Language Design for Implementing Process Scheduling Hierarchies

(Invited Paper)

Julia L. Lawall
DIKU, University of Copenhagen
2100 Copenhagen Ø,
Denmark
julia@diku.dk

Gilles Muller
Obasco Group, EMN/INRIA
44307 Nantes Cedex 3,
France
gilles.muller@emn.fr

Hervé Duchesne
Obasco Group, EMN/INRIA
44307 Nantes Cedex 3,
France
herve.duchesne@emn.fr

ABSTRACT

Standard operating systems provide only a single fixed scheduler, which does not meet all possible application scheduling needs. More flexibility can be achieved using a hierarchy of schedulers, allowing multiple schedulers to coexist in a single operating system (OS). Bossa is a framework for facilitating the implementation and deployment of OS process schedulers. In this paper, we describe the features of Bossa that enable the creation and management of a scheduling hierarchy. These features include a domain-specific language for implementing schedulers and a type system for describing requirements on scheduler behavior. The use of the domain-specific language eases scheduler development and enables scheduler verification. We have found that the approach allows programmers, even students who are not kernel or scheduling experts, to easily and safely implement and deploy schedulers that meet specific application needs.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.4.7 [**Operating Systems**]: Organization and Design; F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Verification

Keywords

Domain-Specific Languages, Process scheduling, Scheduling hierarchies, Operating System extension, Verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'04, August 24–26, 2004, Verona, Italy.

Copyright 2004 ACM 1-58113-835-0/04/0008 ...$5.00.

1. INTRODUCTION

An operating system (OS) process scheduler controls when and for how long each process is given access to the CPU. Standard OSes, such as Linux and Windows, provide a single fixed scheduler, which is intended to provide adequate service to most kinds of applications. Such a scheduler typically guarantees that every process that is able to run will eventually get access to the CPU, and controls the frequency and duration of this access using a priority. Nevertheless, this kind of scheduling strategy does not address all scheduling requirements. For example, it is not enough to know that a video player will eventually obtain access to the CPU; instead, the player must have access to the CPU within the precise intervals that correspond to its frame rate. As another example, a system administrator may want to restrict all of the processes generated by a multithreaded Web server to only a fixed percentage of the CPU, to prevent the processes generated by a flood of requests from overwhelming the system. Such a restriction cannot be expressed by a priority mechanism.

To meet such varied scheduling needs, multiple schedulers must be able to coexist in a single OS. One solution is to multiplex the CPU among a set of process schedulers, where the strategy for choosing among them is controlled by yet another scheduler. We refer to such a scheduler of schedulers as a *virtual scheduler*. To allow multiple strategies for choosing between schedulers, this approach can be generalized to the use of a hierarchy of schedulers [11]. Figure 1 illustrates a hierarchy in which a Fixed-Priority virtual scheduler gives a multimedia process scheduler priority over all other schedulers, and a Proportion virtual scheduler divides the remaining CPU time between a scheduler for Web-server processes and a scheduler for ordinary processes. This hierarchy both ensures that a video player managed by the scheduler for multimedia applications gets access to the CPU at regular intervals and constrains all of the processes generated by a Web server to execute within a fixed percentage of the CPU time.

Although the use of a hierarchy permits multiple schedulers to coexist, the problem remains of how to implement such a hierarchy. Modifying a standard OS to replace the existing scheduler by a hierarchy is a difficult task, requiring a deep knowledge of the internals of the OS and low-level programming in an environment that provides little debugging support. Frameworks have been proposed that provide

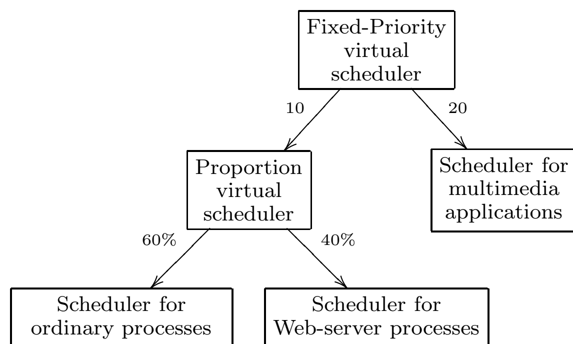

Figure 1: A scheduling hierarchy. The Fixed-Priority virtual scheduler gives priority 10 to the Proportion virtual scheduler and priority 20 to the scheduler for multimedia applications. The Proportion virtual scheduler gives its child schedulers 60% and 40% of the available CPU time, respectively.

an API for constructing a hierarchy, but do not formally describe what each API function should do or address the difficulty of programming at the OS kernel level [3, 4, 5, 10, 11].

Bossa is a framework for implementing and deploying both process schedulers and virtual schedulers in a standard OS that has been augmented with a Bossa run-time system [1, 2, 7]. This framework provides a domain-specific language (DSL) for implementing scheduling policies. The DSL guides and simplifies the implementation of a scheduling policy and permits verification that the decisions taken by the implementation are coherent with the behavior of the target OS. For example, the Bossa verifier ensures that a scheduler cannot elect a process that is blocked waiting for an I/O operation. Aspects of OS behavior relevant to this verification are factorized into a type system, making the Bossa framework OS-independent [8].

In this paper, we present the Bossa support for virtual schedulers. The contributions of this paper are as follows:

- We present an infrastructure for loading schedulers, constructing a scheduling hierarchy, and managing communication within a scheduling hierarchy.
- We describe language constructs provided by the Bossa DSL for specifying the interaction between a virtual scheduler and its child schedulers.
- We present an approach to deriving types for virtual schedulers from types that describe the process-level semantics of OS scheduling events.
- We briefly describe the process of verifying that a scheduler satisfies these types and present compiler optimizations that use type information. The verification process is described in more detail elsewhere [8].
- We show that little overhead is incurred for the use of a scheduling hierarchy.

The rest of this paper is organized as follows. Section 2 gives an overview of process scheduler behavior. Section 3 describes the Bossa infrastructure for constructing and interacting with a scheduling hierarchy. Section 4 presents the Bossa DSL, including the constructs particular to virtual schedulers. Section 5 gives an overview of the type system, and addresses the problem of inferring types for virtual schedulers. Section 6 describes the use of these types in verification and optimization of a virtual scheduler. Section 7 provides some benchmarks assessing the performance of virtual schedulers. Finally, Section 8 presents related work and Section 9 concludes.

2. SCHEDULER BEHAVIOR

The heart of a process scheduling policy is an algorithm for electing a process from the set of processes that are eligible to have access to the CPU. While the exact algorithm is specific to each policy, a feature that is common to all policies is the need to determine the set of eligible processes. The eligibility of a process depends at least on its ability to run based on its recent interaction with the OS. For example, when a process requests to read from a file, it is unable to run, and thus ineligible, until the disk makes the file data available to the OS. Subsequently, the interrupt indicating that the file data is available makes the process newly able to run. A scheduling policy can impose further criteria for eligibility. For example, a round-robin policy makes a process ineligible when it has used up its time slice.

The need to identify the set of eligible processes implies that a scheduler must be aware of the actions in the kernel that affect process eligibility and that it must record the effect of these actions for subsequent use in making scheduling decisions. In the Bossa framework, the OS kernel informs the scheduler of scheduling-related kernel actions via a set of event notifications [7]. Events indicate the need to elect a new process, as well as kernel actions that affect process eligibility, such as process creation, termination, blocking, and unblocking. A Bossa scheduler provides a set of handlers for these events. A scheduler also defines a set of *process states* that event handlers use to record the effect of events on process eligibility.

