PROMETHEUS — A Compositional Modeling Tool for Real-Time Systems *

Gregor Gössler
Gregor.Goessler@imag.fr
VERIMAG, 2, av. de Vignate, 38610 Gières, France

Abstract

PROMETHEUS is a modeling tool allowing the user to specify and compose real-time systems, by means of synchronization and priorization of actions, with a scheduler specified in a high-level description language. The resulting model is checked for consistency of the priorities, safety and liveness properties which can — up to a certain degree — be guaranteed by PROMETHEUS. The composed system can be output in several formats.

1 Introduction

On the background of the growing complexity of real-time systems, it is a crucial, but more and more complex task to guarantee the absence of unwanted interference between the processes, which make the system behavior hard or impossible to predict. Priority functions [3, 1] are an intuitive and powerful means of modeling coordination between processes. Their composability allows a modular specification of different aspects of coordination — for example, functional aspects such as mutual exclusion, as well as non-functional aspects such as scheduling algorithms. These modules can be composed and checked for consistency, that is, absence of contradictory specifications.

PROMETHEUS is a modeling tool based on priority functions as a model for coordination between processes. Both real-time processes and a scheduler can be specified in a high-level modeling language. The language is sufficiently general to specify most frequently used scheduling policies such as rate monotonic scheduling (RMS, [9]), earliest deadline first (EDF, [9]), and the priority ceiling protocol (PCP, [10]). The possibility of defining and instantiating scheduler templates allows to establish a library of schedulers. Process templates simplify the specification of multiple occurrences of a process type with the same untimed transition structure, but different timing constraints.

Detailed diagnostics, mainly about liveness properties of the component systems, and inconsistent priority functions, accelerate debugging and help gaining confidence in the correctness of the specification.

The composed system can be output in the *intermediate format* IF [5]. Figure 1 gives an overview of the context of PROMETHEUS, and the toolset connected to the validation environment IF. The latter has been developed at VERIMAG, and uses

*to appear in proc. Workshop on Real-Time Tools (RT-TOOLS'2001)

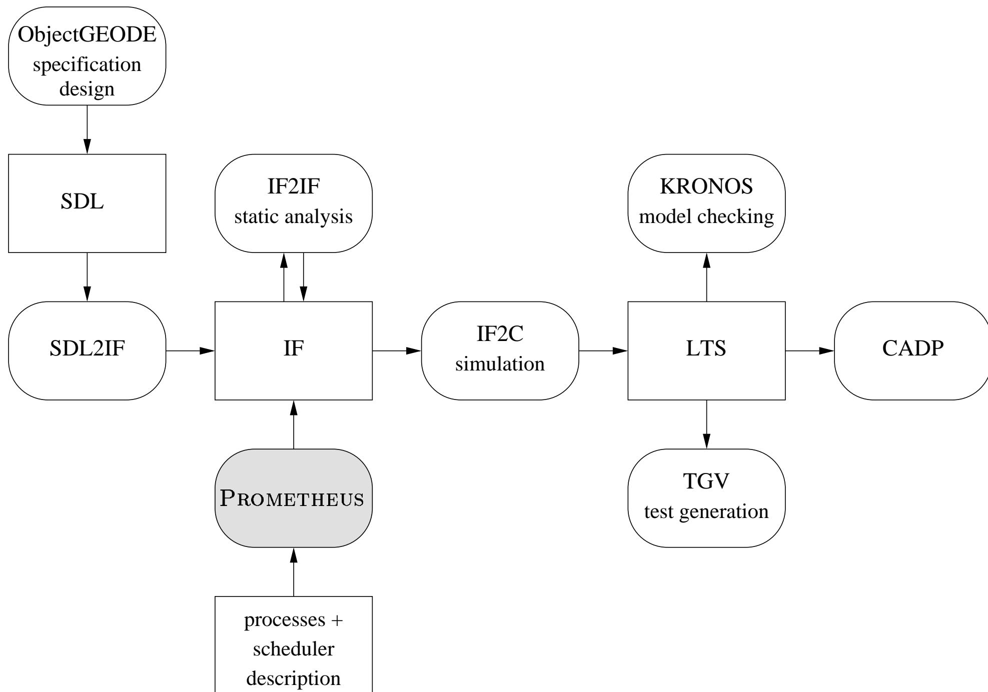

Figure 1: The context of PROMETHEUS.

