A Model for Time-Awareness

Francesco Fiamberti, Daniela Micucci,
Alessandro Morniroli, and Francesco Tisato

University of Milano - Bicocca, Viale Sarca 336, Milan, Italy
{fiamberti,micucci,tisato}@disco.unimib.it,
alessandro.morniroli@gmail.com

Abstract. Time-aware activities are characterized by a set of time-related aspects, independently from the involved application domain. For example, an activity may need to reason on facts that are held to be true in specific time intervals, or it may need to be executed at precise time instants. In this paper we present a temporal model capturing these concepts and their relations. The model is described by means of an UML formalization, enriched with OCL constraints where needed. The model turns into a set of architectural abstractions that makes time-related concepts visible at the application level. This eases the analysis and implementation of time-aware systems and enables adaptivity so that temporal constraints may be dynamically met.

Keywords: time-aware activities, time, timeline, clock, UML.

1 Introduction

It is widely recognized that a sound software architecture based on a proper set of *architectural abstraction*s ([1] and [2]) can bridge the gap between requirements and specifications [3]. Architectural abstractions are design abstractions representing architectural aspects of a system. Well-known examples of architectural abstractions are the *architectural styles* as detailed in [4].

Domain-specific architectural abstractions capture aspects that are relevant in a specific domain so that they can be treated at the architectural level. Under this perspective, time-aware systems [5] are systems that have to deal with *time-related* issues when accomplishing domain-related tasks. For example, a time-aware system may include activities whose activation is *time-driven*, activities that need to reason on *time-stamped facts*, and activities whose *execution period* must be controllable. Therefore, such kind of systems claim for a set of suitable domain-specific architectural abstractions that allows both the temporal behavior of the system to be represented and time-related concepts to be treated as first class entities at the application level. The abstractions should rely on a sound temporal model aimed at specifying all the significative concepts (including their characteristics and relations) that allow a time-aware system to reason on time-stamped information, to know what time it is, and to observe and control the execution period of its time-driven activities.

M. Bajec and J. Eder (Eds.): CAiSE 2012 Workshops, LNBIP 112, pp. 70–84, 2012.

Different research communities agree that time ought to emerge as a first-class concept because of its relevance in the application domain. From the ontologies' research community point of view, the temporal dimension is central in many information sources. Unfortunately, languages like OWL (Ontology Web Language) [6] and RDF (Resource Description Framework) [7] provide a limited support for enriching data with temporal information. As stated in [8], there are many drawbacks in lacking such a support: a minor expressivity of the temporal aspect of the data, a difficult automatization of the validation process and a reduced temporal expressivity of deductive rules and queries on data. As a result, ontologies often cannot fully express the required temporal knowledge, forcing the adoption of ad-hoc solutions.

Different proposals have been conceived to represent temporal aspects in both OWL and RDF. An up-to-date detailed survey may be found in [8]. Dealing with RDF, many extensions to the model have been proposed so that temporal information can be represented (e.g., [9]). According to such extensions, several query languages have been conceived so that new queries are made available exploiting temporal extensions (e.g., [10]). As stated in [11], temporal description logics are not able to represent temporally-changing information, since they are geared towards synchronic relationships, not diachronic ones. This is one of the reasons why researchers do not try to extend the OWL model, but try to propose extensions on the top that allow temporal information to be modeled. In this direction, an interesting result is Time-OWL ([12] and [13]), an ontology of temporal concepts aimed at describing the temporal content of Web pages and the temporal properties of Web services. The ontology includes topological properties of instants and intervals, measures of duration and the meaning of clock and calendar terms. Even if several researchers have extended the OWL description logic model (e.g., [14] for a survey) by adding temporal modeling, there not exists an agreement of a standard. Moreover, few research has been conducted regarding query languages supporting such extensions. A recent step towards the integration of temporal information and related query support is [8], that proposes a methodology and a set of tools for representing and querying temporal information in OWL ontologies.

From a software engineering perspective, the focus is on various aspects related to the construction of real-time system, ranging from modeling tools to schedulability analysis tools up to concrete frameworks.

As stated in [15], 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 [16], this is principally 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.

