Extending ASMETA with Time Features

Andrea Bombarda[1(✉)], Silvia Bonfanti[1], Angelo Gargantini[1], and Elvinia Riccobene[2]

[1] Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione, Università degli Studi di Bergamo, Bergamo, Italy
{andrea.bombarda,silvia.bonfanti,angelo.gargantini}@unibg.it
[2] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy
elvinia.riccobene@unimi.it

Abstract. ASMs and the ASMETA framework can be used to model and analyze a variety of systems, and many of them rely on time constraints. In this paper, we present the ASMETA extension to deal with model time features.

1 Introduction

Abstract State Machines (ASMs) [8] have been used to model several real case studies [2,3,5]. The framework ASMETA [4] supports the design and analysis of ASM models; it offers a wide set of features for model validation, verification, and code generation. However, many real systems, especially those in the safety-critical and cyber-physic domains, rely on time constraints. Modeling and validating these kinds of systems using ASMETA may be difficult since it does not offer primitives explicitly designed for dealing with time. According to the ASM definition of monitored locations, time is a monitored function whose value is written by the environment and read by the machine. Till now, the user has been asked to act as the environment and directly set time values when required; alternately, boolean monitored functions have been used to model passed time events. This user-based way of time supplying can be an annoying and error-prone activity and would require suitable constraints to guarantee time correctness, such as that time is a monotonic increasing function. Moreover, if the specification uses multiple time units (like seconds and minutes), it is left to the user to set them in a coherent way.

In this paper, we present the ASMETA library TimeLibrary that introduces *time* as special monitored functions and the concept of *timers*. Moreover, ASMETA is now extended to handle time in different ways (behind the above-mentioned already existing ways): *i)* reading the time from the machine hosting the simulation; *ii)* allowing the user to set the simulation time unit and enter the time values as a normal monitored function, in case exact time instants chosen by the user are needed to simulate critical behavior; *iii)* automatically increasing the time values at each machine step according to parameters initially set by the user.

A. Raschke and D. Méry (Eds.): ABZ 2021, LNCS 12709, pp. 105–111, 2021.
https://doi.org/10.1007/978-3-030-77543-8_8

```
module TimeLibrary
import StandardLibrary
export *
signature:
  abstract domain Timer
  enum domain TimerUnit={NANOSEC,
          MILLISEC, SEC, MIN, HOUR}
  monitored mCurrTimeNanosecs: Integer
  monitored mCurrTimeMillisecs: Integer
  monitored mCurrTimeSecs: Integer
  monitored mCurrTimeMins: Integer
  monitored mCurrTimeHours: Integer
  controlled start: Timer−> Integer
  controlled duration: Timer −> Integer
  controlled timerUnit: Timer −> TimerUnit
  derived currentTime : Timer−> Integer
  derived expired: Timer −> Boolean
```

```
definitions:

  function currentTime($t in Timer) = if (timerUnit($t)=NANOSEC) then
    mCurrTimeNanosecs
    else if (timerUnit($t)=MILLISEC) then mCurrTimeMillisecs
    else if (timerUnit($t)=SEC) then mCurrTimeSecs
    else if (timerUnit($t)=MIN) then mCurrTimeMins
    else if (timerUnit($t)=HOUR) then mCurrTimeHours
    endif endif endif endif endif
  function expired($t in Timer) = (currentTime($t) >= start($t) + duration($t))

  macro rule r_reset_timer($t in Timer) = start($t) :=
    currentTime($t)
  macro rule r_set_duration($t in Timer, $ms in Integer) =
    duration($t) := $ms
  macro rule r_set_timer_unit($t in Timer, $unit in TimerUnit) =
    timerUnit($t) := $unit
```

Code 1. ASMETA TimeLibrary

Our approach is inspired by the timing mechanism provided in other formal notations [9] and in other ASM frameworks. For instance, CoreASM[1] introduces the TimerPattern: it uses the monitored location *now* to save the current system time and has appropriate *TimerAssumptions* on *now* evolution and whatever unit assumptions [8]. However, CoreASM explicitly only links *now* to the machine clock and it manages only times expressed in milliseconds and nanoseconds. Other ASM time mechanisms have been proposed starting from the seminal work in [11]; e.g., a simulator for real-time reactive ASMs was presented in [1], while the TASM approach specifying duration of rule execution appeared in [12], and its extension with events and observers in [13]. A general study of timing for ASMs can be found in [10].