For a scheduling policy to function correctly, the process state transitions performed by the handlers must model the behavior associated with the corresponding OS event. To ensure safety, this property must be verified statically. Because not all OSes behave in the same way, Bossa provides a type system for expressing the set of state transitions allowed for each event. Types amount to pre- and post-conditions on event-handler behavior. Checking these types requires that the verifier can identify the set of states used by a given policy, determine the semantics of each state, and identify state transitions in the event handler code. These operations are difficult in scheduling code written using a general-purpose language, where states and state transitions can be implemented in arbitrary ways.[1] The Bossa DSL includes specific abstractions for declaring states and implementing state transitions, making it possible to statically identify the state transitions performed by a handler and to check that these satisfy the required types [2, 8].

This description of scheduler behavior pertains to process schedulers, which manipulate processes directly. In the rest of this paper, we show how to extend this behavior to virtual schedulers.

[1]For example, in Linux 2.4 the state of a process is indicated in part by the `state` field and in part by the `policy` field of the associated process structure.

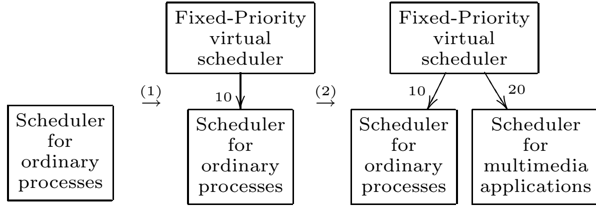

Figure 2: The construction of a scheduling hierarchy

```
int mount(char *parent_name, char *child_name,
          int child_property_count,
          int *child_properties);

int mount_root(char *new_root_name,
               int child_property_count,
               int *child_properties);

void unmount(char *scheduler_name);
```

Figure 3: The Bossa API for adding schedulers to and removing schedulers from a hierarchy.

3. HIERARCHY MANAGEMENT

Bossa allows applications to create, dismantle, and move around in a hierarchy, according to their scheduling needs. We first describe the infrastructure implementing these operations, and then describe how the OS interacts with the resulting hierarchy.

3.1 Application interaction with a hierarchy

A scheduling hierarchy has the form of a tree with a single virtual scheduler as the root (or a process scheduler as the root if it is the only scheduler in the hierarchy), virtual schedulers as the internal nodes, and process schedulers as the leaves. Figure 2 illustrates the construction of such a hierarchy. A hierarchy initially consists of a single scheduler, known as the *default scheduler*, that is statically linked with the kernel. Other schedulers can be either statically linked with the kernel or dynamically loaded as kernel modules. A user-level process can add a virtual scheduler at the root of the scheduling hierarchy, as shown by transition (1) in Figure 2, and a process scheduler or virtual scheduler as the child of an existing virtual scheduler, as shown by transition (2). In each case, attributes, such as the priorities shown in Figure 2, may need to be specified for the child scheduler. A scheduler can be removed from the hierarchy if it has no children, or if it is a virtual scheduler at the root of the hierarchy and has only one child scheduler. Figure 3 shows the API for carrying out these operations.

Every process is initially managed by the default scheduler or by the scheduler of its parent. A scheduler defines `attach` and `detach` interface functions to enable processes to change scheduler. A process changes scheduler by invoking the `attach` function of the new scheduler with the required process attributes. If the attachment succeeds, *i.e.*, if any admission criterion is satisfied, then the new scheduler invokes the `detach` function of the original scheduler, ensuring that every process is managed by exactly one scheduler at a time. For example, once a video player has created the hierarchy shown in Figure 2, it can move from the scheduler for ordinary processes to the scheduler for multimedia applications by invoking this scheduler's `attach` function with information about the player's frame rate and CPU requirements.

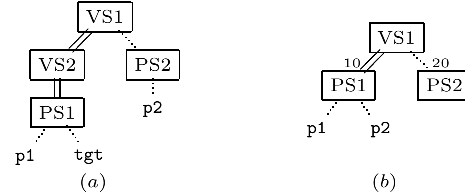

Figure 4: (a) An event notification indicating a change in the eligibility of a process is forwarded toward the process scheduler managing the affected process, `tgt`. (b) An event notification indicating the need to elect a new process is forwarded to the elected child scheduler. Double lines indicate event forwarding.

3.2 OS interaction with a hierarchy

In the Bossa framework, the OS interacts with the scheduler via a set of event notifications. To generalize this approach to a hierarchy of schedulers, we must consider how these event notifications should propagate through the hierarchy. The receipt of an event notification affects the eligibility of the processes managed directly or indirectly by the children of a virtual scheduler. We thus adapt the notion of process state to describe properties of these child schedulers.

Event notifications are received at the root of the hierarchy and forwarded along a path through the hierarchy toward a process scheduler among the leaves. Some cases are shown in Figure 4. If the event notification indicates a change in the eligibility of a process, such as process blocking or unblocking, then the event must be forwarded toward the scheduler managing the affected process, as illustrated in Figure 4a. Any scheduler that receives the event notification can also preempt the child scheduler that is directly or indirectly managing the running process, if any. Such preemption has the effect of propagating a series of preempt notifications toward the process scheduler directly managing this process. On the other hand, if the event notification indicates the need to elect a new process, then each scheduler that receives the event notification elects one of its child schedulers and forwards the event to this scheduler, until the event notification reaches a process scheduler. This process scheduler then elects a new process to have access to the CPU. Figure 4b shows the case of a Fixed-Priority virtual scheduler (VS1) that forwards such an event notification to the highest-priority child scheduler that is managing any eligible processes.

States are used in a virtual scheduler to record the existence of eligible processes among the processes managed directly or indirectly by each child scheduler. The result of forwarding an event notification is the new *scheduler state* of the child scheduler, which indicates whether this scheduler is directly or indirectly managing the running process, or any processes that are ready to run. The virtual scheduler uses this information to update its view of the state of the child scheduler.

```
states = {
  RUNNING running : process;
  READY   ready   : sorted queue;
  READY   expired : queue;
  READY   yield   : process;
  BLOCKED blocked : queue;
  TERMINATED terminated;
}

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

Figure 5: Extracts of the implementation of the Linux process scheduler

```
states = {
  RUNNING running : scheduler;
  READY   ready   : public sorted queue;
  READY   yield   : scheduler;
  BLOCKED blocked : queue;
}

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

Figure 6: Extracts of the implementation of the Fixed-Priority virtual scheduler

4. THE BOSSA DSL

In the Bossa framework, process schedulers and virtual schedulers are implemented using a DSL. This DSL defines a scheduling policy in terms of a collection of declarations, event handlers, and interface functions. Declarations describe global variables, process attributes, process states, and the relative ordering of processes. Event handlers describe the interaction with the OS. Interface functions describe the interaction with user-level processes. We focus on the states and event handlers, which determine the correctness of the interaction between the scheduling policy and the target OS. As examples, we use extracts of Bossa implementations of the Linux process scheduler and the Fixed-Priority virtual scheduler for use with a Bossa-enabled version of the Linux kernel.

4.1 States

A process scheduler records the eligibility of processes using a set of process states. As shown in Figure 5, the states defined by the Linux scheduling policy are `running`, `ready`, `expired`, `yield`, `blocked`, and `terminated`. Each state is annotated with a *state class*, indicating the eligibility of processes in that state. The state classes are fixed by the Bossa DSL and there must be at least one state in each state class. The `RUNNING` state class indicates that a process in the given state is running.[2] The `READY` state class indicates that the processes in the given states are able to run. The `BLOCKED` state class indicates that the processes in the given states are not able to run. The `TERMINATED` state class indicates that the processes in the given states are terminating. State classes describe the semantics of the associated states, and are used in the verification process described in Section 6. Each state declaration also includes information about the data structure used to store the set of processes in the state.

