Mapping of State Machines to Code: Potentials and Challenges

Mehrdad Saadatmand
Mälardalen Real-Time Research Centre (MRTC),
Mälardalen University,
Västerås, Sweden
mehrdad.saadatmand@mdh.se
& Alten AB,
Sweden
mehrdad.saadatmand@alten.se

Antonio Cicchetti
Mälardalen Real-Time Research Centre (MRTC),
Mälardalen University,
Västerås, Sweden
antonio.cicchetti@mdh.se

***Abstract*—There is a big number of testing methods which are based on the state machine formalism. State machines serve as a strong means in behavior modeling of computer systems. However, this strength is lost as we go down the abstraction level from models to code. This is essentially due to the inherent semantic gap between state machine models and code, and that it is generally a challenging task to track states and transitions at runtime. In this paper, we discuss the benefits and challenges of having a mechanism for mapping state machines to code. The main intention with such a mechanism is to enable tracking of state changes at runtime. As we explain in this paper, the mapping provides for several important testing features such as verifying the actual runtime behavior of the system against its state machine models. Its importance becomes further emphasized remembering that model-based static analysis techniques rely on models as the source of information and design assumptions, and therefore, any mismatch between the actual behavior of the system and its models can also imply invalidation of the analysis results.**

Keywords–State Machines; Runtime Verification; Behavioral Modeling.

I. Introduction

Applying Model-Based Development (MBD) methodology [1][2] helps to cope with the ever-increasing complexity of computer systems. It does so by raising the abstraction level, enabling analysis at earlier phases of development and automatic code generation. In Model-Based Testing (MBT), models serve as an explicit representation of the intended behavior of a system from which test cases are generated [3] [4]. Test cases are then executed to detect failures and to verify if the intended and actual behavior of a system differ.

State machines [5] serve as a modeling formalism for behavioral description of different types of systems (particularly reactive systems) and are used extensively in model-based testing techniques. State machine models can thus capture the expected behavior of a system. In testing the behavioral aspects, it is necessary to be able to determine whether the runtime behavior of the system is in compliance with its specified behavior represented and described using state machine models. This is regardless if the code is manually developed or automatically generated from system models. For this purpose, there needs to be a mechanism to establish a mapping between state machines and code in order to verify that the code at runtime is actually behaving correctly according to the state machine model in terms of its internal states and transitions.

In [6], we have introduced and developed an approach for testing the behavior of automotive embedded systems, by comparing against the Timed Automata (TA) [7][8] specification models that are used to describe the internal behavior of system components, modeled using EAST-ADL language [9] (the term *state machine* is used in this paper as a synonym to also refer to a timed automaton whenever the main concern is only the states and transitions in the model regardless of the timing specifications). In the approach, timed automata models are analyzed to identify if different properties hold or not. As a result, a trace is generated consisting of a sequence of states and transitions serving as a witness or counter-example of the performed analysis. To verify the actual behavior of the system, it is then checked at runtime if the system traverses and goes through the exact order of states and transitions as in the trace file. To achieve this, as part of the approach we have defined a mechanism for mapping state machines to code. The mapping is currently done manually, which is not that scalable especially when the size of code grows. In this paper, we focus on the mapping mechanism and discuss its importance and the capabilities it provides for testing the behavior of systems based on our findings in [6]. We introduce our specific way of implementing the mapping mechanism along with its features and limitations. Moreover, other different possible solutions to implement such a mapping mechanism and the related challenges are also described and identified. In short, the main intention with this paper is to highlight the benefits of having a mapping mechanism between state machine models and code (as part of our research project results); particularly that establishing such a mapping can require early design decisions and following certain rules in the code to enable tracking states and transitions at runtime.

The remainder of the paper are structured as follows. In Section II, background context and motivation of this work is presented. Related work and possible solutions for the mapping mechanism, along with the challenges and potentials of having such a mapping mechanism are discussed in detail in Section III. Finally, Section IV concludes the paper and there, we also discuss the future directions of this work.

