On Designing a Target-Independent DSL for Safe OS Process-Scheduling Components

Julia L. Lawall[1], Anne-Françoise Le Meur*[1], and Gilles Muller[2]

[1] DIKU, University of Copenhagen, 2100 Copenhagen Ø, Denmark
{julia,lemeur}@diku.dk
[2] Ecole des Mines de Nantes/INRIA, 44307 Nantes Cedex 3, France
Gilles.Muller@emn.fr

Abstract. Developing new process-scheduling components for multiple OSes is challenging because of the tight interdependence between an OS and its scheduler and because of the stringent safety requirements that OS code must satisfy. In this context, a domain-specific language (DSL), designed by a scheduling expert, can encapsulate scheduling expertise and thus facilitate scheduler programming and verification. Nevertheless, designing a DSL that is target-independent and provides safety guarantees requires expertise not only in scheduling but also in the structure of various OSes. To address these issues, we propose the introduction of an OS expert into the DSL design process and the use of a type system to enable the OS expert to express relevant OS properties.
This paper instantiates our approach in the context of the Bossa process-scheduling framework and describes how the types provided by an OS expert are used to ensure that Bossa scheduling components are safe.

1 Introduction

A domain-specific language (DSL) is a programming language dedicated to a given family of problems, known as a *domain*. Such a language provides high-level abstractions allowing the *DSL programmer* to focus on what to compute rather than on how to perform this computation [7]. Specifically, the DSL approach relieves the programmer both of constructing code appropriate for a given target and of ensuring that this code satisfies target-specific requirements. Instead, these issues are addressed by the compiler and verifier of the DSL.

Commonly, the responsibilities of designing a DSL and providing the associated compiler and verifier are delegated to a *domain expert*. Such an expert has a broad view of the algorithms relevant to the domain, and thus can select appropriate language abstractions. Nevertheless, this kind of knowledge is not sufficient to create a complete implementation of the language. Constructing a compiler and verifier requires low-level understanding of the target. When there are multiple targets, taking them all into account multiplies the expertise required and the complexity of the compiler and verifier implementations.

* Author's current address: Université des Sciences et Technologies de Lille, LIFL, INRIA project Jacquard, 59655 Villeneuve d'Ascq, France.

In this paper, we examine these issues in the context of the DSL provided by the Bossa framework for implementing operating system (OS) process-scheduling components [14, 16]. The goal of Bossa is to facilitate the integration of new schedulers in existing OSes, both general-purpose OSes, such as Linux and Windows, and those that address special needs, such as OSes for real-time and embedded systems. In the domain of scheduling, there is a substantial gap between expertise in the domain itself and expertise in the target, *i.e.*, a given OS. Scheduling policies are typically developed and presented at a theoretical level, where the main concerns are high-level properties such as liveness and CPU utilization. Implementing a scheduling policy, on the other hand, requires a deep knowledge of the interaction between the OS and its scheduler, and this interaction differs for each OS. Few scheduling experts possess both expertises, particularly across the range of OSes targeted by Bossa.

Our approach To address the need for OS expertise and for OS independence in the design of a DSL for process scheduling, we propose two extensions to the DSL design process. First, we introduce an *OS expert*, *i.e.*, an expert in the target OS, who provides information about OS behavior relevant to the domain of scheduling. Second, we propose that the scheduling expert define a *type system* to be used by the OS expert to create types describing this information. The scheduling domain expert constructs a compiler and a verifier that use these types in generating code appropriate to the target OS and in checking that the behavior of a scheduling policy implementation satisfies OS requirements. This approach allows the DSL to be targeted to many OSes, without complicating its implementation, and eases subsequent evolution of the language to new OSes. In the context of process scheduling, we focus on the problems of codifying the behavior that a target OS requires from its scheduler and of checking that a given scheduler implementation satisfies these requirements, as these are the main sources of OS dependence in this domain.

The specific contributions of this paper are:

- We present a novel approach to the design of a DSL for process scheduling. This approach incorporates an OS expert and a type system to make the language implementation target-independent.
- We instantiate this approach in the context of the DSL of the Bossa process-scheduling framework. In this context, we present a type system that enables an OS expert to express OS information relevant to scheduling.
- We show how to exploit the information provided by the OS expert in the verifier, by presenting a static analysis that checks whether a Bossa scheduling policy satisfies the OS requirements specified using the type system.
- We illustrate the use of the analysis on typical Bossa code.

The rest of this paper is organized as follows. Section 2 presents an overview of our approach and instantiates this approach for the Bossa framework. Section 3 presents the type system used in Bossa and the corresponding analysis. Section 4 illustrates the various features of the analysis on a realistic example. Finally, Section 5 describes related work and Section 6 concludes.

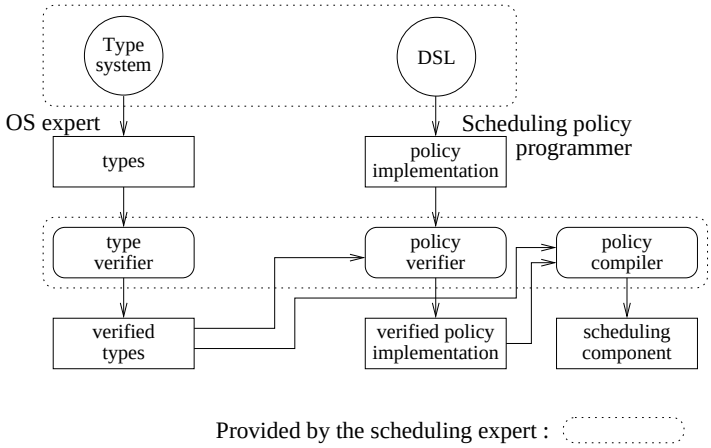

Fig. 1. Verification and compilation tools

2 DSL Design Approach

We first give an overview of our approach, and then instantiate it in the context of the Bossa framework. The languages and tools used in our approach are illustrated in Figure 1.

2.1 Methodology

Based on an analysis of the range of scheduling policies, the scheduling expert identifies the kinds of programming abstractions and properties of OS behavior that are relevant to the scheduling domain. He then designs a DSL incorporating the identified abstractions and codifies the range of properties in a type system. Having designed the language and the type system, the scheduling expert implements a verifier and compiler that are parameterized by type information. The verifier checks that the implementation of a scheduling policy satisfies the provided types, while the compiler uses type information to optimize the code generated to produce the scheduling component.

Each OS has its own properties and conventions. To make these apparent, the OS expert describes the interface that must be provided by the implementation of a scheduling policy and uses the type system to create types describing the expected behavior of each definition required by this interface. The OS expert then configures the verifier and compiler developed by the scheduling expert with this type information. The result is a verifier and compiler specific to the given OS. The types are also given as documentation to the scheduling policy programmer to guide scheduler development. Even though this approach implies that the resulting scheduling components are OS-specific, in practice there are substantial opportunities for code re-use between policy implementations for different OSes.

Because understanding OS behavior is difficult, we propose that the scheduling expert also provide the OS expert with a verifier to check basic properties of

```
1  states = {                    10  On unblock.preemptive {
2    RUNNING running : process;  11    if (e.target in blocked) {
3    READY ready : select queue; 12      if ((!empty(running))
4    READY expired : queue;      13          && (e.target > running)) {
5    READY yield : process;      14        running => ready;
6    BLOCKED blocked : queue;    15      }
7    TERMINATED terminated;      16      e.target => ready;
8  }                             17    }
9                                18  }
```

Fig. 2. Extract of the Linux policy

the types. Such a verifier may check, for example, that a complete set of types is provided and that the types satisfy certain properties generic to scheduling. Details of this verifier are out of the scope of this paper.