IF as an intermediate format for timed asynchronous systems, and integrates tools operating on different levels of abstraction.

At the *specification level*, systems can be described in a high-level description language such as SDL [8]. The translator SDL2IF allows an automatic translation to the IF language.

At the *intermediate level*, the system is represented as a set of parallel communicating processes. At this level, the tool IF2IF allows optimization based on static analysis techniques.

The tool IF2C generates a simulator that uses techniques such as partial order reduction and on-the-fly model-checking to explore the state space of the IF specification, giving access at the *semantic level*, to the corresponding labeled transition system (LTS). The latter can be analyzed using the tool suite CADP [6], including the minimization and comparison tool ALDEBARAN based on bisimulation, and the alternating-free μ-calculus model-checker EVALUATOR. At the same level are also available the TCTL model-checker KRONOS [11], and the test generator TGV [7].

2 Modeling Real-Time Systems

2.1 Modeling Real-Time Processes

Consider the two periodic processes P_1 and P_2 modeled as timed systems [4], as shown in figure 2. We suppose them to share two non-preemptable resources r_1 and r_2. In

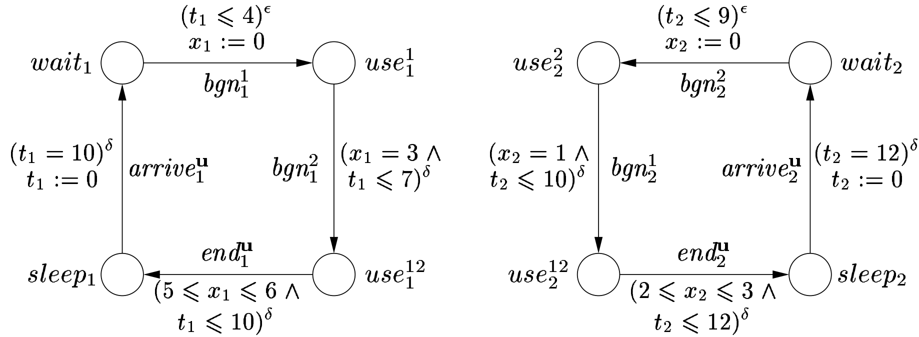

Figure 2: Two processes sharing two non-preemptable resources.

the control states $sleep_1$, $wait_1$, use_1^1, and use_1^{12}, process P_1 is sleeping, waiting for r_1, using r_1, and using both resources, respectively. The control structure of process P_2 is nearly symmetric, with the difference that P_2 acquires first r_2 (state use_2^2) and then r_1 (state use_2^{12}).

Each transition is labeled by an action, a predicate on the clock valuations called *guard*, and a set of clocks to be reset. Actions with an upperscript **u** are *uncontrollable*, that is, they cannot be controlled by the scheduler, whereas all other actions are *controllable* and can be disabled by the scheduler. The guard specifies when the transition is enabled; its exponent (δ or ϵ) is the *urgency type* specifying when the transition becomes urgent and *must* be taken: a *delayable* transition must be taken before it becomes disabled forever, whereas an eager transition must be taken as soon as it is enabled. Transitions are instantaneous; in any control state, time can progress until a transition leaving the control state becomes urgent.

Consider process P_1. From state $sleep_1$, the uncontrollable transition $arrive_1$ leading to state $wait_1$ is taken as soon as the value of clock t_1 reaches 10. t_1 is reset by $arrive_1$; it is used to measure the time elapsed since the last occurrence of $arrive_1$. Hence, the process has an inter-arrival time of 10. The controllable transition bgn_1^1 leading from $wait_1$ to use_1^1 and acquiring r_1 must be taken as soon as possible, and at last at $t_1 = 4$; it resets another clock x_1 measuring the time passed in states use_1^1 and use_1^2. The two remaining transitions bgn_1^2 and end_1 acquire r_2 and free both resources, respectively.

In [3] a property called *structural liveness* has been defined that implies liveness and that can be checked locally at low cost on processes as the conjunction of three more elementary structural properties. Under certain hypotheses on the composition of structurally live processes, the composed system is guaranteed to be structurally live. PROMETHEUS implements these results.

