LLFSMs on the PRU: Executable and Verifiable Software Models on a Real-Time Microcontroller

Fisher Grubb[1(✉)], Vladimir Estivill-Castro[2], and René Hexel[1]

[1] Griffith University, Brisbane, QLD 4111, Australia
fisher.grubb@gmail.com, rhexel@griffith.edu.au
[2] Universitat Pompeu Fabra, 08018 Barcelona, Spain
vladimir.estivill@upf.edu

Abstract. The Internet of Things (IoT) has hugely expanded. There are more devices than people and device numbers are still growing exponentially. These IoT devices are becoming more important and integrated into our lives, and their sophisticated behaviour has prompted the term "Internet of Behaviours". Model-Driven Software Engineering (MDSE) and especially formal verification are applied to safety-critical systems, such as automotive and aerospace, to ensure behaviour correctness, but their use is not common place in the deployment of IoT devices. This paper introduces the deployment of executable and verifiable models for boards with hybrid CPUs; such as Asymmetrical Multi Processing (ASM), including the programmable real-time units (PRUs) in the TI BeagleBone series boards. To the best of our knowledge, this is the first time Model-Driven Software Engineering has been used for applications in PRUs. Thus, we lift the level of abstraction from the use of assembly and C to even beyond C++, using logic-labelled state-machines (LLFSMs). We benchmark MDSE with an existing PRU real-time application [1].

1 Introduction

Computer power is steadily increasing in small devices. A clear example of this capability is the Beagle-bone Black and its Programmable Real-time Unit (PRU) [2]. The PRU is extremely interesting because it offers the developer the capability of real-time programming. Real-time applications have become more in demand in transportation, medical devices, aerospace and the Internet of Things (IoT) [3–7]. Despite Texas Instruments offering a C/C++ compiler [8], applications are only built using direct coding. This contrasts with more modern Model-Based Systems Engineering (MBSE) [9] and Model-Driven Software Development (MDSD) [10], whose advantages include faster development, meaningful validation, increased quality, and a road-map to simulation and verification. Model-driven Engineering (MDE) is considered a crucial approach to minimise the development complexity for cyber-physical systems [11].

L. Borzemski et al. (Eds.): ICSEng 2021, LNNS 364, pp. 391–402, 2022.
https://doi.org/10.1007/978-3-030-92604-5_35

In this paper, we show, to the best of our knowledge, the first, direct, bare-metal deployment of executable behaviour models. Previously, logic-labelled finite-state machines (LLFSMs) had been used with an operating system scheduler [12] and in FPGAs [13]. We would like to highlight two immediate advantages. First, the arrangement of logic-labelled finite-state machines can be scheduled in a time-triggered approach. This enables predictable execution, calculation of the worst-case execution time (WCET), and most importantly, compliance with real-time requirements (that is, guarantees to meet hard deadlines) [14]. Second, we can produce Kripke structures and formally verify the system without semantic gaps [14]. That is, the Kripke structure verified by the model checker has exactly the same semantics as its execution.

This paper focuses on running verifiable software on a real-time processor sub-system of the Sitara family of system-on-a-chip (SoC) processors, in particular the powerful Beagle-Bone AI (four core PRU subsystem, compared to two previously). These PRUs aid the main ARM Cortex-A as high-speed, general-purpose microcontrollers with real-time calculations and IO interaction with external hardware. Examples include audio processing with sub-millisecond latency [15], real-time sensing and control [16], deterministic execution and enabling real-time target-detection via a frequency-modulated ultrasonic sensor [17], collection and processing of real-time high-resolution hydro-acoustic data [18], improving performance of a three-phase micro-inverter [19], and accelerating visible light communication [20].

We will discuss MDSE in Sect. 2, contrasting our approach with standards such as the UML and with tools such as Simulink. Section 3 details concurrently executing LLFSMs and scheduling arrangements. We illustrate LLFSMs by also describing our case study, re-engineering a file transmission system using LLFSMs, inspired by the demo-software released as part of *Purple VLC* [20]. We benchmark MDSE with an existing PRU real-time application [1]. We illustrate that LLFSM models are abstract enough for the same model to be compiled for the PRU as well as a common development host environment, where testing and verification can be performed. In Sect. 4, we show that models can directly generate SMV code, not only for testing and formal verification by tools such as NuXMV, but also for simulation by such model checkers, minimising semantic gaps. Section 5 provides final remarks.

