types are described 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 type of the `unblock.preemptive` event when used with Linux 2.4. The type 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 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 verify that the `unblock.preemptive` handler presented above satisfies this type. 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. 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 *(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 excerpt of an 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
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;
  ...
END
```

This machine specifies a family of systems all having the same abstract properties with respect to the parameter `ELEM`. The clause VARIABLES defines the representation of the state of the machine. The clause INVARIANT constrains the domain of these variables. The initial state of the machine is specified in the INITIALISATION clause. The services provided by a machine are specified in the clause OPERATIONS.

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

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

An abstract specification can be materialized as an implementation by a mechanism of refinement. The abstract machine acts as the interface of the implementation with respect to client machines. Regardless of the refinement levels, the user of a machine is always concerned by the variables and the operations defined at the abstract machine. The invariant of a refinement relates the abstract variables to the concrete ones and is called the "coupling invariant".

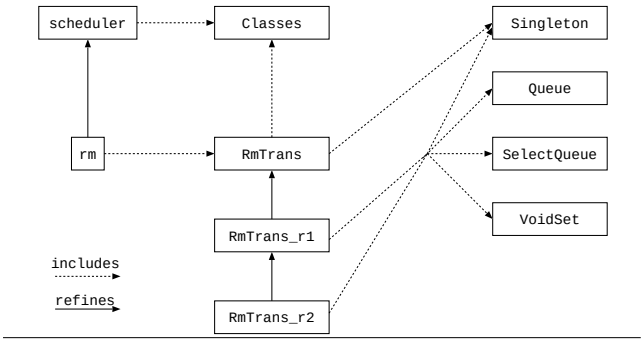

Figure 2. Architecture of the B project

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.

4. Expressing Bossa specifications in B

We now consider how Bossa event types and scheduling policies 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 scheduling policy is then translated into several levels of refinement of this abstract scheduler. Verifying the correctness of a Bossa scheduling policy amounts to verifying these refinements, which requires discharging a set of automatically generated proof obligations. We use the RM policy presented in Section 2 as an example.

Figure 2 represents the architecture of the B project used in the conformance verification of the RM scheduling policy.

- The `Classes` machine included by the `scheduler` machine defines state classes and the transitions between them.
- The `scheduler` machine describes an abstract scheduler specified by Bossa event types.
- The `rm` machine describes the RM 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.

4.1 Encoding the event types

The event types are defined in terms of the Bossa 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. The `Classes` machine also defines state transition operations. These operations either move a process from one class to another or allow an unbounded number of state changes between two given state classes.

MACHINE `Classes`
SETS `Process`
VARIABLES
`Running, Ready, Blocked, Terminated, running`
INVARIANT
`Running` $\subseteq$ `Process` $\wedge$ `Ready` $\subseteq$ `Process`
$\wedge$ `Blocked` $\subseteq$ `Process` $\wedge$ `Terminated` $\subseteq$ `Process`
$\wedge$ `running` $\in$ `Process`
$\wedge$ `Running` $\cap$ `Ready` $= \emptyset$ $\wedge$ `Running` $\cap$ `Terminated` $= \emptyset$
$\wedge$ `Running` $\cap$ `Blocked` $= \emptyset$
$\wedge$ `Ready` $\cap$ `Terminated` $= \emptyset$ $\wedge$ `Ready` $\cap$ `Blocked` $= \emptyset$
$\wedge$ `Terminated` $\cap$ `Blocked` $= \emptyset$
$\wedge$ (`Running` $\neq \emptyset \Rightarrow$ `Running` $=$ {`running`})
INITIALISATION
`Running, Ready, Blocked, Terminated := ∅,∅,∅,∅`
`||` `running` $:\in$ `Process` *// running belongs to Process*
OPERATIONS
`CBlockedToTerminated(tgt)` $\triangleq$
PRE `tgt` $\in$ `Blocked` **THEN**
`Blocked := Blocked - {tgt}`
`||` `Terminated := Terminated` $\cup$ {`tgt`}
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 rule for `unblock.preemptive` presented in Section 2. This rule 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.

`Unblock_preemptive(tgt)` $\triangleq$
PRE `tgt` $\in$ (`Running` $\cup$ `Ready` $\cup$ `Blocked`) **THEN**
SELECT `tgt` $\in$ `Blocked` $\wedge$ `Running` $\neq \emptyset$ **THEN**
CHOICE `CRunningBlockedToReadyReady(tgt)`
OR `CBlockedToReady(tgt)`
END
WHEN `tgt` $\in$ `Blocked` $\wedge$ `Running` $= \emptyset$ **THEN**
`CBlockedToReady(tgt)`
WHEN `PTRUE` **THEN** `skip`
END
END

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.

More details about the B model can be found in an expanded version of this paper [4].

4.3 Proof automation

The proof obligations generated for the preceding Bossa/B development are not automatically proved by the provers available with Atelier B. In the context of our study, some proof formulae can be instantiated in WS1S (weak second order logic of one successor). These can be decided automatically [11]. Currently, we use the Mona tool [8] to decide these proof obligations. More precisely:

- The event type state space consists of one set of processes per Bossa class from which given processes can be removed or inserted. These transitions can be expressed in WS1S.
- The scheduling policy state space is obtained by splitting the abstract state classes into disjoint subsets. State transitions are derived from the Bossa scheduling policy code. Here again, the refinement relation relating the abstract and concrete space can be expressed in WS1S.

5. 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, 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., 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 a certified Bossa compiler. We will also consider how this approach can be applied to other DSLs.

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, Apr. 2005.

[3] P. Behm, P. Desforges, and J.-M. Meynadier. Météor : An industrial success in formal development. In D. Bert, editor, *B'98: second Int. B Conference, Montpellier*, volume 1393 of *Lecture Notes in Computer Science*, page 26. Springer-Verlag, 1998.

[4] J.-P. Bodeveix, M. Filali, J. Lawall, and G. Muller. Formal methods meet domain specific languages. In *Fifth International Conference on Integrated Formal Methods (IFM), Eindhoven Netherlands*, Lecture Notes in Computer Science. Springer-Verlag, Nov. 2005. (to appear).

[5] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, pages 9–17, 2000.

[6] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems.* Wiley, West Sussex, England, 2002.

[7] C. Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, 1999.

[8] 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.

[9] J. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Third Int. 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] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, San Diego, CA, Oct. 2000.

[11] W. Thomas. Automata on infinite objects. In J. Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–192. MIT Press, 1990.

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, and in part by the Danish Research Council, grant number 21-05-0545.