The paper is structured as follows. In Sect. 2 we present the main functionalities we have introduced to deal with time, namely the TimeLibrary, with its *monitored functions* and the *Timer*. Section 3 reports the different approaches to simulate the time and shows the results of simulation in different case studies, such as a simple clock and the well-known Sluice Gate Control case study. Future works are outlined in Sect. 4.

2 Time in ASMETA

In ASMETA framework, we have introduced the TimeLibrary[2] which contains the basic constructs necessary to introduce time features in ASMETA specifications: *i)* monitored functions to manage the time in different time units (nanoseconds, milliseconds, seconds, minutes, and hours); *ii)* an abstract domain Timer useful to introduce user-defined timers; *iii)* some functions and rules to operate on timers, like to check if a desired amount of time is passed, to reset and start a timer, and to set the timer duration and time unit. The proposed solution allows users to use different time units in the same ASM model and it guarantees consistency

[1] https://github.com/CoreASM/coreasm.core/tree/master/org.coreasm.engine/src/org/coreasm/engine/plugins/time.

[2] https://github.com/asmeta/asmeta/blob/master/asm_examples/STDL/TimeLibrary.asm.

between them during model simulation. Moreover, our mechanism assures that in a defined state, all the time functions refer to the same time instant, no matter what time unit is used.

A simple example using the time monitored functions is shown in Code 2, representing a clock that displays at each step current hours, minutes, and seconds.

```
asm simpleClock
import TimeLibrary

signature:
  controlled clockHours: Integer
  controlled clockMins: Integer
  controlled clockSecs: Integer
```

```
definitions:
  main rule r_main =
    par
      clockHours:=mCurrTimeHours mod 24
      clockMins:=mCurrTimeMins mod 60
      clockSecs:=mCurrTimeSecs mod 60
    endpar
```

Code 2. Time example: return current time

Measuring the absolute time is useful, but often, systems require that actions are executed if a desired amount of time is passed. For this purpose, timers are available in the TimeLibrary (see Code 1)[3], and, in the following, we will show how to use them. In the Sluice Gate Control case study (a well-known case study proposed in [7]) there are two timers, one to check if 10 min are passed before closing the gate and one if 3 h are passed before opening the gate. We have declared one function for each timer (see Code 3). Timers are initialized in the initial state, in terms of duration and time unit (using duration($t in Timer) and timerUnit($t in Timer) controlled functions). Moreover, during the initialization phase, the user can (if it is required by the specification) start the declared timer (using the currentTime($t in Timer) library function). The user uses the function expired($t in Timer) from the TimeLibrary (see line 12 in Code 3) to check if the timer passed as parameter is expired. While, when the timer must be reset in the specification, it can be done using the rule r_reset_timer($t in Timer) (see line 15 in Code 3), which takes the timer to reset as parameter. Moreover, in the specification duration of a timer can be changed (using the rule r_set_duration($t in Timer)) as well as its time unit (using the rule r_set_timer_unit($t in Timer)).

```
1  asm sluiceGateGround
2  import TimeLibrary
3  signature:
4   enum domain PhaseDomain = { FULLYCLOSED, FULLYOPEN }
5   dynamic controlled phase: PhaseDomain
6   static timer10MinPassed: Timer
7   static timer3hPassed: Timer
8
9  definitions:
10 main rule r_Main =
11  par
12   if phase=FULLYCLOSED and expired(timer3hPassed) then
13    par
14     phase := FULLYOPEN
15     r_reset_timer[timer10MinPassed]
16    endpar
17   endif
   if phase=FULLYOPEN and expired(timer10MinPassed) then
    par
     phase := FULLYCLOSED
     r_reset_timer[timer3hPassed]
    endpar
   endif
  endpar

 default init s0:
  function duration($t in Timer) =
   if $t = timer10MinPassed  then 10
   else if $t = timer3hPassed then 3 endif endif
  function timerUnit($t in Timer) =
   if $t = timer10MinPassed then MIN
   else if $t = timer3hPassed then HOUR endif endif
  function start($t in Timer) = currentTime($t)
  function phase = FULLYCLOSED
```

