Advances in Transformation of MARTE Profile Time Concepts in Model-Driven Software Development

Anna Derezinska(✉) and Marian Szczykulski

Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland
a.derezinska@ii.pw.edu.pl

Abstract. UML models can be extended with time concepts from the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile. In the Model-Driven Software Development, elements enhanced by stereotypes corresponding to time concepts can be transformed into code and assisted by appropriate library support during an application development and execution. We discuss several issues of the MARTE time concept interpretation and realization in an MDSD approach. Selected solutions were implemented in FXU, a tool for building C# applications based on UML classes and state machines. Realization of the MARTE support was verified in case studies.

Keywords: Model transformation · Code generation · UML · State machines · MARTE profile · Time modeling · MDD · MDSD

1 Introduction

The main ideas behind Model-Driven Software Development (MDSD) are preparation of comprehensive models of a system, their transformation, and forwarding them to execute [1]. Models can combine structural and behavioral descriptions (e.g. UML classes, components, state machines [2]), with features of a given application domain.

Among MDSD approaches, we focus on creation of a self-contained program application. The main steps of an assumed development process are preparation of models, transformation of models into a code in a general purpose language, and building of an executable application. The target application is based on an automatically generated code, specialized libraries supporting the modeled notions, and an additional code. The application can be run in a standard software development environment. In this paper, we do not discuss direct simulation of models, or executing of some intermediate forms of model transformations [3].

Design and development of domain models can be supported by UML profiles [2]. A profile includes a set of concepts denoted by stereotypes. Model elements are extended using stereotypes and their tagged values that define additional properties. In this way an element meaning can be enhanced while the UML meta-model remain unchanged (in most cases). The Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile belongs to profiles published by the OMG consortium [4].

© Springer International Publishing AG 2017
R. Silhavy et al. (eds.), *Software Engineering Trends and Techniques in Intelligent Systems*, Advances in Intelligent Systems and Computing 575, DOI 10.1007/978-3-319-57141-6_42

This profile has been widely used in system modeling, and UML/MARTE models were transformed into different implementations [5–9]. Some generation tools support MARTE notions, mainly in the Hardware Description Languages: SystemC, VHDL, Verilog. Models using MARTE, especially with detailed description of time behavior, can be applied in different domains. Therefore, we have analyzed transformation of classes and state machines extended by MARTE into a general purpose language. However, there is a lack of detailed analysis of such transformations, especially in the context of state machine models. Moreover, practical realization of an MDSD process required solving of some interpretation issues, e.g. dealing with semantic variation points of UML [2, 10, 11] and selecting working semantics of profile stereotypes.

The main contribution of this paper is presentation of interpretation and transformation of a MARTE subset, namely time concepts specified in MARTE::Time for classes and state machines. The proposed approaches were realized in an extended version of FXU (Framework for eXecutable UML) [12, 13] and verified in cases studies. The FXU tool supports code generation and execution of UML classes and all notions of state machines into C# code. Behavior of classes specified by state machines with MARTE refinements is reflected in a target C# application, which can act as a final implementation or an operational prototype. Thus, we can verify specification ideas, improve productivity and reduce time-to market.

The paper is organized as follows. In the next Section, we present interpretation and transformation issues of MARTE time concepts used in an MDSD process. Implementation of the MARTE support and its verification are briefly described in Sect. 3. We discuss related work in Sect. 4 and conclude the paper in Sect. 5.

2 Interpretation and Transformation of MARTE Time

Time concepts have been already presented in the standard OMG profile for Schedulability, Performance and Time (SPTP) [14], which was used in early UML versions (1.x). Simple time notions (*Time*, *TimeExpression, TimeObservation*) have been included in the UML specification since version 2.0 [2]. As the former SPTP profile was not consistent with new UML versions, a new extended OMG profile was developed. The Modeling and Analysis of Real-Time Embedded Systems profile (MARTE) [4, 15] can be used in UML 2.x and SysML models. Detailed concepts of a time domain are specified in a package of the foundation part, called *MARTE::Time*.

With the MARTE profile we can access physical and logical time structure using clocks. They refer directly to a time base on the time model. A clock has a set of units that can be accepted. A clock is an abstract concept specialized as a logical clock or a chronometric clock, which is assigned to a physical time. Time of a logical clock is usually counted in a number of ticks.

