computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

PROGRAMMING AUTONOMOUS BEHAVIOR OF REACTIVE METERING SYSTEMS BY TIMED AUTOMATA

Lukáš Krejčí

Faculty of Electrical Engineering, Czech Technical University in Prague,
Technická 2, 166 27 Praha, krejclu6@fel.cvut.cz

Abstract: The paper presents a new, innovative approach of programming of autonomous behavior of reactive metering systems. The presented method is based on safely timed automata defined by UPPAAL team. This modeling language is extended with event monitoring, utility functions for asynchronous operations invocation and supervising capabilities. Additionally, appropriate metering operations querying principle for metering systems is proposed. Finally, a new method of timed automata systems simulation is presented. This method is based on the principle of random interleaving of automata execution order and probabilities balancing in order to ensure fairness of automata execution. Advantages of presented methods as well as their basic principles are summarized and demonstrated on a case study of AMM network data concentrator. On this case study, it is shown, that proposed methods allow to effortlessly define the autonomous behavior of a data concentrator in the understandable and easily modifiable way, thus they lack major disadvantages of the currently used approach. *Copyright © Research Institute for Intelligent Computer Systems, 2017. All rights reserved.*

Keywords: AMM, smart metering, data concentrator, autonomous behavior, timed automata, UPPAAL.

1. INTRODUCTION

The Aim of this paper is to present new method of programming of autonomous behavior of reactive metering systems using timed automata, extending the method proposed in [1]. Principles of proposed methods are demonstrated on a case study of Advanced Metering Management (AMM) network data concentrator.

Typical AMM network is composed from two distinct parts. Metering part, which consists of various metering devices interconnected via wired or wireless networks, and data centers part consisting of various user data centers, usually connected via the TCP/IP based network. Data concentrator unit is a device, which serves as an interconnection point between metering and data centers parts of AMM network.

Data concentrator unit (DCU) purpose is to gather data from metering part and distribute it further to the numerous data centers in opposite part of AMM network. Currently, following DCU types are common:

- Transparent DCU, which functionality is similar to network switch/router.
- Autonomous DCU, which is, in addition, capable to perform independent metering management operations.

Transparent DCU simply routes requests and responses between data centers and respective metering network. Consequently, if the transparent DCU is used within AMM network, all metering and data upload operations are managed by data centers.

The autonomous DCU is able to manage metering and data upload operations by itself, however, such a DCU is also able to operate in transparent mode in order to enable advanced management of metering devices. Autonomous DCU usually embeds an internal database system providing the metering data storage.

Currently, autonomous behavior of DCU is typically defined by a fixed set of tasks, as in case of [2] and [3]. Each task usually defines one specific AMM operation (e.g. data reading or writing) tied together with some data upload operation. User can parameterize, which of these tasks should be executed, as well as trigger conditions of each task and, for some particular tasks, number of retry attempts in case of task failure. The task trigger condition is characteristically temporal, that means it specifies moment of first activation and task periodicity. Additionally, there are several more built-in tasks within the autonomous DCU that are hidden from user and thus cannot be parameterized. Synchronization of metering device clock is an example of such a task.

Above described approach has several drawbacks. As there is no possibility of specifying relations among tasks, user have to trust that a DCU manufacturer had implemented some precautious measures that would prevent deadlocks or other unpredictable conditions. However, even if these measures are provided, they can fail. In such case, DCU can become unresponsive for several minutes. As the hidden tasks exist within DCU, tasks execution is not transparent enough and prediction of exact behavior of DCU is very hard for its users. Additionally, fixed set of tasks with limited parameterization capabilities usually do not fit exact requirements of final user. Finally, today DCUs usually support only a single metering and data center network type, as well as support single, commonly proprietary, data center communication protocol. Therefore, users are forced to adapt their AMM networks to DCU behavior.

In order to avoid above stated problems in future, new generation of autonomous DCU software facility and suitable autonomous behavior definition approach was developed.

2. PROJECT BACKGROUND

Since one of the main flaws of the current DCU design approach is limited configurability and extensibility, software facility with modular architecture was developed (Fig. 1).