2.2 Scheduling Expertise

The design of the Bossa framework has been based on an extensive study of scheduling policies, both those found in commercial OSes and those developed for special needs, such as multimedia and real-time applications. A scheduling policy describes how to elect a new process for execution when requested to do so by the kernel. Whether a process is eligible for election depends on its current state (*i.e.*, running, ready to run, or blocked). Thus, a scheduling policy also typically specifies how to react to process state changes. Requests to elect a new process and process state changes can be viewed as *events*. The Bossa framework is organized around a set of *event notifications* that are generated by the kernel[3] and handled by a scheduling component.

In the role of the scheduling expert, we have designed the Bossa DSL and the type system for use within this framework. The Bossa DSL organizes the implementation of a scheduling policy as a collection of event handlers, of which one elects a new process and the others respond to process state changes. The types of the Bossa type system amount to pre- and post-conditions that describe the possible states of relevant processes when an event notification is generated and the allowed effects of the corresponding handler on these process states. Checking of these types ensures that each handler treats all possible input configurations and adjusts process states as required by the OS.

The Bossa DSL. The Bossa DSL provides facilities for declaring process states and associated data structures, for managing the association of processes to states, and for performing other kinds of relevant computation (*e.g.*, arithmetic). We illustrate the DSL using excerpts of a Bossa implementation of the Linux scheduling policy (Figure 2).

The process states used by the Linux policy are defined in lines 1–8 of Figure 2. Each state is associated with a *state class*, indicating the schedulability

[3] A kernel must be prepared for use with Bossa by inserting these event notifications at the points of scheduling-related actions. We have addressed this issue elsewhere [1].

Process schedulers	Lines of code
The Linux 2.4 scheduling policy	201
Progress-based scheduling [21]	234
Lottery scheduling (one currency) [22]	128
Earliest-Deadline First	124
Rate Monotonic (RM)	134
RM + aperiodic processes (polling server)	262

Table 1. Code size of some Bossa schedulers

of the processes in the given state. The state class `RUNNING` describes the process that is currently running. The state class `READY` describes a process that is ready to run (runnable). The state class `BLOCKED` describes a process that is blocked, waiting for a resource. The state class `TERMINATED` describes a process that is terminating. Each state declaration also indicates whether a single variable (`process`) or a queue (`queue`) should be used to record the set of processes currently in the state. One state in the state class `READY` is designated as `select`, indicating that processes are elected for execution from this state. State classes describe the semantics of the declared states, and are crucial to the verification process, as described in Section 3.

The handler for the `unblock.preemptive` event is shown in lines 10-18 of Figure 2. This event is generated by the Bossa version of the Linux OS when a process unblocks and the scheduler is allowed to preempt the running process. Bossa event handlers are written using a syntax that amounts to a restricted version of C, but includes operators specific to the manipulation of process states. The `unblock.preemptive` handler first checks whether the unblocking process, referred to as the *target* process or `e.target`, is currently in the `blocked` state. If so, lines 12–15 determine whether some process is running, and if so, whether this running process should be preempted. These lines first test whether there is a process in the `running` state, using the Bossa primitive `empty` (line 12), and if so check whether the priority of the target process is greater than that of the running process (line 13). If both tests succeed, the state of the running process is changed to `ready` (line 14), indicating that the running process should be preempted. Finally, the state of the target process is also changed to `ready` (line 16), indicating that the target process is newly ready to run.

The Bossa DSL has been used to implement a variety of scheduling policies. Implementations are concise, as shown in Table 1. A grammar of the language and some example policy implementations can be found at the Bossa web site.[4]

The Bossa type system. The Bossa type system allows the OS expert to describe pre- and post-conditions on the behavior required by a particular OS, in a policy-independent way. Accordingly, types are expressed using state classes, rather than the states defined by a specific policy implementation. In addition to the state classes used in state declarations, the state class `NOWHERE` is used to describe a process that is being created and thus not yet managed by the scheduler.

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

A type consists of a collection of rules having the form *input* → *output*, defined as follows:

Input configuration:	*input*	::= [in_1 , ... , in_n]
Input component:	*in*	::= [p_1 ,...,p_n] in c_{in} | [] = c_{in}
Input state classes	c_{in}	::= RUNNING | READY | BLOCKED | NOWHERE
Output configuration:	*output*	::= [out_1 , ... , out_n]
Output component:	*out*	::= [p_1 ,...,p_n] in c_{out} | [p_1 ,...,p_n] in c_{out}! | c_{out}!
Output state classes	c_{out}	::= RUNNING | READY | BLOCKED | TERMINATED
Process names:	*p*	::= src | tgt | p1 | ... | pn

An input configuration describes possible process states on entering the handler and an output configuration describes process states required on exiting the handler. An input configuration can specify that specific processes are in a given state class ([p_1 , ... ,p_n] in c_{in}) or that there are no processes in a given state class ([] = c_{in}). Output configurations are similar, but c_{out}! indicates that processes may change state within the given state class. The names `src` and `tgt` refer to the source[5] and target processes of the event, respectively. The names `p1`, ..., `pn` refer to arbitrary processes distinct from the source and target processes. State classes and process names cannot be duplicated in a configuration and only process names occurring in the input configuration can occur in the output configuration.

As an example, an OS expert might use the following type to describe the behavior of a naive unblock event handler:

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

The first rule indicates that the target process is initially blocked, and that the event handler should change the state of this process such that the policy subsequently considers the process to be runnable. No other state change is allowed. The second rule indicates that if there is currently some running process p, then the event handler can change the state of both the running process and the target process such that both processes are subsequently considered to be runnable. Again, no other state change is allowed.

The pseudo-parallelism introduced by interrupts must be accounted for when analyzing the interactions between event handlers. To describe possible run-time interactions between handlers, the type system includes automata to allow the OS expert to specify possible sequences of events. The transitions in these sequences are specified to be interruptible or uninterruptible. The type information also includes a list of the events that can occur during interrupts.

2.3 OS Expertise

The OS expert identifies the interface and types particular to an OS that has been prepared for use with Bossa. We have created a version of Linux 2.4 for

[5] An event has a source process if the originator of the event is a process other than the processes affected by the event (the target process). For example, the event corresponding to process creation has both a source and a target process.

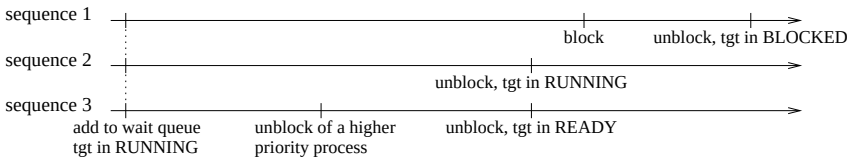

Fig. 3. Event sequences involving `unblock.preemptive`

use with Bossa that uses the following 11 basic events: process creation, process termination, blocking, three variants of yielding, two variants of unblocking, clock tick, timer expiration, and process election [1]. The OS expert declares one type for each event.