MARTE (Modelling and Analysis of Real-Time and Embedded systems) [17] and its predecessor SPTP (Profile for Schedulability, Performance, and Time) [18], are UML profiles designed to face real-time aspects of a system from a model-based perspective. 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 [19]. MARTE models time-related concepts with specific elements like clocks and provides two models that respectively allow to describe the execution platform and the allocation of the functional components to the resources. AADL [20] is an analysis and design language that allows not only to define a representation of the software architecture, but also to formally define the syntax and semantics so that the representation can be verified and validated [21]. Both the approaches, however, only provide modeling capabilities but no embedded tools to directly implement the system. Languages like Giotto [22] and SIGNAL [23] extend existing paradigms to include time-related issues. However, such issues are managed at compile time, preventing the temporal behavior of the system from being adaptive. Similar to Giotto, PTIDES (Programming Temporally Integrated Distributed Embedded Systems) [16] is a programming model for distributed embedded systems based on a global, consistent notion of time. Finally, [24] proposes a modular modeling methodology to specify the timing behavior of real-time distributed component-based applications. It allows building models of the resources and of the software components, which are reusable and independent from the applications that use them.

The key idea behind our proposal is that time-related concepts should be full-edged first-class concepts, which directly turn into basic architectural abstractions 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. More important is that, thanks to abstractions, time emerges at the application level. Therefore, it is possible to build adaptive systems that dynamically change their behavior by observing timed facts, the current time and the execution speed of the activities. Particularly interesting is that a dynamic change of the system behavior may include a modification of the execution speed of its activities and that is driven at the application level.

The paper proposes the model underlying the architectural abstractions by means of UML diagrams enriched with OCL [25] constraints when required. The UML language has been chosen for two main reasons. First, UML is widely used to design concrete systems. Thus, the use of this language already at the analysis phase simplifies the subsequent implementation of a framework reifying the model [26]. Secondly, many timing issues cannot be described by means of languages like OWL (in particular TimeOWL), since they must not convey information to the applications, being related to dynamics only.

The proposed model is applied to a simplified case study dealing with the process of beer making. The case study, even though simple from an applicative point of view, turns out to be significant with respect to the timing requirements that must be met.

The paper is organized as follows: Section 2 introduces the temporal model; Section 3 presents a case study dealing with the artisanal production of beer and how the proposed temporal model may help in realizing some of the major aspects of the problem; finally, Section 4 presents conclusions and outlines future work.

2 The Temporal Model

This section presents the temporal model formalizing its constituting elements, their properties and their relations by means of UML diagrams (both static and dynamic) enriched with OCL constraints when needed.

2.1 Basic Performer Types

*Time Sensitive Performer*s are entities that *perform* domain-related activities and that are someway related to the concept of time. As sketched in Figure 1, *Time Sensitive Performer*s are classified according to the relation they have with the concept of time, and precisely:

- A *Time Conscious Performer* needs to reason on facts placed in a temporal context (*Timed Fact* in Figure 1), without any reference to when such a reasoning is actually realized. The typical example of a *time conscious* activity is the off-line analysis of time-stamped data
- A *Time Observer Performer* needs to read current time from a clock (*Clock* in Figure 1). Again, this property is in no way related to the mechanism that triggers the execution of the activity. As an example, any activity that samples and stores data with a timestamp is a *time observer* activity
- Finally, the only specialization that concerns activation mechanisms is the *Time Driven Performer*, reifying a generic activity whose execution is triggered at specified time instants. This definition includes both periodic and

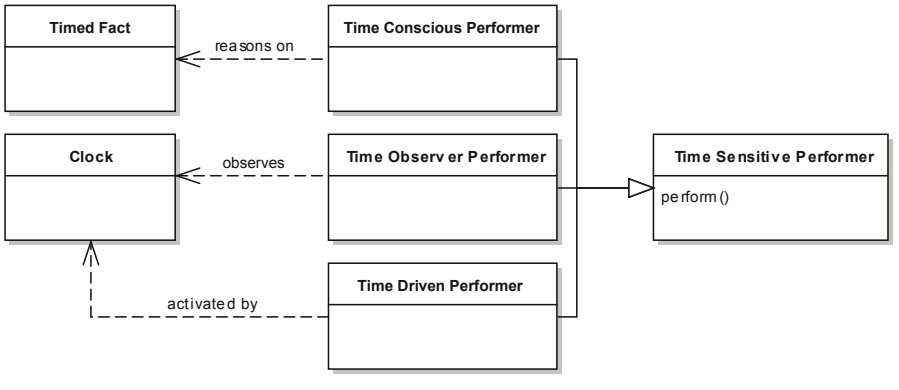

Fig. 1. Performers and their relations with time concepts

aperiodic activations. An example of *time driven* activity is the periodic sampling of data by a sensor.

The following subsections present the time-related abstractions needed to fulfill the requirements all the basic performer types claim.

2.2 Time Consciousness

In any actual software system, time is necessarily discrete. The indivisible time quantum is named *Grain*. A *Timeline* is a sequence of grains, labeled by consecutive integer numbers. The number associated with each grain is stored in the *value* attribute. A *Time Interval*, defined on a timeline, is a subset of consecutive grains belonging to that timeline.

