Automatic Verification of Bossa Scheduler Properties

Jean-Paul Bodeveix[a], Mamoun Filali[a,1], Julia L. Lawall[b] and Gilles Muller[c]

[a] *IRIT, Université Paul Sabatier, Toulouse, France*

[b] *DIKU, University of Copenhagen, Copenhagen, Denmark*

[c] *OBASCO Group, Ecole des Mines de Nantes-INRIA, LINA, Nantes, France*

Abstract

Bossa is a development environment for operating-system process schedulers that provides numerous safety guarantees. In this paper, we show how to automate the checking of safety properties of a scheduling policy developed in this environment. We find that most of the relevant properties can be considered as invariant or refinement properties. In order to automate the related proof obligations, we use the WS1S logic for which a decision procedure is implemented by Mona. The proof techniques are implemented using the FMona tool.

Keywords: scheduling, refinement, model-checking, WS1S

1 Introduction

A Domain-Specific Language (DSL) is a language designed around the precise constructs and abstractions that are relevant to a specific domain [8]. Such a language captures domain expertise, guiding the programmer in the development of programs that are concise, high level, and expressed in terms of common domain abstractions. This technology has proved its value in easing program development in a variety of areas in both research and industry [16].

In addition to easing the task of the programmer, DSL programs have the potential to be highly safe and efficient, because the language can be tuned to ease verification and optimization and because verification and optimization tools can be tuned to common domain idioms. Nevertheless, creating a verifier or optimizer for a programming language remains a difficult task, which is further complicated in the case of DSLs by the need to start from scratch for each new language. A solution is to build on existing general-purpose verification and optimization tools, but there

[1] Email: filali@irit.fr

is little experience in this area. In this paper, we contribute to the understanding of this area by presenting a case study in using an existing verification tool to construct a verifier for the Bossa DSL for implementing operating system (OS) process schedulers [17].

Bossa. Process scheduling is an old problem, but there is no single scheduling policy that is perfect for all applications. Indeed, increasingly demanding applications in areas such as multimedia and embedded systems have motivated many new scheduling algorithms [10,22,23]. As these algorithms are not available in standard OSes, the developer of an application that requires a specific scheduling policy must implement it himself at the OS level. This is difficult and error-prone, requiring substantial expertise in the target OS.

To address this issue, we have developed the Bossa environment for developing operating system process schedulers [17]. Bossa provides two DSLs: a *specification language* for describing the scheduling requirements of a given OS and a *programming language* for implementing scheduling policies. Both languages offer abstractions dedicated to the scheduling domain, allowing OS scheduling behavior and scheduling policies to be described in a high-level and natural way. Bossa has been ported to Linux and Chorus [12], and has been used in research [9] and teaching.

Bossa has a well-defined semantics and an ad-hoc verifier [14]. The verifier uses abstract interpretation to check that the specification of OS scheduling requirements is consistent and that a Bossa scheduling policy satisfies the specified requirements. The verification ensures that a scheduling policy considers as eligible for execution only those processes that are able to run. Some other properties are also checked, such as the absence of null-pointer dereferences. These checks ensure that a Bossa scheduler does not crash or hang the OS. Other properties, such as fairness and liveness could have been considered, but we have chosen to focus on properties relating to the interaction with the OS, as these are typically outside the expertise of the scheduling algorithm designer, and thus fit well with the goal of DSLs to encapsulate expertise.

This paper. As the Bossa verifier is hand-crafted and mixes the encoding and checking of the relevant properties, we have found that it is hard to maintain, especially when extending the Bossa programming language. We have thus become interested in implementing the Bossa verifications using an existing verification engine. Previously [5], we have studied how the Bossa methodology could be considered within the B [1] formal method. This study showed that some of the verifications made by Bossa could be expressed as proof obligations of B invariants or refinements. However, most of these proof obligations were not discharged automatically by the provers available in atelier B [7]: a tool supporting the refinement based B development method [1]. The reasons include the lack of suitable decision procedures and abstraction generators, particularly given the fact that the number of processes manipulated by a Bossa policy is unbounded. We thus needed a logic that could express Bossa properties in a natural way, to facilitate extension to new features, and a verification engine that would provide a decision procedure for this logic.

In this paper, we show that the WS1S logic, via the interface of the FMona tool [4], is well-suited for expressing Bossa code and properties. A decision procedure for WS1S is provided by the Mona verification tool [13]. We can thus automatically verify all of the properties considered in our previous work, and cover other properties considered by the Bossa verifier. The rest of this paper is organized as follows. Section 2 gives an overview of Bossa and presents the associated verification problem in more detail. Section 3 introduces the tools and techniques that we use. Section 4 considers the verification of the specification of OS requirements. Section 5 presents the expression of scheduling policies and their verification with respect to the specification of OS requirements. Section 6 considers the relation between the verification methods that we use and some others. Section 7 concludes and presents some future work.