In the following subsections, time concepts in the context of an MDSD approach will be discussed. They correspond to stereotypies from the *MARTE::Time* profile. For each stereotype, we give (i) a short description, (ii) various interpretations with hints to realization of model to code transformation and run-time library, and (iii) a usage example. Examples refer to a case study of a dish washer controller (Sect. 3).

2.1 Stereotype *TimedDomain*

The *TimedDomain* stereotype can be assigned to a package treated as a container including definitions of clock types and objects of clocks. Such packages can be nested one in another. If a clock type or a clock object are placed in a package without this stereotype, many interpretations are possible. Lack of the *TimedDomain* stereotype can be ignored, and definitions of clock types and clock objects are allowed in any package. In another approach clock types and objects of clocks are disregarded if they are not located in a package extended with the *TimedDomain* stereotype.

In a practical realization, the above approaches can be combined into a hybrid one. Classes specifying clock types should always be generated, regardless being included in a *TimedDomain* package or not. This solution is motivated by a fact, that clock types can be used for different purposes in a model and their code should always be offered. A more restricted rule is proposed for clock objects, which have to be placed in an adequately stereotyped package. Those objects are used only for time measurement, therefore should always be placed in a package assigned to this domain.

An example of the *TimedDomain* stereotype associated with a package is shown in Fig. 1. The package contains a *Dishwasher Timer* class and an enumeration.

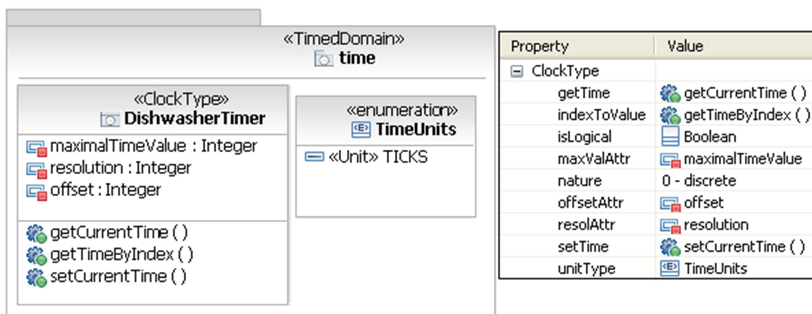

Fig. 1. Examples of the *TimeDomain* and *ClockType* stereotypes

2.2 Stereotype *ClockType*

The *ClockType* stereotype can be assigned to any class that specifies a type of a clock. There are several tagged values used for detailed specification of a clock type. If these values are fixed, they can be defined directly in a model. Otherwise tags are defined as attribute values and operations in a class. In this way, tagged values can differ in dependence on a clock instance. Tags of *ClockType* have the following meaning:

- *isLogical* - a clock type: *true* for a logical clock, *false* for a physical one (an attribute to get system time is necessary, e.g. *System.DateTime* for C#),
- *nature* - represents the discrete or dense time nature (logical clocks have always discrete time),
- *maxValAttr* – after reaching this value, time is counted from the beginning (logical clock only),

– *setTime/getTime* – operations for changing/reading time value (logical clock only),
– *indexToValue* – an operation to map a time index (number of an event) to a real time value (logical clock only),
– *ResolAttr* – resolution of an associated clock (defined for discrete time only),
– *offsetAttr* – offset of the associated clock expressed in the default time units,
– *unitType* – a set of supported time units.

A class with the *ClockType* stereotype can be transformed into a code as any other class in a program. Its tags are implemented using corresponding methods and fields to store appropriate values. The class is also supplemented with additional data and methods to control a clock behavior in accordance to the *MARTE::Time* specification.

Exemplary application of *ClockType* is shown in Fig. 1. The *DishwasherTimer* class is specified as *ClockType* with a set of tags. Tag names and values are listed in an additional window. An enumeration defines time units accepted by this clock type. Each item should be denoted by the *Unit* stereotype belonging to a package of Non-functional Properties Modeling *MARTE::NFPs*.

2.3 Stereotype *Clock*

An instance of a class stereotyped with *ClockType* is labelled with the *Clock* stereotype. This kind of object should be located in an object diagram defined in a package with the *TimeDomain* stereotype. It is used to access time by other elements from the *MARTE::Time*, namely *TimedProcessing, TimedEvent,* and *TimedValueSpecification*.