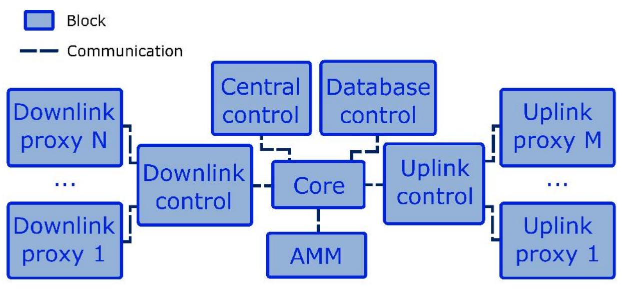


Fig. 1 – Structure of developed DCU software facility

The software facility is divided into blocks. Each block is executed as independent process within host operating system and performs set of specific operations. The following block types were defined:

- Downlink blocks responsible for communication with metering devices.
- Uplink blocks responsible for communication with data centers.
- Database control block responsible for internal database system.
- AMM block responsible for all AMM operations.
- Central control block responsible for autonomous behavior of DCU.
- Core block, which serves as message dispatcher and supervisor of other blocks.

Each mentioned block is configurable by its own XML based configuration file, so changes of user requirements can be easily reflected in modifications of these files. Furthermore, software facility designed in such way can be easily extended with new types of blocks developed in future, e.g. when support of new metering devices network is added by introducing a new type of the downlink proxy block. Additional details about software facility, e.g. structure of each block or used inter-process communication mechanisms, are beyond scope of this paper.

Since extensible and configurable DCU software facility was designed, new suitable approach of DCU autonomous behavior description was necessary. Similar effort was made by authors of [4], [5] and [6], who proposed method for programming of autonomous embedded systems based on Petri nets. However, disadvantage of Petri nets is complicated incorporation of timing. Therefore, approach that is more suitable for reactive metering systems was found. Proposed approach utilizes timed automata systems developed by UPPAAL team [9].

3. TIMED AUTOMATA SYSTEMS

In the following text, the timed automata systems developed by UPPAAL team [9] are described. UPPAAL team extended existing theory of timed automata, described in [7] and [8], with discrete variables and synchronization capabilities. As defined by UPPAAL, timed automata systems are a powerful tool for the model-checking verification [10] of various systems. Moreover, timed-automata systems can be used as modeling languages for model-based test-generation approaches (e.g. [11], [12] and [13]). Each timed automata system is basically a set of Finite State Machines (FSM) driven by a system of transition labels and automata variables/constants. These automata are always simulated in a discrete time flow. The timed automata systems can be described in XML files with the standard schema defined by UPPAAL team.

As mentioned before, timed automata are driven by a system of automata variables and transition labels, represented as text strings, referencing automata variables or constants. These labels have two purposes. They decide if labeled transition can be executed and define operations that are done upon transition execution. The following types of transition labels exist:

- Synchronization label, which purpose is to specify relation among automata. This label can reference only single variable. Synchronization can be invoked on particular variable by usage

of "!" operator placed after variable name. A variable can be awaited for synchronization by usage of "?" operator placed after variable name.
- Guard condition label, which decides, when labeled transition can be executed. It has form of arithmetic expression. Labeled transition is evaluated as executable if, and only if this expression is evaluated as true or guard condition label is an empty string.
- Assignment operations label specifies operations performed upon transition execution. Operation can be a call of a defined function, or assignment of value to a variable.

As previously mentioned, definitions of functions can be part of a timed automata system. Functions can be declared as globally or locally visible and typically perform calculations on the automata variables or constants.

The automata variables or constants (denoted also as variables in following text) can be declared as globally or locally visible and are of following types:
- Clock variable (denoted as "clock"), which value is increased over time, is used for timing of automata. Can be referenced within guard condition or assignment operations labels.
- Integer variable (denoted as "integer"). Can be referenced within guard condition or assignment operations labels.
- Boolean variable (denoted as "boolean"). Can be referenced within guard condition or assignment operations labels.
- Unicast synchronization channel variable (denoted as "chan"), which is used for synchronization of run among automata. Can be referenced within synchronization labels only. When synchronization is invoked on variable of this type, only single transition awaiting for synchronization on same variable is randomly executed among all awaiting transitions.
- Broadcast synchronization channel variable (denoted as "broadcast chan"), which has similar purpose as unicast synchronization channel variable. However, opposed to it, upon synchronization invocation on particular variable, all awaiting transitions are executed.