Being *Fact* a domain-dependant data, a *Timed Fact* is a fact associated to a time interval representing the fact's temporal context. All the introduced concepts are sketched in Figure 2.

The concepts related to time consciousness would be suitable to be represented by means of TimeOWL. However, using two formalisms to describe the model we propose would generate confusion. Therefore, UML has been chosen as the single modeling language, since, as previously anticipated, it is able to express all the concepts underlying the model.

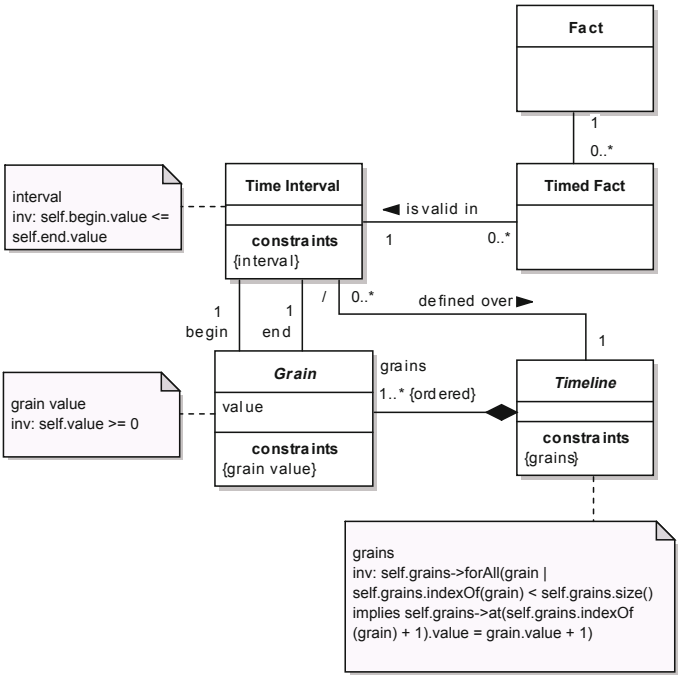

Fig. 2. Concepts related to time consciousness

2.3 Time Observability

Figure 3 sketches the abstractions related to the concept of time observability. In particular, a *Timer* is a source of events that are interpreted as equally spaced in time. Timers can be arranged in a hierarchy, in which every descendant timer has exactly one reference timer. The root of this hierarchy is the *Ground Timer*, which generates events that are understood as marking the flow of the real external time. On the other hand, a *Virtual Timer* is a kind of timer that counts (through its *internal counter*) the number of events it receives from its reference timer and in turn generates events every time the number of received events equals its *period*. This behavior is shown in the state diagram presented in Figure 4. A particular case of *Periodic Entity* (i.e., entities interested in receiving events generated by a timer) is the *Clock*, whose purpose is to keep track of current time on a timeline. Every time a clock receives an event generated by its timer, it advances the current time (*current time*) on its timeline, as depicted in Figure 5.

