Petri nets as a means to validate an architecture for time aware systems

Francesco Fiamberti, Daniela Micucci, and Francesco Tisato

D.I.S.Co., University of Milano-Bicocca, Viale Sarca 336, 20126, Milano, Italy
{francesco.fiamberti, daniela.micucci, francesco.tisato}@disco.unimib.it

Abstract. Time aware systems claim for an explicit representation of time-related concepts, so that they can be observed and possibly controlled at run-time. The paper identifies a set of architectural abstractions capturing such concepts related to time and classifies the base activities performed by a time aware system. Our proposal has been formalized using two different modeling techniques: UML and Petri nets. The former has been chosen to model the static structure of both the abstractions and the entities performing time aware activities. The latter have been exploited to model the dynamics of a time aware system.

Keywords: real-time, architectural abstractions, UML, Petri nets

1 Introduction

Time aware systems [1] deal with *time-related* issues when accomplishing domain-related tasks. For example, a time aware system includes activities whose activation is *time driven*, activities that need to reason on *timestamped facts*, and activities that need to know *what time it is*. Therefore, time ought to emerge as a first-class concept because of its relevance in the application domain.

As stated in [2], only recently model-based development has begun focusing on timing aspects of a system in addition to its functional and structural ones. However, the proposed approaches tend to specify the requirements with respect to timing focusing on a specific solution. Moreover, the satisfaction of timing requirements is verified only during the test phase of the development process. As stated in [3], this is due to the fact that model-driven approaches applied to embedded systems in early design phases do not always rely on a systematic and rigorous methodology that includes specifying and verifying timing requirements.

To overcome the above drawbacks, several modeling techniques have been proposed. MARTE [4] is an UML profile designed to face real-time aspects of a system from a model-based perspective. UML [5] is a standardized general-purpose modeling language used to specify the artifacts of a software system. A UML profile customizes UML for a specific purpose or domain by using extension mechanisms able to modify the semantics of the meta-model elements [5]. [6] exploits MARTE (the logical time concept) and the CCSL language [7] to specify the causal and temporal characteristics of the software as well as the hardware

parts of the system. However, modeling capabilities need to be supported by tools that directly implement the system.

Languages like Giotto [8] and SIGNAL [9] extend existing paradigms to include time-related issues. Close to Giotto, PTIDES [3] is a programming model for distributed embedded systems based on a global, consistent notion of time. However, such approaches allow time-related issues to be managed at compile time only, preventing the temporal behavior of the system from being adaptive.

The key idea behind our proposal is that time-related concepts should be first-class concepts, which directly turn into basic architectural abstractions [10] supported by a running machine. In this way, it is possible to explicitly treat time-related aspects from the analysis of the requirements to the test phase of the life cycle of a system. Even if UML [5] is a well-known modeling language in the software engineering area, the features of Petri nets [11] make them a suitable tool to model the dynamics of a time aware system. Indeed, when a set of time driven entities must be performed, they must be enabled in a deterministic way. However, once they have been enabled, their actual execution can be nondeterministic, that is, their execution order should have no significance and should not affect the system behavior. Therefore, we used UML to describe the architectural abstractions and the time-related activities (static structure), and Petri nets to describe the dynamics of a time aware system.

The paper is organized as follows. Section 2 introduces time-related abstractions by means of UML class diagrams. Section 3 identifies the three base entities performing time aware activities by means of UML class and state diagrams. Section 4 discusses the dynamics of a time aware system exploiting Petri nets. Finally, Section 5 presents concluding remarks.

2 Time-related abstractions

To give a flavor of the proposed model, the following simplified example will be used. Consider a road gate equipped with a camera. The gate provides access to an area with traffic restrictions. In particular, car transits are only allowed at night and for a restricted set of vehicles. Every second, the camera must acquire a frame, which must be stored with the acquisition timestamp. Finally, the acquired frames are elaborated offline to detect infractions.

The described scenario is an example of a time aware system. A *time aware* system reifies the following time aware activities:

- a *time driven* activity is triggered by events that are assumed to model the flow of time. In the proposed example, the acquisition activity must be time driven in order to acquire frames at predefined time instants.
- a *time observer* activity observes "what time it is". Thus, the acquisition activity in the example must also be time observer, since it needs the correct timestamp for every acquired frame.
- a *time conscious* activity reasons on facts placed in a temporal context, no matter when the computation is realized. In the example, the infraction

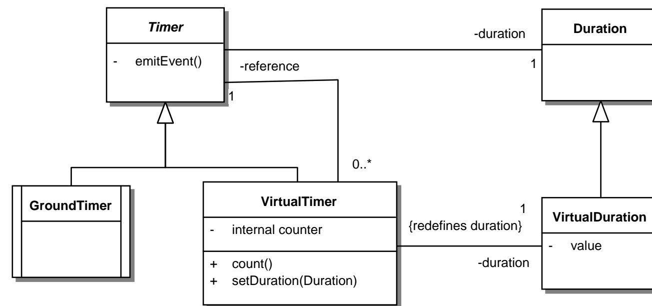

Fig. 1: Concepts related to timers

detection activity is time conscious, as it performs an offline elaboration of timestamped frames.