2 Bossa in a Nutshell

In this section, we give an overview of the Bossa DSLs, and then consider issues that arise in showing that a Bossa scheduling policy satisfies a specification.

2.1 The Bossa DSL for programming scheduling policies

A process scheduler is the part of an OS that is responsible for electing processes to have access to the CPU. To do this, it must keep track of the set of processes that are eligible for election and have a means of electing one of them. A scheduling policy describes the strategy taken by the scheduler in these operations. These operations are thus the focus of the constructs and abstractions provided by the Bossa DSL for programming scheduling policies.

Rate Monotonic (RM) [15] is a scheduling policy often used to manage periodic processes in real-time systems. When a process should be elected to have access to the CPU, this policy picks the eligible process with the shortest period. Figure 1 shows an extract of the Bossa implementation of this scheduling policy. The implementation declares the process attributes (line 2), the process states (lines 3-10), the ordering criteria (line 11) and the event handlers (lines 12-30). The process attributes record policy-specific information about each process. In the RM policy, this includes the process's period. The ordering criteria, which is only provided for priority-based policies, specifies the relative priority of processes. In the RM policy, this specifies that the process with the shortest period should be given the highest priority. We describe the process states and the event handlers in the rest of this section.

The RM policy defines six process states: `running`, `ready`, `yield`, `blocked`, `computation_ended`, and `terminated`. A process managed by the policy is always in exactly one of these states. Each process state is associated with a *state class*, which describes the eligibility of processes in the state. The state classes are as follows:

- `RUNNING`: active processes (at most one on a uniprocessor),
- `READY`: processes eligible for election,
- `KERNEL_BLOCKED`: processes that are ineligible due to their recent interaction with the OS (*e.g.*, request for I/O),

- `POLICY_BLOCKED`: processes that are ineligible due to their recent interaction with the scheduler (*e.g.*, quantum expired),
- `KERNEL_POLICY_BLOCKED`: processes that are ineligible due to their recent interaction with both the OS and the scheduler,
- `TERMINATED`: terminated processes

A policy may define multiple states within each state class, to express further refinements required by the policy. For example, the RM policy defines the `ready` and `yield` states in the `READY` state class, to distinguish processes that are unconditionally ready to run (`ready`) from those that should only run if no other process is available (`yield`). Finally, a data structure is associated with each state: a `process` variable for a state that can contain at most one process or a `queue` for a state that can contain any number of processes.

Event handlers describe how the scheduling policy reacts to various OS events that affect the eligibility of processes. Examples include process blocking, which makes a process ineligible, and process unblocking, which makes a process eligible again. The set of events that must be handled is specific to the targeted OS. For Linux 2.4, a policy must provide handlers for ten events.

```
scheduler RM = {                                                                              1
  process = { time period; cycles wcet; timer period_timer; int missed_deadlines; }          2
  states = {                                                                                  3
    RUNNING running : process;                                                                4
    READY ready : select queue;                                                               5
    READY yield : process;                                                                    6
    KERNEL_BLOCKED blocked : queue;                                                           7
    POLICY_BLOCKED computation_ended : queue;                                                 8
    TERMINATED terminated;                                                                    9
  }                                                                                          10
  ordering_criteria = { lowest period }                                                      11
  handler (event e) {                                                                        12
    /* event block.*: e.target blocks */                                                     13
    On block.* { e.target => blocked; }                                                      14
    /* event unblock.preemptive: e.target unblocks */                                        15
    On unblock.preemptive {                                                                  16
      if (e.target in blocked) {                                                             17
        e.target => ready;                                                                   18
        if ((!empty(running)) && (e.target > running)) { running => ready; }                 19
      }                                                                                      20
    }                                                                                        21
    /* event yield.user.*: e.target wants to yield to any eligible process */               22
    On yield.user.* { e.target => yield; }                                                   23
    /* event bossa.schedule: e.target OS requests process election */                       24
    On bossa.schedule {                                                                      25
      if (empty(ready)) { yield => ready; }                                                  26
      select() => running;                                                                   27
      if (!empty(yield)) { yield => ready; }                                                 28
    }                                                                                        29
  }                                                                                          30
  interface = { ... } }                                                                      31
```

Fig. 1. Extract of the Bossa Rate Monotonic scheduling policy

We describe the `unblock.preemptive` handler (lines 14-19) in detail, as it illustrates most of the language constructs. This handler is triggered when a process unblocks and the OS allows the scheduling policy to preempt the running process, if desired. The handler manipulates the process `e.target`, which is the process that is unblocking. It first checks that this process is currently blocked (`e.target in blocked`, line 17). If so, the handler changes the state of the unblocking process to `ready` (line 18) and then checks whether the running process should be preempted (line 19). For the latter, it tests whether there is a running process (`!empty(running)`) and whether the priority of the unblocking process is greater than that of the running process (`e.target > running`). If both conditions are satisfied, then the state of the running process is changed to `ready`, requesting that it be preempted.