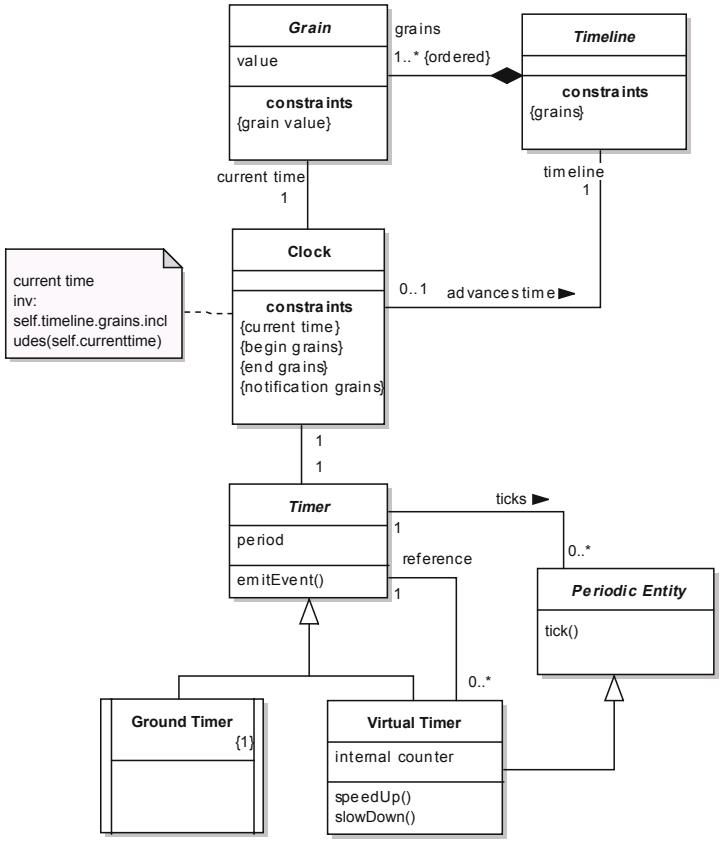

Fig. 3. Concepts related to time observability

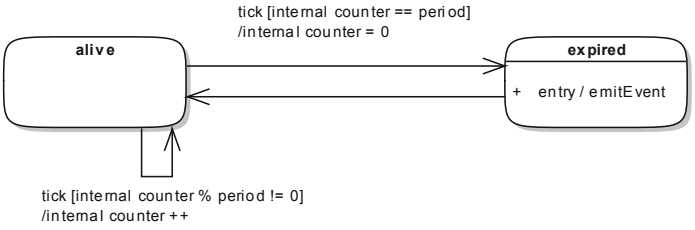

Fig. 4. Virtual timer behavior

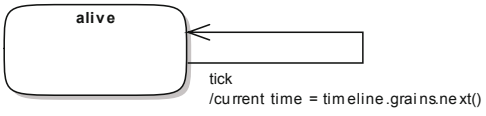

Fig. 5. Clock behavior

2.4 Time Triggering

A *Clock* is an entity that on one hand can be used by time observer entities to read current time, and on the other hand can be used to trigger the execution of *Time Driven Performers*. It is useful to remark again that *Time Conscious Performers* and *Time Observer Performers* are activated by some generic entities unrelated to time. The additional concepts that underlie the activation of *Time Driven Performer*s by *Clock*s are shown in Figure 6. A clock is possibly associated to a set of *Time Driven Performer*s that must all be always activated together at predefined time grains, specified by the clock's *notification grains* attribute. In order to allow the use of the defined concepts for the definition of standard hard real-time systems, it is useful to associate to every *Time Driven Performer* a list of *Performer Execution* entities, each composed of a *begin* time grain (the instant when the performer's execution can be triggered) and an *end* grain (representing the deadline for that particular execution of the performer). Of course, several constraints are required to guarantee the consistency of the picture:

- For every execution of a performer, the deadline must obviously be in the future with respect to the corresponding activation. That is, the value of the *end* attribute of every *Performer Execution* must be greater than or equal to the value of the corresponding *begin* grain
- The set of *begin* grains for all performers must be the same (since the clock activates all the associated performers at every emitted event), and must coincide with the set of the clock's *notification grains*

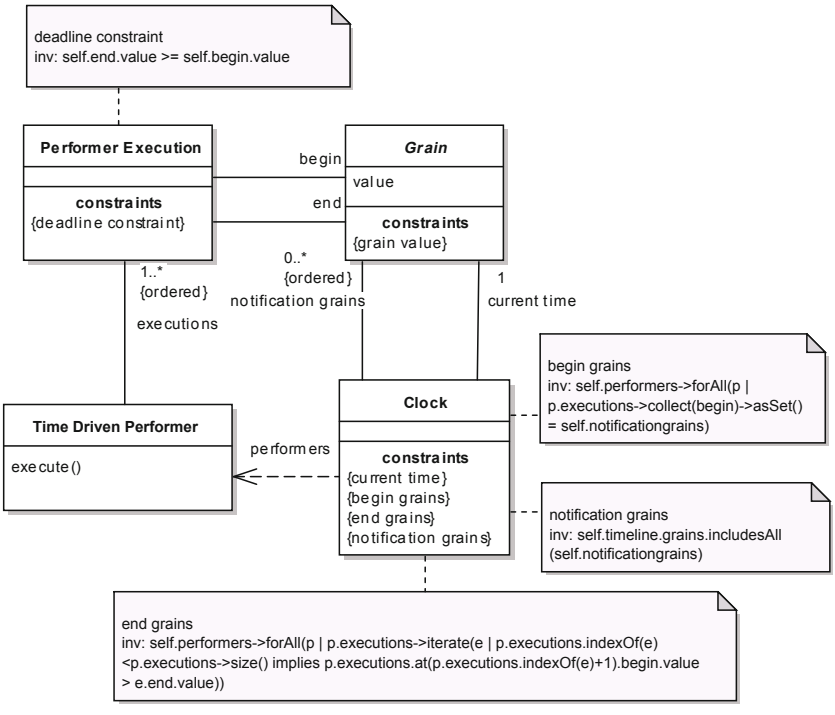

Fig. 6. Concepts related to time triggering

– To ensure that every execution of a performer is terminated before the next one is triggered, for every performer and for every execution of that performer, the value of the *end* grain must be less that the value of the *begin* grain of the next execution, if present