There are additional properties of a clock. A reference class with the *ClockType* stereotype is defined in the *type* tag. A default time unit used by the clock is given in the *unit* tag. The unit has to belong to a set of units listed in the appropriate clock type. A chronometric clock has a time *standard* specified with a tag. It is equal to one of predefined values from the *TimeStandardKind* MARTE Library [4–Annex D.3.].

Code generation of the *Clock* stereotype requires transformation not only of classes but also of object diagrams. This facility can be restricted to objects that are annotated with the *Clock* stereotype and placed in a package with the *TimeDomain* stereotype. In a class defining a clock type, a static method can be generated for any clock object. The method returns an instance of the clock type specified with parameters defined in an object diagram. Usage of the *Clock* stereotype is illustrated in Fig. 2. A *timer* object is an instance of the *DishwasherTimer* class (Fig. 1).

Fig. 2. An instance of *DishwasherTimer* - an example of the *Clock* stereotype

2.4 Stereotype *TimedProcessing*

The *TimedProcessing* stereotype can be assigned to any element that has behavior specified by its start and end points, or by a duration time. For example, such a behavioral element can be a whole state machine, an action (*Do, Entry, Exit*) in a state, or an action labelling a transition between states. A stereotyped element is associated with a clock using tag *on*. The stereotype can be refined with several tags: *duration, finish, start*. There are many possible interpretations of specified time behavior in dependence of these tags.

1. *Start* and *finish* tags are specified and a *duration* tag is not. In general, events given in *start* and *finish* stand for beginning and ending points of the behavior. Moreover various interpretation cases are possible:
   (a) After encountering of a start event behavior is started and ended with a finish event. However, behavior can be started and/or ended also in another way. Several solutions can be chosen if a start event happens during an active behavior:
      - behavior cannot be activated once again as it is already in progress,
      - a new behavior instance is launched, e.g. in a separate thread,
      - the behavior is reactivated after its end.

      A *finish* event encountered while the behavior is not active. Then, we can
      - ignore the *finish* event,
      - save the *finish* event in a buffer/queue and wait for activation of the behavior.

      Then, the behavior is ended due to the *finish* event.

   (b) A behavior can be started and ended only via *start* and *finish* events. Though, not all *start/finish* events can launch/end a behavior.
   (c) Combination of cases (a) and (b), i.e. each occurrence of a *start/finish* event starts/ends a behavior and it is not allowed to activate or deactivate any behavior in another way.
   (d) *Start/finish* events are generated when a behavior starts or ends. This interpretation is opposite to above ones, where the events acted as triggers.
2. A *duration* tag is specified and *start* and *finish* tags are not. In general, a behavior should last for time equal to a *duration* tag value. Time is counted with a clock specified in an *on* tag. However, different interpretations can be used:
   (a) The exact duration time will be forced. If a behavior is longer than stated by the *duration* tag, it is interrupted. Otherwise, if a behavior is due to finish before elapsing of duration time, it is prolonged to be as long as the duration value.
   (b) A behavior can lasts no longer than specified by *duration* value. Otherwise it will be interrupted.
   (c) A behavior should last for an interval equal at least to the *duration* value. Otherwise an error occurs (e.g. an exception is raised) or the behavior end will be postponed to the required moment.

3. All *start*, *finish* and *duration* tags are specified. In this case, we expect a behavior to start in the same moment as the *start* event occurrence, and end at the *finish* event occurrence. But additionally, passing time is restricted by the *duration* tag. If we select the variant when the event *finish* triggers the behavior end, different interpretations are still possible:
   (a) Each *finish* event is ignored during an interval of the *duration* time, which is counted since a *start* event occurrence.
   (b) If a *finish* event occurred and a time interval counted from the *start* event occurrence is shorter than the *duration* value, an error is generated.

Our recommendations depend of a meta-model element to which the stereotype is assigned. In case of a state machine, a *start* event is interpreted as in 1(a), and *finish* as 1(d). When an action in a state or on a transition is concerned, both *start* and *finish* are handled according to 1(d). The *duration* tag will be processed as 2(b) in all cases.

Usage of the *TimedProcessing* stereotype is shown in Fig. 3. The *DishWasher Controller* state machine is specified with this stereotype and its tags: *start* and *finish*. The tags have their events specified. The state machine begins is activity after occurrence of the *start* event. After the end of the behavior, the *finish* event will be launched.