The language also provides operations on integers and time, loops over queues, etc. It does not provide unbounded loops or recursive functions.

2.2 The Bossa DSL for specifying OS scheduling requirements

A scheduling policy must interact with the target OS at the lowest level. At this level, OSes vary widely, and thus the definition of a scheduling policy must be tuned to the target OS. Thus, the set of event handlers, the requirements on their behavior, and the possible interactions between them are not fixed, but specified for each OS by an expert in the OS's process management behavior. This specification consists of a set of *event types* and an *event automaton*.

The event types describe the set of required event handlers and the requirements on their behavior. The latter are specified in terms of preconditions and postconditions on the states of relevant processes, such as the process generating the event (the source) or the process affected by the event (the target). Each event type rule describes a mapping of processes to states that can occur when the event is generated and the set of mappings of processes to states that are allowed on completion of the event handler. The specification is expressed in terms of the state classes, to be policy independent. As an example, the requirements on the `block.*` handler might be expressed as follows:

```
block.* : [tgt in RUNNING] -> [tgt in KERNEL_BLOCKED]
```

This rule specifies that when a process (`tgt`) blocks, it must be in a state of the `RUNNING` state class, and the handler must put the blocking process in a state of the `KERNEL_BLOCKED` state class, to record that the process is ineligible. This rule, however, is not sufficient to capture the possible interactions between the scheduler and the OS that can occur in *e.g.* Linux 2.4. The appendix presents the more complex Linux 2.4 `block.*` event type.

The event automaton describes the sequences in which the OS generates the various events. For example, to terminate a process in Linux 2.4, the OS first blocks the terminating process, then elects some other process, and eventually generates a `process.end` event to remove the process from the scheduler. Some sequences are uninterruptible; for example, Linux 2.4 always requests the election of a new process immediately after blocking a process, with no interruption possible between them. For the interruptible sequences, the OS expert also provides information about which events can occur during interrupts. The Bossa event type compiler augments the event automaton with all permutations of these events at each step in any interruptible sequence.

2.3 The Bossa verification problem

The Bossa verifier applies various consistency checks to the OS specification and to scheduling policies. We describe these checks below.

For the OS specification, the main goal is to ensure that the event types are consistent. Bossa distinguishes between *context sensitive* and *automatic* events. Context-sensitive events occur only if the mapping of processes to states satisfies the preconditions of the corresponding event types. An example is unblocking, which originates at the OS level, where process states are known. There are no consistency requirements in this case. Automatic events occur regardless of the current mapping of processes to states. An example is blocking, which derives from actions at the user level where process states are unknown. Consistency requires that the event type specify at least one allowed behavior for each mapping of processes to states that can hold when the event occurs. To check this, the verifier takes into account all possible sequences of events as specified by the event automaton, and all possible effects of the events along these sequences, as specified by the event type rules.

For a policy, the main goal is to ensure that the event handlers are well defined (*e.g.*, no null-pointer dereferences) and that they respect the event types. In this, the verifier uses the event types, instantiated according to the states defined by the policy, and the event automaton, to identify inter-handler effects. For each mapping of processes to states that is allowed by the event types and reachable according to the event automaton, the verifier analyzes each execution path through the event handler to determine the effect on the process states. The resulting mapping of processes to states must be compatible with the postconditions specified by the event type rules. This part of the verifier has been presented in detail previously [14].

Both forms of verification rely on analysis of all possible execution paths, suggesting that model checking would be appropriate. Standard model checking techniques, however, are limited to finite state spaces. In the case of Bossa, the size of the state space is determined by the number of processes, which in general is not bounded. We thus turn to the Mona tool [13], which is able to reason in the presence of unknown integer values.

3 The Mona verification tool

We first review the Mona tool [13] and then describe transition systems, properties, and verification techniques in this setting.

3.1 Mona

Mona implements a decision procedure for the logic WS1S, defined as follows:

Definition 1 (The WS1S logic) *Let* $\{x_1, \ldots, x_n\}$ *be a set of first-order variables and* $\{X_1, \ldots, X_n\}$ *a set of second-order variables. A minimal grammar for the WS1S logic as follows:*

- *a term t is defined by:* $t ::= 0 \mid x_i \mid s(t)$*, where s is the successor symbol.*

- *a formula f is defined by:*

$$f ::= \ t \in X_i \mid \neg f \mid f \wedge f \mid \exists_1 x_i.\ f \mid \exists_2 X_i.\ f \quad \textit{(1st and 2nd order quantification)}$$

A closed formula is WS1S valid if its interpretation over the set $\mathbb{N}$ of natural numbers is valid. First-order variables denote natural numbers and second-order variables denote finite subsets of $\mathbb{N}$. The logic is decidable.

To express properties relevant to Bossa, we use the FMona [4] high-level interface for Mona. This interface allows declaring enumerated types, records with updates, quantification over finite structured types, and parameterized higher-order macros. FMona code is automatically translated into Mona.