Note that the particular case of periodic activities is included in this description. In fact, a periodic activity can be reified by a *Time Driven Performer* with the following properties:

– The set of *executions* is unbounded
– Every execution is characterized by the same difference between the values of the *end* and *begin* grains
– The *begin* grains are equally spaced in time. Of course, the *notification grains* of the associated clock must share this property too

The details of the management of performer executions by a clock are shown in the sequence diagram of Figure 7. Every time a clock is ticked by its timer, it first advances the time it is keeping. Then, it checks if the current time is contained in the set of *notification grains*, that is, if it must trigger its associated performers at this time. If this is the case, the clock triggers the execution of every associated performer by invoking the corresponding *execute* action.

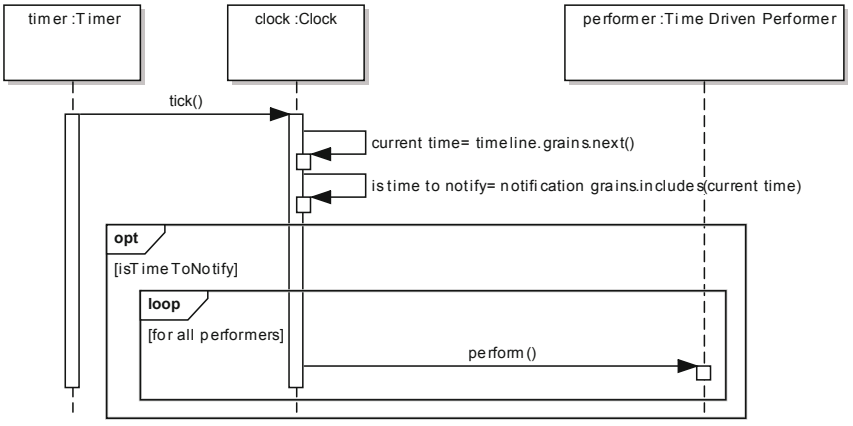

Fig. 7. Clock behavior

The state diagram of a *Time Driven Performer* is presented in Figure 8. The performer is left in its *waiting* state until the associated clock triggers an execution. The performer then goes to the *running* state, whose entry point is the performer's *perform* action. Once the performer completes its execution, it goes back to the *waiting* state until the next execution is triggered.

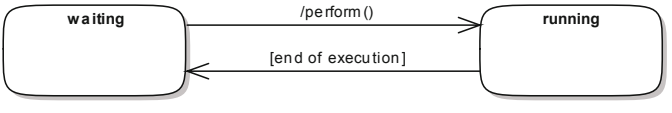

Fig. 8. State diagram of a time driven performer

An interesting feature of the model is the capability to dynamically vary the activation speed of time driven performers at run-time. This requires that the clock advance speed can be modified. At the aim, the two actions *speedUp* and *slowDown* have been defined for a *Virtual Timer* that respectively decrease and increase the timer's period.

2.5 Advanced Performer Types

More complicated performer behaviors can be realized by combining two or more of the time-related properties of Figure 1, as sketched in Figure 9. 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 performers that are triggered at the same grain (either because they are connected to the same clock or because the corresponding clocks activate them at the same time) does not depend on the order in which the executions are