II. Background & Motivation

This work has been performed in the scope of the the Combined Model-based Analysis and Testing of Embedded Systems (MBAT) European project [10] consisting of 38 project partners. One of the main goals in MBAT is to provide a more efficient and effective Verification & Validation solution for embedded systems by exploiting the synergy between model-based analysis and testing. Brake-By-Wire (BBW) system from Volvo is one of the industrial use-cases that are addressed in MBAT. In a BBW system, mechanical parts and hydraulic connections between the brake pedal and each wheel brake are replaced by electronic sensors and actuators. Anti-lock Braking System (ABS) is usually an inherent functionality provided by BBW systems [11] whose purpose is to prevent the locking of wheels by controlling braking based on *slip rate*. There is a threshold for the slip rate beyond which the brake actuator is released and no brake is applied (otherwise the requested brake torque is used).

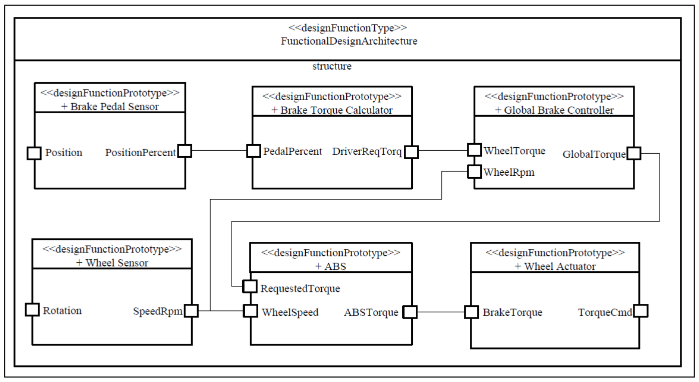

<<designFunctionType>>
FunctionalDesignArchitecture
structure
<<designFunctionPrototype>>
+ Brake Pedal Sensor
Position
PositionPercent
<<designFunctionPrototype>>
+ Brake Torque Calculator
PedalPercent
DriverReqTorq
<<designFunctionPrototype>>
+ Global Brake Controller
WheelTorque
WheelRpm
GlobalTorque
<<designFunctionPrototype>>
+ Wheel Sensor
Rotation
SpeedRpm
<<designFunctionPrototype>>
+ ABS
RequestedTorque
WheelSpeed
ABSTorque
<<designFunctionPrototype>>
+ Wheel Actuator
BrakeTorque
TorqueCmd

Figure 1. Components composing a BBW system [6].

The system is modeled using EAST-ADL. In Figure 1, a simplified model of the system for only one wheel is depicted. The internal behavior of system components are captured and represented using Timed Automata (TA). Since BBW is a real-time system with different timing requirements, the use of TA models enable to also include timing specifications. Considering the purpose of a BBW system, it is generally considered as a safety-critical, distributed real-time (embedded) systems. A timed automata model, designed in UPPAAL tool [12], describing the internal behavior of the ABS component of BBW system is shown in Figure 2. In this model, *y* is a clock whose specification on the states indicates the amount of time units that can be spent in each state (non-deterministically, between 0 and the specified value) before a transition has to be made to the next state. These timing specifications are naturally derived from high level timing requirements of the BBW system and its components. The values in the TA model here are just samples, and the exact values for each implementation of the BBW system might be different.

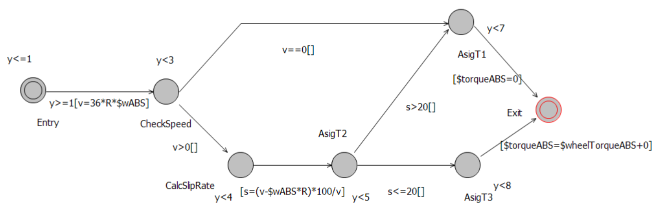


Figure 2. Timed automata model of the ABS component [6].

A. *Testing Goals*

We have developed a testing methodology [6] in order to verify the runtime behavior of the BBW system against the desired behavior represented in the form of state machine and automata models. To do so, test cases are generated from automata models as UPPAAL trace files. These traces serving as *abstract* test cases are then transformed into *concrete* ones, which are essentially executable test scripts. By executing the concrete test cases the runtime behavior of the system is verified in terms of state changes. In other words, it is checked that the order of states of the system at runtime matches what is specified in the models.