As an example, we consider the definition of a notion similar to superposition [6]. We suppose that we have an automaton, the states of which are identified by the `Location` type, and a relation `tr` over the data type `Data`. The superposition of `tr` to a transition is a new relation that is defined over the "superposed" type **record**`{d:Data; w:Location;}` as follows:

```
pred superpose(type Data, type Location, pred(var Data d,d') tr,
    var Location l,l', var record{d: Data; w: Location;} s,s') =
  s.w = l ∧ s'.w = l' ∧ tr(s.d,s'.d);
```

Partial application is possible. For example, if `l1` and `l2` are locations, then `superpose(tr,l1,l2)` is a predicate over pairs of the superposed type. Type parameters are automatically synthesized.

3.2 Transition systems and properties

We define a transition system as a triple composed of a state space, an initialisation predicate and a binary transition relation. In our case, systems are parameterized by the number of processes. Mona can analyze properties of such systems even if the parameter has not been instantiated, which is the essential advantage of Mona over traditional model checking in our context. In this paper, we consider safety properties expressed as invariants. Following Mona terminology, we do not distinguish between a property and a predicate.

Definition 2 (Invariant) *A predicate P is said to be invariant with respect to a transition system if it is true in the transition system's initial states and is preserved by its transition relation.*

Invariance properties rely on stability and do not consider actual executions. When invariance cannot be established, we can consider a weaker property: the predicate *always true* [19], i.e., satisfied over all the reachable states.

Definition 3 (always true predicate) *A state s is said to be reachable with respect to a transition system S if there exists a sequence $s_0, \ldots, s_n = s$ such that s_0 is an initial state of S and for each $i < n$ the states s_i and s_{i+1} are related by the transition relation. The predicate P is said to be always true if it is satisfied by each reachable state.*

In FMona, we introduce the macro `always_true`, which is parameterized by the type of the state space, the predicate we wish to check as always true, the initialisation predicate, and transition relation:

```
pred always_true(type State,pred(var State s) p,
    pred(var State s) init,pred(var State s,s') tr) =
  all State s: reachable(init,tr,s) ⇒ p(s);
pred reachable(type State, pred(var State s) init, pred(var State s,s') tr, var State s) =
  ex array nat of State A: ex nat i: A[i]=s ∧ init(A[0]) ∧
     all nat j where j < i: tr(A[j],A[j+1]);
```

3.3 Verification techniques

We use *refinement* to prove that a concrete specification is correct with respect to an abstract specification, *abstraction* to prove that a property holds in a concrete system, and *iteration* to calculate fixed points. We recall the definitions of refinement and abstraction, and show how iterative techniques are implemented within FMona.

Definition 4 (Refinement) *A transition system S_c (said to be concrete) refines a transition system S_a (said to be abstract) if there exists a relation φ between the states of the concrete transition system and the states of the abstract one such that:*

- *Each initial state of the concrete transition system is related by φ to an initial state of the abstract transition system.*
- *Given a concrete state* `c` *and an abstract state* `a` *related by φ, then for each element* `c'` *related to* `c` *by the concrete transition relation, there exists an element* `a'` *related to* `a` *by the abstract transition relation.*

Definition 5 (Abstraction) *Let $C = (S_c, I_c, \rightarrow_c)$ be a transition system, S_a a state space called "abstract" and φ a relation over $S_c \times S_a$. The abstraction of C through φ is a transition system $(S_a, I_a, \rightarrow_a)$ where*

- *I_a is the image by φ of I_c: $I_a = \varphi(I_c)$,*
- *$\rightarrow_a$ is the set of images by φ of pairs connected through $\rightarrow_c$.*

$$s_a \rightarrow_a s'_a \triangleq \exists s_c\ s'_c :\ \varphi(s_c, s_a) \land \varphi(s'_c, s'_a) \land s_c \rightarrow_c s'_c$$

The implementation of iterative techniques. Most of the temporal properties that we are interested in can be expressed as fixed points [2]. However, since we do not have general decidability results for reaching the fixed point, we must provide a bound on the number of iterations required to reach it. The macro `backward` is parameterized by the number of iterations N, the transition system defined by the predicates `init` and `tr` and the predicate to verify `inv`. Starting from a state characterized by the negation of `inv`, after n iterations (through the recursive macro `iterate`) of the inverse of the transition relation, we check that a fixed point has actually been reached (through the `stable` predicate) without reaching any initial state. In this way, we verify that a path between an initial state and a state satisfying ¬`inv` does not exist.

```
pred backward(type S,var nat N,pred(var S s) init,pred(var S s,s') tr, pred(var S s) inv) =
    check_bwd(init,tr,iterate(N,NOT(inv),inverse(tr)));
pred iterate(type S, var nat N, pred(var S s) start, pred(var S s,s') tr, var S s) =
  if N = 0 then start(s) else ex S s': iterate(N−1,start,tr,s') ∧ (s' = s | tr(s',s)) endif;
pred check_bwd(type S, pred(var S s) init, pred(var S s,s') tr, pred(var S s) bad)=
    stable(bad,inverse(tr)) ∧ all S s: bad(s) ⇒ ∼init(s)
```

4 Verifying the OS Specification using Mona

We now present the translation of the event types and event automaton into FMona and how some of their properties are expressed. The properties include the preservation of the representation invariant and of the number of processes in the system, and the satisfaction of event preconditions. The translation of the event types and event automaton has been automated. The properties are automatically checked by Mona for any number of processes.