Entry action in the *Prewash* state is also specified with the *TimedProcessing* stereotype. In this case, the *duration* tag is defined by a literal with the *TimedValue Specification* stereotype (Sect. 2.6).

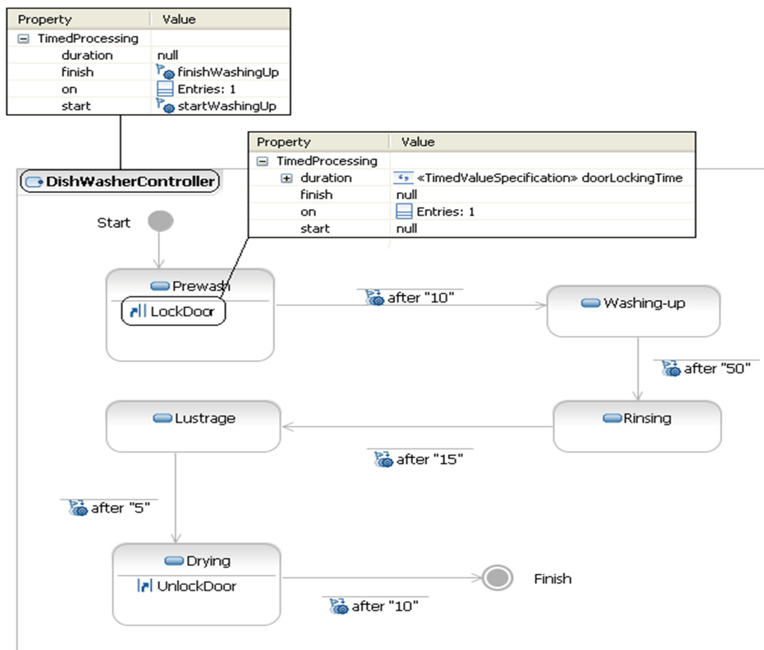

Fig. 3. State machine of *DishWasher Controller* - examples of the *TimedProcessing* stereotype

2.5 Stereotype *TimedEvent*

Any time event from the *CommonBehaviours::SimpleTime* package of UML can be specified by the *TimedEvent* stereotype. Therefore, additional data of a time event can be specified, or a cyclic time event is created.

Value of a timed event determines when the event is to be generated for the first time. When an event flag *isRelative* is false, its value denotes an absolute time instant presented by the associated clock. Otherwise the event value defines a time between an event instance generation and its entering a queue. Then, the *every* tag denotes a duration time between event occurrences. The number of occurrences is limited by the *repetition* tag. Value of a timed event is defined by a CVS expression (Clocked Value Specification) [4 – Annex C].

Realization of this stereotype requires handling of time events that are placed into appropriate event queues of state machines. The same event can be generated many times, if necessary. Different realizations of time events influence performance of a target application. Variants of time event handling in MDSD were presented in [16].

Several transitions in the dish washer state machine are labelled with time events (Fig. 3). These events are extended with *TimedEvent* stereotypes. An example of a transition and the properties of its event are shown in Fig. 4. Values of tags *every* and *repetition* are empty, as the event is not repeatable. The event occurs after 10 time units in accordance to an associated clock (tag *on*). This time is measured since entering the *Prewash* state.

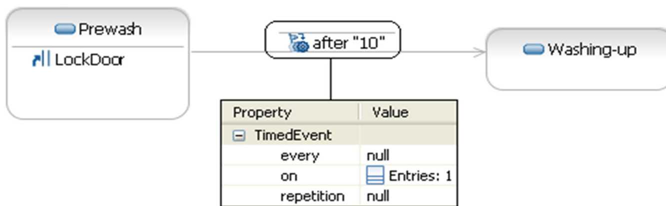

Fig. 4. A transition of *DishWasher Controller* (Fig. 3) with the *TimedEvent* stereotype

2.6 Stereotype *TimedValueSpecification*

This stereotype can be assigned to any value in a UML model (*Classes::Kernel::ValueSpecification*). *TimedValueSpecification* denotes that a corresponding value is interpreted as a time value of a clock referenced by an *on* tag. Meaning of the stereotyped value depends on the *interpretation* tag:

- *Duration* –value of a time interval passing after an event,
- *Instant* – value of a time instant in a given clock,
- *Any* – a duration or instant value in accordance to a TSL (Time Specification Language) expression.