Some of the steps that constitute the approach are as follows:

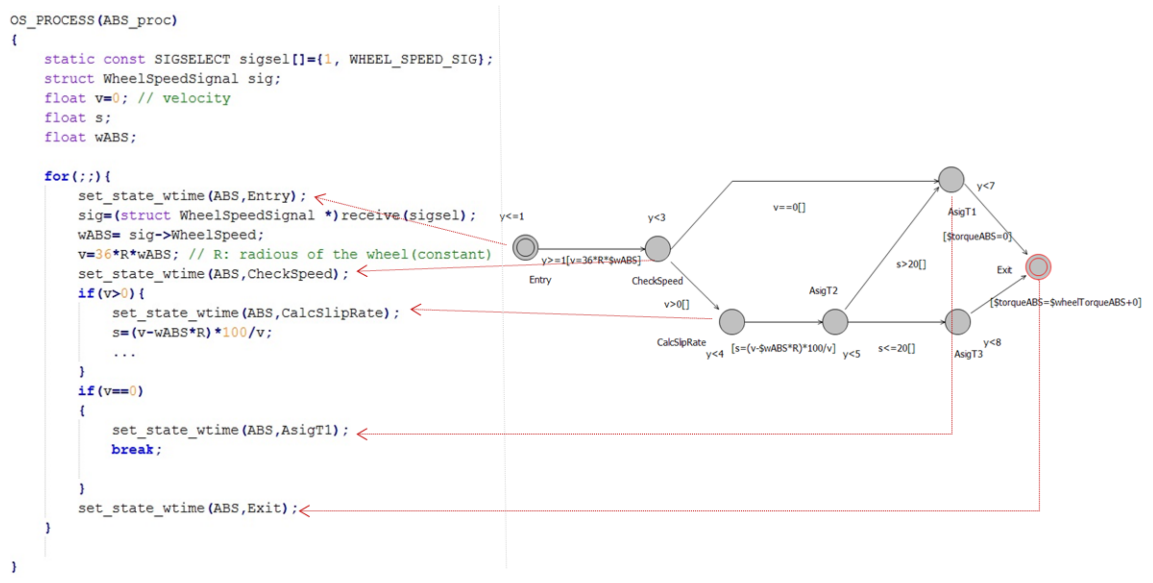
- Based on the automata models, C/C++ enumerations (enum) that represent each state machine and their internal states are generated. These enumeration structures are stored in a C/C++ file along with the definition of a helper function called `set_state(StateMachine,State)`. The file is then included in the implementation code of the target application (i.e., to be tested).
- The states in the automata model are mapped to the code using the above helper function. This is done by adding calls to the `set_state()` helper function at places in the code where a state change occurs. The helper function basically logs the new state belonging to the specified state machine and thus enables to keep track of state changes at runtime.
- According to the automata model, a test script is generated which verifies that the order of state changes (logged using the helper function) match the model. If so, then the result of the test is determined as *pass*, otherwise a *fail* verdict is decided.

This helps to gain more confidence that the behavior of the system is actually as specified and expected at the modeling level. One of the motivations behind our approach is that the models are used for different types of model-based analysis. If the runtime behavior of the system deviates from and does not match the behavioral models, the result of the analyses that have been performed assuming such behavioral models will be violated and not valid anymore.

III. State Machine Mapping

A. *Challenges*

To provide a mechanism for tracking state changes at runtime, the `set_state(StateMachine,State)` helper function that was introduced in the previous section is used to *map* state machine models to code. This mapping step is needed to keep track of different states and how they change at runtime, which is currently done in a manual way. Figure 3 shows how this mapping is done by *annotating* the code and adding calls to the helper function in it. The code shown here is C/C++ code for the ABS component written on OSE Real-Time Operating System (RTOS), which is a commercial and industrial real-time operating system developed by Enea [13]. OSE offers the concept of direct and asynchronous message passing for communication and synchronization between tasks using *send* and *receive* APIs.



```
OS_PROCESS(ABS_proc)
{
    static const SIGSELECT sigsel[]={1, WHEEL_SPEED_SIG};
    struct WheelSpeedSignal sig;
    float v=0; // velocity
    float s;
    float wABS;

    for(;;){
        set_state_wtime(ABS,Entry);
        sig=(struct WheelSpeedSignal *)receive(sigsel);
        wABS= sig->WheelSpeed;
        v=36*R*wABS; // R: radious of the wheel(constant)
        set_state_wtime(ABS,CheckSpeed);
        if(v>0){
            set_state_wtime(ABS,CalcSlipRate);
            s=(v-wABS*R)*100/v;
            ...
        }
        if(v==0)
        {
            set_state_wtime(ABS,AsigT1);
            break;

        }
        set_state_wtime(ABS,Exit);
    }

}
```

Figure 3. Mapping of states to the code (ABS function) [6].