4.1 Translation of the event types

We represent a Bossa state class by the set of processes associated with it. The state of the system is then represented by the record `Classes` where each field represents a state class. The additional state class `NOWHERE` represents processes that are not managed by the scheduler.

```
var nat NProc;  #   maximal number of processes
type Proc = ... NProc;  #   interval type: 0..NProc-1
type Classes = record {
  RUNNING, READY, KERNEL_BLOCKED, POLICY_BLOCKED, KERNEL_POLICY_BLOCKED,
  TERMINATED, NOWHERE: set of Proc;};
```

We represent the event type associated with a given event by a “before-after” predicate that is the disjunction of the relations associated with the various type rules. For example, if `block.*` had the single type rule:

```
block.* : [tgt in RUNNING] -> [tgt in KERNEL_BLOCKED]
```

then the following predicates would be automatically generated:

- `Eblock_`, which defines a before-after relation between elements of type `Classes`, parameterized by the source (`src`) and target (`tgt`) processes. (`block.*` does not have a source process, but the signatures of the translated types are all the same, for simplicity.)

  ```
  pred Eblock_(var Proc src, tgt, var Classes s,s') =
    (({tgt} ⊆ s.RUNNING) ∧ (s' = s with {
      RUNNING := s.RUNNING \ {tgt}; READY := s.READY \ {tgt};
      KERNEL_BLOCKED := s.KERNEL_BLOCKED \ {tgt} ∪ {tgt};}));
      POLICY_BLOCKED := s.POLICY_BLOCKED \ {tgt};
      KERNEL_POLICY_BLOCKED := s.KERNEL_POLICY_BLOCKED \ {tgt};
      TERMINATED := s.TERMINATED \ {tgt};
      NOWHERE := s.NOWHERE \ {tgt};
  ```

  To simplify the FMona code generation, the processes occurring on the left side of a rule are always subtracted from every state class. This approach is mainly useful when the left hand side contains a disjunction of filters, as it is then unnecessary to examine them separately. The FMona code is verbose, but does not need to be simplified since Mona generates an equivalent automaton-based internal representation for all equivalent formulas.

- `block_`, which is the disjunction of the transitions `Eblock_` for any possible values of `src` and `tgt`.

  ```
  pred block_(var Classes s,s') = ex Proc src, tgt: Eblock_(src,tgt,s,s');
  ```

4.2 Verification of intra-event properties

The event type should not allow a scheduling policy to put a process in more than one state, to put multiple processes in the state of the `RUNNING` state class, or to drop processes. These properties are straightforwardly encoded as invariant properties or as the preservation of a state expression.

4.3 Verification of inter-event properties

The sequencing of events is described by the Bossa event automaton. Within this automaton, Bossa distinguishes between automatic events and context-sensitive events. The event types must be constructed such that at every point in the automaton where an automatic event appears, the state of the system obtained by every path to that point is compatible with at least one of the preconditions indicated by the event's type.

To check this property, we must take into account the complete dynamics of the system. This is represented by the combination of the event automaton, the event types, and the system state (`Classes`) to describe the mapping of processes to state classes before and after each event. The resulting transition relation is defined on the state space `NClasses`:

```
type Location = ... 12; #   number of automata states
type NClasses = record { d: Classes; w: Location; };
```

The transition relation itself, `NNext`, is defined by superposing the relations describing the event types (`block_`, etc.) to the transitions in the event automaton. The disjunction of the representations of the event types for the events occurring in interrupts is superposed to automata states that represent interruptible points in the event sequences. The representations of the remaining type rules are superposed to the corresponding automaton transitions.

```
pred interrupts(var Classes s,s') = #   interrupt events
   unblock_preemptive(s,s') | unblock_timer_target_(s,s') | ...;
pred NNext(var NClasses s,s') = #   Labeling of automaton transitions by events
   superpose(interrupts,2,2,s,s') | superpose(interrupts,4,4,s,s') | ...
 | superpose(block_,0,1,s,s') | superpose(bossa_schedule,1,4,s,s') | ...;
```