2.2 Modeling with Priorities

Priorities are widely used in modeling formalisms to restrict system behavior, especially for conflict resolution and scheduling, or to reduce non-determinism. We adopt the approach discussed in [4, 3], providing a general framework for dynamic priorities on the actions of a timed system of processes *TS*.

A *priority order* is a strict partial order $\prec$ on the set of actions of *TS*. *Priority*

functions associate priority orders with subsets of states of the system. More formally, a priority function pr is a finite set of pairs $\{(C^j, \prec^j)\}_{j \in J}$, where for any $j \in J$, C^j is a *state constraint*, that is, a predicate on the control states and clock valuations of TS, specifying when the priority order $\prec^j$ applies.

Priority functions are a natural and intuitive means for expressing functional properties such as mutual exclusion or atomicity, as well as non-functional aspects of process interaction such as scheduling policies. It is often desirable to model properties as priority functions in a modular manner, and to compose the priority functions. Therefore, we first define a composition operator on priority orders. Given two priority orders $\prec^1$ and $\prec^2$, we represent by $\prec^1 \oplus \prec^2$ the least priority order, if it exists, that contains $\prec^1 \cup \prec^2$. $\prec^1 \oplus \prec^2$ is undefined if the relation $\prec^1 \cup \prec^2$ contains a circuit, indicating contradictory priority functions. Two priority orders $\prec^1$ and $\prec^2$ for which $\prec^1 \oplus \prec^2$ is defined, are called *compatible*.

We extend the operator $\oplus$ to priority functions. Let pr_1 and pr_2 be two priority functions. Let s be some control state of TS, and $\mathbf{x}$ be a clock valuation. If $\prec^i$ is the priority order associated by pr_i with the system state $(s, \mathbf{x})$, for $i \in \{1, 2\}$, then the priority function $pr_1 \oplus pr_2$ associates with $(s, \mathbf{x})$ the priority order $\prec^1 \oplus \prec^2$, if it is defined.

The composition operator $\oplus$ allows a modular description of different behavioral aspects as a set of priority functions. Their composition provides a priority function integrating the different aspects of process interaction.

Moreover, the composition of priority functions helps detecting design flaws at an early stage, since it allows to check consistency of the priority functions to be composed. This is because the composition is undefined for system states for which the priority functions are contradictory, as in the concluding example. This is an important argument in favor of modeling and integrating as many interactional aspects as possible by priority functions, in order to detect possible inconsistencies. For example, modeling both atomicity of action sequences, and resource allocation under the priority ceiling protocol in the formalism of priority functions, allows to integrate both policies in one formal description, and to detect inconsistencies.

Mutual exclusion. Mutual exclusion can be modeled with priorities. Consider the two processes P_1 and P_2 above with critical sections $C_1 = \{use_1^1, use_1^{12}\}$ and $C_2 = \{use_2^{12}\}$ respectively, where resource r_1 is used. Whenever P_1 is ready to enter C_1, and P_2 is already in C_2, then P_2 must leave C_2 before P_1 can enter C_1, and vice versa. This order is expressed by the priority function disabling actions entering C_1 (resp. C_2) whenever some transition leaving C_2 (resp. C_1) will eventually be enabled in the current control state.

2.3 Modeling Scheduling Policies

Many existing scheduling policies distinguish between rules assigning process priorities to resolve conflicts between processes on the one hand, and admission control rules deciding whether some process is eligible for resource allocation on the other hand. For example, the priority ceiling protocol schedules the process with the highest current priority among the processes that are waiting for the processor, whereas a process P_i is eligible for the allocation of a free resource if the current priority of P_i is higher than the priority ceilings of all resources currently allocated to processes other than P_i. As discussed in [2], we formalize this decomposition of a scheduling policy as a

conjunction of two priority functions: $pr_{\mathsf{pol}} = pr_{\mathsf{adm}} \oplus pr_{\mathsf{res}}$, where pr_{adm} defines an admission control that restricts resource allocations according to a given condition, whereas pr_{res} resolves conflicts between two or more processes waiting for the same resource.