Drivenness, *observability*, and *consciousness* can be enabled by means of three well distinguished architectural abstractions: *Timer*, *Clock*, and *Timeline*.

2.1 Timer

A *Timer* is a cyclic source of events, all of the same type: two successive events define a *duration*. A timer generates events by means of its *emitEvent* operation.

A *Virtual Timer* is a timer whose event generation is constrained by the behavior of its *reference* timer: it counts (by means of the *count* operation) the number of events it receives from its reference timer and generates an event when this number equals a predefined *value*. The duration is specialized to *virtual duration*. Timers can thus be arranged in hierarchies, in which every descendant timer has exactly one reference timer. The root of every hierarchy is a *Ground Timer*, which is a timer whose duration is not constrained by the duration of another timer. Therefore, the duration of a ground timer can be interpreted as intervals of the real external time, so that the events generated by a ground timer can be interpreted as marking the advance of time. Finally, the *setDuration* operation allows the duration of a virtual timer to be modified, thus varying the speed at which events are generated. Figure 1 sketches the described concepts.

2.2 Clock

A *Clock* counts (by means of its *increment* operation) the events it receives from the associated timer. The event count is interpreted as the clock's *current time* (see Figure 2). Thus, time is not a primitive concept but it is built from events.

2.3 Timeline

A *Timeline* is a data structure (thus intrinsically discrete) constituting a static representation of time as a numbered sequence of *grains*. A grain is an elementary unit of time identified by its *index* and whose interior cannot be inspected.

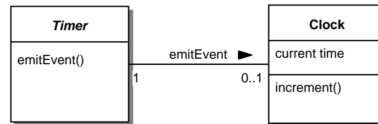

Fig. 2: Concepts related to clocks

A *Time Interval*, defined on a timeline, is a subset of contiguous grains belonging to that timeline. A *virtual timeline* is a timeline whose grains (*virtual grains*) have a *duration* that can be expressed as a time interval in the associated *reference* timeline. Timelines can thus be arranged in hierarchies. The root of every hierarchy is a *Ground Timeline*, which is a timeline whose grain durations are not constrained by the grains of another timeline. In each hierarchy, the ground timeline is therefore the only one whose grains can be interpreted as an elementary time interval in an arbitrary ground reference time (e.g., the "real" time from the application viewpoint).

A *Fact* is an assertion regarding the system domain. A *Timed Fact* is a fact associated to a time interval representing the fact's interval of validity. Therefore, timelines are histories of timed facts. Figure 3 sketches all the described concepts.

By connecting a clock with a timeline, it is possible to interpret as *present time* on the associated timeline the grain whose index equals the clock's current time (see Figure 4). Every time the clock receives an event from the connected timer, it advances the present time on the corresponding timeline by one grain. The clock also defines the concepts of *past* and *future* in the associated timeline: the grains with *index* less than *current time* belong to the past and the grains with *index* greater than *current time* belong to the future.

3 Time aware entities

The identified abstractions enable the design of the following *time aware entities* (see Figure 5), which reify the activities of a time aware system:

- *Time driven entity*: an entity whose activation is triggered by a virtual timer
- *Time observer entity*: an entity that reads current time from clocks
- *Time conscious entity*: an entity that reads/writes timed facts on a timeline without any reference to when such a management is actually realized

More articulated behaviors can be obtained by combining the three basic entities. For example, a *time driven time conscious entity* is an entity that is triggered by a virtual timer (time driven) and reads/writes timed facts (time conscious).

3.1 Time driven entities

A time driven entity is associated to its *activating* timer, and it may be in two states: running and idle. A time driven entity enters the *running* state when its