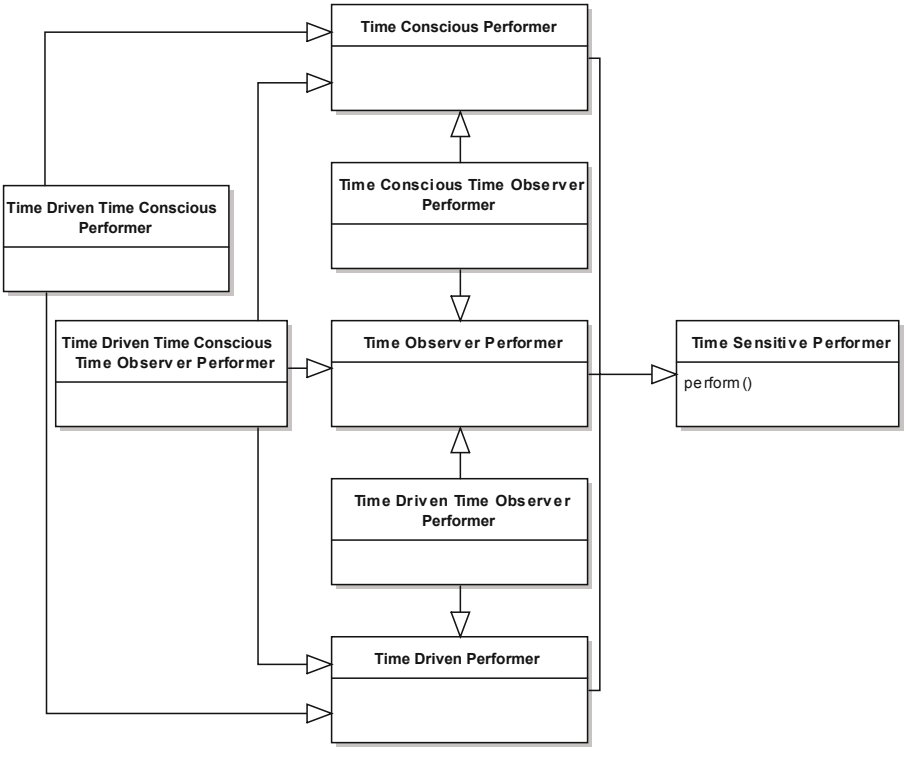

Fig. 9. Advanced performer types

actually managed, which depends on 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 conscious performers that are triggered simultaneously share the same view of the timelines in which they are interested, to avoid the situation of a performer that reads timed facts written by a performer triggered at the same time just because the latter was granted higher execution priority by the low-level scheduler.

Since every *Performer Execution* is characterized by a time interval delimited by the *begin* and *end* grains, a possible solution for this consistency problem is that all performers read timed facts at the beginning of the execution interval and write timed facts only at the end of the interval (i.e., at the deadline for that execution), even if the actual execution ends before the deadline. The performer structure that realizes this mechanism is described by the state diagram of Figure 10. A time driven time conscious performer is at first in the *waiting* state. When the associated clock triggers an execution (by invoking the *execute* action), the performer goes to the *reading* state, where it can read timed facts from timelines by means of its *observe* action. Once this phase has been completed, the performer spontaneously goes to the *running* state, analogous to the homonymous state for a time driven performer (Figure 8). At the end of

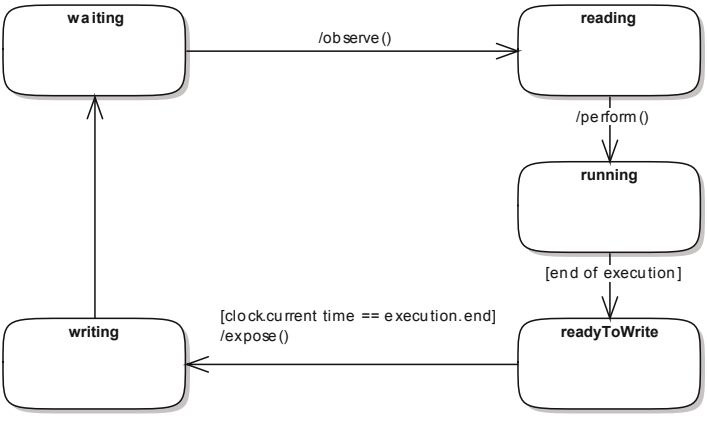

Fig. 10. State diagram of a time conscious time driven performer

the execution, instead of directly writing timed facts to timelines, the performer goes to the *readyToWrite* state, where it stays until the end of the interval for the current execution. Once the deadline is reached, the performer goes to the *writing* state, where it writes timed facts by means of the *expose* action. Once the writing operation terminates, the performer goes back to the *waiting* state until the next execution is triggered.

In an actual implementation, the concrete component in charge of the management of performer executions must thus guarantee that when the execution of a set of performer is triggered, all the performers of the set read timed facts before any is allowed to start the actual execution, and that every performer writes timed facts only at the end of the validity interval for its execution.

3 A Case Study: The Brewing Process

This section introduces a simplified example of how the concepts introduced in Section 2 can be easily applied to build a time-aware system. The case study concerns an automated plant for artisanal production of beer. Even though it may appear a simple application, it is particulary significative from the point of view of the involved temporal constraints and requirements.