As primary purpose of the timed automata is to allow the model-checking verification of the described systems, verification tool is provided by UPPAAL team. This tool allows finding the state coverage, as well as is able to detect several hazardous conditions, such as the presence of deadlocks.

4. BEHAVIOR DESCRIPTION

The data concentrator unit autonomous behavior can be described as a set of processes, which produce various actions. These processes are typically executed on regular basis, which means they are driven by time. However, DCU is a reactive system and thus these processes must deal with various events. These events are usually bound to metering part of AMM network and provide information about numerous conditions related to the specified metering devices. Produced actions are mostly related to data manipulation operations, like metering information reading or their upload to data centers.

An example of the typically encountered event is the metering device state update. This particular event occurs in several situations, e.g. when metering device is connected to the network, or when the number of metering devices connected to the network reaches the predefined maximum. An example of the action commonly performed by DCU is the meter reading. This action performs upload of some AMM related data from metering device to the DCU internal database.

Because the processes defining DCU autonomous behavior are driven by time, each process can be described as a single timed automaton, as defined by UPPAAL team. Consequently, it's possible to describe autonomous behavior of DCU as a timed automata system. As timed automata systems are described using XML files with the defined schema, these files can be used for autonomous behavior definition as well. However, timed automata systems do not suit for processing of external inputs (events) or producing outputs (actions); therefore, developing a way for definition of these bindings to underlying system in timed automata sets was necessary.

4.1 EVENTS HANDLING

Since DCU have to handle incoming events as soon as possible after they are raised, events are comparable to the broadcast synchronization channels of the timed automata systems. Thus, similar approach can be used for awaiting the events. Consequently, new timed automata variable type, denoted as "_event", was introduced. The event variables can be used in similar manner as broadcast synchronization channels. This means following:
- The event variable can be awaited for synchronization in the same way as the synchronization channel type variables by usage of "?" operator in a transition synchronization label.
- When synchronization is invoked on some event variable, every transition with valid guard condition awaiting synchronization on this

particular variable has to be executed.

- Synchronization on specific event variable is invoked, when notification about event linked to this particular variable is received.

As the synchronization on the event variable is invoked only by occurrence of specific event, it, opposed to the synchronization channel variables, cannot be invoked explicitly by usage of "!" operator in a transition synchronization label.

As various types of events exist within DCU system, the event variable must be configured for monitoring specific event before its first usage in a transition synchronization label. Hence, new built-in synchronous function "_evt_subscribe(…)" was introduced. This function can be referenced within an assignment operations label of transition only.

Fig. 2 shows example of the event variable initialization and usage.

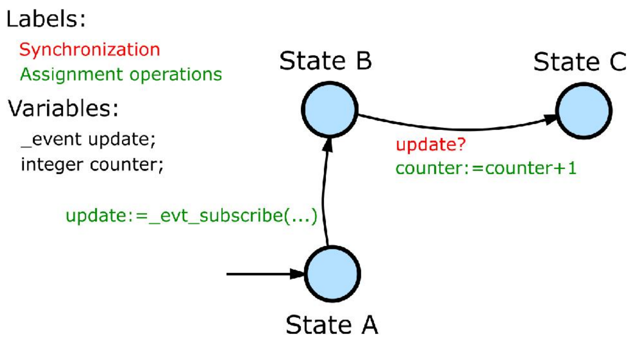


Fig. 2 – Example of the event variable usage

4.2 ACTIONS PRODUCTION

As already mentioned, DCU produces various actions. These actions have to be issued as some transition is being executed. Therefore, an assignment operation is suitable for specifying actions. The following built-in functions have been introduced to cover several actions produced by DCU:

- Function "_amm_read(…)" allowing to read specified AMM data from specified metering device and store it in internal database.
- Function "_amm_write(…)" allowing to write specified AMM data to specified metering devices.
- Function "_amm_invoke(…)" allowing to invoke specified operations within specified metering devices.
- Function "_amm_clock_sync(…)" allowing to synchronize internal real-time clock of specified metering devices with internal real-time clock of DCU.
- Function "_report_upload(…)" allowing to query specified information from internal database system, build report in specified format and upload it to specified data center.