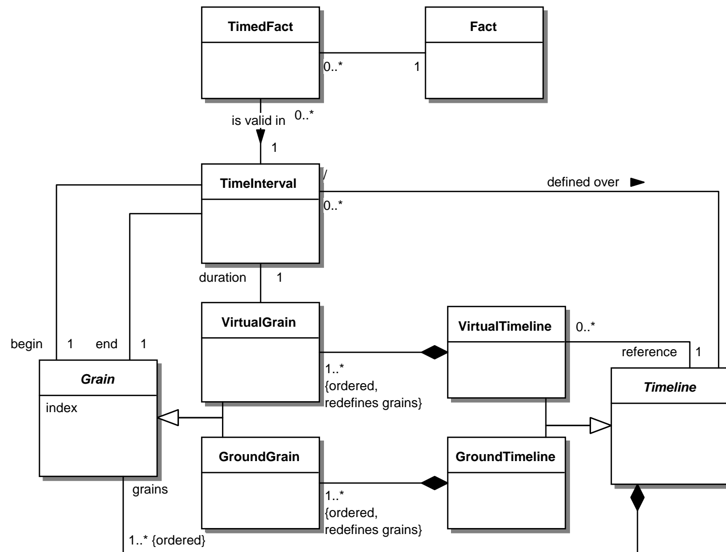

Fig. 3: Concepts related to timelines

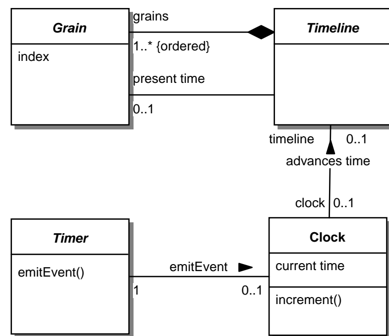

Fig. 4: Connection of a clock to a timeline

activating timer emits an event. In this state, it performs its domain-dependant operation. At the end of the execution, the entity goes back to the *idle* state.

This simple model assumes that the deadline for an execution coincides with the beginning of the next execution. To adapt the model to the general case where deadlines temporally precede the beginning of the next execution, it is possible to associate a second timer to each time driven entity, as sketched in Figure 6. When the deadline timer emits an event, the associated time driven entity must have already completed the *perform* operation. It follows that a time driven entity must include an additional state, denoted *terminated*.

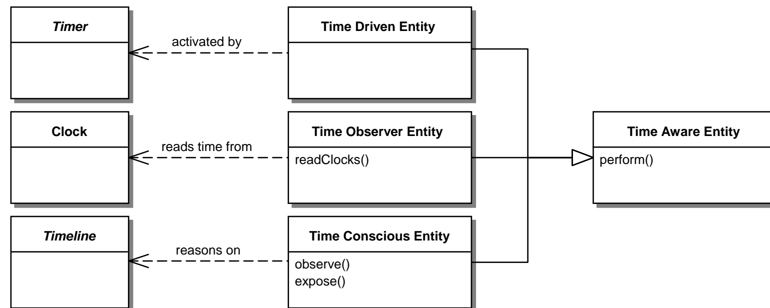

Fig. 5: Entity classification according to the relation with the basic concepts

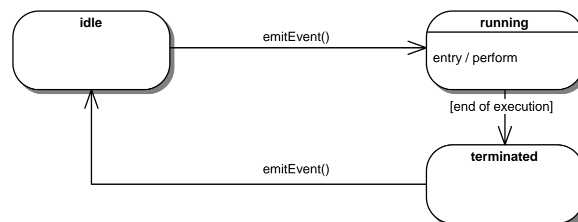

Fig. 6: State diagram for a time driven entity

3.2 Time driven time conscious entities

Some care must be used to guarantee consistency when designing entities that are both time driven and time conscious. In fact, it is desirable that the behavior of all the entities that are triggered simultaneously does not depend on the order in which the executions are actually managed, which may be affected by low-level details such as the number of available cores or the particular scheduling algorithm that is being used. Therefore, it is necessary to guarantee that all the time driven time conscious entities that are triggered simultaneously share the same view of the timelines, to avoid the situation of an entity that reads timed facts written by another entity triggered simultaneously just because the latter was granted higher execution priority by the low-level scheduler.

A possible solution is that all entities read timed facts immediately when they are activated by the activating timer and write timed facts only when they receive an event by the deadline timer, even if the actual execution ends before the deadline. The state diagram in Figure 7 enriches Figure 6 by introducing effects in the transitions triggered by timers: the effect of an event from the activating timer is the reading of facts by means of the *observe* operation, whereas the effect of an event from the deadline timer is the writing of facts by means of the *expose* operation. In an actual implementation, the concrete component in charge of the execution of entities must guarantee that when the execution of a set of entities is triggered, all the entities read timed facts before any one of

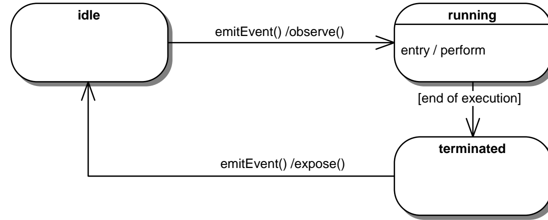

Fig. 7: State diagram for a time driven time conscious entity

them is allowed to start the actual execution, and that every entity writes timed facts only at the deadline for its execution.

4 Time aware systems

This section presents the behavior of time aware systems exploiting Petri nets. First, the dynamics of a timer hierarchy will be discussed. Afterwards, the model of the activation of time driven and time conscious entities will be presented.