A virtual scheduler is concerned not with the eligibility of individual processes, but with the ability of each child scheduler to elect a process. As shown in Figure 6, the states defined by the Fixed-Priority policy are `running`, `ready`, `yield`, and `blocked`. State classes are used to describe the eligibility of the processes managed by the schedulers in these states. The state class `RUNNING` means that the scheduler in the associated state is managing the running process. The state class `READY` means that the schedulers in the associated states are not managing the running process, but are managing some processes that are able to run. The state class `BLOCKED` means that the schedulers in the associated states are not managing the running process or any process that is able to run. The remaining annotations associated with a state declaration describe the implementation of the state, except for the keyword `public`, which is described below.

[2]Bossa is currently targeted toward monoprocessors. Thus, a policy must declare exactly one state in the `RUNNING` state class and there may be at most one process in this state.

4.2 Event handlers

The event notifications of the kernel either inform the scheduler of changes in the eligibility of processes or request that the scheduler initiate a change in the eligibility of a process by electing a new process. To illustrate how a Bossa scheduler reacts to changes in process eligibility, we consider the `unblock.preemptive` event, which occurs when a process is unblocking and when the scheduler may, if desired, preempt the running process.

In a process scheduler, the handlers manipulate processes directly. A handler may check the state of a process using the construct *exp* `in` *state*, change the state of a process to *state* using the construct *exp* `=>` *state*, and check whether there are no processes in a given state using the construct `empty(`*state*`)`. The use of these constructs is illustrated by the `unblock.preemptive` handler of the Linux scheduling policy shown in Figure 5. This handler checks whether the unblocking process, referred to as the *target process* and designated as `e.target`, is in the `blocked` state. If so, it changes the state of the unblocking process to `ready`. If there is a running process and this process has a lower priority than the unblocking process, then the running process is preempted by changing its state to `ready` as well. If the unblocking process is not currently blocked, which can occur due to particularities of the Linux kernel, there is no further action.

In a virtual scheduler, the handler of the `unblock.preemptive` event must forward the event notification to the process scheduler managing the unblocking process. Event forwarding is implemented using the following constructs:

`next(`*exp*`)`   *exp* `=> forwardImmediate()`

The expression `next(`*exp*`)` returns the child scheduler that is

directly or indirectly managing the process *exp*. The statement *exp* `=> forwardImmediate()` is a variant of the state transition operation *exp* `=>` *state* that forwards the event to the child scheduler designated by *exp*. As a result of the event forwarding, the child scheduler returns its scheduler state. If this scheduler state does not correspond to the state class of the child scheduler's current state, then the forwarding operation ends by selecting a new state for the child scheduler. If there is only one state in the state class corresponding to the child's scheduler state, then that state is used. Otherwise, the state designated as `public` is used. The handler can subsequently change the state to another state in the same state class, if desired.

The `unblock.preemptive` handler of the Fixed-Priority scheduling policy forwards the event notification to the child scheduler managing the unblocking process, and possibly preempts the running process. This behavior is analogous to that of the Linux scheduling policy, except that here the scheduler manipulates the schedulers managing the unblocking and running processes, rather than manipulating these processes directly. The handler begins by checking whether the scheduler managing the unblocking process is currently in the `blocked` state, indicating that it is not managing any processes that are able to run. In this case, forwarding the unblock event will cause the unblocking process to be newly considered to be able to run, and thus the scheduler will move from the `blocked` state to the `ready` state, making it newly eligible for election. The policy then considers whether the affected scheduler has higher priority than the child scheduler managing the running process, if any, and if so preempts that scheduler by changing its state to `ready`. The remainder of the handler simply forwards the event to the child scheduler managing the unblocking process. This forwarding is done regardless of the state of the unblocking process, which is not known to the virtual scheduler. When the event notification reaches the process scheduler managing the unblocking process, that scheduler will use its record of the state of the process to determine the appropriate state transition.

5. CONSTRAINTS ON SCHEDULER BEHAVIOR

A Bossa scheduler must provide handlers for the set of event notifications generated by the target OS kernel and each such handler must implement state transitions that correspond to the semantics of the associated event. The set of events and the corresponding allowed process state transitions are specified by an expert in the target OS as a collection of *event types*. We first describe these process-level types and then present an algorithm to generate types for virtual schedulers from this information. The algorithm is presented informally using an example. More information is available in the appendix. Event types are used for verification and optimization, as described in Section 6.

5.1 Types for process schedulers

The event types provided by the OS expert describe the required effect of the scheduling hierarchy on process states. These types thus describe the required behavior of process schedulers. To be policy independent, types are described in terms of state classes rather than states. The Bossa compiler then instantiates these types in terms of the states defined by a given policy for use in verification and optimization. In addition to the types, the OS expert describes possible sequences of events and the set of events that can occur during interrupts. This information is used to identify the possible execution paths through the policy and to restrict the analysis to instantiations that can occur within these execution paths.

The type for the `unblock.preemptive` event for the Linux OS is shown below:

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

This type consists of four rules describing the state transitions allowed for the unblocking process, referred to as `tgt` (target), and in one case for a process `p` in the `RUNNING` state class. In each case, no other state transitions are allowed. In the first rule, the state of the unblocking process is specified to change from a state in the `BLOCKED` state class to a state in the `READY` state class, indicating that the process is newly able to run. The second rule describes preemption; the transition of the unblocking process is the same as in the previous rule, but the process in the state in the `RUNNING` state class is moved to a state in the `READY` state class. The remaining rules indicate that when the target process is not actually blocked, no state transition is allowed.

5.2 Types for virtual schedulers

The types for process schedulers presented above describe the allowed effect of each event handler on the states of the child processes. Analogously, the types for virtual schedulers describe the allowed effect of each event handler on the states of the child schedulers. When an event notification is propagated down the hierarchy, it eventually reaches a process scheduler, and thus the types for virtual schedulers must be consistent with the types for process schedulers.

The event types for virtual schedulers are inferred from the process-level event types provided by the OS expert. To illustrate some of the issues involved, we consider the following process-level type rule for the `unblock.preemptive` event:

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

The inference of types for virtual schedulers must account for the fact that the processes mentioned in a type rule may be managed by one or more of the child schedulers of the virtual scheduler. Some possibilities are illustrated in Figure 7, where in the first case the processes `p` and `tgt` are both managed by the same process scheduler, while in the other cases they are managed by different process schedulers. Furthermore, the inference must determine the effect of the state transitions ultimately carried out at the process level on the scheduler states of these child schedulers. For example, in the case where `p` and `tgt` are managed by different process schedulers the effect of the event on the child scheduler managing `tgt` may change its scheduler state from `BLOCKED` to `READY`, if the child scheduler is initially only managing blocked processes, but may also cause the scheduler state to remain `READY`, if the scheduler managing `tgt` is initially also managing some processes in the `READY` state class. These scenarios are illustrated by the second and third examples in Figure 7.

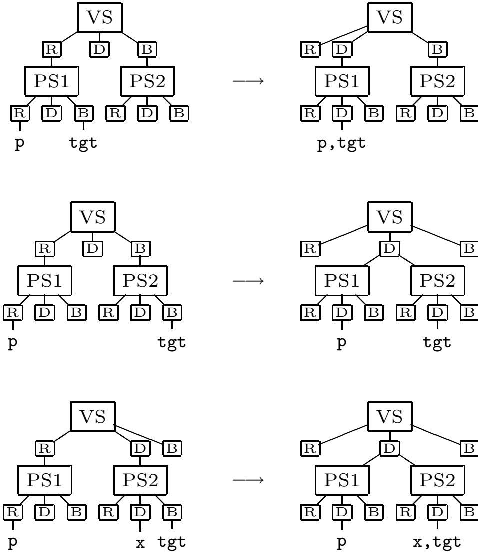

Figure 7: The effect of process state transitions associated with the `unblock.preemptive` event on the states associated with the child schedulers of a virtual scheduler. `R` represents the states of the `RUNNING` state class, `D` represents the states of the `READY` state class, `B` represents the states of the `BLOCKED` state class.