2 The Case for MDSE and Modelling with LLFSMs

Modelling is seen as such a fundamental activity to systems engineering that the MBSE initiative [21] of the International Council of Systems Engineering (INCOSE) emphasises a repository approach for systems engineering, built on a meta-schema of modelling and quality assurance through formal models offering consistency validation across different diagram types and viewpoints. Such is the power of modelling that large industrial consortia endorse MBSE [21], particular for the model-driven development of complex systems. Although MBSE strongly suggests to automatically derive executables from models, typically this is not

fully possible. Tools (e.g., Modelica [22], Ptolomeo [23] and Simulink [24]) have made MBSE ideas widely adopted, but use several abstractions and assumptions. Formal models capture requirements into specifications, enabling validation of requirements, generating test plans (from unit-tests to integration and acceptance tests). Showing that the model is correct (even if not directly executable) establishes the model as a formal specification. Simulation is also advantageous, allowing semi-automatic tests, by comparing the results of the simulation against the implementation. This raises confidence in the correctness of the implementation modulo the semantic gap between the assumptions in the simulation and the actual implementation [25]. For instance, Simulink has been recently heralded as a tool for MDSE and formal verification of software for safety-critical systems, for example in aerospace applications [26], which illustrate some of the issues with Simulink MDSE. In some cases, the Simulink model is analysed with formal verification tools (Simulink Design Verifier (SLDV) [27], or NuSMV [28]), but later, is manually (humanly) programmed in assembly or C. Intensive testing between the implementation and the model needs to be carried out to gain confidence the implementation meets the specification [25,29].

These practices imply further semantic gaps: implementation code is not directly extracted from the model. Also, when C or C++ is generated by Simulink, it contains large amounts of boilerplate code implementing specific interpretations, e.g. for port activation and signal propagation, in the semantics of Simulink's diagrams, often making these executables unsuitable for PRU real-time applications.

We should also comment on the potential use of the Unified Modelling Language (UML) for MDSE of code that shall run on processors such as the PRU. Unsurprisingly, the dominant role the UML has taken as a standard has resulted in a series of proposals for UML MDSE of embedded systems [30–32] The promise of these proposals is that of MDSE: standardisation, capturing and validating requirements, platform independence, automatic code generation, UML-extension. However, the disadvantages rapidly materialise [32]: too many diagrams, imprecise semantics, and not enough formality for embedded systems. UML has been used for several embedded software applications [33, Section 18.9], but usually as a high level design, before defining more formal models such as process algebra or coloured Petri nets. UML has seen a number of revisions aimed at simplification, but many [31,34] still identify numerous imprecisely defined and overlapping concepts. For instance, although SysML [35] (a general-purpose modelling language for systems engineering, defined as an extension of a subset of UML) is a simpler language (only 8 diagram types compared to of 14 in UML), SysML inherits similar challenges with regards to semantics and model execution [31]. Similarly, the Foundational Subset for Executable UML Models (fUML) has been criticised because of its lack of clarity. Along the same lines, UML MARTE, which originally was designed for model-based design and analysis of real-time and embedded software of cyber-physical systems [36], is seen more as a specification tool than to construct executable models [31]. Many other disadvantages of using UML for development on FPGAs are discussed

elsewhere [13]. Thus, UML is far from being a modelling language that enables formal verification and direct execution without semantic gaps in hardware such as the PRU. Examples of such semantic gaps include the noticeable discrepancies when running Papyrus-RT UML models with their nuXmv simulation [37], or the observation that value-domain LTL properties may hold with one semantics but not with another [38]. Issues with the semantics of UML's state charts have been raised long ago [39]; however, matters have not improved. Posse and Dingel [40, Section 6] cite over 30 references attempting to produce a formal semantics for UML-RT (a subset of UML for real-time systems). Moreover, although Papyrus-RT is supposed to be a reference implementation to directly generate C++ code from models, this requires the Eclipse CDT, which in itself requires high level process and thread management in a multi-threaded operating system (at the moment only Linux and Cygwin), which definitely rules it out for bare-metal, embedded systems.

Because the PRUs are programmable in assembly language (`pasm`), C, or C++, and lack any higher-level tools, the cycle-level operations (Cyclops) suite has been proposed [41]. This suite provides a web-browser-based integrated development environment (IDE) for the PRU, a library of common functions for interfacing with various peripherals, and a C library for time synchronisation with the main processor clock via a Linux kernel module. However, the Cyclops approach forces the programmer into an even more restricted, text-based programming [41]. The Cyclops authors argue this language combines the convenience of C with the determinism of `pasm`; but the Cyclops language supports only a small feature set: variable assignment, arithmetic expressions, if/else conditional statements, and while-loops, but not for-loops, functions, or floating-point arithmetic. The authors of Cyclops suggests that this "forces the programmer to write simple code more appropriate for a real-time application, with more deterministic runtimes" [41].