4.1 Timer hierarchy

Before describing a complete timer hierarchy, we will detail the internal behavior of a virtual timer. Figure 8 shows the subnet modeling the timer `T2`, of duration 3, without descendant timers. In the initial marking, a token is present in place `p1`. The first time a token is put into place `Event to T2`, transition `t1` is enabled and fires, putting a token into both `p2` and `T2 updated`. The structure made of `p*` and `t*` works as an internal counter, and the token to `T2 updated` is needed to allow the external system to be notified that the timer completed its update operations (simple increment of the internal counter or event generation). When the internal counter equals the timer's duration, transition `Emit event` is fired. At the end, a token is put into place `T2 updated`. If a clock is connected to the timer, after every generated event, the corresponding transition `increment clock time` is fired before the end of the timer's update. Note also that the gray arcs in Figure 8 are required for the correct behavior of the Petri net, but do not have any particular time-related semantics. For example, the arcs from the place `Event to T2` to transitions `t1`, `t2` and `t3` allow such transitions to fire only one at a time, when a token is present in place `Event to T2`.

Figure 9 shows the subnet of `T1`, a timer with descendants (`T3` and `T4`). Unlike in the previous case, transition `Emit event` puts tokens into all the places `Event to T*` of the descendant timers, whose places `T* updated` are joined in transition `Join descendant timers`, after which the system's behavior is the same as in the case without descendant timers. No assumptions are made on the order in which descendant timers are updated.

Figure 10 shows a Petri net modeling a four-level hierarchical timer structure. Place `Ground timer event` receives a token that is interpreted as the flow

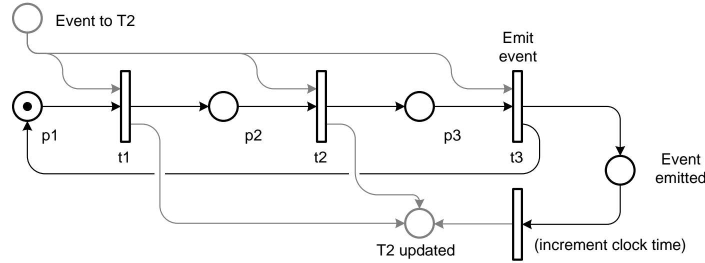

Fig. 8: Subnet for a timer with no descendants

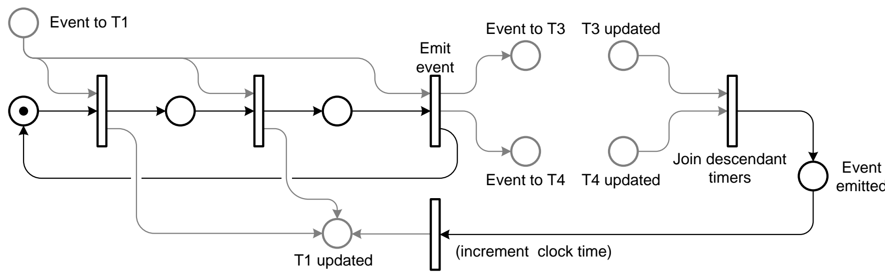

Fig. 9: Subnet for a timer with descendants

of real time. If the previous update of the system has been completed (a token is present in place `All timers updated`), transition `Start execution` is enabled. When the transition is fired, a token is sent to place `Event to T*` of every virtual timer directly connected to the ground timer, enabling internal update operations. When a timer emits an event, its update can terminate only when all its possible descendant timers' updates have been triggered and completed. Once the timer has been updated, a token is put into the corresponding place `T* updated`. Places `T* updated` are joined in transition `Join direct descendants of ground timer`, whose firing terminates the atomic update of all timers by putting a token into place `All timers updated`. This token enables transition `Start execution` when the next token is produced in place `Ground timer event`. Thanks to the recursive structure of virtual timers, only direct descendants of the ground timer need to be joined in transition `Join direct descendants of ground timer` to ensure atomicity of all timers' updates. In Figure 10, for clarity the details regarding the internal structure of timers have been hidden and represented by means of simple transitions shown in gray.

