Compositional Modeling in Metropolis

Gregor Gössler and Alberto Sangiovanni-Vincentelli

University of California at Berkeley, Dept. of EECS
{gregor,alberto}@eecs.berkeley.edu

Abstract. METROPOLIS is an environment for the design of heterogeneous embedded systems. The framework is based on a general system representation called the METROPOLIS meta-model. This model forms the backbone of the software system and is used to integrate a variety of analysis and synthesis tools. Compositional modeling is a powerful method for assembling components so that their composition satisfies a set of given properties thus making the verification problem much simpler to solve. We use the meta-model to integrate the PROMETHEUS tool in METROPOLIS for supporting compositional modeling and verification of METROPOLIS specifications and present a first set of results on a non-trivial example, a micro-kernel real-time operating system, TinyOS.

1 Introduction

METROPOLIS [4] is a design environment for embedded systems. It supports a methodology that favors the reusability of components by explicitly decoupling the specification of orthogonal aspects over a set of abstraction levels. More precisely, computation, communication, and coordination are separated by having them described by different entities: processes, media, and schedulers. METROPOLIS proposes a formalism called *meta-model* that is designed so that various computation and communication semantics can be specified using common building blocks [8]. The meta-model supports progressive refinement of components, their communication and coordination. It allows executable code in a Java-like syntax as well as denotational formulas in temporal and predicate logic, so that the right level of details of the design can be defined at each abstraction. The METROPOLIS architecture encompasses a compiler front-end to translate a meta-model specification into an intermediate representation, and a set of back-end tools to support tasks such as synthesis, refinement, analysis, and verification of the model.

Building systems which satisfy given specifications is a central problem in systems engineering. Standard engineering practice consists in decomposing the system to be designed into a set of cooperating components. Sometimes this decomposition is dictated by the functionality of the system: for example, a network of sensors and actuators is naturally partitioned into components. We are interested in assessing whether the global behavior of the system satisfies given specifications. An essential problem to solve is how to compose the components. If indeed a rigorous design methodology is used when assembling the

system from components, then the verification problem can be solved either by construction or using formal methods. Unfortunately, designers are used to ad hoc design methodologies that almost always lead to solutions that must be validated by simulation, rapid prototyping and testing. In some cases, it is possible to solve the composition problem by synthesizing a controller or supervisor that restricts the behavior of the components [16] so that the overall system behaves correctly by construction or is amenable to formal analysis. Both verification at the global system level and synthesis techniques have well-known limitations due to their inherent complexity or undecidability, and cannot be applied to complex systems. As an alternative to cope with complexity, *compositional* modeling techniques have been studied. By compositional modeling we understand that the components of a (real-time) system are modeled in such a way that important liveness and progress properties are preserved when a new component is added to the system.

We are interested in the design of complex systems such as an embedded system for automotive power-train control or a wireless network of sensors and actuators. We have experience in setting up a platform-based design methodology supported by METROPOLIS for these applications that uses extensively the principles of successive refinement as a way of simplifying substantially the verification problem. However, there is much room for compositional methods that, in addition to the successive refinement principle, can improve substantially the design process in an unobtrusive way [1].

In this respect, we are motivated by the analysis of a particular network of sensors and actuators being designed at the Berkeley Wireless Research Center. The components of this network are small, inexpensive embedded computers called nodes, which can be distributed over a building, measure parameters such as temperature, and communicate with each other over a low-power wireless device. TinyOS [10] is an operating system for these embedded systems that provides basic functionality, such as task scheduling, message passing, and interrupt handling, while being extremely small (less than a kilobyte) and supporting a modular structure of the application. Since, in this wireless network, direct communication is only possible over short distances, most nodes cannot communicate with each other directly. A sensing and routing application running under TinyOS on each node is in charge of periodically requesting data from a sensor, transmitting this data, and routing incoming messages towards their destination. The question we would like to answer in an environment like METROPOLIS is whether the nodes do not deadlock and operate in a safe mode. Applying standard verification techniques is a hard problem because of the complexity of their behavior. It is then essential to apply techniques such as compositional modeling to see whether these problems can be solved in a substantially better way.

[1] Let a component C satisfy a property P_C that guarantees the composition of C with the rest of the system to satisfy a property P. When C is refined into a set of components such that their composition satisfies P_C, then the entire system still satisfies P. This principle allows incremental verification of the system as it is refined.