The key observation in the inference of virtual scheduler event types is that a virtual scheduler can only forward an event to the child scheduler managing the target process or cause a `preempt` event notification to be forwarded to the child scheduler managing the running process. Thus, for a given process-scheduler type rule, any process that changes state must be managed by the child scheduler managing the target process, unless it is the running process and is preempted, in which case it can be managed by either the same scheduler or a different one. Based on these observations, the inference algorithm begins by creating a set of possible configurations, each reflecting a different distribution of the processes mentioned by a given type rule among the child schedulers according to these constraints. As an example, the configurations generated from the above type rule are as follows:

```
⟨[tgt in BLOCKED], [p in RUNNING], ∅⟩
⟨[p in RUNNING, tgt in BLOCKED], −, ∅⟩
```

In each configuration, the first component describes the processes managed by the scheduler managing the target process, the second component describes the processes managed by the scheduler that is preempted, if any, and the third component describes any other processes mentioned by the rule (in our example, there are none):

The next step is to determine the state of each child scheduler, before and after the event notification. The state of the scheduler depends not only on the states of the processes mentioned in the type rule, but also on the states of any other processes that the scheduler might be managing. From each configuration, the inference algorithm creates a set of instantiations reflecting the possibility that each state class that is not constrained by the type rule may or may not be empty. A few of the instantiations resulting from the first of the above configurations are as follows:

```
⟨[[] = RUNNING, s1 in READY, tgt in BLOCKED],
 [p in RUNNING, s2 in READY, s3 in BLOCKED], ∅⟩
⟨[[] = RUNNING, [] = READY, tgt in BLOCKED],
 [p in RUNNING, s2 in READY, s3 in BLOCKED], ∅⟩
```

The instantiations represent process states on event notification. To determine the corresponding output configurations, in each case, the first component is matched against the complete set of type rules for the event itself, and the second component, if any, is similarly matched against the type rule for the `preempt` event. From the first instantiation above and the type for `unblock.preemptive` given in Section 5.1 we obtain the following:

```
⟨[[] = RUNNING, s1 in READY, tgt in BLOCKED]
 → [[] = RUNNING, [tgt,s1] in READY],
 [p in RUNNING, s2 in READY, s3 in BLOCKED]
 → [[] = RUNNING, [p,s2] in READY, s3 in BLOCKED], ∅⟩
```

For each of the pairs of input and output configurations, the output configuration is again instantiated to reflect the possibility that each state class about which there is no information may or may not be empty. The following instantiated tuples result from the tuple above, reflecting the fact that the output configuration of the *target* component of this tuple puts no constraint on the `BLOCKED` state class:

```
⟨[[] = RUNNING, s1 in READY, tgt in BLOCKED]
 → [[] = RUNNING, [tgt,s1] in READY], [] = BLOCKED,
 [p in RUNNING, s2 in READY, s3 in BLOCKED]
 → [[] = RUNNING, [p,s2] in READY, s3 in BLOCKED], ∅⟩
⟨[[] = RUNNING, s1 in READY, tgt in BLOCKED]
 → [[] = RUNNING, [tgt,s1] in READY], s4 in BLOCKED,
 [p in RUNNING, s2 in READY, s3 in BLOCKED]
 → [[] = RUNNING, [p,s2] in READY, s3 in BLOCKED], ∅⟩
```

A scheduler state is then computed for each input and corresponding output configuration, giving the effect on a single child scheduler. The information about the various schedulers is then combined to construct virtual-scheduler event-type rules. For example, from both of the above tuples, we obtain the rule:

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

The processes named `s`i are not mentioned in the resulting rule, as they share the same schedulers as the `p` and `tgt` processes.

The complete set of virtual scheduler types inferred from the process scheduler type rules for `unblock.preemptive` are as follows:

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

All of the inferred type rules enforce the basic effect of unblocking, as indicated by the fact that the child scheduler managing the target process is not in a state of the `BLOCKED` state class at the end of the event handler.

6. VERIFICATION AND OPTIMIZATION

The Bossa compiler first verifies statically that a policy satisfies the event types, and thus the requirements of the target OS, by analyzing the state transitions performed by the possible execution paths through the handlers. During the verification process, information is collected about the possible states of the values of various process-typed expressions. The compiler then uses this information to perform some optimizations during code generation.

6.1 Verification

The purpose of the Bossa verification is to ensure that for each mapping of processes to states that can occur at run time, the effect of each event handler is compatible with the expectations of the target OS. The verification process is oriented toward precision rather than efficiency, as properties of the domain of scheduling and restrictions in the Bossa language, such as the absence of recursion, imply that event handlers have a simple structure. Accordingly, several kinds of information are stored in the representations of processes and states, and the analysis avoids introducing approximations, when possible. We consider a simplified version of the language in which the only values are booleans and processes and there are no assignments. This language and the verification process have been described formally elsewhere for process schedulers [8]. Virtual schedulers do not introduce any new issues. We thus give only an overview and an example here.

The verifier relies on an analysis that proceeds by abstract interpretation of the handler code. Abstract values are used to represent the individual processes and the sets of processes in each state. The abstract representation of a process consists of information about the process name, such as `tgt` for the target process, and the starting state of the process, when this information is known. The abstract representation (*contents description*) associated with a state is $[\,]$, if it is known that there are no processes in the state, or a pair $\langle must, may\rangle$, otherwise. In the latter case, *must* is a set of abstract processes describing the set of processes that are known to be in the given state and *may* is a set of abstract processes describing the set of processes that may be in the given state. This information allows the analysis to determine the current state of specific processes, such as the target process, and the starting and ending states of arbitrary processes that change state in the course of the handler.

We present the analysis rules for only two constructs, the boolean expression `empty(`*state*`)` and the statement `if` *exp* `then` $stmt_1$ `else` $stmt_2$. These constructs are sufficient to illustrate the main points of the analysis. The expression `empty(`*state*`)` is true if and only if there does not exist any process in the state *state*. The analysis rules are as follows:

$$\frac{\Sigma(state) = [\,]}{\Phi, \Sigma \vdash_{\rm b} \texttt{empty}(state) : true} \quad (1)$$

$$\frac{\Sigma(state) = \langle\{pd_1, \ldots, pd_n\}, may\rangle \quad n > 0}{\Phi, \Sigma \vdash_{\rm b} \texttt{empty}(state) : false} \quad (2)$$

$$\frac{\Sigma(state) = \langle\emptyset, may\rangle}{\Phi, \Sigma \vdash_{\rm b} \texttt{empty}(state) : \langle\Sigma[state \mapsto [\,]], \mathsf{add}(\sqcup may, state, \Sigma)\rangle} \quad (3)$$

The expression is analyzed with respect to a *variable environment* Φ mapping each variable to an abstract process describing its value, and a *state environment* Σ, mapping each state to a contents description. The result of the analysis is a boolean value, if one can be determined, and otherwise it is a pair of state environments, representing the current state environment extended with any information that can be inferred from the truth or falsity of the test expression, respectively. If the contents description associated with *state* is $[\,]$, then the result of `empty(`*state*`)` is known to be true (rule (1)). Similarly, if the set of processes that must be in *state* is known to be non-empty, then the result of the test is known to be false (rule (2)). If neither of these conditions holds, then the value of the test is unknown (rule (3)). The result is one state environment in which $[\,]$ is associated with *state* and another state environment in which the information associated with the *state* is updated such that its "must" set contains an upper bound of its current "may" information. This approach is analogous to the use of positive and negative information in partial evaluation [6].