4.2 Time driven entities

Time driven entities can be executed by associating them to timers. Every time a timer emits an event, it sends a signal to all its associated entities, which behave consequently according to their internal state. To make the concepts clear, we

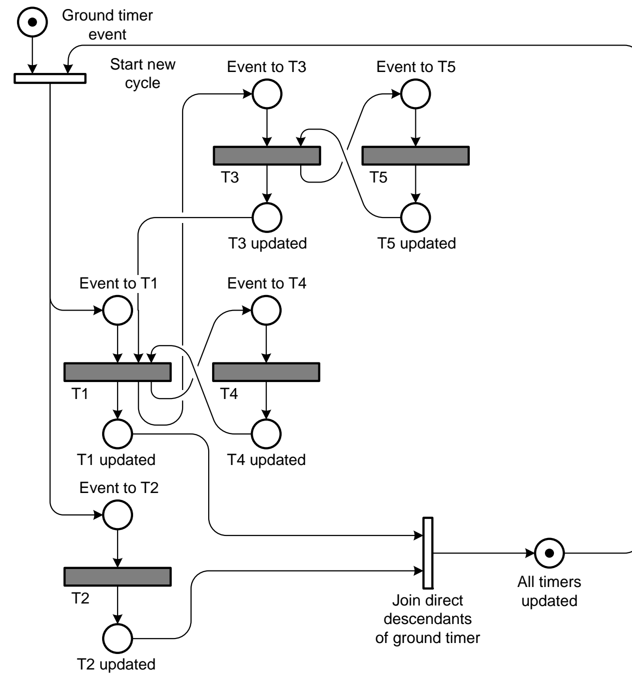

Fig. 10: Example of hierarchical timer structure

assume that the relative deadlines of the time driven entities coincide with the beginning of the next execution. Figure 11 sketches an example of such a system, where timers `T3`, `T4`, and `T5` have been hidden for the sake of readability. The assumption is made that the execution of a time driven entity is an instantaneous action. Thus, if the actual time taken by an entity is not negligible with respect to the smallest time scale in the system, the execution considered here is made only of the (instantaneous) set of operations needed to decouple the actual actions from the main system flow (e.g., the operations needed to start a new thread where threads are available).

Figures 12 and 13 show the structures of timers `T2` and `T1` respectively in presence of time driven entities: when an event is generated, a token is put into place `T* time driven entities to be enabled`, to enable the execution of the entities associated to `T*`. Note that habilitation does not mean immediate execution: the actual execution of all the time driven entities can be started only once all the timers have been updated, as will be explained later. To ensure that the entities have been enabled, a token is required in place `T* time driven entities enabled` for the transition in input to place `T* updated` to fire.

Figure 14 shows how atomicity of all timers'update can be granted. A copy of place `All timers updated` is available for every subnet modeling a group of time driven entities associated to the same timer. Every entity group has a `T* time`

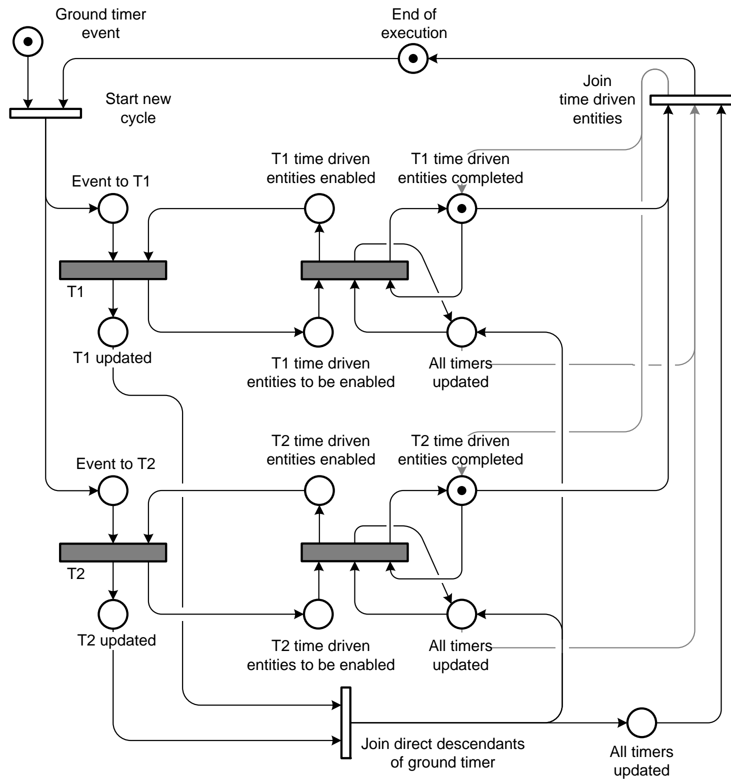

Fig. 11: Time aware system with time driven entities

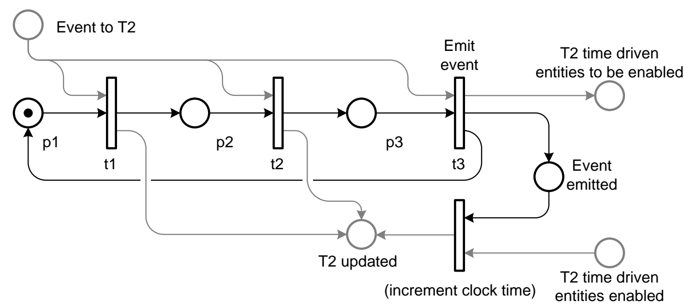

Fig. 12: Subnet for a timer with associated time driven entities

`driven entities completed` place, where the presence of a token indicates that no additional actions are required for the group. This is needed to deal with the (typical) situation where only a subset of all the timers emit an event. All places `T* time driven entities completed` are initialized with a token, and a token