Code 3. Use of Timer in Sluice Gate Control specification

[3] Note that $t denotes the variable t in the AsmetaL notation.

3 Simulating Time

Besides time modeling, ASMETA framework supports three different mechanisms to handle time during simulation: *i)* the time is read from the machine hosting the simulation; *ii)* the user enters the values for time as normal monitored functions; *iii)* the time is automatically increased at each step by a predefined value.

The first mechanism allows the user to run the specification without entering the value of time monitored functions because the time is obtained from the Java 8 Date/Time API *Instant.now()* and automatically assigned to the time monitored functions. Sometimes, and especially if the specification would require long time intervals like hours or very short time intervals like nanoseconds, if the real time is used during the simulation, it may be unfeasible or impractical for the user to check what happens at specific instants of time. In this case, the second mechanism is most suitable: the user specifies the time unit he wants to run the specification and enters the desired time when required. If the specification uses more than one time with different time units, the others are automatically derived. In case the user wants to execute the specification and automatically increment the time by a predefined value at each step, the third approach can be used. The user has to define the time step and time unit, then the system automatically increments the time of the set delta value at each running step. If times have other time units compared to the one set by the user, they are automatically derived. The desired mechanism is set in the ASMETA → Simulator preferences from Window menu in Eclipse, as shown in Fig. 1.

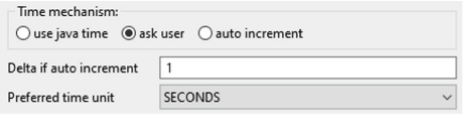
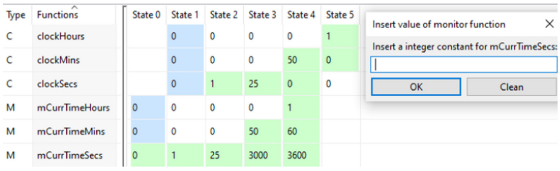


Fig. 1. Simulator settings

In the following, some simulation examples using all methods are shown. The simulation of Code 2 using Java time is shown in Fig. 2. For the entire simulation, the user had to wait 1 h. To fasten checking what happens at specific instants of time, the second method is the most suitable because the user specifies at each step the time and one step is executed with the inserted value (see Fig. 3).

Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6
C	clockHours		0	0	0	0	0	1
C	clockMins		0	0	1	8	59	0
C	clockSecs		2	25	2	20	18	2

Fig. 2. Clock simulation in "Java Time" mode



Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5
C	clockHours		0	0	0	0	1
C	clockMins		0	0	0	50	0
C	clockSecs		0	1	25	0	0
M	mCurrTimeHours	0	0	0	0	1	
M	mCurrTimeMins	0	0	0	50	60	
M	mCurrTimeSecs	0	1	25	3000	3600	