As mentioned, the mapping step is currently done manually. While this works fine for systems with a small code base, it has a big impact for the scalability of the general approach. From this perspective, the manual mapping step can be considered a bottleneck in the automation of the whole approach. Therefore, there is a need and big interest in automating the mapping step by finding a solution to map state machine models to code in order to track state changes at runtime.

B. Potentials

So far, it was discussed how having such a mapping mechanism can help to track the runtime behavior of the system with respect to its internal state changes. Information on internal state changes basically provides an insight about execution flow in the system. On the other hand, state machines used in the modeling phase represent the desired and expected behavior of the system. The mapping mechanism enables to compare the actual versus expected behavior and identify violations. Moreover, it also becomes possible to identify at which state and during which transition a violation has occurred. This, in turn, can help with debugging and better pinpointing the root cause of the problem than the case where there is no such mapping; hence identifying the *vicinity* and localization of potential defects. We have discussed and demonstrated these features in detail in [14], as a method for checking architectural consistency.

Moreover, having the mapping mechanism and being able to track state changes at runtime brings along other interesting testing capabilities. One of such capabilities is to verify timing properties and clock constraints in real-time systems. We have introduced and demonstrated it in [15]. The idea is basically that not only each state change is recorded, but they are also time stamped. This way, it becomes possible to measure the time difference between each pair of states during the actual execution of the system and at runtime. This information is then used to compare the timing behavior of the system versus the timing and clock constraints that are specified in the timed automata models. For example, a clock constraint can be defined in the timed automata model specifying that the system may spend time and remain in a state only for a certain period of time and then it has to make the transition to the next state. Such timing requirements are of great importance in designing real-time systems, but are also hard to actually test at such granularity and level of detail. Timed automata are a formal way of capturing such timing requirements and constraints, which are then used also for analyzing the temporal correctness of the system design. However, despite performing static analysis, at runtime situations may still occur, which lead to the violation of assumptions that have been taken into account for performing the analyses; hence invalidation of the analysis results [16][17]. This further emphasizes the need to be able to actually test a system with respect to its extra-functional properties; and particularly in this case, timing properties which are not as easy and straightforward to perform as functional testing. The approach we have introduced in [15] is one solution towards this purpose, which is based on the state machine mapping mechanism in order to test clock constraints in real-time systems.

C. Related Work & Possible Solutions

There is not much discussion in the literature on establishing a mapping and relationship between state machines and code for the purposes mentioned in this paper. Walkinshaw et al. in [18] discuss the problem of rarely maintaining state machine models during software development by emphasizing and drawing attention to their importance and role in state-based testing techniques. They introduce an approach based on symbolic execution to reverse engineer state transitions from code. However, what we discussed here can be considered as opposite of their approach and with the purpose of tracking state changes at runtime; which their introduced reverse engineering approach does not provide. Moreover, the mapping from the direction of state machine models to code and then runtime tracking of state changes helps to identify situations where the behavior of code deviates from what is specified by the model as the correct behavior. In contrast, merely extracting state machine models from the code just creates the model of how the code behaves, which can contain and represent a wrong behavior. What might be possible here is to reverse engineer state machine model of the code and then compare it with a designed state machine model representing the correct and expected behavior that the system should conform to. In [19], the DiscoTect method and tool are introduced. DiscoTect engines observe and filter system events during execution and at runtime to discover and construct the architecture of the system based on derived states and transitions. The ultimate goal in DiscoTect is to highlight inconsistencies between the implementation and intended architecture by providing the discovered architecture, which can then be compared with intended one. SMArTIC [20] is also an architecture and method for specification mining which is similar to DiscoTect, particularly that the discovered system specification is derived in the form of a finite state automaton. Other examples of such dynamic analysis techniques that derive state machine models from actual program executions are ADABU [21] and GK-tail [22]. The latter aims to capture the interplay between data values and component interactions by annotating state machine models with conditions on data values. FSMGen tool which is introduced in [23] utilizes a symbolic execution technique to statically analyze TinyOS program codes in order to derive state machine models of the system. The advantage that static analysis methods have over dynamic ones (e.g., the ones mentioned above) is that dynamic approaches can capture and analyze only particular runs of an application, while applications can generally have an infinite number of execution traces. All these mentioned approaches try to construct state machine models from code. However, as described earlier, in this paper, our focus is on the other direction which is from existing models (used in model-based analysis) to code and establishing mapping between them.