2 Design Flow for Compositional Modeling in Metropolis

The basic question we address in this paper is how to link compositional modeling methodologies and tools with a general framework like METROPOLIS. For the verification of models given in an expressive modeling language like the meta-model of METROPOLIS, it is in general necessary to represent high-level constructs in a more basic formalism on which verification can be carried out. The choice of their representation is crucial for the applicability of compositional reasoning. Not all meta-model constructs can be modeled so that existing compositionality results can be applied. We therefore need to subset the language of the meta-model.

Sub-setting the language allows to leverage the compositional modeling tool PROMETHEUS [13]. In PROMETHEUS real-time processes, their synchronization, and scheduling policies can be specified in a high-level modeling language. For the interfacing with METROPOLIS, the parser front-end for processes has been replaced with a parser for the METROPOLIS meta-model.

PROMETHEUS constructs the real-time system incrementally by first analyzing the behavior of the processes, then taking into account their synchronization on shared resources, and finally applying the specified scheduling policy. The resulting model is compositionally checked for safety, liveness, and timing properties, by analyzing the properties of the processes and deriving properties of the system. Detailed diagnostics are given to accelerate debugging and help gaining confidence in the correctness of the specification.

The main limitation of the compositional modeling principles implemented in PROMETHEUS comes from its conservative character. In case a property cannot be guaranteed nor refuted by PROMETHEUS, help from "classical" verification techniques and tools is still needed. This help is provided by the validation platform IF [7] in which PROMETHEUS is integrated, allowing the user to export models towards a large range of existing tools in order to perform static analysis, simulation, model-checking, or generation of tests on the IF specification generated by PROMETHEUS.

In this paper, we are interested in assessing how compositional modeling can be adopted in the framework of METROPOLIS, as well as exploring and validating new methods for compositional design and verification. The design flow and interaction between the tools is sketched in fig. 1 and can be described as follows:

1. The system designer provides a restricted meta-model specification that is compatible with the PROMETHEUS formalism, and optionally a high-level description of a scheduling policy to be applied to the system.
2. The meta-model specifications are transformed in the PROMETHEUS formalism and the model is analyzed by PROMETHEUS. If a scheduler is specified, the behavior of the model is accordingly restricted. PROMETHEUS generates diagnostics about properties such as consistency of the requirements, safety, liveness, non-zenoness etc., and outputs dynamic priority rules describing the optional scheduling policy.

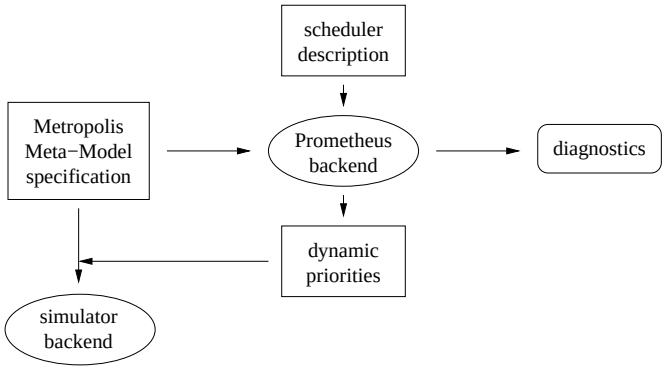

Fig. 1: Design flow and interaction between the tools.

3. The model can be executed using a simulator back-end. If desired, the model is equipped with a generic scheduler applying the priority rules generated by PROMETHEUS. The model can also be exported towards the validation platform IF, which is not shown in the figure.

The remainder of the paper is organized as follows. Section 3 gives an informal overview of the modeling formalism used by PROMETHEUS. In section 4 we show how some important constructs of the METROPOLIS meta-model are represented in the modeling formalism of PROMETHEUS such that compositional analysis techniques apply. Section 5 presents a case study, and section 6 discusses our approach and future work.

3 Modeling in Prometheus

In PROMETHEUS, real-time systems are modeled by timed systems with dynamic priorities. We give here a simplified presentation; a formal discussion can be found in [5, 12].

3.1 Timed Systems

We model real-time processes as timed systems, a variant of timed automata [2]. Consider the timed system of fig. 2 modeling a process with period T and execution time E. Each transition is labeled by an action name, a guard, and possibly one or more clocks being reset. A *state* of a timed system is a tuple $(s, \mathbf{x})$, where s is a control state, and $\mathbf{x}$ is a *clock valuation*. The timed system can evolve in two different ways: by letting time pass (which means that the values of all clocks increase uniformly), or by taking some transition that is *enabled* — which means that its guard is verified for the current clock valuation — and resetting the specified clocks to zero. Taking a transition is instantaneous.