Fig. 3. Clock simulation in "ask user" mode

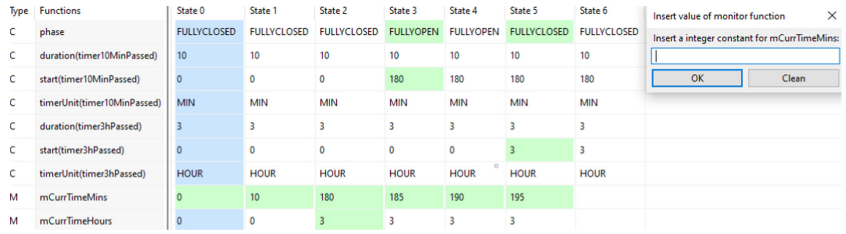
Type	Functions	State 0	State 1	State 2	State 3	...	State 119	State 120	State 121	State 122	State 123
C	clockHours		0	0	0		0	1	1	1	1
C	clockMins		0	1	1		59	0	0	1	1
C	clockSecs		30	0	30		30	0	30	0	30
M	mCurrTimeHours	0	0	0	0		1	1	1	1	1
M	mCurrTimeMins	0	1	1	2		60	60	61	61	62
M	mCurrTimeSecs	30	60	90	120		3600	3630	3660	3690	3720

Fig. 4. Clock simulation in "auto increment" mode with Delta = 30 and Time Unit = SECONDS

Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5
M	mCurrTimeMins	0	75	183	189	193	
C	duration(timer10MinPassed)	10	10	10	10	10	10
C	phase	FULLYCLOSED	FULLYCLOSED	FULLYCLOSED	FULLYOPEN	FULLYOPEN	FULLYCLOSED
C	start(timer10MinPassed)	0	0	0	183	183	183
C	duration(timer3hPassed)	3	3	3	3	3	3
C	timerUnit(timer10MinPassed)	MIN	MIN	MIN	MIN	MIN	MIN
C	timerUnit(timer3hPassed)	HOUR	HOUR	HOUR	HOUR	HOUR	HOUR
C	start(timer3hPassed)	0	0	0	0	0	3
M	mCurrTimeHours	0	1	3	3	3	

Fig. 5. Sluice Gate simulation in "Java Time" mode

The advantage is that the required simulation time is lower. Moreover, this is useful in case the user wants to check the behavior of the modeled system when the clock returns erroneous values (such as decreasing time between two consecutive steps). The last simulation method automatically increments the time at each step by a given value, and an example is shown in Fig. 4. The simulation is performed using a delta time equals to 30 s. As expected the change of clockHours function occurs at State 120. To show the use of timers, we have simulated the Sluice Gate specification using the three methods available. The first method requires the user to wait three hours before changing from FULLYCLOSED to FULLYOPENED. After ten minutes the gate moves back to FULLYCLOSED state in which it remains again for three hours (see Fig. 5). Note that controlled functions at state i are updated due to monitored values (including time) observed at state $i-1$. Since the instants of time when state changes occur are well known, we have simulated the specification using the second method (see Fig. 6) where the time is set at each state by the user. In this case, the simulation is faster because we do not have to wait the specification



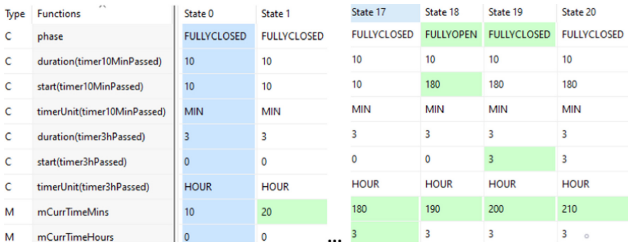
Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6
C	phase	FULLYCLOSED	FULLYCLOSED	FULLYCLOSED	FULLYOPEN	FULLYOPEN	FULLYCLOSED	FULLYCLOSED
C	duration(timer10MinPassed)	10	10	10	10	10	10	10
C	start(timer10MinPassed)	0	0	0	180	180	180	180
C	timerUnit(timer10MinPassed)	MIN	MIN	MIN	MIN	MIN	MIN	MIN
C	duration(timer3hPassed)	3	3	3	3	3	3	3
C	start(timer3hPassed)	0	0	0	0	0	3	3
C	timerUnit(timer3hPassed)	HOUR	HOUR	HOUR	HOUR	HOUR	HOUR	HOUR
M	mCurrTimeMins	0	10	180	185	190	195	
M	mCurrTimeHours	0	0	3	3	3	3	

Fig. 6. Sluice Gate simulation in "ask user" mode

time. In Fig. 7 the specification is simulated with the auto increment method, the delta is set to 10 and the time unit to minutes. Using this approach, the simulation is run and at each step the minutes are incremented by 10.