Time expressions are written in TSL, a part of VSL (Value Specification Language) [4–Annex B], therefore realization of the stereotype requires translation of such expressions according to the defined grammar.

3 Support for the MARTE Profile in FXU

Framework for eXecutable UML (FXU) was developed as a first tool that supported transformation of classes and state machines into C# code [12]. Its main goal was transforming all notions of state machines into an executable code. Its functionality was enhanced within consecutive versions [13, 16].

In general, FXU consists two parts: FXU Generator that transforms UML class and state machine models into corresponding C# code, and FXU Run-Time Library that implements state machine concepts and is incorporated into a final application. In order to make the tool more flexible and to combine elements of MARTE profile, the tool architecture was refactored. It was made extendable by a set of plug-ins. A plug-in component is responsible for interpretation of elements (stereotypes, tagged values) of a given profile, insertion of appropriate changes of a model and generation of an additional code to realize the profile. Appropriate extensions were also integrated with the Run_Time Library.

Using the new FXU architecture, the tool was facilitated with MARTE code generation. The extended library supports run-time realization of the profile notions according to interpretations given in the previous Section.

FXU with MARTE was used in different case studies. One of them was related to a home alarm system combining features of two models: an intrusion alarm and a fire alarm [17]. Experiences of the model and application development influenced selecting among variants of MARTE stereotype realization.

Examples presented in this paper originate from a case study of a dish washer, mainly its controller. A logical clock is used in the model, therefore an activity time is independent from a physical time. A whole cycle of a dish washer work lasts for 90 logical time units, e.g. assuming 1 min for a unit it makes 90 min. In the state machine of the dish washer controller, movements between consecutive states are realized according to specified time requirements. The washing process starts with the *startWashingUp* event and ends with the *finishWashingUp* event, as stated in the tags of the *TimedProcessing* stereotype of the state machine.

Experimental verification of the FXU with MARTE was also carried out on various models aiming at utilization of all concepts from the *MARTE:Time* profile, in particular: extensive usage of time events, managing of time-driven processes, testing of various clock types, etc. Another case study was based on an example of a 4 stroke engine [18]. The performed experiments confirmed a proper utilization of time concepts in the model-driven application development.

4 Related Work

Models with the MARTE profile are widely applied in system modeling and analysis. Therefore, different transformations, mainly into domain targets, were proposed.

UML/MARTE models were used in HW/SW co-design approaches. In [5] models were transformed into SystemC executable used in simulation to verify a target VHDL. Results were applied in an FPGA solution of multimedia embedded systems.

Rapid prototyping of heterogeneous embedded HW/SW systems under consideration of timing and power aspects was presented in [6]. MARTE/UML models were transformed to IP-XACT and further into SystemC. Executable specification was used in an estimation and simulative analysis of timing and power properties of a system.

Generation of the System-Level Architecture Model (S-LAM) from a UML model with the MARTE profile is presented in [7]. Data-parallel applications were designed to be executed on a massively parallel System-on-Chip. The Gaspard2 tool supports transformation of MARTE/UML into OpenCL, a standard for parallel computing [8].

It should be noted, that in the papers discussed above, no information about dealing with state machines and the MARTE interpretation issues are given. State machines were taken into account in [9] where the MODCO transformation tool was presented. However, this approach covers only a small subset of UML state diagram constructs, supporting neither hierarchy nor concurrency.

There are CASE tools that support modeling with UML profiles, including MARTE, e.g. IBM RSA (since v. 7.0), Papyrus, MagicDraw, etc. but only some of them deal also with transformation of MARTE models, like Papyrus.

In the contrary to other approaches, our target is not a domain language, but a general purpose language, namely C#. Moreover, in the code generation and building an application we focus on state machine transformation. We take into account all features of state machines, including complex states with orthogonal regions, history, all pseudostates, etc.

5 Conclusions

Modeling of a system with time notions, its automatic transformation and building of an application combined with the support of modelled concepts gives an opportunity to create a well-specified reliable application. Therefore, we discussed transformation variants of MARTE time concepts that were implemented in an MDSD tool. It transforms classes and state machines into C# and supports building an application.