The analysis of a conditional statement uses the following rules:

$$\frac{\Phi, \Sigma \vdash_{\rm b} bexp : true \quad \Phi, \Sigma \vdash_{\rm s} stmt_1 : S}{\Phi, \Sigma \vdash_{\rm s} \texttt{if (} bexp \texttt{) } stmt_1 \texttt{ else } stmt_2 : S} \quad (4)$$

$$\frac{\Phi, \Sigma \vdash_{\rm b} bexp : false \quad \Phi, \Sigma \vdash_{\rm s} stmt_2 : S}{\Phi, \Sigma \vdash_{\rm s} \texttt{if (} bexp \texttt{) } stmt_1 \texttt{ else } stmt_2 : S} \quad (5)$$

$$\frac{\begin{array}{c}\Phi, \Sigma \vdash_{\rm b} bexp : \langle true_env, false_env\rangle \\ \Phi, true_env \vdash_{\rm s} stmt_1 : S_1 \quad \Phi, false_env \vdash_{\rm s} stmt_2 : S_2\end{array}}{\Phi, \Sigma \vdash_{\rm s} \texttt{if (} bexp \texttt{) } stmt_1 \texttt{ else } stmt_2 : S_1 \cup S_2} \quad (6)$$

These rules exploit the precise information produced by the analysis of a boolean expression such as `empty(`*state*`)`. If the result of the analysis of the boolean expression is a boolean value, only the corresponding branch of the conditional is subsequently analyzed (rules (4) and (5)). On the other hand, if the result of the analysis of the boolean expression is a pair of state environments, then the component of the pair representing the case where the test is true is used in the analysis of the "then" branch, and the component of the pair representing the case where the test is false is used in the analysis of the "else" branch (rule (6)). The result of the analysis of each branch is a set of state environments. When both branches are analyzed, the result of the analysis of the conditional statement is the union of the two sets of resulting state environments. The rest of the handler is then analyzed with respect to each of these environments individually. The analysis thus follows a "meet over all paths" strategy [9].

As an example, we consider the following simplified version of the `unblock.preemptive` handler of the Fixed-Priority policy:

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

analyzed with respect to an empty variable environment Φ and the following state environment Σ:

$$\begin{array}{l}\{\texttt{running} \mapsto \langle\{(\mathsf{x}, \texttt{running})\}, \{(\mathsf{x}, \texttt{running})\}\rangle, \\ \;\texttt{ready} \mapsto \langle\emptyset, \{(\mathsf{x}, \texttt{ready})\}\rangle, \\ \;\texttt{blocked} \mapsto \langle\{(\texttt{tgt}, \texttt{blocked})\}, \{(\mathsf{x}, \texttt{blocked})\}\rangle\}\end{array}$$

For conciseness, this state environment only includes the states `running`, `ready`, and `blocked`, as these are the only

states that are relevant to the example. The must information in this state environment indicates that it is known that there is a child scheduler in the `running` state, that it is unknown whether there are any child schedulers in the `ready` state, and that the target process is known to be managed by a child scheduler in the `blocked` state.

The analysis begins with the test expression `next(e.target) in blocked`. Because the must information of the `blocked` state indicates that the child scheduler managing the target process is indeed in this state, the analysis of this expression returns *true*, and thus, by rule (4), the analysis of the enclosing conditional statement considers only the "then" branch. The analysis next considers the test expression `empty(running)`. By rule (2), the result of the analysis is *false*, because the must information of the `running` state is non-empty. Thus, by rule (5) for conditional statements, only the "else" branch is analyzed. This branch changes the state of the scheduler in the `running` state to `ready`, producing the following set of state environments as the result of analyzing the inner conditional statement:

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

The handler next forwards the event to the child scheduler managing the target process. To determine the resulting state of this child scheduler, the analysis identifies type rules for the given event that are compatible with the current state environment. According to the set of type rules for `unblock.preemptive` inferred in Section 5.2, the scheduler managing the target process ends up in a state of the `READY` state class. The Fixed-Priority policy defines two such states, `ready` and `yield`, and this scheduler is not currently in either of them. Thus, the `public` state is chosen, which is `ready`. The result of updating the set of state environments according to this state transition is thus:

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

which is the result of analyzing the handler. This combination of initial state environment and final state environment is compatible with the type rule:

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

inferred in Section 5.2. Analysis with respect to other state environments compatible with the input configurations of the type rules proceeds similarly.

6.2 Optimization

The main optimizations performed by the Bossa compiler are in the implementation of the state transition operations and the specialization of sequences of events. These optimizations are driven by the state information collected during the analysis. The optimizations apply to both process schedulers and virtual schedulers.

A state transition operation *exp* `=>` *state* removes the given process or scheduler from the data structure associated with its current state and adds it to the data structure associated with its new state. As shown in the examples of the Linux policy (Figure 5) and the Fixed-Priority policy (Figure 6), a state can be implemented as either a queue or a process/scheduler variable (an array is also possible), and thus the exact implementation of a state transition operation depends on the kinds of data structures associated with the states involved. The Bossa syntax makes the destination state explicit, but the source state is determined from the current state of the affected process or scheduler, *exp*. For each state-transition operation, the analysis collects the set of possible source states, and specialized code is generated accordingly.

Compilation of the event-forwarding state-transition operation *exp* `=> forwardImmediate()` used in virtual schedulers is more complex. Unlike *exp* `=>` *state*, this statement does not indicate the destination state. Indeed, the destination state depends on the scheduler state returned by the child scheduler at run time. Nevertheless, the types can be used to restrict the set of states that must be taken into account. For example, when forwarding of the event occurs in an `unblock.preemptive` handler, the types inferred in Section 5.2 indicate that the generated code need only take into account the possibility that the resulting scheduler state is `RUNNING` or `READY`, but not `BLOCKED`. This optimization reduces the number of tests performed by the virtual scheduler.

A Bossa policy specifies a disjoint set of scheduling event handlers, but in the kernel, some sequences of events can occur atomically. For example, when a process blocks, another process is elected immediately. The Bossa compiler creates specialized handlers based on information in the event types about such atomic sequences. The sequence of handlers is analyzed as a single unit, implying that information inferred during the analysis of one handler is propagated to the analysis of subsequent ones in the sequence. This information may imply that some tests on process states are always true or always false. The analysis keeps track of such test expressions, which are removed during code generation, thus creating a handler that is specialized to the given event sequence. This approach creates an efficient implementation, without requiring the programmer to duplicate code, create optimized instances, or even be aware of the relevant kernel properties.

7. EVALUATION

We first present a micro-benchmark that measures the overhead introduced by the use of a scheduling hierarchy, and then assess the performance of a scheduling hierarchy in the context of a video player and a Web server.