Another approach to enable tracking of state changes at runtime could be to implement or generate the code in the form of a state machine. In other words, the code is originally designed and written in the form of states and transitions; i.e., an implementation and code representation of the state machine. The Windows Workflow Foundation [24] provides tracking APIs, which make it (easily) possible to implement such an approach as demonstrated in [25]. Another example of this approach could be to have a variable to keep the current state and a switch-case structure (in C/C++) to choose execution blocks based on its value. In [26], where we have presented a more complete and extended version of our testing methodology, this approach is used. In other words, the code contains necessary variables to keep track of different states at runtime. A feature of this mapping approach is that if the code is automatically generated from the models, it can be made to insert and include the necessary variables to keep track of states and transitions as part of the code generation process.

Finally, automation of the manual mapping approach that was introduced in previous sections of the paper can serve as another solution. This requires an 'intelligent' tool, which goes through the code and tries to identify parts that match a state from the model (e.g., based on the guards, actions, and other information in the state machine model). The accuracy of such a tool needs to be considered carefully. For this reason, it may be made as a semi-automatic tool with user interactions to confirm whenever, for example, several matching points are detected for a state (false positives). A feature of this approach though is that it can be very helpful when there is already some code available (as opposed to the code generation approach discussed above), for instance, in legacy systems.

An advantage of the mapping approach in general is that the instrumentation of the code that is done to achieve the mapping can be done just to test the system and removed afterwards from the final product and before the actual deployment. The impacts of such instrumentation and how it may affect test results, particularly, in real-time systems need to also be taken into account, as we have discussed with more details in [15]. Moreover, it should be investigated if a state change always corresponds to only one location in the code, particularly when the target system is parallel or distributed, e.g., in multicore scenarios. In other words, the mapping in some systems might not always be one-to-one but also one-to-many.

IV. Conclusion

In this paper, we discussed the idea of mapping state machine models to code to enable tracking state changes at runtime. Moreover, the advantages and potentials that such a mapping can offer for testing were also presented along with the possible implementation solutions as well as the challenges that exist in implementing it. A manual establishment of mapping between state machines and code is currently being considered as part of a testing methodology for the Volvo's Brake-By-Wire use-case in the MBAT European project. However, the main challenge is that while such a manual mapping might work for a small system, it will not be scalable for systems with large code bases, and therefore, needs to be automated. In summary, the main goal of this paper has been to highlight the benefits and uses of having the mapping mechanism, discuss its feasibility, and encourage research on methods for automatic establishment of the mapping as well as its further use in testing.

V. Acknowledgements

This work has been supported by the MBAT European Project [10] and also through the ITS-EASY industrial research school [27]. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no 269335 (see Article II.9. of the JU Grant Agreement) and from the Swedish Governmental Agency for Innovation Systems (VINNOVA). We would also like to thank Raluca Marinescu and Dr. Cristina Seceleanu for their technical tips and support for this work.

References

[1] B. Selic, "The pragmatics of model-driven development," Software, IEEE, vol. 20, no. 5, Sept 2003, pp. 19–25.

[2] J. Bezivin, "On the unification power of models," Software Systems Modeling, vol. 4, no. 2, 2005, pp. 171–188.

[3] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Software Testing, Verification and Reliability journal, vol. 22, no. 5, Aug. 2012, pp. 297–312.

[4] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One evaluation of model-based testing and its automation," in Proceedings of the 27th international conference on Software engineering, ser. ICSE, New York, USA, 2005, pp. 392–401.

[5] T. S. Chow, "Testing software design modeled by finite-state machines," IEEE Trans. Softw. Eng., vol. 4, no. 3, May 1978, pp. 178–187.

[6] R. Marinescu, M. Saadatmand, A. Bucaioni, C. Seceleanu, and P. Pettersson, “EAST-ADL Tailored Testing: From System Models to Executable Test Cases,” Mälardalen University, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-278/2013-1-SE, August 2013.