These built-in functions are, by principle, asynchronous, so finding a method allowing supervising their results was desirable. Suitable method, inspired by C# 5.0 Task Parallel Library [14] and asynchronous API [15], was found and new timed automata variable type denoted as "_action" was introduced. The action variables can have their values assigned only by built-in functions listed above. In order to allow awaiting completion of the asynchronous actions represented by each action variable, the following properties similar to the event or the broadcast synchronization channels are assigned to the action variables:

- The action variable can be awaited for synchronization in same way as the event variables by usage of "?" operator in transition synchronization label.
- When synchronization is invoked on some action variable, all transitions with valid guard condition awaiting synchronization on this particular variable have to be executed.
- Synchronization on specific action variable is invoked, when the asynchronous action represented by this particular variable is completed.

Similarly to the event variables, synchronization on specific action variable cannot be invoked explicitly by usage of "!" operator in a transition synchronization label, because an invocation is done implicitly upon completion of the represented asynchronous action.

As an ability of asynchronous action success or failure detection is desirable, the action variables can be referenced within a guard condition label. In such case, each referenced action variable is interpreted as a Boolean expression. This Boolean expression is evaluated as true value, if, and only if asynchronous action represented by this variable has been completed successfully; otherwise it is evaluated as false value. Similarly, the action variable can be referenced within an assignment operations label for reading.

None of the asynchronous actions described above provides a return value, however, some actions defined in a future versions could have it. Considering that, it is possible to use action variable on the right side of assignment operation as well. Action variable used in this way has value of the action represented by the given action variable. If the referenced action has not yet completed, this variable has zero value.

Fig. 3 shows example of the action variable usage.

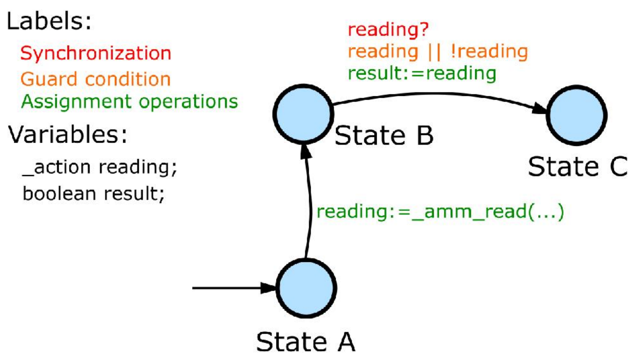


Fig. 3 – Example of action variable usage

4.3 ADDITIONAL CONSIDERATIONS

The timed automata clock variables are powerful tool that can be used for timing certain DCU operations. However, in several cases, some form of absolute time reference is required, e.g. while performing certain action every 10th minute of each 6th hour. Hence, the following synchronous built-in functions were introduced:

- Function "_rtc_get_seconds(...)" returning number of current second in current minute.
- Function "_rtc_get_minutes(...)" returning number of current minute in current hour.
- Function "_rtc_get_hours(...)" returning number of current hour in current day.
- Function "_rtc_get_wday(...)" returning number of current day in current week.
- Function "_rtc_get_mday(...)" returning number of current day in current month.
- Function "_rtc_get_month(...)" returning number of current month in current year.
- Function "_rtc_get_year(...)" returning number of current year.

These functions can be referenced within a guard condition and an assignment operations labels.

Since the unicast synchronization channel variables are stochastic by nature, they are not supported in order to prevent unpredictable conditions in the timed automata runtime. The broadcast synchronization channel variables provide similar functionality and are, opposed to the unicast channel variables, deterministic.

Finally, only the normal locations (i.e. states of a timed automaton) are supported, because the urgent and committed locations, defined by UPPAAL team, lacks the usage in case of the DCU system.

4.4 VERIFICATION

If autonomous behavior of DCU is described in presented form, it can be verified using UPPAAL verification tool in order to find state coverage and detect possible deadlocks. Due to the newly introduced variable types and built-in functions some modifications are necessary before the verification. These adjustments can be done by some sort of preprocessing tool. They should replace the introduced event and action variables by the standard variables and define or unroll the built-in functions.

Each event variable can be replaced by a broadcast synchronization channel variable. In order to simulate the events raising during the verification, a new automata should be added to the system (one automaton for each event variable). These additional automata would randomly invoke synchronization on the broadcast synchronization channel variables that replaced the event variables.

The action variables can be replaced in similar way; however, a single action variable should be replaced by one broadcast synchronization channel variable, one Boolean variable and two integer variables in order to trace action information (type, completion, success and return value).