Micro-benchmark. A scheduling hierarchy propagates event notifications from the root of the hierarchy to the process scheduler managing the target process. To assess the cost of these layers of communication, we consider hierarchies having the form of a sequence of virtual schedulers with a single process scheduler as the leaf. Each virtual scheduler is a null scheduler that simply forwards every event to its only child. As a test application, we use the `lat_ctx` benchmark,[3] which passes a token around a ring of processes, each of which traverses an array of some fixed size. Because this application performs so little computation between context switches, it represents a worst case for scheduler performance. Figure 8 shows the overhead introduced in this benchmark for between 1 and 3 virtual schedulers,

[3] `http://www.bitmover.com/lmbench/`

	Min	Max
Linux: Player only	0.005	0.048
EDF: Player only	0.019	0.024
Linux: Player, make	0.009	22.683
EDF: Player, make	0.017	0.018

Table 1: Mplayer: distance between video and audio.

which is typically the limit of what is needed in practice. The cost per added virtual scheduler increases roughly linearly. The Fixed-Priority and Proportion virtual schedulers used in our example hierarchy (Figure 1) are only slightly more complex than the null scheduler considered here, and have similar performance.

MPEG video display with reservations. On a lightly loaded system, a video player can achieve the frame rate required by the video by sleeping for an appropriate interval after processing each frame. On a heavily loaded system, however, the player needs to reserve a portion of CPU time within each interval, to ensure not only that it receives adequate access to the CPU but also that it receives this access at the appropriate rate.

We consider the use of the video player `mplayer`[4] with a scheduling hierarchy consisting of a Fixed-Priority scheduler at the root, and the Linux scheduler and an Earliest Deadline First (EDF) reservation-based scheduler at the leaves. The Linux scheduler has lower priority than the EDF scheduler. Only the video player runs on the EDF scheduler. We slightly modified `mplayer` to construct the hierarchy, attach itself to the EDF scheduler, and yield at the end of processing each frame. Table 1 shows the performance of the player using Bossa on the *Matrix Reloaded* trailer with and without reservations when competing with Linux kernel compilation. The performance is measured as the difference in the percentage of the complete audio and video that has been treated so far. Under the Linux scheduling policy, the video falls far behind the audio in the presence of kernel compilation. With EDF, the player maintains good performance.

Web server isolation. The use of a scheduling hierarchy allows running multiple web servers in isolation, so that if one is under attack or flooded, the others still react with guaranteed performance. We have implemented this approach using a Proportion virtual scheduler controlling two Linux schedulers each running a different Apache server. One is allocated 75% of the CPU, while the other is allocated the remaining 25%. As shown in Table 2, when flooding one server, the proportion of pages served by each of the two servers nearly matches its CPU allocation. When flooding both servers simultaneously by two different clients, the 25% server serves a little bit more than its allocation, resulting in 107% of the pages served by a single server.

	Request rate (req/s)	% of Apache alone
Apache alone	975	100%
One server flooded Apache 75%/25%	732/253	75%/25%
Both servers flooded Apache 75%/25%	722/321	75%/32%

Table 2: Apache performance (two servers).

8. RELATED WORK

Several scheduling frameworks allow more than one scheduler to coexist, with the choice between them made using a built-in fixed-priority strategy. Vassal allows a user-defined scheduler to be loaded into Windows, where it is given priority over the standard Windows scheduler [3]. The S.Ha.R.K. kernel has no fixed scheduler and instead allows the loading of scheduling modules [5]. Neither framework addresses the need to specify other policies for choosing between multiple schedulers. Furthermore, neither approach provides for verification of scheduler behavior.

The work closest to ours is the HLS framework for developing scheduling hierarchies [10]. HLS provides an API for implementing new schedulers, analogous to Bossa's set of event notifications, but does not provide a DSL or any verification of scheduler behavior. Thus, programming a scheduler remains error-prone and requires a deep knowledge of the behavior of the target OS. In addition to the problem of implementing a hierarchy, HLS considers the problem of reasoning about a composition of schedulers [11]. Based on declarations describing the behavior of individual scheduling policies, HLS infers guarantees about the scheduling behavior of a hierarchy. We are considering how to incorporate this reasoning system into Bossa.

CPU inheritance scheduling is an approach in which ordinary threads act as schedulers by donating their CPU time to other processes [4]. This approach is very flexible, but again there is no verification of scheduler behavior.

9. CONCLUSION

In this paper, we have presented aspects of the Bossa scheduling framework that allow the construction and manipulation of a scheduling hierarchy. Key features of this approach are the use of a DSL for the programming of schedulers and the use of a type system for specifying requirements on scheduler behavior. The DSL simplifies programming, and more crucially allows static verification that a scheduler satisfies the required types.

The specific focus of this paper is on generalizing ideas that were developed in the context of individual process schedulers to the context of a hierarchy of schedulers. We conjecture that the approach, particularly the work on the type system, can be used in other contexts to generalize a class of policies that are usually presented in a monolithic way to a hierarchical structure. Currently, we are considering the problem of managing multiple OS resources with the goal of allowing user-level management of energy usage. We will consider how the approach developed here can be applied to this setting.

Acknowledgments

We thank the PEPM 2004 organizers for the invitation to present this work, and John Hatcliff, Anne-Françoise Le Meur, and Mads Sig Ager for comments on a draft of this paper.

[4] `http://www.mplayerhq.hu`

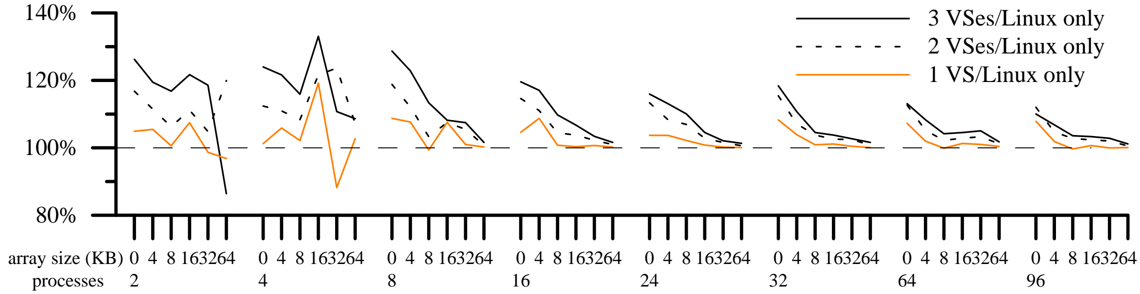

Figure 8: Virtual scheduler overhead in `lat_ctx`.

Availability

Bossa and all material described in this paper are available at the Bossa web site: `http://www.emn.fr/x-info/bossa/`

10. 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.

[2] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, Mar. 2002.

[3] G. M. Candea and M. B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, Aug. 1998.

[4] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 91–105, Seattle, WA, Oct. 1996.

[5] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[6] R. Glück and A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis. Proceedings*, volume 724, pages 112–123. Springer-Verlag, 1993.

[7] J. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *ACM SIGOPS European Workshop 2002 (EW'2002)*, pages 54–61, Saint-Emilion, France, Sept. 2002.

[8] J. L. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, Oct. 2004. To appear.

[9] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[10] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[11] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, Dec. 2001.

APPENDIX

A. INFERRING VIRTUAL-SCHEDULER EVENT TYPES

The goal of the inference algorithm is to derive event types for virtual schedulers from the process-level event types provided by the OS expert. A type consists of a collection of type rules $i \rightarrow o$, defined as follows:

Input configuration:	i	::= `[`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:	o	::= `[`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`

Among the input components, `[`p_1`,...,`p_n`] in` c_{in} indicates that the processes represented by p_1, ..., p_n are known to be in some states of the state class c_{in} and, `[] =` c_{in} means that there is no process in any state of the state class c_{in}. Among the output components, `[`p_1`,...,`p_n`] in` c_{out} means that the processes represented by p_1, ..., p_n are required to be in states of the state class c_{out}, `[`p_1`,...,`p_n`] in` c_{out}`!` means the same but that additionally other processes are allowed to change state within the state class c_{out}, and c_{out}`!` means that processes are only allowed to change state within the state class c_{out}. An input or output configuration contains at most one entry for each allowed state class.