In most cases, the types of the Linux events are intuitive. For example, the type of an unblocking event requires that a blocked process change from a state in the `BLOCKED` state class to a state in the `READY` state class. Nevertheless, the sequences of events that can occur due to interrupts in the Linux kernel imply that sometimes an event handler can receive processes that are not in the states expected by an intuitive description of the handler behavior. We illustrate this issue using the `unblock.preemptive` event. Some possible event sequences involving this event are shown in Figure 3. In the Linux kernel, a process blocks by first placing itself on a wait queue and then requesting to be blocked. A process can be unblocked as soon as it is on a wait queue. If the process has already blocked, as illustrated by the first sequence, then this process, *i.e.*, the target process from the point of view of the handler, is in a state in the `BLOCKED` state class, and the rules presented in Section 2.2 apply. If the process has not yet blocked, then it is normally in a state in the `RUNNING` state class, as illustrated by the second sequence, but may be in a state in the `READY` state class, *e.g.*, if it has been preempted by the unblocking of a higher priority process, as illustrated by the third sequence. In the latter two cases the process should remain in its current state. The complete type for `unblock.preemptive` is thus as follows:

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

The Bossa type verifier (Figure 1) checks that this type is consistent with the declared event sequences.

The considerations that go into the development of the types for `unblock.preemptive` require a deep understanding of a range of Linux mechanisms (blocking, interrupts, *etc.*). A correct policy implementation can, however, often be constructed based on only the knowledge of what cases should be treated and what transitions are allowed. This information is made available to the programmer by the inclusion of the types in the Bossa documentation. Even when more information about kernel behavior is needed to define an event handler, the types are still useful in signaling the possibility of unexpected behavior.

3 Static Analysis for Bossa

The Bossa verifier connects a policy implemented using the DSL provided by the scheduling expert to the types provided by the OS expert. Specifically, the verifier checks that the policy implementation satisfies the types, and is thus coherent with the behavior of the target OS. We first present the semantics of a core subset of the DSL and then present the type checking analysis used by the verifier. This analysis relies critically on features of the DSL that make process-state properties explicit, such as the classification of process states into state classes and the use of explicit state names in operations that test state contents and implement state changes.

3.1 Semantics of the Bossa DSL

The types provided by the OS expert describe changes in process states. We thus consider a core subset of the Bossa DSL that focuses on the operations that relate to the states of processes. The syntax of this core language is as follows:

Statements: $stmt ::=$ exp `=>` $state$ | `foreach (` x `in` $state$ `)` $stmt$
| `if (` $bexp$ `)` $stmt_1$ `else` $stmt_2$ | `{`$stmt_1$ `;` $stmt_2$`}` | `{ }`
Process expressions: $exp ::=$ `e.source` | `e.target` | x | $state$ | `select()`
Boolean expressions: $bexp ::=$ `empty(`$state$`)` | exp `in` $state$ | exp_1 `>` exp_2

The only constructs not previously presented are `e.source`, which refers to the source process, `foreach`, which iterates over the set of processes in a given state, and `select()`, which elects the highest priority process in the state designated as `sorted`.

The big-step operational semantics [18] of this language is given in Figure 4. The semantics is defined in terms of the following judgments:

Statements:	$\phi, \sigma \vdash stmt \rightarrow \sigma'$
Process expressions:	$\phi, \sigma \vdash exp \rightarrow process$
Boolean expressions:	$\phi, \sigma \vdash bexp \rightarrow bool$

An auxiliary judgment $\pi, \phi, \sigma \vdash^x stmt \rightarrow \sigma$ is used to control the iteration in the semantics of `foreach`. These judgments refer to the following semantic objects:

Variable environment:	$\phi : (var \cup \{\texttt{e.source}, \texttt{e.target}\}) \rightarrow process$
State environment:	$\sigma, \sigma' : state \rightarrow \mathcal{P}(process)$
Process:	$process : process\_id \times priority$

The set of processes, *process*, is finite. A state environment σ partitions the set of processes, *i.e.*, a process cannot be in more than one state at a time. Each process is associated with a unique identifier *process_id* and a priority drawn from some totally ordered domain. By a slight abuse of notation, we use σ^{-1} to designate a function that maps a process to its current state as defined by σ. The semantics also uses the function state_info(*state*), which returns the kind of data structure (queue or process variable) implementing *state*.

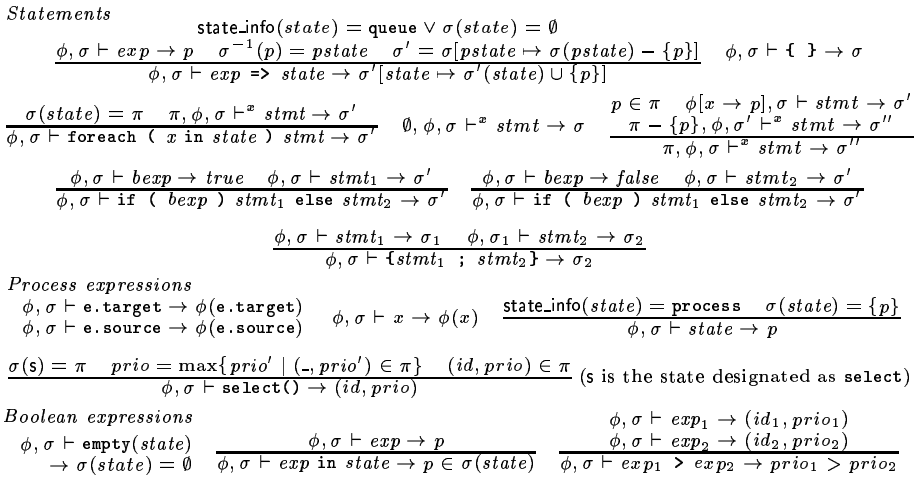

Fig. 4. Semantics of a core subset of Bossa

3.2 Analysis of Bossa Programs

The analysis performed by the Bossa verifier is based on a number of well-known dataflow-analysis techniques, including inferring information from conditional-test expressions, not merging the results of analyzing conditional branches, and maintaining a precise representation of the contents of data structures. In the context of Bossa, the simple program structure (*e.g.*, the absence of function calls) and the use of domain-specific constructs imply that these techniques can be used more effectively than is typically possible in a general-purpose language. The result is an analysis that is precise enough to identify policy implementations that satisfy the types provided by the OS expert, while detecting implementations that violate OS requirements.

The analysis is inter-procedural (*i.e.*, inter-handler) following a graph of possible execution paths through the policy implementation derived from the event sequences given by the OS expert. Each step in the analysis considers a pair of a handler and an abstract state environment that has not previously been explored. The analysis simulates the behavior of the handler on the state environment, producing a set of possible resulting state environments. These environments are then checked against the type of the handler. If the type is satisfied, then all possible successor handlers are paired with each of the resulting state environments, and pairs that have not been considered previously are added to the set of pairs to explore.

The analysis manipulates a set of abstract values. We first present these abstract values, then present some useful functions, and finally present the analysis rules. The analysis has been implemented in the Bossa verifier.

Abstract values. The information contained in the abstract values is determined by the kinds of properties described by the types. Types describe the starting and ending state classes of various processes, notably the source and target processes,

as well as the allowed transitions between state classes and within the states of a state class. Accordingly, the abstract values keep track of the state of the source process and the state of the target process and permit to determine both the original and current state of each process.