By contrast, we advocate the use of higher levels of abstraction, but simultaneously executable and verifiable models, following the practice of MDSE for complex software design. Research has confirmed that MDSE has numerous benefits, including reduced development time [42–44]. We will demonstrate MDSE for software on bare metal, i.e. the PRU. MDSE equates development to the construction of the software specification. We postulate the use of logic-labelled finite-state machines (LLFSMs) to achieve this higher-level conceptual thinking of software development.

3 Using Logic-Labelled Finite-State Machines (LLFSMs)

LLFSMs are a form of executable state machines where systems behaviour is still modelled as states and transitions, but transitions are labelled by Boolean expressions. Complementary to UML statecharts [33] (and its variants), and the event-driven nature of UML, LLFSMs take a time-triggered approach [45,46] (the only events are system-clock events). Time-triggered systems have the advantage that they are easier to prove correct [47]. As opposed to an event-driven system, where there are call-backs or interrupts that are enacted as events

arrive, LLFSMs can be considered as polling based. This is suitable for bare-metal implementations of embedded systems and in particular for the PRU, where all communication with the ARM processors in the Beagle-Bone is performed using polling mechanisms.

LLFSMs are executable models of software behaviour under a precise semantics [14]. UML's statecharts are direct descendants of behaviour models of reactive systems as proposed by STATEMATE [48] and then OMT [49]. Mellor proposed Executable UML [50,51], creating a subset of UML's statecharts where the model would translate immediately to implementation. LLFSMs are state machines, with the particular aspect that they are not event-driven. All transitions are labelled by Boolean expression only (and this includes function calls, or to more sophisticated aspects that return true versus not true—such as expert systems, theorem proving systems, and other tools that enable a deliberative nature and not a purely reactive nature). Such conceptualisation of finite-state models appeared in the LISP Augmented Finite State Machines of the subsumption architecture for robotic systems [52], and is present in the Ptolomeo modelling suite [23]. It is also evident in avionic systems where transition tables indicate conditions as the Boolean expression that shift the system from one mode to another [26].

To further explain LLFSMs and their execution on the PRUs we also introduce our case study that was inspired by the *Purple VLC* system for data transmission. We extend beyond the *Purple VLC* demo where only a string of 100 characters was transmitted by placing it in the command line of a program for the ARM-side (linux-debian) of the system. We enable transmission of binary files or text files of arbitrary size. Our system consists of 4 programs, each constructed as an LLFSM. A file residing on the ARM-side (linux-debian) is transmitted using the RPMessage interface (`/dev/rpmsg_30`) to PRU-0. The behaviour (on the PRU-0) partitions the stream into packets of 24 bytes that can be forwarded to the PRU-1 though the XFR broadside bus, using Register Transfers. The PRU-1 reassembles the packets to produce a stream for a second RPMessage interface. A second program on the ARM (linux-debian) reads from its corresponding RPMessage interface (`/dev/rpmsg_31`) and captures the corresponding file. To monitor the data transmission status, we used LEDs in our demonstration software [1] (see https://youtu.be/fEuk4mN8cfU).

Figure 1 shows the first executable LLFSM that is compiled with a standard C++ compiler, say `gcc` (in our case, we used Ubuntu on an Intel for development), but for our demonstration we compiled and executed on the ARM's debian. This illustrates the powerful abstractions from executable environments that MDSD produces. The notation of LLFSMs has many elements in common with UML. In particular, each machine has an initial pseudo-state indicated by the small solid filled circle. Execution will commence here. A scheduler awards each turn in round robin fashion to all machines in the same arrangement. In our case study all arrangements consist of only one machine; however, our scheduler (ucFSM) can execute more than one machine concurrently (on the ARM or on the PRU) but in sequential order in a single thread.