The starting point of the algorithm is a type, $\{i_1 \rightarrow o_1, \ldots, i_n \rightarrow o_n\}$. The algorithm is applied to each rule $i \rightarrow o$ in this type, individually. Throughout, we abbreviate `RUNNING` as R, `READY` as D, `BLOCKED` as B, `NOWHERE` as N, and `TERMINATED` as T. The presentation of the algorithm is somewhat informal. The algorithm has been implemented as part of the Bossa framework.

A.1 Distributing processes among the child schedulers

The first step of the algorithm is to distribute the processes mentioned in a type rule among the child schedulers of a virtual scheduler. Two child schedulers are distinguished: the child scheduler *target* managing the target process to which the current event must be forwarded and the child scheduler *preempt*, if any, managing a process that is to be preempted. Other processes are distributed among some set of child schedulers *other*.

Based on a given type rule $i \rightarrow o$, a preprocessing step constructs the set of states that are initially known to be empty, the set of processes (at most one) known to be in the `RUNNING` state class, and an environment mapping each process named by i to its input and output state classes:

$$
\begin{aligned}
&\mathit{empties} = \{c \mid \texttt{[] = } c \in i\}\\
&\mathit{in_running} = \{p \mid \texttt{[}p\texttt{] in RUNNING} \in i\}\\
&\mathit{env} = \{(p, (ic, oc)) \mid p \in i(ic) \wedge\\
&\qquad\qquad (p \in o(oc) \vee (ic = oc \wedge \forall c.p \notin o(c)))\}
\end{aligned}
$$

In the definition of *env* and subsequently, we define $i(c)$ as $\emptyset$, if either `[] =` c is part of i or there is no information about c in i, and as $\{p_1, \ldots, p_n\}$ if `[`$p_1, \ldots, p_n$`] in` c_{in} is part of i. $o(c)$ is defined similarly.

The set *distrib* of all possible distributions is defined as follows:

$$
\begin{aligned}
&\mathit{distrib} =\\
&\{\langle \mathit{target}, \mathit{preempt}, \mathit{other}\rangle \mid\\
&\quad \langle T, P, O\rangle \in \mathsf{partition}(\mathsf{domain}(\mathit{env}), 3) \wedge\\
&\quad \mathit{target} =\\
&\qquad \{(c, \mathit{ps}) \mid c \in \{R, D, B, N\} \wedge\\
&\qquad\qquad \mathit{ps} = \mathsf{check}(c, \{p \mid p \in T \wedge \mathit{env}(p) = (c, _)\})\} \wedge\\
&\quad \mathit{preempt}' =\\
&\qquad \{(c, \mathit{ps}) \mid c \in \{R, D, B, N\} \wedge\\
&\qquad\qquad \mathit{ps} = \mathsf{check}(c, \{p \mid p \in P \wedge c = R \wedge\\
&\qquad\qquad\qquad \mathit{env}(p) = (R, D)\})\} \wedge\\
&\quad \mathit{preempt} = \mathsf{check_preempt}(\mathit{preempt}')\\
&\quad \mathcal{O} = \mathsf{partition}(O, |O|) \wedge\\
&\quad \mathit{other} =\\
&\qquad \{\{(c, \mathit{ps}) \mid c \in \{R, D, B, N\} \wedge\\
&\qquad\quad \mathit{ps} = \mathsf{check}(c, \{p \mid p \in O' \wedge \mathit{env}(p) = (c, c)\})\} \mid\\
&\qquad O' \in \mathcal{O}\} \wedge\\
&\quad \texttt{tgt} \in \bigcup \mathsf{range}(\mathit{target}) \wedge\\
&\quad \bigcup \mathsf{range}(\mathit{target}) \cup \bigcup \mathsf{range}(\mathit{preempt}) \cup\\
&\qquad \bigcup\{\bigcup \mathsf{range}(O) \mid O \in \mathit{other}\} = \mathsf{domain}(\mathit{env})\}
\end{aligned}
$$

The function $\mathsf{partition}(s, n)$ partitions the set s into a sequence of n possibly empty subsets. The definition of *distrib* uses the functions check and check_preempt, defined as follows:

$$
\begin{aligned}
\mathsf{check}(c, \mathit{ps}) = {} & \texttt{[]}, \text{if } c \in \mathit{empties}\\
& \texttt{[]}, \text{if } c = R \wedge \mathit{ps} \neq \mathit{in_running}\\
& \mathit{ps}, \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{check_preempt}(\mathit{env}) = {} & -, \text{if } \mathit{env} = \emptyset\\
& \mathit{env}, \text{otherwise}
\end{aligned}
$$

The function $\mathsf{check}(c, \mathit{ps})$ discards *ps* if the state class c is known to be empty, converts $\emptyset$ to `[]` when some process (managed by some other scheduler) is known to be in the `RUNNING` state class, and otherwise returns *ps* unchanged. The function $\mathsf{check_preempt}(\mathit{env})$ converts the information about *preempt* into the invalid marker "−" in the case where there is no preempted process.

A.2 Instantiation

The next step is to instantiate each state class about which there is no information as either empty or not empty. The result is the set *instantiates*, defined below. In the computation of *instantiates*, the function new() generates a fresh process name. The predicate $\mathsf{fresh}(p)$, used in Section A.4, returns true only for process names generated using this function.

$$
\begin{aligned}
&\mathsf{instantiate}(s) =\\
&\quad \{\{(R, \mathit{ps}_R), (D, \mathit{ps}_D), (B, \mathit{ps}_B), (N, \mathit{ps}_N)\} \mid\\
&\qquad \mathit{ps}_R \in \mathsf{instantiate}'(R, s) \wedge \mathit{ps}_D \in \mathsf{instantiate}'(D, s) \wedge\\
&\qquad \mathit{ps}_B \in \mathsf{instantiate}'(B, s) \wedge \mathit{ps}_N \in \mathsf{instantiate}'(N, s)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{instantiate}'(c, s) = {} & \{\texttt{[]}, \mathsf{new}()\}, \text{if } s(c) = \emptyset\\
& \{s(c)\}, \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{instantiations} =\\
&\quad \{\langle \mathit{target}, \mathit{preempt}, \mathit{other}\rangle \mid\\
&\qquad \langle T, P, O\rangle \in \mathit{distrib} \wedge\\
&\qquad \mathit{target} \in \mathsf{instantiate}(T) \wedge\\
&\qquad (P = - \Rightarrow \mathit{preempt} = -) \wedge\\
&\qquad (P \neq - \Rightarrow \mathit{preempt} \in \mathsf{instantiate}(P)) \wedge\\
&\qquad \mathit{other} = \{o' \mid o \in O \wedge o' \in \mathsf{instantiate}(o)\}
\end{aligned}
$$

A.3 Computing output configurations

For each instantiation, $\langle \mathit{target}, \mathit{preempt}, \mathit{other}\rangle$, we next compute output configurations resulting from forwarding the current event notification to the scheduler represented by *target* and forwarding a `preempt` notification to the scheduler represented by *preempt*. For this, we consider the configuration (*target* or *preempt*) and each of the type rules. For conciseness, we only present the computation of output configurations from a single configuration *config* and type rule $i \rightarrow o$. The result is a set of output configurations, corresponding to the different ways that process names in *config* can be matched to the process names in i.