A *process description*, *pd*, is a pair of one of the forms (src, *state*), (tgt, *state*), (x, *state*), (x, *class*), or (x, $\top$), where src, tgt, and x are constants that represent the source, target, and any other process, respectively. Process descriptions are ordered as follows:

$$(\mathsf{src}, \mathit{state}), (\mathsf{tgt}, \mathit{state}) \sqsubseteq (\mathsf{x}, \mathit{state}) \sqsubseteq (\mathsf{x}, \mathit{class}) \sqsubseteq (\mathsf{x}, \top)$$

(x, *state*) $\sqsubseteq$ (x, *class*) only holds if *state* was declared to be in the state class *class*. If the second component is the name of a state or state class, it is the state or state class of the process at the start of the handler. If this component is $\top$, the state of the process at the start of the handler is unknown (the starting state of the source or target process is always known). A process description can either represent a specific process or one of a set of possible processes. Those of the form (src, *state*) or (tgt, *state*), or of the form (x, *state*) where *state* is represented as a process variable, can only represent a single process. Such a process description is said to be *unique*. The predicate unique(*pd*) is true only for these process descriptions.

The analysis is defined in terms of an abstract state environment that associates with each state a *contents description*, *cd* of the form [] or $\langle \mathit{must}, \mathit{may} \rangle$. The description [] indicates that it is known that there are no processes in the state. A description $\langle \mathit{must}, \mathit{may} \rangle$ describes the set of processes in a state using "must" and "may" information, analogous to that used in alias analysis [15]. Specifically, *must* is a set of process descriptions of which each element represents a distinct process that is known to be in the state, and *may* is a set of process descriptions that describes a safe approximation (superset) of all processes that may be in the state. Redundant information is not allowed in may information; for example, may information can contain only one of (tgt, *state*) and (x, *state*). For convenience, the functions must and may return the corresponding components of a contents description of the form $\langle \mathit{must}, \mathit{may} \rangle$, and return $\emptyset$ for the [] contents description. The maintaining of both must and may information often enables the analysis to return a precise result (*i.e.*, *true* or *false*) as the result of analyzing the domain-specific boolean expressions `empty(`*exp*`)` and *exp* `in` *state*.

The ordering on contents descriptions is defined as follows:

$$[] \sqsubseteq [] \qquad \frac{\mathit{must}_2 = \{\mathit{pd}'_1, \ldots, \mathit{pd}'_n\} \quad \exists (\mathit{pd}_1, \ldots, \mathit{pd}_n) \in \mathsf{choose}(\mathit{must}_1, n). \forall i. \mathit{pd}_i \sqsubseteq \mathit{pd}'_i \quad \forall \mathit{pd} \in \mathit{may}_1. \exists \mathit{pd}' \in \mathit{may}_2. \mathit{pd} \sqsubseteq \mathit{pd}'}{\langle \mathit{must}_1, \mathit{may}_1 \rangle \sqsubseteq \langle \mathit{must}_2, \mathit{may}_2 \rangle}$$

$$[] \sqsubseteq \langle \emptyset, \mathit{may} \rangle$$

The function choose(π, n) returns all possible tuples of n distinct elements of π. Descriptions lower in the ordering are more informative than those higher in the ordering. This ordering is a partial order and extends pointwise to state environments. Based on this ordering, an upper bound of two contents descriptions is computed as follows. $[] \sqcup [] = []$. Otherwise $\mathit{cd}_1 \sqcup \mathit{cd}_2 = \langle \alpha \cup \beta \cup \gamma \cup \delta, \mathsf{may}(\mathit{cd}_1) \uplus$

$\mathsf{may}(cd_2)\rangle$ where α, β, γ, δ, and $\uplus$ are defined as follows:

$$
\begin{array}{l}
must_1 = \mathsf{must}(cd_1), \quad must_2 = \mathsf{must}(cd_2), \\
must'_1 = \{(\mathsf{x}, s) \mid (p, s) \in must_1 - must_2\}, \quad must'_2 = \{(\mathsf{x}, s) \mid (p, s) \in must_2 - must_1\}, \\
must''_1 = \{(\mathsf{x}, c) \mid (\mathsf{x}, s) \in must'_1 - must'_2 \wedge \text{state } s \text{ in state class } c\}, \\
must''_2 = \{(\mathsf{x}, c) \mid (\mathsf{x}, s) \in must'_2 - must'_1 \wedge \text{state } s \text{ in state class } c\}, \\
\alpha = must_1 \cap must_2, \quad \beta = must'_1 \cap must'_2, \quad \gamma = must''_1 \cap must''_2 \\
\delta = \{(\mathsf{x}, \top)\}, \text{ if } |\alpha \cup \beta| < \min(|must_1|, |must_2|). \text{ Otherwise, } \emptyset. \\
\pi_1 \uplus \pi_2 = \{pd \mid pd \in \pi_1 \cup \pi_2 \wedge \forall pd' \in ((\pi_1 \cup \pi_2) - \{pd\}).pd' \not\sqsupseteq pd\}
\end{array}
$$

Some useful functions. The main purpose of the analysis is to track changes in process states. For this, the analysis uses two key functions, add and remove, that add a process description to the information known about a state and remove a process description from the information known about a state, respectively, in some abstract state environment Σ, mapping states to contents descriptions.

There are two cases in the definition of add, depending on whether the process description is unique:

$$
\begin{array}{l}
\text{if } \mathsf{unique}(pd), must = \mathsf{must}(\Sigma(state)), may = \mathsf{may}(\Sigma(state)) : \\
\quad \mathsf{add}(pd, state, \Sigma) = \Sigma(\{state \mapsto \langle \{pd\} \cup must, \{pd\} \uplus may\rangle\} \\
\qquad\qquad \cup \{state' \mapsto \langle must', may' - \{pd\}\rangle \mid \\
\qquad\qquad\qquad state' \neq state \wedge \Sigma(state') = \langle must', may'\rangle\})
\end{array}
$$

$$
\begin{array}{l}
\text{if } \neg\mathsf{unique}(pd), must = \mathsf{must}(\Sigma(state)), may = \mathsf{may}(\Sigma(state)) : \\
\quad \mathsf{add}(pd, state, \Sigma) = \Sigma[state \mapsto \langle \{pd\} \cup must, \{pd\} \uplus may\rangle]
\end{array}
$$

In both cases, the process description is added to the must and may information (in the latter case, the use of $\uplus$ implies that the process description is only added if it is not already described by some process description in the may information). If the process description is unique, then adding it to the must information of the current state implies that it cannot be in any other state. It is an invariant of the analysis that when add is used, if the process description is unique, it cannot be in the must information of any state. It can, however, be part of the may information, due to the use of the $\sqcup$ operation. The add operation thus removes the process description from the may information of the other states.