Our goal is to check that every reachable state of the system satisfies the preconditions of the automatic events allowed from this state. However, computing the reachable states of the system cannot be automatic in general because the state space is parameterized by the number of processes. Thus, we use a finite abstraction of the state space where a boolean is associated to each class indicating whether the class is empty (`false` value) or not.

```
type AClasses = record {
  RUNNING, READY, KERNEL_BLOCKED, POLICY_BLOCKED, KERNEL_POLICY_BLOCKED,
  TERMINATED, NOWHERE: bool;};
```

This type is extended by the state locations of the automaton:

```
type NAClasses = record { d: AClasses; w: Location;};
```

The abstraction relation between the state space of classes and its abstraction `AClasses` contains the representation invariant `inv` (4.2) and the definition of each boolean field:

```
pred state_abs(var Classes s, var AClasses a) =
  inv(s) ∧ (a.RUNNING ↔ s.RUNNING ≠ ∅) ∧ (a.READY ↔ s.READY ≠ ∅) ∧
  (a.KERNEL_BLOCKED ↔ s.KERNEL_BLOCKED ≠ ∅) ∧
  (a.POLICY_BLOCKED ↔ s.POLICY_BLOCKED ≠ ∅) ∧
  (a.KERNEL_POLICY_BLOCKED ↔ s.KERNEL_POLICY_BLOCKED ≠ ∅) ∧
  (a.TERMINATED ↔ s.TERMINATED ≠ ∅) ∧ (a.NOWHERE ↔ s.NOWHERE ≠ ∅);
```

The predicate `nabs` extends the abstraction relation to the state space of the superposed automaton. It takes as argument a concrete value of type `NClasses` and an abstract value of type `NAClasses`, and maps state locations to themselves and abstracts the process sets attached to each class to booleans.

```
pred nabs(var NClasses c,var NAClasses a)= c.w=a.w ∧ state_abs(c.d,a.d);
```

The satisfaction of preconditions can now be expressed at the abstract level: each concrete counterpart of an abstract state must satisfy the preconditions of automatic events. We first introduce a predicate `check_pre` over concrete states of type `NClasses`, asserting that the guards of automatic events allowed by the automaton at the current state are satisfied. The abstract level predicate `check(var NAClasses a)` expresses that all the concretisations of `a` through `nabs` satisfy `check_pre`.

Backward analysis of the abstract system, which terminates here after one step, verifies this property on each reachable abstract state and thus on a superset of the reachable concrete states. Given macros `AInit` and `ANext` that compute the abstraction of the initialisation and the transition relation of the concrete system, the property is checked by the following assertion:

```
backward(1, ANext(NNext,nabs), AInit(NInit,nabs),check);
```

Remarks

- This study revealed an error in the abstraction of the behaviour of the Linux kernel as described by Bossa event types. The analysis performed by Bossa tools did not detect the error because it performs a less accurate analysis: it considers an abstraction of state classes using three values (empty, nonempty, unknown), and no verifications are performed on unknown states.
- Even if it is theoretically possible to compute the set of reachable states of the abstract system, its computational complexity makes doing so hard. This is why we have applied an iterative method. Its convergence is however guaranteed because the abstract state space is finite, even though the transitions are parameterized by the number of processes.
- The analysis performed here amounts to model checking, but because the transition relations over the state space `NAClasses` are parameterized, a finite-state model checker is not directly applicable. The use of a finite state model checker would require a program transformation, which is usually hard to validate. In our proposal, abstraction is specified at the semantic level. The correctness of the method relies on a simple meta-level theorem.

5 Verifying Scheduling Policies using Mona

We now show how the process states and event handlers defined by a Bossa scheduling policy (see Figure 1) are translated into FMona, and how their conformance to the event types is verified.

5.1 Bossa specification of process states and event handlers

Process-state declarations are automatically translated into FMona as a type declaration, which associates each state with a set of processes, and a gluing invariant relating states and classes. This gluing invariant also includes a representation invariant for states, building on the one for state classes:

```
type Bstates = record {
  running: set of Proc;
  ready: set of Proc;
  yield: set of Proc;
  blocked: set of Proc;
  computation_ended: set of Proc;
  terminated: set of Proc;
};
```

```
pred Bstates2Classes(var Bstates c,
    var Classes a) =
  inv(a) ∧ #   representation invariant
    a.RUNNING = c.running
  ∧ a.READY = c.ready ∪ c.yield
  ∧ c.ready ∩ c.yield = ∅
  ∧ a.KERNEL_BLOCKED = c.blocked
  ∧ a.POLICY_BLOCKED = c.computation_ended
  ∧ a.KERNEL_POLICY_BLOCKED = ∅
  ∧ a.TERMINATED = c.terminated;
```

The translation of event handlers to FMona relies on a weakest precondition calculus over the Bossa statements. An abstraction of subexpressions that cannot be translated into FMona (arithmetic expressions, etc.) is also performed. State updates and emptiness checks are preserved.