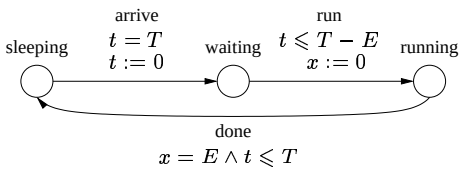

Fig. 2: Timed system modeling a simple process.

An enabled transition becomes *urgent* and must be taken before it gets disabled forever.

From control state *sleeping*, transition *arrive* leading to control state state *waiting* becomes enabled and urgent when the value of clock t reaches T. t is reset by *arrive*; it is used to measure the time elapsed since the last occurrence of *arrive*. Transition *run* leading from *waiting* to *running* must be taken by $t = T - E$ and resets another clock x measuring the time passed in state *running*. Finally, with transition *done*, the process returns to *sleeping* after having spent E time units in *running*.

We use a *flexible* parallel composition $\|$ of timed systems [6]. *Maximal progress* semantics ensures that synchronization transitions are taken whenever this is possible; a synchronizing transition may interleave only when no synchronization is possible. We refer to the parallel composition of timed systems as a *timed system of processes*.

3.2 Coordination

Priorities are widely used for scheduling [15, 17], and for conflict resolution in modeling formalisms for concurrent systems, such as process algebras, see for example [3, 9]. In most approaches, absolute priority levels are assigned to processes or transitions. This approach suffers from two problems that take away much of the potential strength of priorities as a modeling tool. First, absolute priority levels lead to models that are not *incremental*, in the sense that adding a process in general requires recomputing the priorities. Second, priority layers are not *composable*, in the sense that two priority assignments expressing two properties cannot be easily composed to a single priority assignment ensuring both properties. For these reasons, partial priority orders between the actions of processes are particularly interesting, since they allow to express *local* properties as priority relations that only apply between certain transitions, without side effect on other transitions.

We adopt the approach developed in [6, 5, 12], providing a general framework for dynamic priorities on the actions of a timed system of processes *TS*. The key to a modular description of different behavioral aspects by dynamic priorities is the composability principle studied in [1].

A *priority order* is a strict partial order $\prec$ on the set of actions of *TS*. A *priority function* associates priority orders with subsets of states of the system. More formally, a priority function pr is a set of priority rules $\{(C^j, \prec^j)\}_{j \in J}$,

where J is a finite index set, and for any $j \in J$, C^j is a predicate on the control states and clock valuations of TS, specifying when the priority order $\prec^j$ applies. We require the C^j to be invariant under the progression of time. A *timed system with priorities* is a tuple (TS, pr).

In order to allow composition of priority functions, 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 orders. We now extend the partially defined operator $\oplus$ to priority functions. Let pr_1 and pr_2 be two priority functions, and $(s, \mathbf{x})$ be a state of TS. 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 $pr_1 \oplus pr_2$ maps $(s, \mathbf{x})$ to $\prec^1 \oplus \prec^2$. A priority function that maps any system state to a priority order is called *well-defined*. Two priority functions are *consistent* if their composition is well-defined.

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 are a natural and powerful means for modeling coordination between processes, including functional requirements such as mutual exclusion or atomicity, as well as non-functional aspects of process interaction like scheduling policies [5, 12]. Modeling and composing different interactional aspects by priority functions helps detecting design flaws at an early stage. Inconsistencies can be backtracked up to a set of contradictory requirements. The diagnostic provided by the composition operation comprises the set of states for which the problem appears, and of the set of conflicting actions.

Scheduler modeling. A general modeling framework for scheduling policies based on priority functions has been discussed in [12, 1] and implemented in PROMETHEUS. It has been shown how frequently used scheduling policies such as rate monotonic (RM) and earliest deadline first (EDF) scheduling [15], and the priority ceiling protocol [17], can be modeled in the scheduler description language of PROMETHEUS, and represented as priority functions.

Consider n instances TS_i, $i \in \{1, \ldots, n\}$ of the periodic process of fig. 2 with periods T_i and execution times E_i. We suppose that they use a shared CPU in the *running* states. Scheduling the processes according to the EDF policy means that the CPU is granted to the waiting process that is closest to its relative deadline. If we assume that the deadline of a process TS_i is equal to its period T_i, and the time elapsed since the beginning of its period is measured by a clock t_i, then the EDF policy is modeled by the priority function

$$pr_{\mathsf{pol}} \;=\; \bigoplus_{i \neq j} \big\{ \big(T_i - t_i < T_j - t_j,\ \{ run_j \prec run_i \} \big) \big\} \,.$$

Intuitively, if TS_i is closer to its deadline than TS_j, then its action run_i is given priority over action run_j.