There are three cases in the definition of remove, depending on whether the process description is unique, and if so depending on the implementation of the state, if any, for which the process description occurs in the must information:

$$
\frac{\mathsf{unique}(pd) \quad \Sigma(state) = \langle must, may\rangle \quad pd \in must \quad state\_info(state) = \texttt{queue}}{\mathsf{remove}(pd, \Sigma) = (\{state\}, \Sigma[state \mapsto \langle must - \{pd\}, may\rangle])}
$$

$$
\frac{\mathsf{unique}(pd) \quad \Sigma(state) = \langle must, may\rangle \quad pd \in must \quad state\_info(state) = \texttt{process}}{\mathsf{remove}(pd, \Sigma) = (\{state\}, \Sigma[state \mapsto [\,]])}
$$

$$
\frac{\begin{array}{c}\neg\mathsf{unique}(pd) \vee \forall state.pd \notin \mathsf{must}(\Sigma(state)) \\ s = \{state \mid \exists pd' \in \alpha(pd).pd' \in \mathsf{may}(\Sigma(state))\}\end{array}}{\mathsf{remove}(pd, \Sigma) = (s, \Sigma\{state \mapsto \langle \mathsf{must}(\Sigma(state)) - \alpha(pd), \mathsf{may}(\Sigma(state))\rangle \mid state \in s\})}
$$

$$
\alpha(pd) = \{pd' \mid pd \sqsubseteq pd' \vee pd' \sqsubseteq pd\}
$$

This function returns both a set of states in which the process represented by the process description pd may occur, and a state environment in which any possible representation of this process has been removed from the must information. In the first two rules, because the process description is unique and occurs in must information, we know the state of the associated process. If this state is implemented as a queue, then the process description is simply removed from the must information in the resulting state environment. If this state is implemented as a process variable, then removing the process makes the state empty. Thus, the information associated with the state is replaced by $[]$. In the final rule, either the process description is not unique, or it does not occur in any must information. In this case, the process may occur in any state whose may information contains a process description related to the given process description, as computed using α. All elements of $\alpha(pd)$ are removed from the must information of such states. All such states are returned as the set of possible states of the process.

Analysis rules. The analysis simulates the execution of a Bossa handler with respect to variable and state environments described in terms of process descriptions and contents descriptions, respectively. The result of the analysis is a set of state environments, which are then checked against the type of the handler to determine whether the handler satisfies OS requirements. The analysis rules are shown in Figure 5 and are defined using the following judgments:

Statements:	$\Phi, \Sigma \vdash_{\mathrm{s}} stmt : S$
Process expressions:	$\Phi, \Sigma \vdash_{\mathrm{e}} exp : (pd, s, \Sigma')$
Boolean expressions:	$\Phi, \Sigma \vdash_{\mathrm{b}} bexp : bool^{+}$

The environments Φ and Σ and the results S, s, and $bool^{+}$ are defined as:

$$\Phi : (var \cup \{\texttt{e.source}, \texttt{e.target}\}) \to pd \qquad S : \mathcal{P}(state \to cd) \qquad s : \mathcal{P}(state)$$
$$\Sigma, \Sigma' : state \to cd \qquad bool^{+} : bool + ((state \to cd) \times (state \to cd))$$

The initial variable environment Φ contains information about the source and target process, derived from the initial state environment under consideration.

Analysis of a statement returns a set of state environments, representing the effect on process states of the various execution paths through the statement. To increase precision, the analysis keeps the results of analyzing individual conditional branches separate, rather than computing a single environment approximating their different effects. This degree of precision is tractable because of the simple structure of Bossa handlers. Analysis of a process expression returns a process description pd describing the value of the expression, a set s of states representing a safe approximation (superset) of the state of this process, and a state environment Σ' that is a safe approximation of the effect of removing the process from its current state. This information is used in the analysis of an `in` expression and in the analysis of a state-change statement. Analysis of a boolean expression returns either a boolean value or a pair of state environments. In the latter case, the pair components represent the current state environment enhanced with information derived from the assumption that the boolean expression is true or false, respectively. The domain-specific boolean expressions

Statements

$$\frac{\Phi, \Sigma \vdash_e exp : (pd, s, \Sigma') \quad \mathsf{state\_info}(state) = \mathsf{queue} \vee \Sigma'(state) = [\,] \quad pd \notin \mathsf{must}(\Sigma'(state))}{\Phi, \Sigma \vdash_s exp \texttt{ => } state : \{\mathsf{add}(pd, state, \Sigma')\}} \quad (1)$$

$$\frac{\Phi, \Sigma \vdash_e exp : (pd, s, \Sigma') \quad \mathsf{state\_info}(state) = \mathsf{queue} \vee \Sigma'(state) = [\,] \quad pd \in \mathsf{must}(\Sigma'(state)) \quad pd' = \min(\{pd' \mid pd' \in (\alpha(pd)) \wedge pd' \sqsupseteq pd \wedge (pd' \notin \mathsf{must}(\Sigma'(state)) \vee pd' = (\mathsf{x}, \top))\})}{\Phi, \Sigma \vdash_s exp \texttt{ => } state : \{\Sigma'[state \mapsto \langle \{pd'\} \cup \mathsf{must}(\Sigma'(state)), \{pd\} \uplus \mathsf{may}(\Sigma'(state))\rangle]\}} \quad (2)$$

$$\frac{\Phi[x \mapsto \sqcup \mathsf{may}(\Sigma(state))], \Sigma \sqcup (\sqcup S) \vdash_s stmt : S}{\Phi, \Sigma \vdash_s \texttt{foreach (} x \texttt{ in } state \texttt{)} \; stmt : S \cup \{\Sigma\}} \quad (3) \qquad \Phi, \Sigma \vdash_s \texttt{\{ \}} : \{\Sigma\} \quad (4)$$

$$\frac{\Phi, \Sigma \vdash_b bexp : true \quad \Phi, \Sigma \vdash_s stmt_1 : S}{\Phi, \Sigma \vdash_s \texttt{if (} bexp \texttt{)} \; stmt_1 \; \texttt{else} \; stmt_2 : S} \quad (5) \qquad \frac{\Phi, \Sigma \vdash_b bexp : false \quad \Phi, \Sigma \vdash_s stmt_2 : S}{\Phi, \Sigma \vdash_s \texttt{if (} bexp \texttt{)} \; stmt_1 \; \texttt{else} \; stmt_2 : S} \quad (6)$$

$$\frac{\Phi, \Sigma \vdash_b bexp : \langle trueEnv, falseEnv\rangle \quad \Phi, trueEnv \vdash_s stmt_1 : S_1 \quad \Phi, falseEnv \vdash_s stmt_2 : S_2}{\Phi, \Sigma \vdash_s \texttt{if (} bexp \texttt{)} \; stmt_1 \; \texttt{else} \; stmt_2 : S_1 \cup S_2} \quad (7) \qquad \frac{\Phi, \Sigma \vdash_s stmt_1 : S_1 \quad S_2 = \cup\{S \mid \Sigma' \in S_1 \wedge \Phi, \Sigma' \vdash_s stmt_2 : S\}}{\Phi, \Sigma \vdash_s \{stmt_1 \texttt{ ; } stmt_2\} : S_2} \quad (8)$$