Bossa event handlers can be partially defined. For example, it is only allowed to put a process in a state designated as a `process` variable if the state is empty. In order to guarantee the wellformedness of event handlers, preconditions are generated in such cases. For example, in the following handler, because the state `yield` is implemented as a variable (see Figure 1, line 6), the transition to `yield` is only valid if `yield` is empty:

```
On yield.user.* { e.target => yield; }
```

The FMona translation of this handler includes the condition `yield=∅` which, when not satisfied, blocks the transition. We will prove that this cannot occur.

```
pred Byield_system_pause_(var Proc src, tgt, var Bstates s,s') =
 ((s.yield = ∅) ∧
  (s' = s with { running := s.running \ {tgt}; ready := (s.ready \ {tgt});
                 yield := s.yield ∪ {tgt}; blocked := (s.blocked \ {tgt});
                 computation_ended := s.computation_ended \ {tgt};
                 terminated := s.terminated \ {tgt};}));
```

5.2 *Verification of a policy*

We now show how to use FMona to express the conformance of a policy with the event types. This property is based on the existence of a refinement relation between transitions over the `Classes` type, said to be abstract, and transitions over the `Bstates` type, said to be concrete. It amounts to instantiating the event type rules, which are defined in terms of state classes, with respect to the states defined by a given policy.

The generic refinement property for preconditioned transitions (used here) is instantiated and expressed as a predicate over the concrete type:

```
pred check_ref(type State_c, type State_a,
  pred(var State_c s_c,var State_a s_a) φ,
  pred(var State_c s_c, s_c') tr_c,
  pred(var State_a s_a, s_a') tr_a, var State_c s_c) =
    (all State_c s_c': all State_a s_a:
      (tr_c(s_c,s_c') ∧ φ(s_c,s_a) ∧ pre(tr_a,s_a))
         ⇒ ex State_a s_a': tr_a(s_a,s_a') ∧ φ(s_c',s_a')) ∧
    all State_c s_c: all State_a s_a: (φ(s_c,s_a) ∧ pre(tr_a,s_a))
      ⇒ pre(tr_c,s_c);
```

This refinement property, however, is not usually valid. For example, a transition towards the `yield` state is only possible if this state is empty. As the event type does not require the associated state class `READY` to be empty, there is no refinement. We must thus take into account executions allowed by the automaton and only check the refinement on concrete states of type `Bstates` of which a given abstraction is reachable. This abstraction is obtained by representing each state by a boolean that is true if the state is not empty. The concrete and abstract states are superposed to those of the automaton. The refinement property is then verified for all concrete counterparts of reachable abstract states. Only allowed transitions from a given state are taken into account. In our case, convergence was obtained after three backward applications of the abstract transition. In general, the convergence of the iterative method is ensured because the abstract state space is finite. The size of the state space can, however, be much larger than the number of iterations required (640 vs. 3 in the case of the RM policy).

6 Related work

In this paper, we have automated the proof of Bossa scheduler properties using the WS1S logic and Mona. It is interesting to remark that, although the worst-case complexity of this logic is non-elementary, the computations required by the considered scheduling structures seem to be feasible. This confirms the results obtained for other domains, such as the static analysis of data structures [11] and software structures [21].

Other methods and proof techniques could have been used. Concerning methods, we mention the B method, which we have used in our previous study [5]. Nevertheless, the approach presented here requires some features that are not supported by B, namely taking into account the event automaton and the use of abstractions. Our use of these features widened the set of properties that could be expressed, while remaining within WS1S logic. This, in turn, allowed the proof obligations to be discharged automatically.

Concerning proof techniques, states could have been represented by counters. Since Mona implements Presburger arithmetic, Mona could have been used. However tools dedicated to arithmetic such as FAST [3] should be more efficient. The use of the WS1S logic allowed us to avoid abstracting sets as counters. Petri nets [20] could also have been used. Although the use of counters is restricted in this case, it should be possible to elaborate models. In both cases, the considered properties cannot be decided automatically and require either making abstractions or applying convergence accelerations.

7 Conclusion

In this paper, we have shown how to use Mona to check Bossa properties that were previously checked by specific tools. This approach separates the generation of proof obligations, which remains Bossa-specific, from the construction of the associated proofs, which is performed by a generic tool. Because the generation of proof obligations is much simpler than proof construction, this approach should make it easier to extend and modify the Bossa verifier as the needs of the language evolve. Extensions can furthermore transparently use features of the automated prover that were not built into the Bossa verifier, such as reasoning about integers. Nevertheless, while this study considers the safety properties checked by the Bossa verifier, it does not address some properties that are used by the Bossa compiler to generate optimized code. For this, a stronger coupling between Bossa and FMona is needed.

Using a general purpose verifier may seem potentially less efficient than a verifier dedicated to Bossa. Nevertheless, there has been substantial research dedicated to improving the efficiency of general-purpose provers, which goes beyond what is typically available for developing tools for a domain-specific language. In practice, we have observed that our approach gives the same performance as the Bossa verifier, around one minute to verify a typical scheduling policy. In particular, our use of iterative techniques, which avoid second-order quantification, greatly improves both the execution time and memory usage.