3.1 Description

The process of making beer is known as *brewing*. Its purpose is to convert starch source (barley malt) into a sugary liquid called wort, that will be boiled with hops and afterwards transformed into an alchoolic beverage by yeast during fermentation. Figure 11 shows a typical brewing configuration, which includes three tuns, used for different purposes during the brewing. The process includes the following three macro-phases:

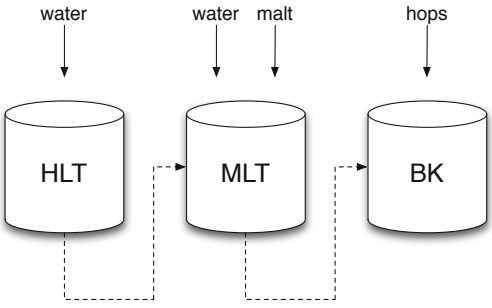

Fig. 11. Typical brewing configuration

- *Mashing* is the process in which milled malted barley is mixed with hot water inside MLT (Mash Lauter Tun) at different temperatures for different time intervals, during which the temperature must be kept constant (e.g., 52 ℃ for 10 minutes, 66 ℃ for 60 minutes, 78 ℃ for 15 minutes)
- *Lautering* aims at transferring the wort obtained after mashing into BK (Boil Kettle), separating solid grains from liquid wort (this is achieved using a false bottom placed in MLT). During this phase, the grains can be washed with hot water coming from HLT (Hot Liquor Tank), to extract more sugar from them (*Sparging*)
- *Boiling* is the last phase during which the wort collected in BK must be boiled (usually for 90 minutes). Besides, one or more hops are added as a source of bitterness, flavor and aroma at different time intervals (e.g., hop 1 after 10 minutes of boiling, hop 2 after 60 minutes, hop 3 after 90 minutes).

3.2 Modeling

In the process described above it is possible to identify at least one *time-related* aspect for each phase of the brewing process that can be addressed using the proposed temporal model:

(I) Boiling: hops need to be added at fixed times during boiling. This is an example of an aperiodic real-time activity. As shown in Figure 12, a *timer* T1, having the ground timer GT as its reference, is given acting as the source of ticks for a *clock* C1, which drives a *time driven performer* P1 that throws hops.

(II) Lautering: while transferring the liquid wort from MLT to BK (and from HLT to MLT if sparging is required), fluid volume sampling must be speeded up in order to provide better feedback. This is an example of a periodic real-time activity whose activation speed needs to be tuned. As shown in Figure 13, a *timer* T2 is given acting as the source of ticks for a *clock* C2, which drives a *time driven performer* P2 that acquires volume samples with different periods (e.g., 500 ms during the lautering (and sparging) phase, 30 s otherwise).

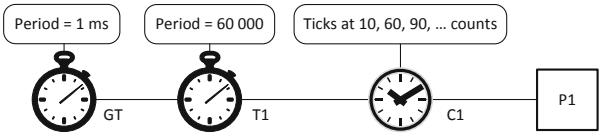

Fig. 12. Example of an aperiodic activity

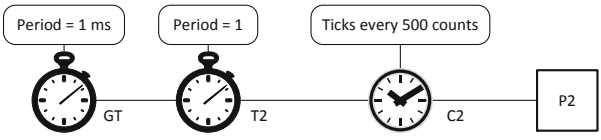

Fig. 13. Example of a periodic activity

(III) Mashing: during mashing, the MLT temperature must be periodically sampled and filtered in order to control the heating system to keep temperature constant. These are respectively examples of time conscious time observer and time conscious activities. As shown in Figure 14, a *timer* T3 is given, acting as the source of ticks for a *clock* C3, which drives a *time driven time conscious time observer performer* P3 that acquires MLT temperature samples with a fixed period (e.g., 500 ms), and saves them in timed facts on a *timeline* TL with the timestamp read from C3. Also, T3 acts as the source of ticks for a second *timer* T4, which in turn acts as the source of ticks for a *clock* C4, which drives a *time driven time conscious performer* P4 that reads from TL the not yet processed samples and computes the corresponding mean with a fixed period (e.g., 15 s).

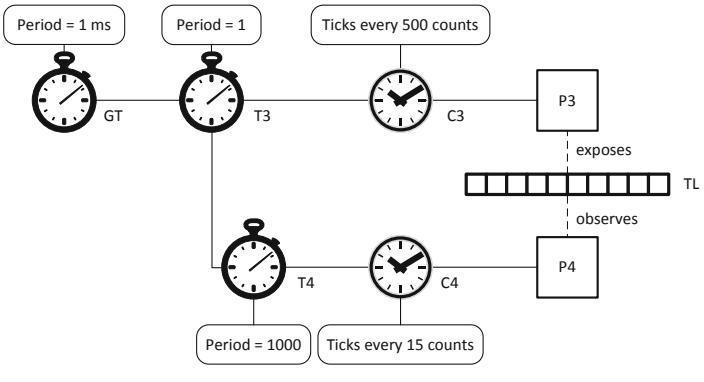

Fig. 14. Example of the use of timelines

4 Conclusions and Ongoing Activities

The paper presents a model that captures both the static and the dynamics aspects of time-related concepts. The model has been applied to design a plant for the artisanal production of beer because of its temporal requirements that well fit with the proposed model.