Process expressions

$$\frac{\Phi(\texttt{e.source}) = pd \quad (s, \Sigma') = \mathsf{remove}(pd, \Sigma)}{\Phi, \Sigma \vdash_e \texttt{e.source} : (pd, s, \Sigma')} \quad (9) \qquad \frac{\Phi(\texttt{e.target}) = pd \quad (s, \Sigma') = \mathsf{remove}(pd, \Sigma)}{\Phi, \Sigma \vdash_e \texttt{e.target} : (pd, s, \Sigma')} \quad (10)$$

$$\frac{\Phi(x) = pd \quad (s, \Sigma') = \mathsf{remove}(pd, \Sigma)}{\Phi, \Sigma \vdash_e x : (pd, s, \Sigma')} \quad (11) \qquad \frac{\mathsf{state\_info}(state) = \mathsf{process} \quad \Sigma(state) = \langle \{pd\}, may\rangle}{\Phi, \Sigma \vdash_e state : (pd, \{state\}, \Sigma[state \mapsto [\,]])} \quad (12)$$

$$\frac{\Sigma(\mathsf{s}) = \langle must, may\rangle \quad may \neq \emptyset \quad |must| \leq 1}{\Phi, \Sigma \vdash_e \texttt{select()} : (\sqcup may, \{\mathsf{s}\}, \Sigma[\mathsf{s} \mapsto \langle \emptyset, may\rangle])} \quad (13) \qquad \frac{\Sigma(\mathsf{s}) = \langle must, may\rangle \quad may \neq \emptyset \quad |must| > 1}{\Phi, \Sigma \vdash_e \texttt{select()} : (\sqcup may, \{\mathsf{s}\}, \mathsf{add}(\sqcup may, \mathsf{s}, \Sigma[\mathsf{s} \mapsto \langle \emptyset, \emptyset\rangle]))} \quad (14)$$

In the above two rules, s is the state designated as `select`

Boolean expressions

$$\frac{\Sigma(state) = [\,]}{\Phi, \Sigma \vdash_b \texttt{empty(}state\texttt{)} : true} \quad (15) \qquad \frac{\Sigma(state) = \langle \{pd_1, \ldots, pd_n\}, may\rangle \quad n \neq 0}{\Phi, \Sigma \vdash_b \texttt{empty(}state\texttt{)} : false} \quad (16)$$

$$\frac{\Sigma(state) = \langle \emptyset, may\rangle}{\Phi, \Sigma \vdash_b \texttt{empty(}state\texttt{)} : \langle \Sigma[state \mapsto [\,]], \mathsf{add}(\sqcup may, state, \Sigma)\rangle} \quad (17)$$

$$\frac{\Phi, \Sigma \vdash_e exp_1 : (pd_1, s_1, \Sigma_1) \quad \Phi, \Sigma \vdash_e exp_2 : (pd_2, s_2, \Sigma_2)}{\Phi, \Sigma \vdash_b exp_1 \texttt{ > } exp_2 : \langle \Sigma, \Sigma\rangle} \quad (18) \qquad \frac{\Phi, \Sigma \vdash_e exp : (pd, s, \Sigma') \quad \mathsf{unique}(pd) \quad pd \in \mathsf{must}(\Sigma(state))}{\Phi, \Sigma \vdash_b exp \texttt{ in } state : true} \quad (19) \qquad \frac{\Phi, \Sigma \vdash_e exp : (pd, s, \Sigma') \quad state \notin s}{\Phi, \Sigma \vdash_b exp \texttt{ in } state : false} \quad (20)$$

$$\frac{\Phi, \Sigma \vdash_e exp : (pd, s, \Sigma') \quad \forall pd' \in \mathsf{must}(\Sigma(state)).pd' \notin \alpha(pd)}{\Phi, \Sigma \vdash_b exp \texttt{ in } state : \langle \mathsf{add}(pd, state, \Sigma), \Sigma\rangle} \quad (21) \qquad \frac{\Phi, \Sigma \vdash_e exp : (pd, s, \Sigma')}{\Phi, \Sigma \vdash_b exp \texttt{ in } state : \langle \Sigma, \Sigma\rangle} \quad (22)$$

Rules (21) and (22) only apply if no preceding rule applies.

Auxiliary functions: unique, must, may, $\uplus$, α, add, and remove are defined in Section 3.2.

Fig. 5. Analysis rules for a core subset of Bossa

`empty(`*state*`)` and *exp* `in` *state* often enable useful information to be incorporated into these environments, which are used in the analysis of a conditional statement. The relationship between the analysis and the semantics is formalized in the appendix.

4 Analysis Example

The analysis must be precise enough to be able to both accept correct handlers and give useful feedback for incorrect handlers. To illustrate the behavior of the

Bossa analysis, we consider the following example:

```
1  On unblock.preemptive {
2    if (empty(running)) { }
3    else running => ready;
4    e.target => ready;
5  }
```

This handler is typical in its size, structure, and use of domain-specific constructs. Nevertheless, it is incorrect for the Linux OS because it does not take into account the possibility that the target process might not be blocked.

We consider the analysis of this handler with respect to a state environment in which the target process is initially in the `ready` state and no information is known about the set of processes in the other states. For conciseness, we only include the `running`, `ready`, and `blocked` states, as only these states are relevant to the handler. The initial state environment is thus represented as follows:

$$\{\texttt{running} \mapsto \langle \emptyset, \{(\texttt{x}, \texttt{running})\}\rangle, \texttt{ready} \mapsto \langle \{(\texttt{tgt}, \texttt{ready})\}, \{(\texttt{x}, \texttt{ready})\}\rangle,$$
$$\texttt{blocked} \mapsto \langle \emptyset, \{(\texttt{x}, \texttt{blocked})\}\rangle\}$$

In this state environment, the must information $\{(\texttt{tgt}, \texttt{ready})\}$ for `ready` indicates that the target process is initially in this state, and the must information $\emptyset$ for the other states indicates that nothing is known about their contents. In each case, the may information indicates that some process `x` may be in the given state and that any such process is initially in the given state itself. Figure 6 illustrates the steps in the analysis of the handler with respect to this state environment.

The analysis begins by analyzing the test expression, `empty(running)` (line 2), of the initial conditional. At this point the contents description associated with `running` is $\langle \emptyset, \{(\texttt{x}, \texttt{running})\}\rangle$, implying that this state is neither known to be empty, nor known to be non-empty. Rule (7) is used, which creates refined state environments, B and C in Figure 6, describing the cases where the state is empty and non-empty. These environments will be used in the analysis of the then and else branches of the conditional, respectively.

The analysis of the then branch, `{}` (line 2), is trivial, and simply returns the current state environment B. The analysis of the else branch, `running => ready` (line 3), uses the first state-transition rule, rule (1). This rule first analyzes the expression `running`, obtaining as the first component of the result the process description *pd* associated with the process in this state, and as the third component of the result a state environment Σ' describing the effect of removing this process from its current state. The result of analyzing the entire statement is then the state environment C obtained by adding the representation *pd* of the process to its new state `ready` in the state environment Σ'.