We assume that for each resource r, conflicts are resolved according to a partial order on the set of processes $\{P_i \mid i \in \{1,\ldots,n\}\}$. The partial order is specified as a set of state constraints $\{\mathcal{C}^r_{ij}\}_{i,j\in\{1,\ldots,n\}}$. When $\mathcal{C}^r_{ij}$ holds, process P_i has priority over process P_j for using resource r. Notice that $\mathcal{C}^r_{ij}$ may depend on clock valuations as well as on control states. As an example, the earliest-deadline-first policy (EDF) on resource r specifies that r is granted to the waiting process that is closest to its relative deadline. If we assume that the deadline of a process P_i is equal to its period T_i, then the EDF policy is defined by

$$\mathcal{C}^r_{ij} \;=\; (T_i - t_i < T_j - t_j) \vee (T_i - t_i = T_j - t_j \wedge i < j) \enspace .$$

The first term of $\mathcal{C}^r_{ij}$ means that whenever two processes P_i and P_j are both waiting for r, then the actions granting r to P_i have immediate priority over the actions granting r to P_j if P_i is closer to its relative deadline than P_j (namely, $T_i - t_i < T_j - t_j$). The second term ensures a strict allocation order in case of conflict (that is, when $T_i - t_i = T_j - t_j$). Applied to the previous two-process example, the priority function modeling EDF scheduling with respect to resource r_1 is

$$\big\{\big(10 - t_1 \leqslant 12 - t_2,\ \{bgn^1_2 \prec bgn^1_1\}\big),\quad \big(10 - t_1 > 12 - t_2,\ \{bgn^1_1 \prec bgn^1_2\}\big)\big\} \enspace .$$

Intuitively, if P_1 is closer to its deadline, then its action bgn^1_1 acquiring r_1 is given priority over action bgn^1_2 allocating r_1 to P_2, and vice versa; in case of equal distance, P_1 wins.

3 Prometheus

3.1 The Architecture of Prometheus

Figure 3 shows the global architecture of Prometheus. As an input, Prometheus reads a file describing the global system configuration including the process declarations, available resources, synchronizing actions, and optionally the name of a file specifying the scheduler to be used. The processes can be specified directly in the system file, or instantiated from process templates. From this description, Prometheus generates an IF specification of the processes, the priority function describing the coordination between the processes, including the scheduling policy, and optionally a timed automaton representing the product timed system with priorities.

3.1.1 The System Description

This file declares the different components of the system, as well as some high-level aspects of their interaction. It declares the available preemptable and non-preemptable resources, the component processes, which can be described directly, or instantiated from templates, and synchronization between actions of the processes. An optional scheduler can be instantiated.

System 1 shows a system description declaring a non-preemptable resource `cpu`, and instantiating three periodic processes that are scheduled under a scheduler called `edf.sched`.

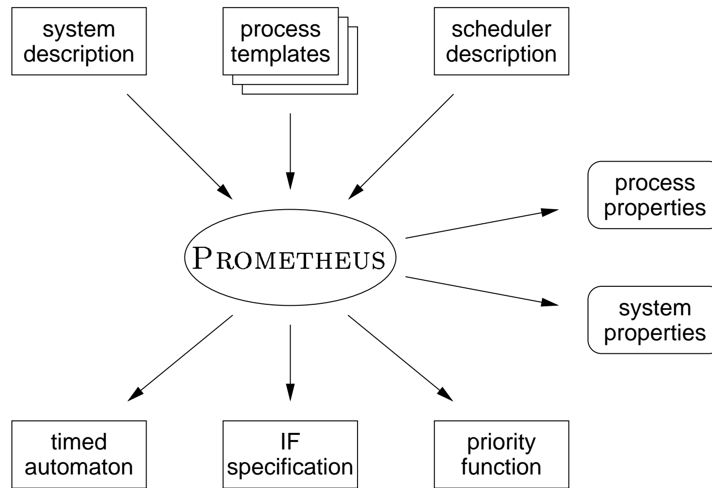

Figure 3: Overview of the tool PROMETHEUS.