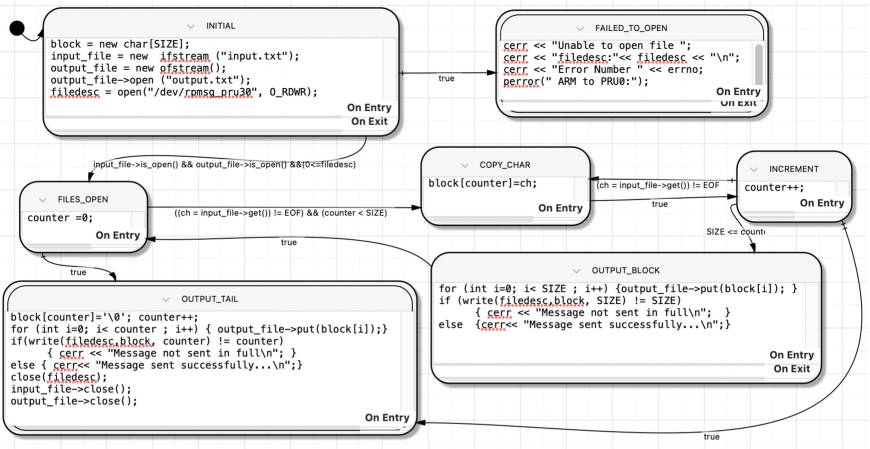

Fig. 1. LLFSM that transmits a text file from the ARM to PRU0.

Thus, at each point during execution, a LLFSM has a current state (the initial pseudo-state to commence). When a machine receives its turn, it executes a ringlet, which is essentially the step of running the code in the current state. Because states can have On-Entry, On-Exit or Internal sections, the ringlet may be different. In our example of Fig. 1 all states have C++ code only in their On-Entry sections. A machine that uses the other sections can be converted to an equivalent machine that uses only On-Entry sections [14]. When a machine first executes the current state, the code in the On-Entry section is executed. Then, the transitions (out of the current state) are evaluated in a predefined static sequence. For example, in Fig. 1 there is a transition from `INITIAL` to `FILES_OPEN` and also from `INITIAL` to `FAILED_TO_OPEN`. However, the arrow in the first transition has no mark, while the transition from `INITIAL` to `FAILED_TO_OPEN` has one mark. Thus, despite this later transition (labelled with the Boolean expression `true`) would always succeed, it will only fire if the first transition evaluates to false (which tests whether the program could set up the RPMessage device `/dev/rpmsg_pru30`). So, the second transition is analogous to the `else` part of an `if` control statement. Similarly, there are three transitions out of the state `INCREMENT`, the third one has two ticks and has target state `OUTPUT_TAIL`. States that have no outgoing transitions are terminal and the scheduler will terminate if that state is reached. In this case `OUTPUT_TAIL` and `FAILED_TO_OPEN` are terminal. When no transition fires, the machine executes the internal section and relinquish its turn until being awarded a turn again. When a transition fires, the code in the On-Exit section runs before updating the current state to the target state of the transition that has fired.

We argue that the behaviour of the program in Fig. 1 is visually apparent. The machine attempts to open 1) the input file for reading, 2) an output file to locally produce a copy of the file and 3) an RPMessage device to send a

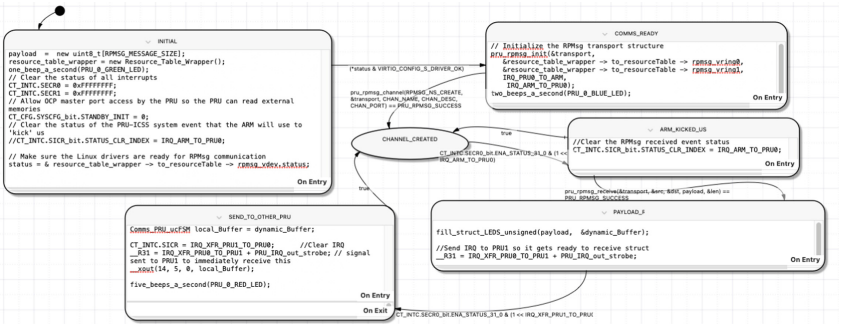

Fig. 2. LLFSM that transmits blocks of 24 bytes in registers from PRU0 to PRU1.

stream of bytes to PRU-0. If it does not succeed, it terminates. Otherwise, by evaluating the first transition out of `FILES_OPEN` it reads a byte. If it obtains the EOF flag and could not count to 24 bytes, it will transfer to the terminal state to `OUTPUT_TAIL` of the file. However, otherwise it transfers to `COPY_CHAR` where the character is added in a buffer, and the `counter` variable is increased in the state `INCREMENT` to account for one character read. In `INCREMENT`, the first transition attempts to read the next character, and while this succeeds, the behaviour loops between `COPY_CHAR` and `INCREMENT` until the buffer is full (that is, the `counter` variable is larger than the buffer size), when that happens, the block is forwarded to the PRU-0 using the RPMessage utilities (in our code there is also the mirroring to an output file that is not strictly necessary for file transmission). A EOF signal in the loop between `COPY_CHAR` and `INCREMENT` will take the behaviour to the `OUTPUT_TAIL` to send the tail of the file (since the input file's size may not be a multiple of 24 bytes). This machine transmits text files and since between the PRUs there is no EOF signal, a null byte is appended at the end of the last block.

