Event types

This document describes the notation used in the event types. An example of the error messages that result from checking the event types is here.

An event type describes possible process states before and after an event notification. The basic form of an event type is as follows:

$$\{\ [p\ op\ class\ ,\ ...]\ ...\ \}\ \text{->}\ \{\ [p\ op\ class\ ,\ ...]\ ...\ \}$$

In such a type, the *configurations* , [*p op class* , ...], to the left of the arrow describe possible process states before the event and the configurations to the right of the arrow describe possible process states after the event. The outer braces can be omitted in either case if there is only one possible input configuration or one possible output configuration.

A configuration consists of at most one *entry* , *p op class* , for each of the possible classes RUNNING, READY, BLOCKED, TERMINATED, and NOWHERE (NOWHERE represents processes that are not yet managed by the scheduler). The possible forms of an entry are:

- [ ] = *class*
- [*p* , ...] in *class*
- [*p* , ...] in *class* !
- *p* in *class*
- *p* in *class* !
- *class* !

Here, *p* represents a process, and can have one of the following forms:

- src: The source process.
- tgt: The target process.
- p: Any process, of which there must be at least one.

The quantification of p does not include any other processes mentioned in the entry. For example, the entry

```
[p, tgt] in READY
```

implies that READY contains at least two processes. All processes mentioned in the input configuration of an event type must also appear in the output configuration of the event type. If more variables are needed, p1, p2, etccan be used, with meaning analogous to p. Variables, *e.g.* tgt or p, are used to indicate the presence of processes and the movement of processes between classes. A change in the class of src or tgt between the input and output type indicates that the corresponding source or target process must move from a state of the class indicated in the input type to a state of the class indicated in the output type. A change in the class of a variable p between the input and output type indicates that processes in the states of the class indicated in the output type may have come from states of the class indicated in the input type. The ! annotation on a type indicates the possibility of state changes within the class. This annotation is only allowed in an output type. *class* ! indicates that processes may change state within the given class. [*p* , ...] in *class* ! and *p* in *class* ! indicate that processes may change state within the given class, in addition to the state changes implied by the variables. In contrast, an output type of the form *e.g. p* in *class* indicates that processes can be added to states of the class, as indicated by the use of the variable p, but that other processes in the states of the class cannot change state.

The following type illustrates the use of p:

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

This type implies that there is at least one state of the READY class that is non-empty and that every process in a state of the READY class must still be in a state of the READY class after the event notification.

The following type, which is the type of the bossa.schedule event, is an example of the use of !:

```
[[] = RUNNING, p in READY] -> [p in RUNNING, READY!]
```

This type implies that there must exist some process in a state of the READY class that the event handler moves to a state of the RUNNING class. READY! in the output type indicates that processes can change state within the READY class.

If a class is not mentioned in an output configuration, there can be no change with respect to the set of processes in states of the class. For example, the following type (a possible type of the yield.system.pause.* event):

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

does not mention READY in the output type and does not mention BLOCKED, TERMINATED, or NOWHERE at all. This implies that the event handler cannot move processes into a state of the READY class and that the event handler cannot have any effect on processes in any BLOCKED state or that either have terminated or are not yet managed by the scheduler.

Error messages

The event types of yield.system.immediate.* are:

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

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

Consider a policy where the states are declared as follows:

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

and the yield.system.immediate.* handler is defined as follows:

```
On yield.system.immediate.* {
  if (e.target in running) {
    e.target => blocked;
  }
}
```

Clearly, this handler does not satisfy the event types, because it puts the running process in blocked, which is a BLOCKED state, rather than in a READY state.

The Bossa compiler traces all of the possible computation paths through the policy, so it typically analyzes each handler more than once. An example of the error message produced on one analysis of this handler is as follows:

```
line 45: yield.system.immediate.*:
from input:
{nowhere_10: nowhere_10, running_0: tgt, ready: ready, blocked: [], terminated: []}
derived from event type input:
{tgt in RUNNING}
obtained output:
{nowhere_10: nowhere_10, running_0: [], ready: ready, blocked: tgt, terminated: []}
which caused the following errors:
In class READY, no process matching tgt
  in output type: {RUNNING = [], tgt in READY!}
```

The Bossa verifier instantiates each possible input type according to the states defined by the given policy and the configurations that can arise in the course of the execution of the policy. For each instantiation, it then simulates the execution of the handler, and then checks the result (still defined in terms of the states defined by the policy) against the possible output types. In this example, the verifier is considering the input type {tgt in RUNNING}. Based on this input type, it has constructed a configuration where there is some unknown thing in each of the nowhere and ready states (numerical suffixes such as _10 result from some renaming done by the compiler, and are unimportant), the target process is in the running state, and the remaining states are empty. From this configuration, it obtains an output configuration in which the running process is now empty and the target process is now in the blocked state. This configuration does not match the only possible output event type, which is {RUNNING = [], tgt in READY!}.