System 1 Three periodic processes, scheduled under EDF.

```
SYSTEM edf_test
        NONPREEMPTABLE cpu;

        periodic.proc
                p1 (0, 2, 5),   // fixed priority, exec. time, period
                p2 (0, 3, 11),
                p3 (0, 4, 17)

        SCHEDULER edf.sched()
END edf_test
```

3.1.2 The Process Description

The description of a process can be part of the system description file, the scheduler description file, or make a file of its own. A process description essentially defines a Petri net with urgency. It can be parameterized by a list of integer parameters that are instantiated in the system description file. In the remainder of the process description, a parameter may replace any integer expression. The resources used by the process are specified by declaring, for each used resource, the set of places where it is used.

Process 1 shows the process template `periodic.proc` included by system 1. Transitions `arrive` and `rl` are uncontrollable, whereas transition `bgn` acquiring the resource `cpu` is controllable. `$T` and `$E` reference the parameters `T` and `E`, respectively.

3.1.3 The Scheduler Description

This file serves to specify the admission control, and conflict resolution policies. More generally, any priority function can be specified. As the process description, the sched-

Process 1 Template `periodic.proc` specifying a simple periodic process.

```
PND periodic (prio, E, T)
        PLACES sleep wait use
        RESOURCE cpu: use
        CLOCKS t x
        INIT: sleep;

        arrive: UNCTRL
                IN: sleep
                OUT: wait
                GUARD t = $T DELAYABLE
                RESETS t;

        bgn:    IN: wait
                OUT: use
                GUARD TRUE EAGER
                RESETS x;

        rl:     UNCTRL
                IN: use
                OUT: sleep
                GUARD x = $E DELAYABLE;
END periodic
```

uler can be parameterized. An optional Petri net with urgency may be used to define schedulers that have a memory, such as event handlers that keep trace of requests that have not been served yet. The scheduler may redeclare preemptable resources of the system description to be non-preemptable.

The *admission control policy* is defined by state constraints $blocking_{ij}$ specifying when process P_i blocks process P_j on some non-preemptable resource, as discussed in [2]. The definition of *blocking* is of the form

```
[FORALL pid1: PROCESS] [FORALL pid2: PROCESS]
              BLOCKING (pid1,pid2) := state_constraint
```

The process identifiers can be universally quantified. `state_constraint` can be prefixed by a list of existential and universal quantifications on the domains of processes, resources, preemptable resources, and non-preemptable resources. It may contain propositions on control states and clock states, as well as the pre-defined predicates **using(P,r)** and **waiting(P,r)** characterizing the control states where process P uses resource r, and process P waits for resource r, respectively. Moreover, integer expressions occurring in *blocking* may use the pre-defined expressions **prio(P)** for the fixed priority of some process P, and **ceiling(r)** for the priority ceiling [10] of some resource r.

A priority function defining the *conflict resolution policy* [1], can be given as a list of priority rules of the form

aid_1 `<0` aid_2 [`IF state_constraint`]

where $\mathtt{aid}_1$, $\mathtt{aid}_2$ are action identifiers, or of the form

[FORALL $\mathtt{pid}_1$: PROCESS] [FORALL $\mathtt{pid}_2$: PROCESS]
pid_1 <0 pid_2 [IF state_constraint]

In the first case, the priority is assigned between the specified actions; in the second case, the priority applies to all actions of the specified processes which acquire some resource. The optional `state_constraint` specifies the system states where the priorities apply. It has the same structure as for *blocking*. In addition, it can contain the predicate *blocking*.

Scheduler 1 shows the EDF scheduler used by system 1: for any two processes P_i, P_j, the actions of the latter acquiring some resource (in the case of system 1, the CPU) dominate the acquiring actions of the further if either P_j is closer to its deadline, or if both processes have the same distance to their deadlines, and the parameter `pid` of P_j is less than that of P_i.

Scheduler 1 Specification of an EDF scheduler.

```
SCHEDULER edf()
    PRIORITIES                      // conflict resolution
        FORALL Pi: PROCESS
        FORALL Pj: PROCESS
            Pi <0 Pj  IF  [ Pi.t - Pj.t < $Pi.T - $Pj.T ] OR
                          [ Pi.t - Pj.t = $Pi.T - $Pj.T ] AND
                                          $Pi.pid > $Pj.pid
END edf
```

Scheduler 2 specifies the priority ceiling protocol, as shown in [2]. In the first part of the scheduler specification, the state constraint *blocking* is defined. The second part specifies with two priority rules the conflict resolution policy. All three state predicates use quantifications.