4.5 EXAMPLES

Fig. 4 depicts example of two particular automata (the build-in functions parameters are omitted).

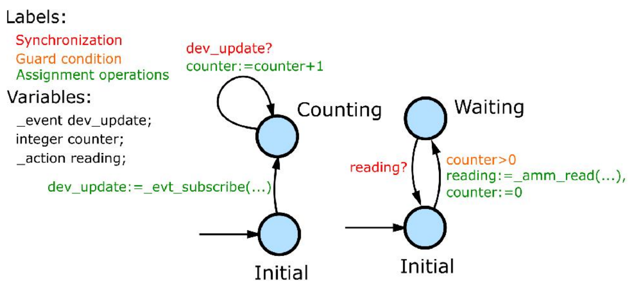


Fig. 4 – AMM application layer testing automata

The automaton on the left side of Fig. 4 simply increases counter when a metering device status update event is raised. Automaton on the right side of Fig. 4 performs AMM application layer test on devices, which have recently been connected to the network and waits for completion of this test.

Fig. 5 depicts another example of used automaton (the built-in functions parameters are omitted).

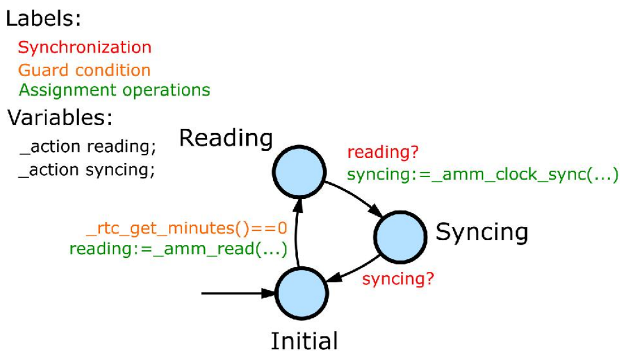


Fig. 5 – Clock synchronization automaton

The automaton in Fig. 5 reads current time from real-time clocks of all active metering devices every hour. After the reading is done, automaton performs clock synchronization on all devices without synchronized real-time clock suitable autonomous behavior definition approach was developed.

5. METERING OPERATIONS QUERING

In order to enable usage of action-production built-in functions, it was necessary to propose a query system allowing to specify which data on which metering devices should be affected by called function. Consequently, an appropriate approach was developed. The proposed principle is based on usage of the Structured Query Language (SQL; originally proposed in [16]; standardized in [17]; described in [18]). The AMM network and metering data within each metering device is represented as SQL data tables with fixed structure and therefore SQL language can be used for specifying parameters of presented metering functions.

The metering network itself is represented as a virtual SQL table with following columns:

- Column “gid” is an integer representing unique metering device global identifier within AMM network. This ID is assigned to the metering devices during their first connection.
- Column “address” is a string representing hardware address of a metering device.
- Column “clock_diff” is an integer representing a difference between internal clock of metering device and the DCU.
- Column “status” is an integer representing an encoded state of metering device. This encoded value covers all of possible states of metering device (e.g. if is the device communicating with DCU on link and application layers, if the device’s internal clock is synchronized with the DCU, etc.).
- Column “link_reliability” is a floating-point value representing reliability of communication with the device on the link layer.
- Column “amm_reliability” is a floating-point value representing reliability of communication with the device on the application layer (i.e. responsibility of AMM operations).

Proposed table structure enables effortless specification of target metering device using various parameters. For example, clock synchronization can be triggered only on such metering devices, which clock difference and link layer communication reliability is in specified range (i.e. query “clock_diff > 60 AND link_reliability > 0.8”).

The metering data themselves are represented as a virtual SQL table, where individual columns correspond to all possible metering information, row identifier and measurement time stamp. Each row of table corresponds to exactly one row in metering device's internal recorder. This structure allows to exactly specify, which values within specified timespan should be manipulated. The configuration data are represented in equivalent manner.

Additionally, in order to allow repetition of failed metering operations, an additional virtual SQL table type has been introduced. Rows of this virtual table corresponds to individual failed metering operations, while columns identify operation type, timestamp and parameters. Using multiple virtual tables of this type (each individual virtual table is bound to one specific metering device), any attempts to re-execute any failed metering operation can be easily described within the program XML file.