In the first step, we create a set of environments, each mapping the process names in *config* in some way to the process names in i for which there is a state change between i and o. The names `src` and `tgt` can only be matched against themselves. For the other process names, we consider all permutations. Π takes the cross product of a sequence of values.

$$
\begin{aligned}
\mathit{moves}(i, o) = \{p \mid {} & \exists c \in \{R, D, B, N\}.(p \in i(c) \wedge\\
& \exists c' \in \{R, D, B, N\}.(p \in o(c') \wedge c \neq c'))\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{create_entry}(c) =\\
&\quad \{\emptyset\}, \text{if } \mathit{config}(c) = \texttt{[]} \wedge (i(c) = \texttt{[]} \vee i(c) = \emptyset)\\
&\quad \{(p_{c_1}, p_{i_1}), \ldots, (p_{c_n}, p_{i_n}) \mid \langle p_{c_1}, \ldots, p_{c_n}\rangle = \mathit{config}(c) \wedge\\
&\qquad \langle p_{i_1}, \ldots, p_{i_n}\rangle = \mathsf{permutation}(\mathit{relevant}, n) \wedge\\
&\qquad \forall j.p_{c_j} = \texttt{src} \Leftrightarrow p_{i_j} = \texttt{src} \wedge\\
&\qquad \forall j.p_{c_j} = \texttt{tgt} \Leftrightarrow p_{i_j} = \texttt{tgt}\},\\
&\qquad \text{if } n \geq |\mathit{relevant}|, \text{ where } n = |\mathit{config}(c)| \text{ and}\\
&\qquad\quad \mathit{relevant} = \{p \mid p \in \mathit{moves} \wedge p \in i(c)\}\\
&\quad \emptyset, \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{envs} = \{r \cup d \cup b \cup n \mid {} &\\
& \langle r, d, b, n\rangle \in \Pi\langle \mathsf{create_entry}(c) \mid c \in \langle R, D, B, N\rangle\rangle\}
\end{aligned}
$$

In the second step, we use each environment in *envs* to modify the configuration *config* according to the transitions described by the rule $i \rightarrow o$. We present this operation for the case of a single environment. To simplify the presentation, we consider the environment to be a list (as in SML). For a given input configuration *config*, a type $i \rightarrow o$, and a corresponding environment *env* a set of output configurations are generated using $\mathsf{update}(\mathit{config}, o, \mathit{env})$, defined as follows:

$$\begin{aligned}&\mathsf{update}(\mathit{config}, o, [\,]) = \mathsf{instantiate}(\mathit{config})\\&\mathsf{update}(\mathit{config}, o, (p_c, p_i) :: \mathit{env}) =\\&\quad \text{let } c_{old} = \mathsf{find}(p_c, \mathit{config})\\&\qquad c_{new} = \mathsf{find}(p_i, o) \text{ in}\\&\quad \mathsf{update}(\mathsf{add}(p_c, c_{new}, \mathsf{rem}(p_c, c_{old}, \mathit{config})), o, \mathit{env})\end{aligned}$$

The functions add and rem add and remove a process to and from a state class, respectively. If c_{old} is R, then rem updates the information about this state class with `[]`, as there can have been at most one running process. Otherwise, rem simply removes the process p_c from the information about the state class c_{old}. Multiple configurations are generated because of the application of instantiate to the final result.

A.4 Constructing type rules

For each element $\langle \mathit{target}, \mathit{preempt}, \mathit{other}\rangle$ of *instantiations*, we use update to create corresponding output configurations for the child schedulers represented by *target* and *preempt*; the output configuration for each element of *other* is just the element of *other* itself, as no state transitions are allowed for these schedulers. Combining *target*, *preempt*, and the elements of *other* each with some possible output configuration gives a set of pairs of input and output configurations $\{(s_{1_{in}}, s_{1_{out}}), \ldots, (s_{n_{in}}, s_{n_{out}})\}$. For each $s \in \{s_{1_{in}}, \ldots\}$, we compute a process name as follows:

$$\begin{aligned}&\mathsf{proc_nm}(s) =\\&\quad \texttt{src-tgt}, \text{if } \exists c.\texttt{src} \in s(c) \wedge \texttt{tgt} \in s(c)\\&\quad \texttt{src}, \text{if } \exists c.\texttt{src} \in s(c)\\&\quad \texttt{tgt}, \text{if } \exists c.\texttt{tgt} \in s(c)\\&\quad \mathsf{concat}(\{p \mid \exists c.\texttt{tgt} \in s(c) \wedge \neg\mathsf{fresh}(p)\}), \text{otherwise}\end{aligned}$$

For each $s \in \{s_{1_{in}}, s_{1_{out}}, \ldots\}$, we also compute a state class as follows:

$$\begin{aligned}&\mathsf{state_class}(s) =\\&\quad R, \text{if } s(R) \neq \texttt{[]}\\&\quad D, \text{if } s(R) = \texttt{[]} \wedge s(D) \neq \texttt{[]}\\&\quad B, \text{otherwise}\end{aligned}$$

This computation is identical to the scheduler state computation described in Section 3.2.

The type rule corresponding to set of pairs of input and output configurations $\{(s_{1_{in}}, s_{1_{out}}), \ldots, (s_{n_{in}}, s_{n_{out}})\}$ is defined as follows:

$$\begin{aligned}&\{(R, ps_{R_{in}}), (D, ps_{D_{in}}), (B, ps_{B_{in}})\} \rightarrow\\&\qquad \{(R, ps_{R_{out}}), (D, ps_{D_{out}}), (B, ps_{B_{out}})\}\end{aligned}$$

For this, we first compute:

$$\begin{aligned}&\forall c \in \{R, D, B\}.\\&\quad ps'_{c_{in}} = \{p \mid \exists j.p = \mathsf{proc_nm}(s_{j_{in}}) \wedge c = \mathsf{state_class}(s_{j_{in}})\}\\&\quad ps'_{c_{out}} = \{p \mid \exists j.p = \mathsf{proc_nm}(s_{j_{in}}) \wedge c = \mathsf{state_class}(s_{j_{out}})\}\end{aligned}$$

Based on this information, we then compute $ps_{c_{in}}$ and $ps_{c_{out}}$, for each state class c. For the inputs:

$$\begin{aligned}ps_{c_{in}} = {}& \texttt{[]}, \text{if } c \in \mathit{empties}\\& ps'_{c_{in}}, \text{otherwise}\end{aligned}$$

For the output of the `RUNNING` state class:

$$\begin{aligned}ps_{R_{out}} = {}& \texttt{[]}, \text{if } R \in \mathit{empties} \wedge ps'_{R_{out}} = \emptyset\\& \texttt{[]}, \text{if } \mathit{is_running} \neq \emptyset \wedge ps'_{R_{out}} = \emptyset\\& ps'_{R_{out}}, \text{otherwise}\end{aligned}$$

The first two cases show that the absence of information about the schedulers in a given state class in the output configuration is not sufficient to conclude that the state class is empty. This is because there could be some child schedulers that are not represented among the $s_1, \ldots, s_n$, and these child schedulers may be in a state of the given state class. If the state class is additionally known to be empty in the input configuration i of the type rule, however, we can conclude that the state class is empty at the end of the handler, because such extra child schedulers cannot change state class. For the `RUNNING` class, there is an additional case, represented by the second rule, where we can conclude that $ps_{R_{out}} = \texttt{[]}$. In this rule, the conditions imply that there was originally a process in the state of the `RUNNING` state class, but this process has moved to a state of another state class, and thus the `RUNNING` state class is now known to be empty.

Finally, for the output of the `READY` or `BLOCKED` state class, *i.e.*, where $c \in \{D, B\}$, we obtain:

$$\begin{aligned}ps_{c_{out}} = {}& \texttt{[]}, \text{if } c \in \mathit{empties} \wedge ps'_{c_{out}} = \emptyset\\& ps'_{c_{out}}, \text{otherwise}\end{aligned}$$