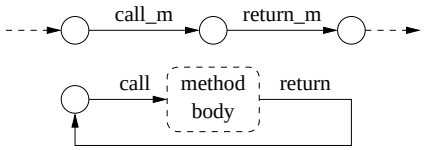

Fig. 3: Method call between process (top) and method (bottom). Transitions *call_m* and *call* synchronize, as well as *return_m* and *return*.

4 Translation of Some Meta-Model Constructs

This section shows how a timed system of processes and a priority function, which can be analyzed by PROMETHEUS, are constructed from a meta-model specification. The current implementation of the PROMETHEUS back-end makes abstraction from data values, keeping only information about control, timing, and coordination. Translation of many meta-model constructs is simplified by the fact that in PROMETHEUS, as in METROPOLIS, computation and coordination are modeled separately. In the translation, components are represented by timed systems, whereas constraints on the coordination of components are modeled by priority functions that are then composed. As both the meta-model and timed systems with priorities have formal semantics, correctness of the translation can be formally proven, which is however beyond the scope of this paper. A back-end tool translating meta-model code into timed systems with priorities has been implemented.

Processes and Media. Each process or medium of a meta-model specification is represented by a timed system in PROMETHEUS, which is constructed bottom-up from the control structure of the process or medium: an atomic statement is represented by a single transition leading from an initial to an end state; a sequence of two statements is modeled by merging the end state of the first statement with the initial state of the second.

The meta-model enforces a strict separation between computation and communication. The only way for a process to communicate with another process is to call an interface method implemented by some medium. This is modeled under PROMETHEUS by decomposing each method call in two transitions, for the call and the return of control. They synchronize with corresponding transitions of the medium, as shown in fig. 3. This translation of method calls requires that any method of a given medium is executed by no more than one process at the same time, such that transition `return` synchronizes with transition `return_m` of a unique process.

Schedulers. The meta-model provides a class **Scheduler** whose instances can be connected to processes in order to coordinate their execution. Meta-model schedulers may contain executable code that is difficult to analyze, and more

expressive than timed systems with priorities. In particular, it cannot exclude deadlocks. For this reason, PROMETHEUS currently does not support the full generality of the class but provides high-level language constructs to build dynamic priority schedulers.

Constraints. The meta-model allows the description of very general constraints including LTL (linear-time temporal logic) and first-order logic formulas. We focus on timing and synchronization requirements, for which the meta-model provides macro notations, and show how they can be modeled.

The maximal rate of an event, and the maximal latency between two events, can be specified by the statements

maxrate ($block$, d) and **maxlate** ($block_1$, $block_2$, d),

where $block$, $block_1$, and $block_2$ are labels referring to blocks of statements. The meaning of **maxrate** is that $block$ may be entered at most every d time units. This is modeled in PROMETHEUS by having some clock x reset by all transitions entering $block$, and constraining the guards of all transitions entering $block$ by $x \geqslant d$. The **maxlate** constraint signifies that $block_2$ must be left at most d time units after $block_1$ has been entered, which is modeled by resetting a clock x at all transitions entering $block_1$, and constraining all transitions leaving $block_2$ by $x \leqslant d$.

Mutual exclusion between two critical sections labeled by $block_1$ and $block_2$ belonging to two different processes P_1 and P_2 are specified in METROPOLIS by the primitive

mutex (P_1, $block_1$, P_2, $block_2$).

We adopt the Petri net notations $\bullet S$ to denote the set of actions entering a set of states S, and $S\circ$ for the set of actions internal to or leaving S. Using the shorthand notation $A \prec B$ with action sets A and B for the order $\{a \prec b \mid a \in A \wedge b \in B\}$, the mutual exclusion constraint is represented by the priority function $pr_{\mathsf{mutex}} = \{(block_1, \bullet block_2 \prec block_1 \circ), (block_2, \bullet block_1 \prec block_2 \circ)\}$. By abuse of notation, $block_1$ and $block_2$ here denote the sets of control states in the timed system representation of the meta-model statements $block_1$ and $block_2$. Whenever P_1 is in $block_1$, and P_2 is ready to enter $block_2$, then P_1 must leave $block_1$ before P_2 can enter $block_2$, and vice versa. This order is expressed by the priority function pr_{mutex} disabling actions entering $block_2$ (resp. $block_1$) whenever some transition of a process in $block_1$ (resp. $block_2$) will eventually be enabled.