A direction which benefits from the discussed approach is rapid prototyping. A final application can cover control and time-related parts of a system functionality. Detailed modeling of other system features can be cumbersome in an MDSD, and therefore postponed to implementation in a programming language. Taking into account verification purposes, the application can be treated as a conceptual prototype, or an operational prototype for further code extension. Moreover, processing of call and time events specified in state machines can be realized within a target application with a satisfactory performance [16].

References

1. Liddle, S.W.: Model-driven software development. In: Embley, D.W., Thalheim, B. (eds.) Handbook of Conceptual Modeling, pp. 17–54. Springer, Heidelberg (2011)
2. Object Management Group, OMG Unified Modeling Language (2015). http://www.omg.org/spec/UML/
3. Dominguez, E., Perez, B., Rubio, A.L., Zapata, M.A.: A systematic review of code generation proposals from state machine specifications. Inf. Softw. Technol. **54**(10), 1045–1066 (2012)
4. Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. version 1.1. (2011). http://www.omg.org/spec/MARTE/
5. de la Fuente, D., Barba, J., Lopez, J.C., Peñil, P., Posadas, H., Sanchez, P.: Synthesis of simulation and implementation code for OpenMAX multimedia heterogenous system from UML/MARTE models. Multimedia Tools Appl., 1–32 (2016). doi:10.1007/s11042-016-3448-5
6. Grüttner, K., Hartmann, P.A., Hylla, K., Rosinger, S., Nebel, W., Herrera, F., Villar, E., Brandolese, C., Fornaciari, W., Palermo, G., Ykman-Couvreur, C., Quaglia, D., Ferrero, F., Valencia, R.: The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration. Microprocess. Microsyst. **37**, 966–980 (2003)
7. Ammar, M., Baklouti, M., Pelcat, M., Desnos, K., Abid, M.: Automatic generation of S-LAM descriptions from UML/MARTE for the DSE of massively parallel embedded systems. In: Lee, R. (ed.) Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015. SCI, vol. 612, pp. 195–211. Springer, Cham (2016). doi:10.1007/978-3-319-23509-7_14
8. Wendell, A., Rodrigues, O., Guyomarch, F., Dekeyser, J.-L.: An MDE approach for automatic code generation from UML/MARTE to OpenCL. Comput. Sci. Eng. **15**(1), 46–55 (2013)
9. Coyle, F., Thornton, M.: From UML to HDL: a Model-Driven Architectural Approach to Hardware-Software Co-Design. Information Systems: New Generations Conference (ISNG) (2005)
10. Prout, A., Atlee, J.M., Day, N.A., Shaker, P.: Code generation for a family of executable modelling notations. Softw. & Syst. Model. **11**, 251–272 (2012)
11. Derezińska, A., Szczykulski, M.: Interpretation problems in code generation from UML state machines - a comparative study. In: Kwater, T. (ed.) Computing in Science and Technology 2011: Monographs in Applied Informatics, pp. 36–50. Depart. of Applied Informatics Faculty of Applied Informatics and Math. Warsaw Univ. of Life Sciences (2012)
12. Pilitowski, R., Derezinska, A.: Code generation and execution framework for UML 2.0 classes and state machines. In: Sobh, T. (ed.) Innovations and Advanced Techniques in Computer and Information Science and Engineering, pp. 421–427. Springer, Dordrecht (2007)
13. FXU Framework for eXecutable UML. http://galera.ii.pw.edu.pl/~adr/FXU/
14. Object Management Group, UML Profile for Schedulability, Performance, and Time Specification, version 1.1. (2005). http://www.omg.org/spec/SPTP/
15. Selic, B., Gerard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE. Developing Cyber-Physical Systems. Elsevier (2014)

16. Derezińska, A., Szczykulski, M.: Performance evaluation of impact of state machine transformation and run-time library on a C# application. In: Kobayashi, S.-y., Piegat, A., Pejaś, J., El Fray, I., Kacprzyk, J. (eds.) ACS 2016. AISC, vol. 534, pp. 328–340. Springer, Cham (2017). doi:10.1007/978-3-319-48429-7_30
17. Derezińska, A., Szczykulski, M.: Application of time concepts from the MARTE profile in a model-driven development case study. Przeglad Elektrotechniczny (Rev. Electr. Eng.) **2015** (11), 178–181 (2015)
18. Andre, C., Mallet, F., Simone, R.: Time modeling in MARTE. In: ECSI Forum on specification & Design Languages (FDL), Barcelona, Spain. ECSI, pp. 268–273 (2007)