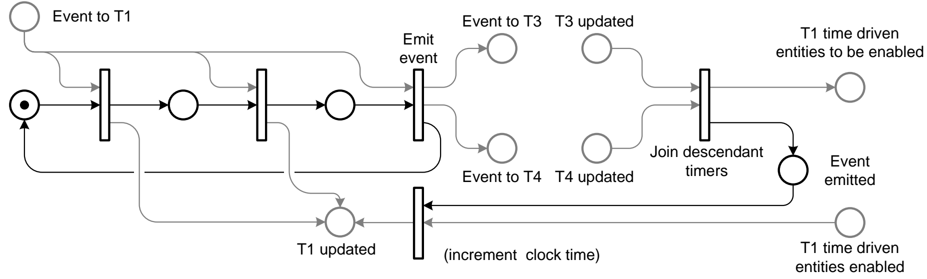

Fig. 13: Subnet for a timer with descendants and associated time driven entities

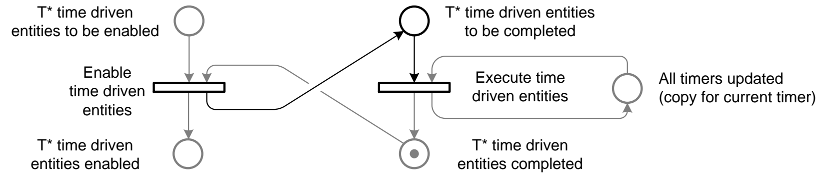

Fig. 14: Model of a group of time driven entities

is always present at the end of every execution. Only in case of an event from the associated timer, place `T* time driven entities completed` is cleared as a consequence of the habilitation of time driven entities. In fact, when a token is put into place `T* time driven entities to be enabled`, transition `Enable time driven entities` fires, removing the token from place `T* time driven entities completed` and putting it into place `T* time driven entities to be completed`. Transition `Execute time driven entities` is not enabled until a token is present in the current timer's copy of place `All timers updated`, that is, until all the timers have been updated. All places `T* time driven entities completed` are joined in transition `Join time driven entities` (see Figure 11). At the end of each global timer update, the time driven entity groups that do not require execution (because their timer did not emit an event) already have a token in place `T* time driven entities completed`.

Transition `Join time driven entities` is not enabled only if some of the time driven entities must still be executed. If this is the case, all the corresponding transitions `Execute time driven entities` are now enabled, removing the token from place `T* time driven entities to be completed` and putting it into `T* time driven entities completed`. No assumptions are made on the possible order in which time driven entities are executed. Once all the executions have been completed, the transition `Join time driven entities` is enabled. Note that in order to prevent an early firing of this transition in the case where no entity groups need to be executed, a copy of place `All timers updated` is present as an input to `Join time driven entities`, so that the global update

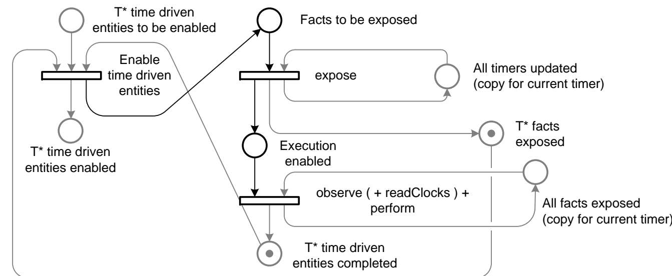

Fig. 15: Model of a group of time driven time conscious entities

of timers must be terminated first even though all the places `T* time driven entities completed` contain a token.

4.3 Time driven time conscious (observer) entities

As stated in subsection 3.2, some care must be used when designing entities that are time driven and time conscious, or time driven, time conscious and time observer. The two cases can be analyzed together, since the differences are limited to the presence of clocks and to the need to read the relevant clocks' current times before starting the executions. Figure 15 contains the model of a group of time driven time conscious entities associated to the same timer, while an example of time aware system is shown in Figure 16.

As introduced in Subsection 3.2, the sequence of operations of a time driven time conscious entity can be organized in three blocks: `expose` (that puts on the right timelines the timed facts computed during the previous execution), `observe` (that reads timed facts from the timelines of interest), and `perform` (the actual execution of the entity's actions). Since no constraints are put on the order in which entities are executed, consistency and predictability of the system's behavior require that timed facts are added on a timeline by an entity only at the end of the time grain of the timer to which the entity is associated, and before any other entity in the system reads facts from the same timelines. So the exposition of all the facts generated by all entities must be realized as an atomic action after all timers have been updated and before the execution of the `perform` of any time driven time conscious entity.