Figure 2 shows the second machine in the demonstration, which runs on PRU0. The diagrams for receiver machine on PRU1 and for the machine on the ARM which receives from PRU1 are part of the downloadable files [1].

4 Formal Verification

One of the aspects that may not be so apparent of the machine in Fig. 1 is that it works for all file lengths, and in particular, when a file has length that is a multiple of 24 bytes. Perhaps in this special case, the transition to end the file may miss the last character. By automatically translating the structure of this LLFSMs to a model checker using tools released for ATL model-to-text transformation [14], we can verify that the `OUTPUT_TAIL` state is never reached unless the EOF flag has been found and that all the files characters are in the buffer. This shows how delicate is that the transition from `INCREMENT` to `OUTPUT_BLOCK` is evaluated first, after the `counter` has been incremented, and

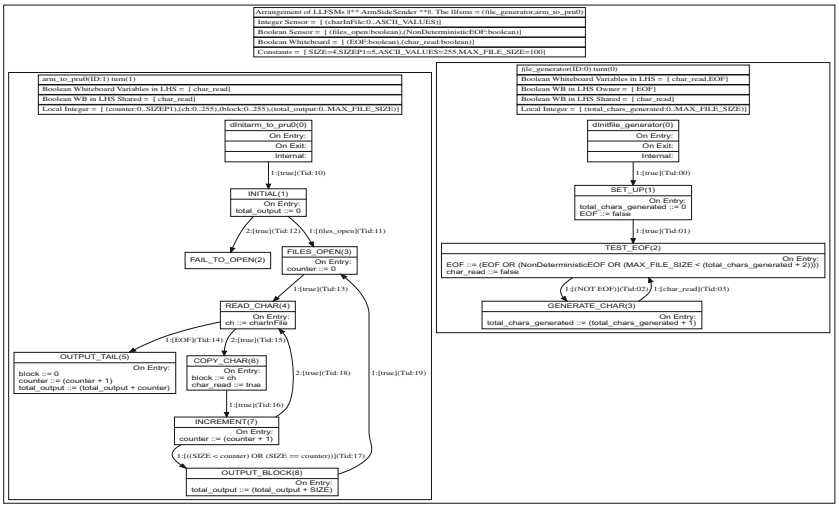

Fig. 3. Arrangement of two LLFSMs that represents the closed model of Fig. 1.

the second transition from `INCREMENT` to `COPY_CHAR` attempts to read a `char` only when there is room in the buffer. If this evaluation results in the EOF flag, then `OUTPUT_TAIL` is reached but with all bytes already in the buffer.

The ATL model-to-text transformation [14] produces NuSMV code that can be simulated with the simulator options of NuSMV, but some adjustment must be made. First, transitions with an assignment such as `ch = input_file->get()` imply an intermediate state (Fig. 3 shows a state `READ_CHAR`). Also, the NuSMV model checker does not simulate files being opened or writing to an RPMessage device. But the NuSMV simulation allows us to raise the EOF signal at any point in time, and in particular, when the file length is a multiple of the buffer size. The simulation in NuSMV (the firing of transitions and the trace trough states) is completely loyal to the C++ version that runs on the arm.

Moreover, we can create a closed model (refer to Fig. 3). This is an arrangement of LLFSMs with one more LLFSM that selects non-deterministically the length of the file, and feeds a file of exactly that length with arbitrary ASCII characters to the machine of Fig. 1. We also can verify the correctness for different buffer sizes. We can then formally verify with NuSMV [28] (the NuSMV code for this verification is also released with our demonstration software) LTL properties such as the following.