Every call to action production functions (described in section 4.2) utilizes the metering devices, metering data and operation retry queries as parameters. Metering devices query parameter is always required. Second parameter can contain metering data query. In cases when second parameter is not used, third parameter must contain operation retry query. Otherwise, third parameter should be left empty. Last parameter of each action production function call is always the timeout of corresponding asynchronous metering operation.

6. AUTOMATA RUNTIME SIMULATION ENGINE

Finally, some method of timed automata systems simulation was necessary to develop. The simplest way would be to use randomizing of transition execution, i.e. selecting random transition in each simulation cycle. However, this approach has two major drawbacks that could cause severe problems, especially in case when malfunctioning Pseudo-Random Numbers Generator (PRNG) is used. If executed transition is selected globally among all possible transitions, then transition-rich automata are preferred. Similarly, if executed transition is selected from randomly selected automaton, then probability of executing transitions of particular automaton decreases with increasing number of automata. Therefore, as DCU can be a complex system described by several timed automata and as even simplest automata are vital to proper DCU functionality (e.g. metering device status update events counting automaton), this approach is inacceptable in the DCU case. Consequently, an appropriate approach of timed automata simulation was developed.

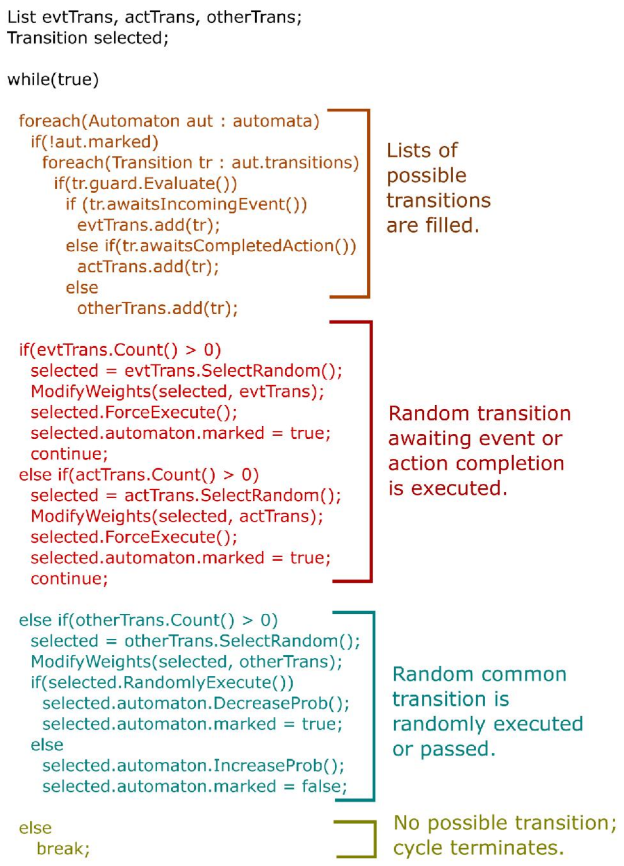
The developed automata simulation approach, called Automata Runtime Simulation Engine (ARSE), attempts to equalize probabilities of transitions and automata execution. During each simulation cycle, ARSE attempts to execute transitions of as many automata as possible, each of them giving the exact probability of execution.

As already mentioned, ARSE attempts to interleave automata in each simulation cycle. In order to achieve this, one transition is randomly selected among all possible transitions. Selected transition is then randomly executed or passed, which is necessary to prevent divergence of state coverages of ARSE and mentioned simple random approach, and particular automaton is marked as simulated. This process is repeated during cycle; however transitions of automata marked as simulated are ignored. When no possible transition is found, the simulation cycle is terminated and all automata are marked as not simulated. Because it is important to properly react to raised events and completion of asynchronous actions in the DCU's use case, ARSE prioritizes transitions awaiting synchronization on event or action variables over other transitions upon transition selection. Also, the prioritized transitions are always executed.

During particular simulation cycle, transitions are randomly selected using weight $w_{t,n}$ (weight of transition t in cycle n). After selection is done, weight of selected transition is decreased using arbitrary non-increasing function $s_{dec}(w_{t,n}, n)$, down to the minimal transition weight, denoted as w_{min}. Afterwards, weight of each non-selected possible transition is increased using arbitrary non-decreasing function $s_{inc}(w_{t,n}, n)$, up to the maximal transition weight, denoted as w_{max}. These weight-balancing operations ensure, that despite usage of malfunctioning PRNG, subsequently ignored possible transitions are more likely to be selected over time.