With reference to Figure 15, for every group of entities, place `T* facts exposed` is initialized with a token. When the associated timer emits an event, the token put into place `Time driven entities to be enabled` enables transition `Enable time driven entities`, which removes the tokens from places `Time driven entities completed` and `T* facts exposed`, putting a token into `Facts to be exposed`. Once all the timers have been updated, so that a token is put into the copy of place `All timers updated` for every group of entities, transition `expose` is fired, putting a token into the corresponding places `T* facts exposed` and `Execution enabled` (whose presence prevents unrequested

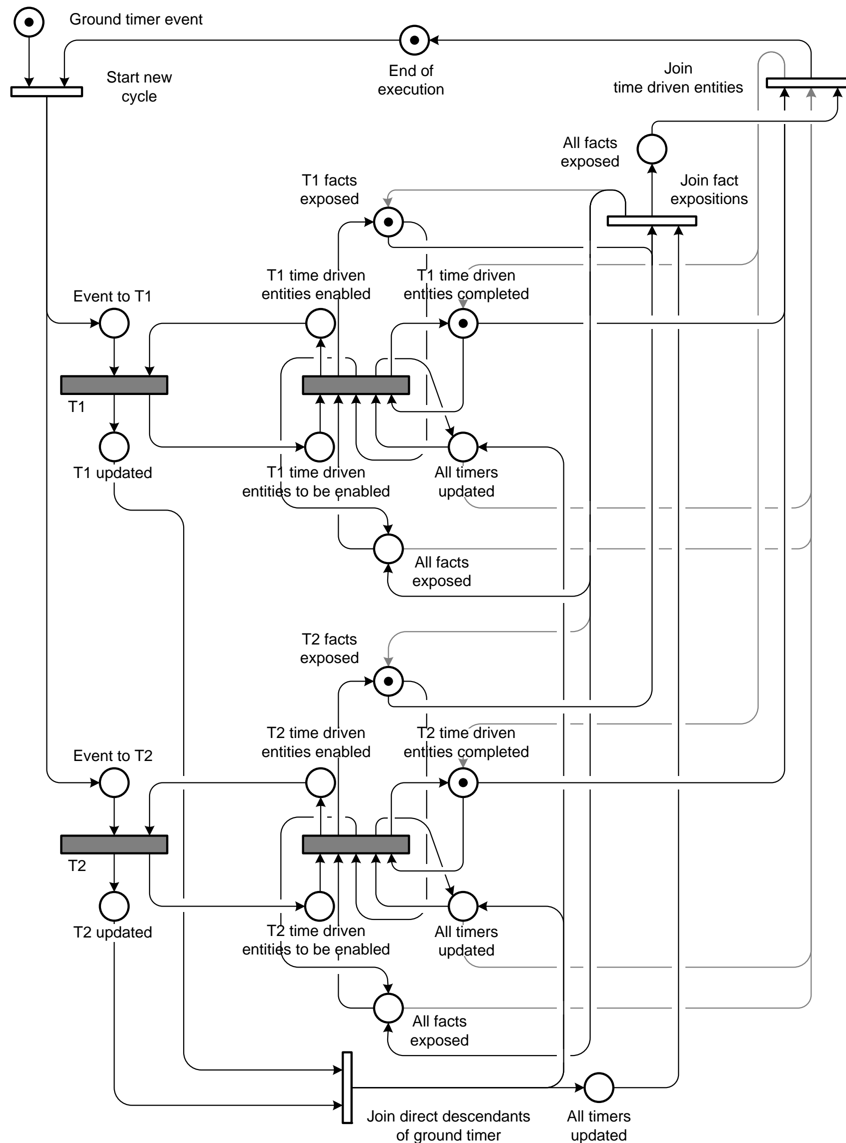

Fig. 16: Time aware system

firing of transition `observe (+ readClocks) + perform` that would be otherwise triggered by the simple presence of a token in place `All timers updated`). All places `T* facts exposed` are joined, together with a copy of `All timers updated`, in transition `Join fact expositions`. Only after all the expositions have been completed, this transition can fire, putting a token into the copy of