1. If `OUTPUT_TAIL` is reached, then end of file must have happened.
   `G (arm_to_pru0.At_OUTPUT_TAIL -> file_generator.At_TEST_EOF & w_b.EOF )`
2. If opening the file succeeds and we reach end of file, we eventually reach the state `OUTPUT_TAIL`.
   `G ( arm_to_pru0.At_FILES_OPEN -> F arm_to_pru0.At_OUTPUT_TAIL)`
3. If `OUTPUT_TAIL` is reached, the number of characters forwarded to the PRU0 is one more than the size of the input file.
   `G (arm_to_pru0.At_OUTPUT_TAIL -> arm_to_pru0.total_output = file_generator.total_chars_generated + 1)`

Similarly, for formal verification of the executable LLFSM for the PRU-0 (the one in Fig. 2) we created a closed model that represents the hand-shake with the ARM program in Fig. 1, the program in the PRU-1 and the debian kernel on the BeaggleBone Black (with what we believe is our faithful interpretation of TI's documentation regarding the rpmsg behaviour). In this case, we can also prove similar properties. In particular, that the PRU0 does not drop packets[1].

1. If `OUTPUT_TAIL` is reached in the LLFSM sending the file from the arm and the machine in PRU-1 completes its handshake with the machine in PRU-0, then the number of blocks forwarded by the arm side to the PRU0 is the same as those received at PRU1.
   ```
   G ((arm_to_pru0.At_OUTPUT_TAIL & pru1_to_arm.At_RESET) -> F ( G
   arm_to_pru0.total_blocks_forwarded = pru1_to_arm.total_blocks_received))
   ```

We should point out that the current formal verification paces the machines that correspond to the environment to the needs of the machine being tested. While this verifies each of our four machines as they would run on their environment (whether PRU-side or ARM-side), it does not verify the integration of the four machines. We leave for further work the formal verification where all machines run at their own arbitrary speed from the others, and for the verification of real-time deadlines. However, the first issue, the verification of the integration, corresponds to an extensions that is not difficult. In this case, what is needed is for the scheduler used for the model checking in the NuSMV Kripke structure to award the turns to machines in the arrangement non-deterministically (rather that in round-robin fashion). We expect to have results on this soon.

5 Final Remarks

We have proposed a systems engineering approach to software development based on executable models that are a common framework to the different processor core designs of the same SoC. This offers the huge benefits of MBSE to application development despite the ARM cores most commonly run Linux, a non-real-time OS, while the PRU cores are specifically real-time running separate tasks concurrently to the ARM. We have illustrated how LLFSMs stream-line development for all processors within a single SoC since also we can compile and test on a separate x86 PC. Moreover, we have shown that they can be directly transformed to simulation and verification on a model checker.

References

1. Grubb, F., Estivill-Castro, V.: LLFSMs on the PRU real-time microcontroller (2021). https://github.com/fgrubb/LLFSMs-on-the-PRU-Real-Time-Microcontroller
2. Molloy, D.: Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux. Wiley, New York (2014)

[1] We are thankful to Miguel Carrillo for ensuring LTL-formulas match the verified property.