Because the value of the test expression `empty(running)` could not be determined, the result of analyzing the conditional (lines 2-3) is the union of the sets of the state environments resulting from the analysis of the branches, according to rule (7). The rule for sequence statements, rule (8), implies that the next statement of the handler, `e.target => ready` (line 4) is analyzed with respect

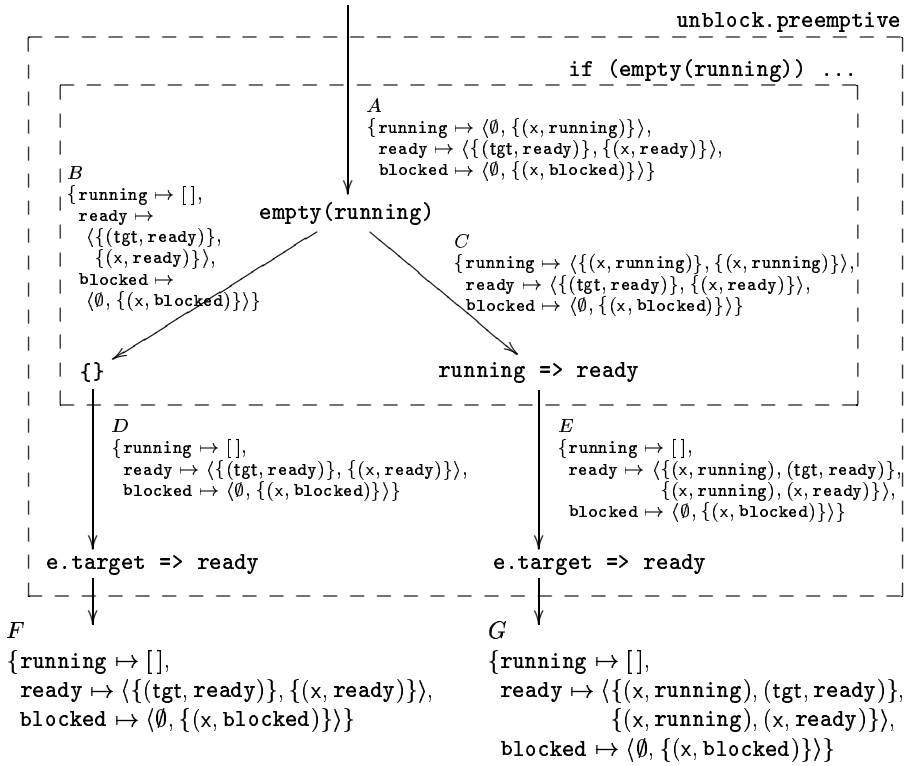

Fig. 6. Steps in the analysis of the `unblock.preemptive` handler

to each of these environments, D and E, individually. The analysis proceeds similarly to that of `running => ready`, and produces the two state environments, F and G, shown at the bottom of Figure 6.

The final step is to compare the two resulting state environments to the type of the event. As described in Section 2.3, the type of `unblock.preemptive` for Linux 2.4 is:

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

We first consider the output state environment F:

$$\{\texttt{running} \mapsto [], \texttt{ready} \mapsto \langle\{(\texttt{tgt}, \texttt{ready})\}, \{(\texttt{x}, \texttt{ready})\}\rangle, \\ \texttt{blocked} \mapsto \langle\emptyset, \{(\texttt{x}, \texttt{blocked})\}\rangle\}$$

To match this state environment against the type rules, the verifier must determine the starting and ending state of each process. The starting state of a

process is stored in the second component of the associated process description. Thus, in environment F, the starting state of the target process is `ready`. The ending state of a process is indicated by the state that is mapped to the contents description containing the given process. Thus, in environment F, the target process ends up in the `ready` state. Other state changes are indicated by the must and may information associated with each state. In environment F, the may information associated with each state contains only process descriptions originating in the state itself. Thus, we conclude that the handler does not change the state of any process. The state environment is thus compatible with the type rule `[tgt in READY] -> []`.

We next consider the output state environment G:

$$\{\texttt{running} \mapsto [], \texttt{ready} \mapsto \langle\{(\texttt{x}, \texttt{running}), (\texttt{tgt}, \texttt{ready})\}, \{(\texttt{x}, \texttt{running}), (\texttt{x}, \texttt{ready})\}\rangle,$$
$$\texttt{blocked} \mapsto \langle\emptyset, \{(\texttt{x}, \texttt{blocked})\}\rangle\}$$

This environment indicates that both the process initially in the `running` state and the process initially in the `ready` state end up in the `ready` state. This behavior is not described by any of the type rules, which require that if the running process is preempted, then the target process is initially in a state of the `BLOCKED` state class. The policy is thus rejected by the Bossa verifier.

The problem with the given handler implementation is that it preempts the running process regardless of the state of the target process, whereas the type only allows preemption when the target process is initially in a state of the `BLOCKED` state class. Indeed, a programmer might write such a handler on the assumption that the target process is always initially blocked, which, in the absence of detailed information about the kernel, is a natural assumption. The type of the event, however, clearly shows that the `RUNNING` and `READY` state classes must be taken into account. A correct version of the handler is:

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

Analysis of this handler yields state environments that satisfy the type of `unblock.preemptive`.

5 Related Work

Recently, there has been increasing interest in compile-time error detection in the context of OS code. CCured [17], Cyclone [12] and Splint [10] check C programs for common programming errors, such as invalid pointer references, relying to a varying degree on user annotations. These approaches provide little or no support for checking domain-specific properties. Meta-level Compilation [9] checks properties that can be described using an extended automata language, and has

been used to find many bugs in OS code. Nevertheless, this approach provides only bug finding; it does not guarantee satisfaction of the checked property. The SLAM model checker has been used to verify properties of device driver code [2]. The effectiveness of all of these strategies depends on the will of the programmer to follow anticipated (or inferred, in the case of Meta-level Compilation [8]) coding conventions. A DSL, on the other hand, restricts the programmer to a limited set of abstractions, thus making feasible more precise verifications.

The SPIN extensible operating system [4] addresses the need to check the safety of OS extensions by requiring that extensions be implemented using a variant of Modula-3, which is a type safe language. Nevertheless, this approach only suffices for properties that can be encoded using the general-purpose Modula-3 types. In particular, there is no framework for declaring and checking high-level domain-specific properties, as we have proposed here.

Numerous DSLs exploit the use of high-level domain-specific constructs to provide verification of domain-specific properties. We present a few examples. Devil is a language for implementing the hardware interaction code in a device driver. The Devil compiler verifies properties of the internal consistency of the Devil code; these checks have been shown to drastically reduce the chance of runtime errors [19]. Nevertheless, there is no verification of the relationship between a Devil specification and the target OS. Several approaches, including Teapot [6], Promela++ [3], and ESP [13], have combined DSLs with model checking to check algorithmic properties, such as liveness. Our focus is on easing the integration of DSL program with a target system rather than on algorithmic properties of the DSL program itself. The design of the Apostle language for parallel discrete event simulation recognizes the importance of connecting DSL constructs to specific type and effect information [5]. Again this information is used to check algorithmic properties.

The Bossa DSL is a stand-alone language, with its own syntax, semantics, parser, verifier, and compiler. An *embedded DSL*, on the other hand, is a controlled extension of an existing language, allowing reuse of host-language processing tools and enabling the mixing of the DSL with host language features [11]. Traditionally, an embedded DSL is simply a library, where constraints of the host language, such as type constraints, ensure that library constructs are used in a consistent, language-like manner. In this variant, the only verification that is possible is verification of properties that can be encoded in a way that they are checked by the host language. It seems awkward, if not impossible, to express high-level domain-specific properties in this setting. In another variant, an embedded DSL is implemented via macros that translate DSL constructs into the host language [20]. These macros can allow host language constructs to appear at specific places in a DSL program giving a controlled combination of the host language and the DSL. Our approach is directly applicable to this setting, as the macro expander can perform verification as well as translation. Nevertheless, if host language constructs are allowed, parts of the DSL program cannot be verified, which is not desirable in a domain such as process scheduling that has stringent safety requirements.