The Bossa verifier checks properties related to the scheduling requirements of the OS. Using our approach, properties concerning the scheduling algorithm and the scheduled application can now also be considered. For instance, does the implementation of a scheduler satisfy a given property? And will an application meet its deadlines under a given scheduler? Finally, the generation of certified code as well as proof annotated code [18] could be considered.

——— This work was supported in part by the Danish Research Council, grant 21-03-0545 and in part by the project CORSS : *Composition et raffinement de systèmes sûrs* of program "ACI Sécurité Informatique" supported by the French Ministry of Research and New Technologies.

Availability Bossa is available at `http://www.emn.fr/x-info/bossa/`.

References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[2] A. Arnold. *Finite transition systems.* Prentice-Hall, 1994.

[3] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast acceleration of symbolic transition systems. In *Proc. 15th Conf. Computer Aided Verification (CAV'2003)*, volume 2725 of *LNCS*, pages 118–121. Springer, 2003.

[4] J.-P. Bodeveix and M. Filali. A generic tool for expressing the development of validations . In *11th Nordic Workshop on Programming Theory NWPT'99*, pages 37–37, Uppsala University, 6-8 October 1999. Bjorn Victor and Wang Yi.

[5] 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*, volume 3771 of *LNCS*, pages 187–206, 29 November-2 December 2005.

[6] K. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[7] CLEARSY. L'atelier B. version 3.6. Technical report, http://www.atelierb.societe.com/.

[8] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Pisa, Italy, Sept. 1998.

[9] J. Cordry, N.Bouillot, and S. Bouzefrane. Bossa et le concert virtuel réparti, intégration et paramètrage souple d'une politique d'ordonnancement spécifique pour une application multimédia distribuée. In *13th International Conference on Real-Time Systems*, Paris, France, Apr. 2005.

[10] 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 (SOSP'99)*, pages 261–276, Kiawah Island Resort, SC, Dec. 1999.

[11] J. Elgaard, A. Møller, and M. I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proc. Programming Languages and Systems, 9th European Symposium on Programming, ESOP '00*, volume 1782 of *LNCS*, pages 182–194. Springer-Verlag, March/April 2000.

[12] Jaluna. Jaluna Osware. http://www.jaluna.com.

[13] N. Klarlund. Mona & fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, LNCS, 1998. 1414.

[14] 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 *LNCS*, pages 436–455, Vancouver, October 2004. Springer-Verlag.

[15] 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.

[16] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[17] G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE 2005 - High Assurance Systems Engineering Conference*, pages 56–65, Heidelberg, Germany, Oct. 2005. IEEE.

[18] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI96*, pages 1–13, Seattle, October 1996.

[19] J. S. Pettersson. Comments on "Always-true is not invariant": assertional reasoning about invariance. *Information Processing Letters*, 40(5):231–233, December 1991.

[20] C. Reutenauer. *The mathematics of Petri nets*. Prentice-Hall, 1990.

[21] A. Sandholm and M. I. Schwartzbach. Distributed safety controllers for web services. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering*, number 1382 in LNCS, pages 270–284. Springer-Verlag, March/April 1998.

[22] 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 (OSDI)*, pages 145–158, New Orleans, LA, Feb. 1999.

[23] 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 (Lake George), NY, Oct. 2003.

A The `block.*` event type for Linux 2.4

The requirements on the `block.*` event in the context of Linux 2.4 are specified as follows:

```
block.*: [tgt in RUNNING]->[tgt in KERNEL_BLOCKED]
block.*: [[] = RUNNING,tgt in READY]->[tgt in KERNEL_BLOCKED]
block.*: [[] = RUNNING,tgt in POLICY_BLOCKED]->[tgt in KERNEL_POLICY_BLOCKED]
```

A process is always executing when it blocks. The first rule thus specifies that when the blocking process (`tgt`) is in the state of the `RUNNING` state class, the handler must put the blocking process in a state of the `KERNEL_BLOCKED` state class, to record that the process is ineligible. An executing process, however, is not always in the state of the `RUNNING` state class. As illustrated by `unblock.preemptive` handler (line 17, Figure 1), a handler can request preemption of the executing process, by changing its state from the state of the `RUNNING` state class to a state of the `READY` state class, if the process is still considered eligible for election, or to a state of the `POLICY_BLOCKED` state class, if not. The remaining rules thus consider the cases where preemption has been requested in this manner. In the former case, the blocking process must be put in a state of the `KERNEL_BLOCKED` state class to record that it is ineligible due to its interaction with the OS. In the latter case, it must be put in a state of the `KERNEL_POLICY_BLOCKED` state class to record that when the process unblocks, becoming eligible from the point of view of the OS, it is still ineligible from the point of view of the scheduling policy.