3. Kushal, K.S., Nanda, M., Jayanthi, J.: Formal methods and tools for safety of critical systems. In: Nanda, M., Jeppu, Y. (eds.) Formal Methods for Safety and Security, pp. 13–21. Springer, Singapore (2018). https://doi.org/10.1007/978-981-10-4121-1_2
4. Bambagini, M., Marinoni, M., Aydin, H., Buttazzo, G.: Energy-aware scheduling for real-time systems: a survey. ACM Trans. Embed. Comput. Syst. **15**(1), 1–34 (2016). https://doi.org/10.1145/2808231
5. Gomaa, H.: Real-Time Software Design for Embedded Systems. Cambridge University Press, Cambridge (2016). https://doi.org/10.1017/CBO9781139644532
6. Rizwan, P., Suresh, K., Babu, M.R.: Real-time smart traffic management system for smart cities by using internet of things and big data. In: International Conference on Emerging Technological Trends (ICETT), pp. 1–7 (2016). https://doi.org/10.1109/ICETT.2016.7873660
7. Malek, Y.N., et al.: On the use of IoT and big data technologies for real-time monitoring and data processing. Procedia Comput. Sci. **113**, 429–434 (2017). https://doi.org/10.1016/j.procs.2017.08.281
8. Texas Instruments: PRU optimizing C/C++ compiler v2.3. Literature Number: SPRUHV7C (2018)
9. Rodrigues da Silva, A.: Model-driven engineering: a survey supported by the unified conceptual model. Comput. Lang. Syst. Struct. **43**, 139–155 (2015). https://doi.org/10.1016/j.cl.2015.06.001
10. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, 2nd edn. Morgan & Claypool, San Rafael (2017)
11. Mohamed, M.A., Kardas, G., Challenger, M.: Model-driven engineering tools and languages for cyber-physical systems-a systematic literature review. IEEE Access **9**, 48605–48630 (2021). https://doi.org/10.1109/ACCESS.2021.3068358
12. Estivill-Castro, V., Hexel, R., Lusty, C.: High performance relaying of C++11, objects across processes and logic-labeled finite-state machines. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) SIMPAR 2014. LNCS (LNAI), vol. 8810, pp. 182–194. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11900-7_16
13. Estivill-Castro, V., Hexel, R., McColl, M.: High-level executable models of reactive real-time systems with logic-labelled finite-state machines and FPGAs. In: International Conference on ReConFigurable Computing and FPGAs, ReConFig, pp. 1–8. IEEE (2018)
14. Carrillo, M., Estivill-Castro, V., Rosenblueth, D.A.: Verification and simulation of time-domain properties for models of behaviour. In: Hammoudi, S., Pires, L.F., Selić, B. (eds.) MODELSWARD 2020. CCIS, vol. 1361, pp. 225–249. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67445-8_10
15. McPherson, A.P., Zappi, V.: An environment for submillisecond-latency audio and sensor processing on BeagleBone Black. J. Audio Eng. Soc. (2015)
16. Anand, A.M., Raveendran, B., Cherukat, S., Shahab, S.: Using PRUSS for real-time applications on Beaglebone Black. In: Third International Symposium on Women in Computing and Informatics, NY, USA, WCI 2015, pp. 377–382. ACM (2015)
17. Kepa, K., Abaid, N.: Development of a frequency-modulated ultrasonic sensor inspired by bat echolocation. In: International Society for Optics and Photonics, SPIE, Bellingham, vol. 9429, pp. 175–182 (2015)
18. Travaglione, B.: Using a single-board microcontroller and ADC to perform real-time sonar signal processing. In: 2nd Acoustical Societies Conference, pp. 1–8 (2016)

19. Götz, M., Gobetti, M.W., Líbano, F.B.: A grid-tie micro-inverter software development based on a low cost multiprocessor platform. In: Brazilian Symposium on Computing Systems Engineering (SBESC), New Jersey, pp. 122–127. CPS IEEE (2015)
20. Yin, S., Smaoui, N., Heydariaan, M., Gnawali, O.: Purple VLC: accelerating visible light communication in room-area through PRU offloading. In: International Conference on Embedded Wireless Systems and Networks, EWSN 2018, Junction, pp. 67–78 (2018)
21. Estefan, J.A.: Survey of model-based systems engineering (MBSE) methodologies. Technical report, INCOSE-TD-2007-003-01, Seattle, WA, USA (2008)
22. Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach, 2nd edn. IEEE Press, Wiley, Hoboken (2015). https://doi.org/10.1002/9781118989166
23. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation Using Ptolemy II. Ptolemy.org (2014)
24. Documentation Simulink: Simulation and model-based design (2020). https://www.mathworks.com/products/simulink.html
25. Jeppu, Y., Rey, G.J., Apte, P.R.: Generating test cases with 100-percent requirements coverage using design of experiments. J. Aerosp. Inf. Syst. **11**(10), 632–648 (2014)
26. Jeppu, N., Jeppu, Y.: Arguing formally about flight control laws using SLDV and NuSMV. In: Nanda, M., Jeppu, Y. (eds.) Formal Methods for Safety and Security, pp. 73–84. Springer, Singapore (2018). https://doi.org/10.1007/978-981-10-4121-1_7
27. Documentation SLDV: Simulink design verifier (2020). http://in.mathworks.com/products/sldesignverifier/
28. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: a new symbolic model checker. Int. J. Softw. Tools Technol. Transf. **2**(4), 410–425 (2000). https://doi.org/10.1007/s100090050046
29. Jeppu, N.: Exploring Simulink design verifier 03. MATLAB Central File Exchange (2021). https://www.mathworks.com/matlabcentral/fileexchange/54945-exploring-simulink-design-verifier-03. Accessed 27 July 2021
30. Kaur, A., Arora, R.: Application of UML in real-time embedded systems. Int. J. Softw. Eng. Appl. (IJSEA) **3**(2), 59–70 (2012)
31. Amissah, M., Toba, A.L., Handley, H., Seck, M.D.: Towards a framework for executable systems modeling: an executable systems modeling language (ESysML). In: Model-driven Approaches for Simulation Engineering Symposium, SpringSim (Mod4Sim) 2018, pp. 9:1–9:12. ACM (2018)
32. Fernandes, J.M., Machado, R.J.: Can UML be a system-level language for embedded software? In: Kleinjohann, B., Kim, K.H., Kleinjohann, L., Rettberg, A. (eds.) Design and Analysis of Distributed Embedded Systems. ITIFIP, vol. 91, pp. 1–10. Springer, Boston, MA (2002). https://doi.org/10.1007/978-0-387-35599-3_1
33. El Ariss, O., Xu, D.: System modeling with UML state machines. In: Handbook of Finite State Based Models and Applications, pp. 371–386. Chapman and Hall/CRC, New York (2012). https://doi.org/10.1201/b13055-19
34. Kobryn, C.: UML 3.0 and the future of modeling. Softw. Syst. Model. **3**(1), 4–8 (2004). https://doi.org/10.1007/s10270-004-0051-4
35. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann, San Francisco (2009)