[7] R. Alur and D. L. Dill, “A theory of timed automata,” Theoretical Computer Science, vol. 126, no. 2, 1994, pp. 183 – 235.

[8] J. Bengtsson and W. Yi, “Timed automata: Semantics, algorithms and tools,” in In Lecture Notes on Concurrency and Petri Nets, ser. Lecture Notes in Computer Science vol 3098, W. Reisig and G. Rozenberg, Eds. Springer–Verlag, 2004.

[9] The ATESST Consortium, “EAST-ADL Profile Specification.” www.atesst.org, Accessed: August 2014.

[10] MBAT Project: Combined Model-based Analysis and Testing of Embedded Systems, http://www.mbat-artemis.eu/home/, Accessed: August 2014.

[11] S. Anwar, “An anti-lock braking control system for a hybrid electromagnetic/electrohydraulic brake-by-wire system,” in American Control Conference, 2004. Proceedings of the 2004, vol. 3, 2004, pp. 2699–2704.

[12] G. Behrmann, R. David, and K. G. Larsen, “A tutorial on Uppaal 4.0,” http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, November 2006.

[13] Enea, http://www.enea.com, Accessed: August 2014.

[14] M. Saadatmand, D. Scholle, C. W. Leung, S. Ullström, and J. F. Larsson, “Runtime verification of state machines and defect localization applying model-based testing,” in Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon) - Proceedings of the WICSA 2014 Companion Volume, ser. WICSA ’14 Companion. ACM, 2014, pp. 6:1–6:8.

[15] M. Saadatmand and M. Sjodin, “Testing of timing properties in real-time systems: Verifying clock constraints,” in Software Engineering Conference (APSEC, 2013 20th Asia-Pacific, vol. 2. IEEE-CPS, Dec 2013, pp. 152–158.

[16] S. Chodrow, F. Jahanian, and M. Donner, “Run-time monitoring of real-time systems,” in Real-Time Systems Symposium, 1991. Proceedings., Twelfth, dec 1991, pp. 74 –83.

[17] M. Saadatmand, A. Cicchetti, and M. Sjödin, “Design of adaptive security mechanisms for real-time embedded systems,” in Proceedings of the 4th international conference on Engineering Secure Software and Systems, ser. ESSoS’12. Eindhoven, The Netherlands: Springer-Verlag, 2012, pp. 121–134.

[18] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe, “Automated discovery of state transitions and their functions in source code,” Journal of Software Testing, Verification & Reliabality, vol. 18, no. 2, Jun. 2008, pp. 99–121.

[19] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, “Discotect: A system for discovering architectures from running systems,” in Proceedings of the 26th International Conference on Software Engineering, ser. ICSE ’04. Washington, DC, USA: IEEE, 2004, pp. 470–479.

[20] D. Lo and S.-C. Khoo, “SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner,” in Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. SIGSOFT ’06/FSE-14. New York, NY, USA: ACM, 2006, pp. 265–275.

[21] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, “Mining Object Behavior with ADABU,” in Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, ser. WODA ’06. New York, NY, USA: ACM, 2006, pp. 17–24.

[22] D. Lorenzoli, L. Mariani, and M. Pezze, “Automatic generation of software behavioral models,” in Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference on, May 2008, pp. 501–510.

[23] N. Kothari, T. Millstein, and R. Govindan, “Deriving state machines from tinyos programs using symbolic execution,” in Information Processing in Sensor Networks, 2008. IPSN ’08. International Conference on, April 2008, pp. 271–282.

[24] Microsoft Windows Workflow Foundation, http://msdn.microsoft.com/en-us/vstudio/jj684582.aspx, Accessed: August 2014.

[25] To use State Tracking with WorkflowApplication, http://wf.codeplex.com/wikipage?title=Tracking%20states%20with%20WorkflowApplication, Accessed: August 2014.

[26] R. Marinescu, M. Saadatmand, A. Bucaioni, C. Seceleanu, and P. Pettersson, “A model-based testing framework for automotive embedded systems,” in 40th Euromicro Conference on Software Engineering and Advanced Applications. Verona, Italy: CPS-IEEE, August 2014.

[27] ITS-EASY post graduate industrial research school for embedded software and systems, http://www.mrtc.mdh.se/projects/itseasy/, Accessed: August 2014.