6 Conclusion

Extending an existing OS with new scheduling strategies is a difficult task, but one that is essential to providing good performance to many emerging applications. In this paper, we have presented the design of a DSL for scheduling and a type-based approach to expressing and checking OS requirements on a scheduling component. By making OS conventions explicit to both the verifier and the scheduling policy programmer, this approach reduces the expertise needed to extend an OS with new scheduling components, enabling programmers to address the needs of applications with specific scheduling requirements.

In future work, we plan to extend this approach to other OS services individually and in combination. An approach that includes multiple OS services could be useful in the context of embedded systems, where for example energy constraints require specific management strategies for multiple resources. Such cooperating extensions must take into account more complex interactions than extensions of a single functionality, making an approach for describing OS requirements even more important.

Acknowledgments We thank Mads Sig Ager, Olivier Danvy and Henning Korsholm Rohde for comments on this paper. This work was supported in part by a Microsoft Embedded research grant and by ACI CORSS. Anne-Françoise Le Meur was supported by a postdoc fellowship from the Danish Research Agency.

Availability A grammar of the complete Bossa language, the Bossa compiler and verifier, a run-time system for Linux, and numerous Bossa scheduling policy implementations are available at `http://www.emn.fr/x-info/bossa`.

References

1. R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an os kernel using temporal logic and aop. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE Computer Society Press.
2. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software, 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, Toronto, Canada, May 2001.
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. B. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–283, Copper Mountain Resort, CO, USA, Dec. 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

5. D. Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In *Proceedings of the ACM SIGPLAN Workshop on Domain Specific Languages*, pages 17–35, Paris, France, Jan. 1997.
6. S. Chandra, B. Richards, and J. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 237–248, 1996.
7. 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.
8. A. Engler, D. Yu, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.
9. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, California, Oct. 2000.
10. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.
11. P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, Victoria, Canada, June 1998.
12. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.
13. 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, USA, June 2001.
14. J. L. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–62, Saint-Emilion, France, Sept. 2002.
15. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
16. G. Muller, J. L. Lawall, L. P. Barreto, and J.-F. Susini. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. Technical report 03/2/INFO, Ecole des Mines de Nantes, 2003.
17. G. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, Jan. 2002.
18. G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Sept. 1981.
19. L. Réveillère and G. Muller. Improving driver robustness: an evaluation of the Devil approach. In *The International Conference on Dependable Systems and Networks*, pages 131–140, Göteborg, Sweden, July 2001. IEEE Computer Society.
20. O. Shivers. A universal scripting framework, or Lambda: the ultimate "little language.". In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism, Pro-*

gramming, Networking, and Security (ASIAN'96), volume 1179 of *Lecture Notes in Computer Science*, pages 254–265, Singapore, Dec. 1996.
21. D. 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.
22. C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11, Monterey, CA, USA, Nov. 1994.

Appendix

Definition 1 (Relation on process descriptions). *For any processes src and tgt, and any initial state environment σ_0, a process p is related to an abstract process description pd, iff $src, tgt, \sigma_0 \models_{\mathrm{p}} p : pd$, which is defined as follows:*

$$\begin{array}{ll} src, tgt, \sigma_0 \models_{\mathrm{p}} src : (\mathsf{src}, \sigma_0^{-1}(src)) & src, tgt, \sigma_0 \models_{\mathrm{p}} p : (\mathsf{x}, \sigma_0^{-1}(p)) \\ src, tgt, \sigma_0 \models_{\mathrm{p}} tgt : (\mathsf{tgt}, \sigma_0^{-1}(tgt)) & src, tgt, \sigma_0 \models_{\mathrm{p}} p : (\mathsf{x}, \top) \\ src, tgt, \sigma_0 \models_{\mathrm{p}} p : (\mathsf{x}, class), \text{if } \sigma_0^{-1}(p) \text{ is in state class } class & \end{array}$$

This relation extends pointwise to variable environments, as $src, tgt, \sigma_0 \models_{\mathrm{v}} \phi : \Phi$.

Definition 2 (Relation on contents descriptions). *For any processes src and tgt, and any initial state environment σ_0, a set of processes π is related to an abstract contents description cd, iff $src, tgt, \sigma_0 \models_{\mathrm{c}} \pi : cd$, defined as follows:*

$$src, tgt, \sigma_0 \models_{\mathrm{c}} \emptyset : [] \qquad \frac{\begin{array}{l} \exists (p_1, \ldots, p_n) \in \mathsf{choose}(\pi, n). \forall i. src, tgt, \sigma_0 \models_{\mathrm{p}} p_i : pd_i \\ \forall p \in \pi. \exists pd \in may. src, tgt, \sigma_0 \models_{\mathrm{p}} p : pd \end{array}}{src, tgt, \sigma_0 \models_{\mathrm{c}} \pi : \langle \{pd_1, \ldots, pd_n\}, may \rangle}$$

This relation extends pointwise to state environments, as $src, tgt, \sigma_0 \models_{\mathrm{s}} \sigma : \Sigma$.

The analysis and the semantics are then related as follows, proved by induction on the structure of the derivation:

Lemma 1 (Process expression). *If $\Phi, \Sigma \vdash_{\mathrm{e}} exp : (pd, s, \Sigma')$ and $src, tgt, \sigma_0 \models_{\mathrm{v}} \phi : \Phi$ and $src, tgt, \sigma_0 \models_{\mathrm{s}} \sigma : \Sigma$ and $\phi, \sigma \vdash exp \to p$, then $src, tgt, \sigma_0 \models_{\mathrm{p}} p : pd$, $\sigma^{-1}(p) \in s$, and $src, tgt, \sigma_0 \models_{\mathrm{s}} \sigma[\sigma^{-1}(p) \mapsto \sigma(\sigma^{-1}(p)) - \{p\}] : \Sigma'$.*

Lemma 2 (Boolean expression with a known value). *If $\Phi, \Sigma \vdash_{\mathrm{b}} bexp : b$ and $b \in \{true, false\}$ and $src, tgt, \sigma_0 \models_{\mathrm{v}} \phi : \Phi$ and $src, tgt, \sigma_0 \models_{\mathrm{s}} \sigma : \Sigma$ and $\phi, \sigma \vdash bexp \to v$, then $b = v$.*

Lemma 3 (Boolean expression with an unknown value). *If $\Phi, \Sigma \vdash_{\mathrm{b}} bexp : \langle \Sigma_t, \Sigma_f \rangle$ and $src, tgt, \sigma_0 \models_{\mathrm{v}} \phi : \Phi$ and $src, tgt, \sigma_0 \models_{\mathrm{s}} \sigma : \Sigma$ and $\phi, \sigma \vdash bexp \to b$, then if $b = true$ then $src, tgt, \sigma_0 \models_{\mathrm{s}} \sigma : \Sigma_t$ and if $b = false$ then $src, tgt, \sigma_0 \models_{\mathrm{s}} \sigma : \Sigma_f$.*

Theorem 1 (Soundness). *If $\Phi, \Sigma \vdash_{\mathrm{s}} stmt : S$ and $src, tgt, \sigma \models_{\mathrm{v}} \phi : \Phi$ and $src, tgt, \sigma \models_{\mathrm{s}} \sigma : \Sigma$ and $\phi, \sigma \vdash stmt \to \sigma'$, then for some $\Sigma' \in S$, $src, tgt, \sigma \models_{\mathrm{s}} \sigma' : \Sigma'$.*