36. Selic, B., Grard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems, 1st edn. Morgan Kaufmann, San Francisco (2013)
37. Sahu, S., Schorr, R., Medina-Bulo, I., Wagner, M.: Model translation from Papyrus-RT into the nuXmv model checker. In: Cleophas, L., Massink, M. (eds.) SEFM 2020. LNCS, vol. 12524, pp. 3–20. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67220-1_1
38. Besnard, V., Brun, M., Jouault, F., Teodorov, C., Dhaussy, P.: Unified LTL verification and embedded execution of UML models. In: 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, New York, NY, USA, MODELS 2018, pp. 112–122. ACM (2018). https://doi.org/10.1145/3239372.3239395
39. Beeck, M.: A comparison of Statecharts variants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994. LNCS, vol. 863, pp. 128–148. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58468-4_163
40. Posse, E., Dingel, J.: An executable formal semantics for UML-RT. Softw. Syst. Model. **15**(1), 179–217 (2014). https://doi.org/10.1007/s10270-014-0399-z
41. Alanwar, A., Anwar, F.M., Zhang, Y., Pearson, J., Hespanha, J., Srivastava, M.B.: Cyclops: PRU programming framework for precise timing applications. In: IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), New Jersey, pp. 1–6. IEEE (2017)
42. Heijstek, W., Chaudron, M.R.V.: Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process. In: 35th Euromicro Conference on Software Engineering and Advanced Applications, Los Alamitos, CA, pp. 113–120. IEEE (2009)
43. Krogmann, K., Becker, S.: A case study on model-driven and conventional software development: the palladio editor. In: Software Engineering 2007 - Beiträge zu den Workshops, Fachtagung des GI-Fachbereichs Softwaretechnik, 27–30 March 2007, Hamburg, GI, LNI, vol. P-106, pp. 169–175 (2007)
44. Papotti, P.E., do Prado, A.F., de Souza, W.L., Cirilo, C.E., Pires, L.F.: A quantitative analysis of model-driven code generation through software experimentation. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 321–337. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38709-8_21
45. Kopetz, H.: The time-triggered model of computation. In: 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279), pp. 168–177 (1998). https://doi.org/10.1109/REAL.1998.739743
46. Kopetz, H., Bauer, G.: The time-triggered architecture. Proc. IEEE **91**(1), 112–126 (2003). https://doi.org/10.1109/JPROC.2002.805821
47. Furrer, F.: Future-Proof Software-Systems: A Sustainable Evolution Strategy. Springer Vieweg, Berlin (2019). https://doi.org/10.1007/978-3-658-19938-8
48. Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts: The Statemate Approach. McGraw-Hill, New York (1998)
49. Rumbaugh, J., Blaha, M.R., Lorensen, W., Eddy, F., Premerlani, W.: Object-Oriented Modelling and Design. Prentice-Hall Inc., Englewood Cliffs (1991)
50. Mellor, S.J.: Executable and translatable UML. Embed. Syst. Program. **16**(2), 25–30 (2003)
51. Starr, L., Mangogna, A., Mellor, S.: Models to Code With No Mysterious Gaps. Apress, Berkeley (2017)
52. Brooks, R.: The behavior language; user's guide. Technical report, AIM-1227, MIT, Artificial Intelligence Lab, Department of Electronics and Computer Science (1990)