Await. The **await** statement of the meta-model is a powerful means to specify synchronization between processes. Its syntax is

await {
  ($guard_1$; $testList_1$; $setList_1$) $statements_1$;
  ($guard_2$; $testList_2$; $setList_2$) $statements_2$;
  ⋮
}

where $guard_i$ are predicates, and $testList$ and $setList$ define a subset of the interface methods of media to which the object in which the **await** statement resides,

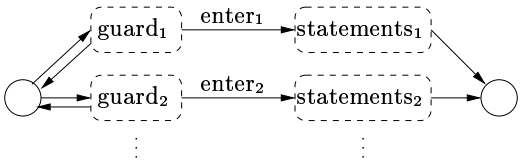

Fig. 4: Modeling the **await** statement.

is connected. $statements_i$ is a block of statements. Intuitively, the semantics is as follows: if a process P comes to the **await** statement, then it can enter the critical section $statements_i$ provided that $guard_i$ is verified, and no other process is in one of the interface methods given in $testList_i$. As long as P is in $statements_i$, no other process can enter any of the interface methods specified in $setList_i$.

One possibility to represent **await** would be using explicit synchronization on semaphores. However, this solution would lead to a model that is difficult to analyze compositionally, especially if other requirements of coordination or timing constraints need to be taken into account. We have therefore chosen to represent the behavioral restriction imposed by **await** entirely by dynamic priorities. The control structure of the **await** statement is represented by the piece of automaton shown in fig. 4. The behavioral restriction is modeled by the priority function

$$\begin{aligned} pr_{\mathsf{await}} \;=\; \bigoplus\nolimits_i \big(\{ guard_i, \quad enter_i \prec testList_i \circ \} \oplus \\ \{ statements_i, \quad \bullet setList_i \prec statements_i \circ \} \big) \end{aligned}$$

saying that from the control states of the timed system modeling $guard_i$, internal and return transitions from one of the methods in $testList_i$ have priority over the transition entering $statements_i$; similarly, in any state that is part of $statements_i$, transitions issued from there have priority over the transitions calling a method in $setList_i$. It can be shown that **mutex** and **await** statements do not introduce any deadly embrace between processes if the priority functions modeling the statements are consistent. Intuitively, a deadly embrace between processes comes from a cyclic waiting relation, which manifests as a circuit in the composition of the priority orders associated with the deadlocking process states.

Thus, a meta-model specification using the currently supported meta-model constructs is automatically translated into a timed system of processes with priorities, which can be analyzed by PROMETHEUS. The compositional modeling methodology discussed in [5, 12] allows to combine parallel composition and dynamic priorities to build live systems from live components. To this end, we define three structural properties whose conjunction is a sufficient condition for liveness, and that are preserved under parallel composition and the restriction with a well-defined priority function. Informally, a timed system is

- *structurally non-zeno* if in any circuit of the discrete transition graph at least one clock is reset, and it is tested against some positive lower bound.

Structural non-zenoness implies that there is a positive lower bound to the execution time of any circuit;
- *locally timelock-free* if from any state, time can pass, or some transition is enabled. Local timelock-freedom excludes the physically unsound behavior where time progress is blocked, and is guaranteed by our model;
- *locally livelock-free* if for any control state, the post-condition of any entering transition implies that some outgoing transition will eventually become urgent.

These properties ensure common-sense requirements relying time progress and occurrence of events in the timed system: time must always diverge (local timelock-freedom); only a finite number of events can occur within some finite amount of time (structural non-zenoness); the system will always progress (local livelock-freedom). *Structural liveness* is defined as the conjunction of the three properties. For example, the timed system of fig. 2 is structurally live if $0 \leqslant E \leqslant T \wedge\ T > 0$.