3.2 How PROMETHEUS Works

On invocation of PROMETHEUS, the processes specified in the system description file are read. For each process, a detailed diagnostic of its liveness properties is printed, including information about its (non)zenoness, its minimal inter-arrival time, and its livelock-freedom. In case some property is not verified, a diagnostic with the concerned transitions and/or states is printed.

Next, PROMETHEUS introduces preemption transitions and places and resuming transitions for all states that use some preemptable resource, and synchronizes resource allocation actions with the compatible preemption actions [2]. According to the declaration of the resource using places of the processes, the priority function modeling mutual exclusion is automatically computed.

After the construction of the synchronized system of processes, the scheduler, if specified, is interpreted, and the priority function pr_{pol} representing the scheduling policy is added to the priority function computed so far. If the priority functions are

Scheduler 2 Specification of a PCP scheduler.

```
SCHEDULER pcp()
    FORALL Pi: PROCESS
    FORALL Pj: PROCESS
        BLOCKING (Pi, Pj) :=        // admission control
            EXISTS r1: NONPREEMPTABLE
            EXISTS r2: NONPREEMPTABLE
                USING (Pi, r1) AND WAITING (Pj, r2) AND
                CEILING (r1) >= PRIO (Pj)

    PRIORITIES                      // conflict resolution
        FORALL Pi: PROCESS
        FORALL Pj: PROCESS
            Pj <0 Pi  IF
                FORALL Pk: PROCESS
                    PRIO (Pi) > PRIO (Pj) AND
                    (PRIO (Pk) < PRIO (Pi) OR NOT BLOCKING (Pj, Pk))

        FORALL Pi: PROCESS
        FORALL Pj: PROCESS
            Pj <0 Pi  IF
                EXISTS Pk: PROCESS
                    PRIO (Pi) < PRIO (Pj) AND
                    PRIO (Pk) > PRIO (Pj) AND BLOCKING (Pi, Pk)
END pcp
```

found to be inconsistent, a warning message is issued informing about the states for which the problem appears, and the set of conflicting actions.

If the option **-d** is selected, then the system is made deterministic by completing the priority orders associated with any system state to a total order on all controllable actions.

Hereafter, a diagnostic about safety and liveness of the composed system is printed, which has been derived from the individual diagnostics and the way the system is composed. In particular, PROMETHEUS analyzes local structural properties of the processes that are a sufficient condition for the unreachability of states where mutual exclusion is violated or the priority function is not defined. If some hypotheses are met, the minimal inter-arrival time, the worst-case completion time, and their ratio are shown for each process. The minimal inter-arrival and maximal execution times are obtained automatically by analyzing the processes.

If the option **-a** has been selected, the product timed system is constructed and written to a file. If one of the options **-i**, **-j** has been selected, the product system is output in the form of a timed automaton, or a timed automaton with convex invariants, respectively. The output of the product timed system allows, for small systems, to directly use KRONOS in order to carry out verification of the constructed system. Without any of the options **-a**, **-i**, or **-j**, the set of processes, together with the synchronizations, is output as an IF specification, and the priority function is written to a separate file.

Execution 1 shows the output of PROMETHEUS for the example of figure 2. Both

Execution 1 The resource allocation example.

```
rechasse> prometheus crossover.system
Properties of P1:
  structurally non-Zeno with minimal loop time 10
  minimal inter-arrival time 10
  locally timelock-free
  timelock-free
  locally livelock-free
  livelock-free
  structurally live

Properties of P2:
  structurally non-Zeno with minimal loop time 12
  minimal inter-arrival time 12
  locally timelock-free
  timelock-free
  locally livelock-free
  livelock-free
  structurally live

Computing mutex priorities.. done.
Warning: priority circuit { P1.bgn2 P2.bgn2 } in (P1.use1 P2.use2 ) .

Product:
  potentially unsafe
  potentially not structurally live
  non-Zeno
  potentially not locally deadlock-free

Writing IF file... done.
Writing priority file... done.
```

processes are found to be structurally live. The minimal inter-arrival times of P_1 and P_2 are found to be 10 and 12, respectively. When composing the processes, PROMETHEUS detects an inconsistency of the priority functions modeling mutual exclusion in state $(P_1.use_1, P_2.use_2)$. Indeed, mutual exclusion on the use of the resources causes the system to deadlock in that state, where each process has acquired one resource, and is waiting for the resource held by the other process. This well-known example illustrates how the composition of priority functions provides diagnostics for contradictory specifications.