The temporal abstractions have been reified in an architecture implementation framework [27] developed in the Java programming language. Even though the language requires a virtual machine, the implementation allows the construction of soft real time systems thanks to the explicit management of the actual running activities performed by the implemented scheduler.

The framework has been successfully exploited for the achievement of the experimental results [28] related to a publish/subscribe platform denoted Space Integration Services [29]. In particular, a set of time driven performers has been implemented that periodically produces a set of publications. A time conscious time observer performer captures such publications and places them in a timeline. Another time conscious time observer performer captures the time in which notifications are ready by placing them in another timeline. Finally, an off-line time conscious performer is in charge of correlating the timed facts of the two timelines.

The main future development of the ideas presented in this paper concerns the re-implementation of the framework in a language that is more suitable for testing in hard real-time applications, such as C/C++. This new implementation could be used for the development of hard real-time applications either in the form of bare-metal, os-less programs for embedded systems, or to be executed on real-time operating systems.

References

1. 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), 314–335 (1995)
2. Kristensen, B.: Architectural abstractions and language mechanisms. In: Proceedings of the Software Engineering Conference, Asia-Pacific, pp. 288–299 (1996)
3. Garlan, D., Perry, D.: Introduction to the special issue on software architecture. IEEE Transactions on Software Engineering 21(4), 1–6 (1995)
4. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
5. Fiamberti, F., Micucci, D., Tisato, F.: An architecture for time-aware systems. In: 2011 IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–4. IEEE (2011)
6. W3C: OWL 2 Web Ontology Language, `http://www.w3.org/TR/owl2-overview/`
7. W3C: RDF Primer, `http://www.w3.org/TR/rdf-primer/`
8. O'Connor, M.J., Das, A.K.: A Method for Representing and Querying Temporal Information in OWL. In: Fred, A., Filipe, J., Gamboa, H. (eds.) BIOSTEC 2010. CCIS, vol. 127, pp. 97–110. Springer, Heidelberg (2011)

9. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing time into RDF. IEEE Transactions on Knowledge and Data Engineering 19(2), 207–218 (2007)
10. Tappolet, J., Bernstein, A.: Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 308–322. Springer, Heidelberg (2009)
11. Krieger, H.-U.: Where Temporal Description Logics Fail: Representing Temporally-Changing Relationships. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) KI 2008. LNCS (LNAI), vol. 5243, pp. 249–257. Springer, Heidelberg (2008)
12. Hobbs, J.R., Pan, F.: An ontology of time for the semantic web. ACM Transactions on Asian Language Information Processing (TALIP) - Special Issue on Temporal Information Processing 3, 66–85 (2004)
13. W3C: Time Ontology in OWL, http://www.w3.org/TR/owl-time/
14. Lutz, C., Wolter, F., Zakharyashev, M.: Temporal description logics: A survey. In: 15th International Symposium on Temporal Representation and Reasoning (TIME 2008), pp. 3–14. IEEE (2008)
15. 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 2010, pp. 239–248. ACM, New York (2010)
16. 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 (RTAS 2007), pp. 259–268. IEEE (2007)
17. OMG: MARTE Modeling and Analysis of Real-Time and Embedded systems
18. OMG: UML Profile for Schedulability, Performance, and Time
19. Group, O.M.: OMG Unified Modeling LanguageTM (OMG UML), Superstructure, http://www.omg.org/spec/UML/2.4.1/
20. SAE: AADL Architecture Analysis and Design Language (2009)
21. de Niz, D.: Diagrams and Languages for Model-Based Software Engineering of Embedded Systems: UML and AADL
22. Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE 91(1), 84–99 (2003)
23. 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)
24. Lopez, P., Medina, J., Drake, J.: Real-time modelling of distributed component-based applications. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2006), pp. 92–99 (2006)
25. Group, O.M.: Object Constraint Language, http://www.omg.org/spec/OCL/2.3.1
26. Fiamberti, F., Micucci, D., Tisato, F.: An Object-Oriented Application Framework for the Development of Real-Time Systems. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 75–90. Springer, Heidelberg (2012)
27. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory and Practice, 1st edn. Wiley (2009)
28. Bernini, D., Fiamberti, F., Micucci, D., Tisato, F.: Architectural Abstractions for Spaces-Based Communication in Smart Environments. Journal of Ambient Intelligence and Smart Environments, Thematic Issue A Software Engineering Perspective on Smart Applications for AmI (in Press)
29. Bernini, D., Micucci, D., Tisato, F.: A platform for interoperability via multiple spatial views in open smart spaces. In: The IEEE Symposium on Computers and Communications, Riccione, Italy, pp. 1047–1052 (2010)