If the timed systems $TS_1, \ldots, TS_n$ obtained by the above translation satisfy one of the structural properties, then their parallel composition $TS = \|\{TS_1, \ldots, TS_n\}$ satisfies the same property. If the priority function $pr = pr_{\mathsf{mutex}} \oplus pr_{\mathsf{await}} \oplus pr_{\mathsf{pol}}$ modeling coordination is well-defined, then the same structural property is still verified by (TS, pr) [5, 12]. The goal of compositional modeling in METROPOLIS is therefore to obtain a meta-model specification that is translated into a set of structurally live timed systems, such that liveness of the composed system is guaranteed by the compositionality results. If any of the properties is not verified by the PROMETHEUS model, this may indicate an unwanted behavior of the meta-model specification. PROMETHEUS checks the structural properties on each timed system, and outputs diagnostics. In case local livelock-freedom is not verified on some timed system, PROMETHEUS propagates the specified timing constraints over the transitions of this component, and checks again. For instance, when specifying a process as in fig. 2, it is possible to constrain only the period and execution time of a process, having the transition *run* automatically restricted by the constraint when execution must begin in order to keep the process live.

5 Case Study: TinyOS

TinyOS [10] is an extremely small (less than a kilobyte) foot-print operating system for embedded systems that provides basic functionality, such as task scheduling, message passing, and interrupt handling, and supports a modular structure of the application. TinyOS has been conceived to run on small, inexpensive embedded computers called nodes, which can be distributed over a building, measure parameters such as temperature, and communicate with each other over a low-power wireless device. Since direct communication is only possible over short distances, most nodes cannot communicate with each other directly. A sensing and routing application running under TinyOS on each node is in charge of periodically requesting data from a sensor, transmitting this data, and routing incoming messages towards their destination.

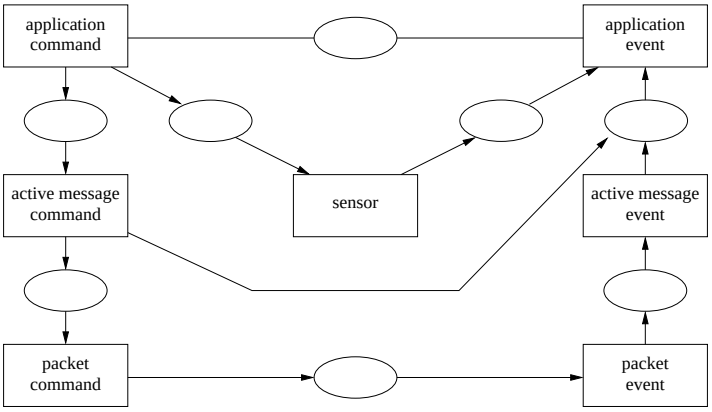

Fig. 5: Simplified METROPOLIS model of the TinyOS application.

A TinyOS application consists of a set of modules that interact through two types of communication: commands, and signaling of events. Both are non-blocking; command invocation only initiates the command execution and returns control to the caller. A TinyOS application therefore has a high degree of logical concurrency. Due to the boundedness of buffers, commands can be refused. In the application we consider, modules form a layered structure: higher-level modules call methods of lower-level modules, which in turn signal events to higher-level modules. In spite of the comparatively small size of the application, verifying its correctness with respect to properties such as liveness is non-trivial. Although TinyOS is not a real-time operating system, its modeling and verification under PROMETHEUS allows to check consistency of the safety constraints and structural liveness, and to generate dynamic priorities in order to simulate its behavior for different timing assumptions and scheduling policies.

We have modeled the sensing and routing application running on one node, in METROPOLIS by 7 processes and 8 media forming a protocol stack as shown in fig. 5. According to the METROPOLIS meta-model, processes (represented by rectangles) communicate by calling methods implemented by media (ovals). In TinyOS, commands and events are handled within the same module, with events preempting the execution of commands. Since the METROPOLIS meta-model assumes exactly one thread per process, and thus excludes intra-process preemption, we have chosen to render the TinyOS semantics by modeling each TinyOS module where command processing can be preempted by events, by two processes: one for executing commands, the other for processing events. Arrows in the figure indicate the direction of command invocation and event signaling of the TinyOS application: *application_command* periodically requests data from the *sensor* module that signals the sampled data to *application_event* as soon as the data are available. *application_command* then broadcasts these data through the protocol stack. Similarly, incoming messages are signaled through the stack

```
process Packet_event {
  port handles rx_byte_ready, tx_byte_ready;
  port signals rx_packet_done, tx_packet_done;
  port p_shared ps;

  void thread() {
    while(true) {
      block(arrival) {
        await {
          (tx_byte_ready.event();;) {
            tx_packet_done.signal();
            tx_byte_ready.clear();
          }
          (tx_byte_ready.event() && !ps.tx_bytes();;) {
            ps.set_tx_bytes (true);
            tx_byte_ready.clear();
          }
          (rx_byte_ready.event();;) {
            block(d1) {}
            rx_byte_ready.clear();
          }
          (rx_byte_ready.event();;) {
            block(d2) {}
            rx_packet_done.signal();
            rx_byte_ready.clear();
} } } } } }
```

Fig. 6: Process `packet_event`.

to *application_event*, and forwarded by *application_command*. Our METROPOLIS model simplifies the actual TinyOS application in that the medium between *packet_command* and *packet_event* abstracts away lower levels of the protocol stack. The description consists of about 700 lines of meta-model code.