Property	Y/N	Property	Y/N
Pure (PUR)	No	Covered by P-invariants (CPI)	Yes
Ordinary (ORD)	Yes	Strongly Covered by T-invariants (SCTI)	Yes
Homogeneous (HOM)	Yes	Structurally Bounded (SB)	Yes
Non-Blocking Multiplicity (NBM)	Yes	Bounded (kB)	Yes
Conservative (CSV)	No	Safe (1-B)	Yes
Structurally Conflict-Free (SCF)	No	Dynamically Conflict-Free (DCF)	Yes
FT0, TF0, FP0, PF0	Yes	No Dead States (DSt(0))	Yes
Connected (CON)	Yes	No Dead Transitions (DTr)	Yes
Strongly Connected (SC)	Yes	Live (LIV)	Yes
Deadlock-Trap Property (DTP)	No	Reversible (Rev)	Yes
Covered by T-invariants (CTI)	Yes		

Table 1: Properties of the proposed Petri nets

place `All facts exposed` for all groups of entities. This enables all transitions `observe (+ readClocks) + perform` (for time observer entities, `readClocks` is executed on all the clocks of interest before `perform`), whose firing puts a token into the corresponding place `T* time driven entities completed`. At this stage, as in the case of pure time driven entities, places `Time driven entities completed` are joined in transition `Join time driven entities`, which fires when all the entity executions have been completed. The gray arcs in Figure 16 involve groups of entities: one connects place `All facts exposed` of each group and transition `Join time driven entities`, to ensure that tokens do not pile up in these places when the corresponding timer does not emit an event. The other is between transition `Join fact expositions` and place `T* facts exposed` of every group, needed to recharge the token for the next execution.

5 Final remarks

The time-related abstractions and the dynamics of a time aware system could have been fully described by means of UML diagrams (i.e., class and state to model the basic abstractions and sequence to model the dynamics). Initially, this was the direction we followed, but we soon realized that the resulting sequence diagrams would be complex and difficult to read. So we decided to use Petri nets, because of their suitability to model the dynamics of a system. The obtained result consists in a set of Petri nets that are simpler and more readable with respect to the corresponding UML sequence diagrams, notwithstanding the need for additional places and transitions that do not have an application semantic but are required for the Petri nets to behave correctly.

Table 1 summarizes the properties of the proposed Petri nets, as defined in [12,13]. Some of the properties cannot be satisfied because of the intrinsic nature of time aware systems (e.g., the proposed nets are not pure because of the presence of loops, which are required to obtain a cyclic behavior).

The proposed models supported the implementation of a Java framework named Time Aware Machine (TAM) [14] that has been used for the experimental testing of the Space Integration Services platform [15]. Currently, the framework is being used in ALARM [16], an architecture based on a time driven mechanism that verifies hypotheses about domain entities against previsions.

References

1. Fiamberti, F., Micucci, D., Tisato, F.: An architecture for time-aware systems. In: 2011 IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA), IEEE (2011)
2. Buckl, C., Gaponova, I., Geisinger, M., Knoll, A., Lee, E.A.: Model-based specification of timing requirements. In: Proceedings of the tenth ACM international conference on Embedded software. EMSOFT '10, ACM (2010) 239–248
3. Zhao, Y., Liu, J., Lee, E.A.: A programming model for Time-Synchronized distributed Real-Time systems. In: 13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07, IEEE (2007) 259–268
4. OMG: MARTE Modeling and Analysis of Real-Time and Embedded systems. http://www.omg.org/spec/MARTE/1.1/PDF/
5. OMG: Unified Modeling Language (UML), Superstructure. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/
6. Peraldi-Frati, M., DeAntoni, J.: Scheduling multi clock real time systems: From requirements to implementation. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on. (2011) 50 –57
7. Mallet, F.: Clock constraint specification language: specifying clock constraints with UML/MARTE. Innovations in Systems and Software Engineering **4**(3) (2008) 309–314
8. Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE **91**(1) (2003) 84–99
9. Gamatié, A., Gautier, T., Guernic, P.L., Talpin, J.P.: Polychronous design of embedded real-time applications. ACM Trans. Softw. Eng. Methodol. **16**(2) (2007)
10. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering **21**(4) (1995) 314–335
11. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
12. Starke, P.H.: Analyse von Petri-netz-modellen. Teubner BG GmbH (1990)
13. Starke, P.H.: INA Integrated Net Analyzer. http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/manual.html
14. Fiamberti, F., Micucci, D., Tisato, F.: An object-oriented application framework for the development of real-time systems. In: 50th International Conference on Objects, Models, Components, Patterns (TOOLS 2012), Springer (2012) 75–90
15. Bernini, D., Fiamberti, F., Micucci, D., Tisato, F.: Architectural Abstractions for Spaces-Based Communication in Smart Environments. Journal of Ambient Intelligence and Smart Environments **4**(3) (2012)
16. Fiamberti, F., Micucci, D., Mobilio, M., Tisato, F.: A Layered Architecture based on Previsional Mechanisms. In: ICSOFT 2013 - Proceedings of the 8th International Joint Conference on Software Technologies (accepted for publication). (2013)