When deciding if selected transition is executed or passed, the particular automaton's probability of execution, denoted as $p_{a,n}$ (execution probability of automaton a in cycle n) is used as a reference. In the case that the transition is executed, this reference probability is decreased using arbitrary non-increasing function $e_{dec}(p_{a,n}, n)$, down to the minimal execution probability, denoted as p_{min}. In the opposite case, the reference probability is increased using arbitrary non-decreasing function $e_{inc}(p_{a,n}, n)$, up to the maximal execution probability, denoted as p_{max}. Similarly to transition weight-balancing operations, these balancing operations ensure, that although malfunctioning PRNG is used, subsequently passed automata are more probable to be executed over time.

Pseudo-code of ARSE's single simulation cycle is depicted on Fig. 6.



```
List evtTrans, actTrans, otherTrans;
Transition selected;

while(true)

 foreach(Automaton aut : automata)
   if(!aut.marked)
     foreach(Transition tr : aut.transitions)
       if(tr.guard.Evaluate())
         if (tr.awaitsIncomingEvent())
           evtTrans.add(tr);
         else if(tr.awaitsCompletedAction())
           actTrans.add(tr);
         else
           otherTrans.add(tr);

 if(evtTrans.Count() > 0)
   selected = evtTrans.SelectRandom();
   ModifyWeights(selected, evtTrans);
   selected.ForceExecute();
   selected.automaton.marked = true;
   continue;
 else if(actTrans.Count() > 0)
   selected = actTrans.SelectRandom();
   ModifyWeights(selected, actTrans);
   selected.ForceExecute();
   selected.automaton.marked = true;
   continue;

 else if(otherTrans.Count() > 0)
   selected = otherTrans.SelectRandom();
   ModifyWeights(selected, otherTrans);
   if(selected.RandomlyExecute())
     selected.automaton.DecreaseProb();
     selected.automaton.marked = true;
   else
     selected.automaton.IncreaseProb();
     selected.automaton.marked = false;

 else
   break;
```

Lists of possible transitions are filled.
Random transition awaiting event or action completion is executed.
Random common transition is randomly executed or passed.
No possible transition; cycle terminates.

Fig. 6 – Pseudo-code of ARSE simulation cycle

7. CONCLUSIONS, RESULTS AND FUTURE WORK

In this paper, the novel approach for a DCU autonomous behavior programming is presented. The described approach utilizes timed automata systems as a suitable way of autonomous behavior description and extends timed automata systems with support of event detection and action production / supervising capabilities. In addition, appropriate method of timed automata simulation runtime (ARSE) is presented. Moreover, as the only DCU-specific parts of presented case study are event types and asynchronous actions definitions, same principles can be reused for programming of autonomous behavior of arbitrary reactive metering device (i.e. device with autonomous behavior describable as set of time-driven processes reacting to raised events and producing specific actions). As the presented methods allow to define autonomous behavior of DCU in the understandable and easily alterable way, we believe that they improve the usability of DCU devices within AMM networks.

In recent time, a limited version of the presented approach was implemented in several real DCU devices, which were beta-tested as the prototypes. Implemented version of ARSE lacks support of the absolute time referencing functions, so automata had to rely on relative timing provided by the clock variables. Additionally, globally visible implicitly existing event variables were used (one for each supported event type) and thus the declaration of event variables is not supported. Similarly, locally visible implicitly existing action variables are used (one variable in each automaton for each supported asynchronous action type) and therefore each declaration of action variable is ignored. Finally, the mentioned weight-balancing and probability-balancing functions are defined as constant functions; however, that is not a big issue since reliable PRNG e.g. based on Mersenne-Twister [19] or RANLUX [20] algorithms can be used and thus a sufficient randomness of the selection of transitions and transition execution/pass decision was achieved.

Described beta test proved that proposed concept of programming of autonomous behavior of metering systems is applicable on existing AMM networks. Moreover, the beta-test has shown that presented principles remove disadvantage of usually cumbersome specification of custom autonomous behavior that currently used DCUs suffer from.