Figure 6 shows the functional METROPOLIS model of process `packet_event`. The process declares five ports over which it can communicate with the media to which it is connected. The interfaces `handles`, `signals`, and `p_shared` provide methods for handling an incoming event, signaling an event to another process, and sharing information with `packet_command`, respectively; their specification is not shown here. The thread of the process repeats an **await** statement with empty *testList* and *setList* in a loop so as to react to incoming events. The **await** guards are not pairwise disjoint; this non-determinism comes from the fact that abstraction has been made from some TinyOS data variables in our model. For example, on a `tx_byte_ready` event signaling that the transmission of a byte has been completed, `Packet_event` reacts either by signaling the successful transmission of a packet (if it was the last byte of the packet) to the layer above, or — if no byte is currently being transmitted — it requests through the medium `p_shared` the transmission of the next byte (`ps.set_tx_bytes(true)`). In both cases, the event is then cleared (`tx_byte_ready.clear()`). The last two clauses of the **await** statement have a similar meaning for the reception of bytes. The blocks `arrival`, `d1`, and `d2` are annotated with timing constraints in the sequel of the model. Such timing constraints, in terms of **maxrate** and **maxlate**, specify the minimal inter-arrival times of the processes, worst-case execution times of the blocks using the CPU, and latencies of the media.

The actual role of TinyOS, that is, scheduling the computation within different modules and their communication with each other, is modeled by a PROMETHEUS scheduler. It declares in which blocks of the model the CPU is used, and gives priority to event handling over command processing as to accessing the CPU.

The meta-model specification and the scheduler are then translated and analyzed by the PROMETHEUS back-end. The meta-model processes are represented by timed systems having between 13 and 34 control states, and up to 41 transitions each. Fig. 7 shows the timed system modeling the process *packet_event*; tim-

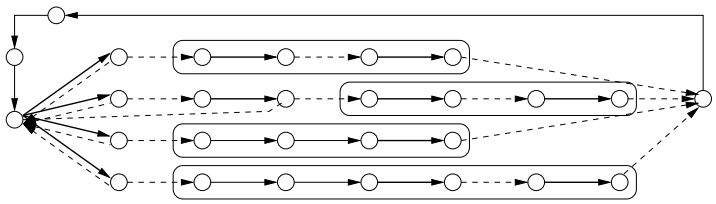

Fig. 7: Timed system modeling the process *packet_event*.

ing information has been omitted. Fat transitions indicate method calls, dashed transitions return from a method call. Boxes show the critical sections of the **await** statement. Its four clauses represent reactions depending on which event has occurred. The guards consist of method calls. Since data are not distinguished, the return transitions nondeterministically enter the critical section, and loop back to the initial state of the **await** statement. The meta-model does not specify how often the guards are evaluated; in our timed system model, an arbitrary lower bound on the delay between two evaluations of the same guard makes the loop structurally non-zeno.

The coordination constraints expressed by the **await** statements, mutual exclusion between the blocks using the CPU, and the scheduling policy are translated into 43 priority functions, and their consistency is verified. By applying the compositionality results, PROMETHEUS determines the model to be deadlock-free, non-zeno, and safe with respect to the mutual exclusion constraints on the CPU. However, there exist states in which a process can potentially stay forever since no timing constraints require it to eventually take a transition. Consequently, the model is not livelock-free. In fact, we have specified minimal, but no maximal inter-arrival times for the processes. After adding appropriate **maxlate** constraints in the METROPOLIS model, PROMETHEUS reports the processes to be structurally live. The timed systems modeling media are still not locally livelock-free: according to their role in METROPOLIS, media passively wait to be called. Applying compositional analysis, the composed system is reported to be deadlock-free, but structural liveness cannot be assured due to media not being locally livelock-free. This problem has been resolved in a later version of PROMETHEUS where methods are not modeled as timed systems of their own, but inlined by the calling process.