4 Future Work

Those presented in this paper are the first results of our effort toward endowing ASMETA with primitives to model and analyze systems with time constraints. In the future, we plan to provide two major improvements: *i)* extending the ASMETA scenario-based validation with new time features; *ii)* automatically mapping ASMETA time primitives into code time primitive, e.g., extending the automatic mapping of ASMETA models into C++ code for the Arduino platform [6].



Type	Functions	State 0	State 1	...	State 17	State 18	State 19	State 20
C	phase	FULLYCLOSED	FULLYCLOSED		FULLYCLOSED	FULLYOPEN	FULLYCLOSED	FULLYCLOSED
C	duration(timer10MinPassed)	10	10		10	10	10	10
C	start(timer10MinPassed)	10	10		10	180	180	180
C	timerUnit(timer10MinPassed)	MIN	MIN		MIN	MIN	MIN	MIN
C	duration(timer3hPassed)	3	3		3	3	3	3
C	start(timer3hPassed)	0	0		0	0	3	3
C	timerUnit(timer3hPassed)	HOUR	HOUR		HOUR	HOUR	HOUR	HOUR
M	mCurrTimeMins	10	20		180	190	200	210
M	mCurrTimeHours	0	0		3	3	3	3

Fig. 7. Sluice Gate simulation in "auto increment" mode with Delta = 10 and Time Unit = MINUTES

References

1. Slissenko, A., Vasilyev, P.: Simulation of timed abstract state machines with predicate logic model-checking. J. Univ. Comput. Sci. **14**(12), 1984–2006 (2008)
2. Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Integrating formal methods into medical software development: the ASM approach. Sci. Comput. Program. **158**, 148–167 (2018)

3. Arcaini, P., Gargantini, A., Riccobene, E.: Rigorous development process of a safety-critical system: from ASM models to Java code. Int. J. Softw. Tools Technol. Transf. **19**(2), 247–269 (2015). https://doi.org/10.1007/s10009-015-0394-x
4. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Softw.: Pract. Exp. **41**, 155–166 (2011)
5. Bombarda, A., Bonfanti, S., Gargantini, A.: Developing medical devices from abstract state machines to embedded systems: a smart pill box case study. In: Mazzara, M., Bruel, J.-M., Meyer, B., Petrenko, A. (eds.) TOOLS 2019. LNCS, vol. 11771, pp. 89–103. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29852-4_7
6. Bonfanti, S., Gargantini, A., Mashkoor, A.: Design and validation of a C++ code generator from abstract state machines specifications. J. Softw.: Evol. Process **32**(2), e2205 (2019)
7. Börger, E.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
8. Böger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56641-1
9. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling Time in Computing. Springer, Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-32332-4
10. Graf, S., Prinz, A.: Time in state machines. Fundam. Informaticae **77**(1–2), 143–174 (2007)
11. Gurevich, Y., Huggins, J.K.: The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. In: Kleine Büning, H. (ed.) CSL 1995. LNCS, vol. 1092, pp. 266–290. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61377-3_43
12. Lundqvist, K., Ouimet, M.: The timed abstract state machine language: abstract state machines for real-time system engineering. J. Univ. Comput. Sci. **14**(12), 2007–2033 (2008)
13. Zhou, J., Lu, Y., Lundqvist, K.: A TASM-based requirements validation approach for safety-critical embedded systems. In: George, L., Vardanega, T. (eds.) Ada-Europe 2014. LNCS, vol. 8454, pp. 43–57. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08311-7_5