Let us now schedule the same system under the priority ceiling protocol (scheduler 2). Execution 2 shows the output of PROMETHEUS. The priority function is still undefined in $(P_1.use_1, P_2.use_2)$, but this time, the priority function representing the priority ceiling protocol makes state $(P_1.use_1, P_2.use_2)$ unreachable. According to the criteria in [3], the composed system is structurally live. Next, the worst-case

Execution 2 The resource allocation example, scheduled with PCP.

```
rechasse> prometheus crossover_pcp.system
Properties of P1:

    [ ... ]

Properties of P2:

    [ ... ]

Computing mutex priorities.. done.
Interpreting scheduler... done.
Warning: priority circuit { P1.bgn2 P2.bgn2 } in (P1.use1 P2.use2 ) .

Product:
  safe
  structurally live
  WCCT:
    P1: (P2, 3, 12)   WCCT=9  Tmin=10  Tmin/WCCT=1.111
    P2: (P1, 6, 10)   WCCT=9  Tmin=12  Tmin/WCCT=1.333

Writing IF file... done.
Writing priority file... done.
```

completion times of both processes are analyzed: P_1 is blocked by P_2 during at most 3 time units. With an execution time of 6, the WCCT of P_1 is 9, and therefore lower than its minimal inter-arrival time `Tmin` of 10. Similarly, P_2 is blocked during at most 3 time units, and is guaranteed to complete before its deadline of 12. Finally, both the IF specification and the priority function are written to files. The whole analysis takes about 0.2 seconds on a PentiumPro under Linux.

4 Conclusion

PROMETHEUS is a useful and efficient support to automatize tedious and error-prone tasks in modeling real-time applications. Its connection to the platform IF allows the user to take advantage of the existing toolset connected to IF, in order to perform static analysis, simulation, model-checking, or test generation on the IF specification generated by PROMETHEUS. The independence between the scheduler description and the specification of the system of processes to be scheduled, introduces a great deal of flexibility, since any of them can be modified independently. This allows, for example, to refine a scheduler, according to the diagnostic provided by PROMETHEUS and other tools, until it ensures liveness of the processes, without the need of modifying the latter.

However, PROMETHEUS is still a prototype. The integration into the IF platform is one-way, since the input language is proper to PROMETHEUS; accepting system descriptions in SDL or IF would further increase its usefulness.

Some models of real-world applications have been treated successfully with PROMETHEUS; the results are convincing. The powerful high-level modeling formalism has

turned out to be of great help for efficiently designing and engineering models of real-time applications, and in particular schedulers. As a side effect, many low-level modeling errors, which can be hard to find, are avoided. Since compositional modeling with priority functions, and the analysis of structural properties, do not require explicit construction of the product system, we think that even complex real-time systems can be tackled using PROMETHEUS.

References

[1] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In M. Joseph, editor, *proc. FTRTFT 2000*, volume 1926 of *LNCS*, pages 106–120. Springer-Verlag, 2000.

[2] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems, special issue on "control-theoretical approaches to real-time computing" (to appear)*, 2001.

[3] S. Bornot, G. Gößler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS 2000*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.

[4] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163:172–202, 2000.

[5] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: A validation environment for timed asynchronous systems. In E. A. Emerson and A. P. Sistla, editors, *Proc. CAV'00*, volume 1855 of *LNCS*, pages 543–547. Springer-Verlag, 2000.

[6] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. CAV '96*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, 1996.

[7] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.

[8] ITU-T. Recommendation Z.100. Specification and Design Language (SDL). Technical Report Z-100, International Telecommunication Union — Standardization Sector, Geneva, 1999.

[9] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[11] S. Yovine. KRONOS: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, 1997.