In spite of tight synchronization between processes and media, the product timed system would have more than half a billion control states, making non-compositional verification a hard problem. PROMETHEUS completes the verification above in less than 5 minutes. Verification of the structural liveness properties is done by verifying them on each component, and applying the compositionality results. The main source of complexity is the check for safety with respect to the **mutex** and **await** constraints. In contrast to formalisms where safety properties are ensured for example by using semaphores, modeling safety properties using priority functions relies on the fact that transitions violating the property, are

disabled by dominating actions. *Structural safety* [12] is a sufficient condition for safety with respect to the invariance of a set of control states, which can be checked compositionally. In contrast to the check for structural liveness, its complexity grows polynomially with the number of transitions in the processes, and with the size of the priority function.

6 Discussion

We have presented a framework and tool support for compositional modeling and analysis of METROPOLIS models. The integration of PROMETHEUS in METROPOLIS constitutes a modeling and verification platform for the application of existing, and the development and experimentation of new methods for compositional modeling and analysis. The METROPOLIS meta-model is a particularly interesting case for compositional modeling and analysis: on the one hand, its expressiveness makes non-compositional verification difficult or infeasible, especially when a higher degree of refinement has been reached. Compositional verification seems a natural way to incrementally verify the model as it is progressively refined. On the other hand, the philosophy of *separation of concerns* encouraged and in part enforced by the meta-model helps making compositional modeling applicable.

PROMETHEUS translates a METROPOLIS model into a timed system of processes and a priority function, keeping descriptions about the components and their coordination separate. The obtained model allows to apply compositionality results in order to verify consistency, safety, and liveness properties of the model. The priority function helps to understand the modeled system and predict its behavior. Future effort is likely to aim at exploiting this rich source of information to verify more, also quantitative, properties of the model.

The TinyOS case study has shown the power of compositionally verifying the liveness and soundness of a non-trivial example, but also current limitations. Future work will explore two complementary directions to obtain stronger compositionality results for "difficult" properties such as schedulability or individual liveness of the processes in the system: first, to extend and generalize compositionality results, for example by applying assume-guarantee-reasoning [14], and by disposing of more information about the interaction between components, e.g. by typing their behavior using *interface automata* [11]. Second, to develop *modeling guide-lines* to enable compositional reasoning: in order for the results to fully apply, help in the form of an adapted programming style is needed. The meta-model has been designed so as to support a variety of design styles, allowing to adopt compositional modeling principles. Some guide-lines are already provided by existing compositionality results, for example, to model coordination in a declarative way (e.g., using **await**) rather than by an operational restriction such as semaphores.

Acknowledgment. The authors would like to thank the referees for their constructive comments.

References

1. 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"*, 23(1/2):55–84, 2002.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. J. Baeten, J. Bergstra, and J. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, IX(2):127–168, 1986.
4. F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Y. Watanabe, and G. Yang. Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model. In *Proc. CODES'02*, 2002.
5. S. Bornot, G. Gössler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS'00*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.
6. S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163:172–202, 2000.
7. 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. Emerson and A. Sistla, editors, *Proc. CAV'00*, volume 1855 of *LNCS*, pages 543–547. Springer-Verlag, 2000.
8. J. Burch, R. Passerone, and A. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Proc. 2nd International Conference on Application of Concurrency to System Design*, 2001.
9. J. Camilleri and G. Winskel. CCS with priority choice. *Information and Computation*, 116(1):26–37, 1995.
10. D. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. In T. Henzinger and C. M. Kirsch, editors, *Proc. EMSOFT'01*, volume 2211 of *LNCS*, pages 114–130. Springer-Verlag, 2001.
11. L. de Alfaro and T. Henzinger. Interface theories for component-based design. In T. Henzinger and C. M. Kirsch, editors, *Proc. EMSOFT'01*, volume 2211 of *LNCS*, pages 148–165. Springer-Verlag, 2001.
12. G. Gössler. *Compositional Modelling of Real-Time Systems — Theory and Practice*. PhD thesis, Université Joseph Fourier, Grenoble, France, 2001.
13. G. Gössler. PROMETHEUS — a compositional modeling tool for real-time systems. In P. Pettersson and S. Yovine, editors, *Proc. Workshop RT-TOOLS'01*. Technical report 2001-014, Uppsala University, Department of Information Technology, 2001.
14. L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems*, 5:190–222, 1983.
15. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.
16. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E. Mayr and C. Puech, editors, *STACS'95*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag, 1995.
17. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.