As the part of future research, the most appropriate forms of the mentioned weight- real-time clock suitable autonomous behavior definition approach was developed.

8. ACKNOWLEDGEMENT

Development described in this paper was supported by the Technology Agency of Czech Republic under grant no. TA03011091.

9. REFERENCES

[1] L. Krejci, "Programming autonomous behavior of AMM network data concentrator by timed automata," in *Proceedings of the IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Application (IDAACS'2015)*, Warsaw, Poland, September 24-26, 2015, Vol. 1, pp. 214-219.

[2] Ormazabal, *Smart Metering*, [product website] [Online]. Available http://www.ormazabal.com/en/your-business/products/advanced-metering?tab=13757&refer=894.

[3] ZIV, *ZIV Metering Solutions*, [product website]. [Online]. Available: http://www.meteringsolutions.ziv.es/ziv/systems.html#AMR?tab=13757&refer=894.

[4] T. Richta, V. Janoušek, "Petri Nets-based development of dynamically reconfigurable embedded systems," in *PNSE'13 – CEUR Workshop Proceedings*, Vol. 2013, Issue 989, pp. 203-217, ISSN 1613-0073, Hamburk, 2013.

[5] T. Richta, V. Janoušek, "Code generation for Petri Nets-specified reconfigurable distributed control systems," in *Proceedings of the 15th International Conference on Mechatronics – Mechatronika 2012*, Prague, 2012, pp. 263-269, ISBN 978-80-01-04985-3.

[6] T. Richta, V. Janoušek, "Operating system for Petri Nets-specified reconfigurable embedded systems," in *Proceedings of the Computer Aided Systems Theory – EUROCAST 2013*, *Lecture Notes in Computer Science*, 8111, Berlin Heidelberg: Springer Verlag, pp. 444-451.

[7] R. Alur, D.L. Dill, "A theory of timed automata," *Theoretical Computer Science*, Vol. 126, Issue 2, pp. 183-235, 1994.

[8] J. Bengtsson, J. Bengtsson, W. Yi, W. Yi, "Timed automata: Semantics, algorithms and tools," *Lecture Notes in Computer Science*, Vol. 3098, pp. 87-124, 2004.

[9] G. Behrmann, A. David, K. G. Larsen, "A Tutorial on Uppaal," *In Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*.

[10] T. A. Henzinger, X. Nicollin, J. Sifakis and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, Vol. 111, No. 2, pp. 193-244, doi: 10.1006/inco.1994.1045, 1994.

[11] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, A. Skou, "Time-optimal real-time test case generation using UPPAAL," in *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software 2003 (FATES'03)*, 2003.

[12] J. Blom, A. Hessel, B. Jonsson, P. Pettersson, "Specifying and generating test cases using observer automata," in *Proceedings of the 4th International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, 2004.

[13] L. Krejčí, J. Novák, "Framework and automated prioritization procedure for model-based testing of automotive distributed systems," in *Proceedings of the 17th Annual International Workshop on Databases, Texts, Specifications and Objects (DATESO 2017)*, 2017.

[14] Microsoft Corporation, *Task Parallel Library*, [Online]. Available: http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx.

[15] Microsoft Corporation, *Asynchronous Programming with Async and Await (C# and Visual Basic)*, [Online]. Available: http://msdn.microsoft.com/en-us/library/hh191443.aspx.

[16] D. D. Chamberlin, R. F. Boyce, "SEQUEL: A Structured English Query Language," in *Proceedings of the ACM SIGFIDET Workshop*

on Data Description, Access and Control. Association for Computing Machinery, 1974, pp. 249-264.

[17] International Organization for Standardization (ISO), "ISO/IEC 9075-1:2008: Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)," 1987.

[18] M. Chapple, "SQL Fundamentals," *Databases*, About.com.

[19] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Transaction on Modeling and Computer Simulation*, Vol. 8, No. 1, pp. 3-30, January 1998.

[20] M. Luscher, "A Portable high quality random number generator for lattice field theory simulations," *Comput. Phys. Commun*, No. 79, 1994.

__Lukáš Krejčí__ was born in Prague in 1990. He received his bachelor's and master's degree from the CTU in Prague, FEE in 2012 and 2014 respectively. Since February 2015, he has been studying the branch of doctoral studies called Measurement and Instrumentation at the Department of Measurement on CTU in Prague, FEE.