Cross-level Co-simulation and Verification of an Automatic Transmission Control on Embedded Processor*

Cinzia Bernardeschi, Andrea Domenici, Maurizio Palmieri, Sergio Saponara
Dept. of Information Engineering, University of Pisa, Italy.

Tanguy Sassolas, Arief Wicaksana, Lilia Zaourar
CEA, LIST, Gif-sur-Yvette CEDEX, France.

Abstract

This work proposes a method for the development of cyber-physical systems starting from a high-level representation of the control algorithm, performing a formal analysis of the algorithm, and co-simulating the algorithm with the controlled system both at high level, abstracting from the target processor, and at low level, i.e., including the emulation of the target processor. The expected advantages are a smoother and more controllable development process and greater design dependability and accuracy with respect to basic model-driven development. As a case study, an automatic transmission control has been used to show the applicability of the proposed approach.

1 Introduction

Simulation is an essential activity in model-driven development (MDD), as it enables developers to implement virtual prototypes of their designs at all required levels of abstraction, until the design has been refined and validated to the point that it can be prototyped in hardware and code.

The existence of design models at different levels of abstraction makes it convenient to use different tools and formalisms for each model. Let us consider, for example, the control part of a cyber-physical system (CPS). This component must implement a high-level control algorithm that can be defined mathematically and modelled and simulated with the well-known tools together with a model of the controlled plant, usually built with the same tools, e.g., with Simulink. From now on, it is tacitly assumed that simulations include a plant model built with

*Work partially supported by the EPI (European Processor Initiative) project, EU-H2020 and by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Department of Excellence).
Published in: Cleophas L., Massink M. (eds) Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops. SEFM 2020. Lecture Notes in Computer Science, vol 12524. Springer, Cham.
The final authenticated version is available online at https://doi.org/10.1007/978-3-030-67220-1_20.

Simulink. Further refinements lead to a lower-level design including programming code and a hardware platform of the target processor(s), including system software/microcode. In this work, the terms *hardware* or *platform* refer to the physical and software infrastructure that executes the control algorithm. At this level of abstraction, it is possible to run the developed programming code on a simulated or real processor architecture. At this point, the hardware platform is the critical issue, as it affects significantly system performance and dependability. Accurate simulation of the platform makes it possible to evaluate hardware from different vendors, compare different architectural solutions, and choose optimal parameter configurations.

Processor simulation, however, requires formalisms and tools that are quite different from those used for high-level design. This mismatch is both conceptual and organisational, since the two levels require different fields of expertise, and is a potential source of issues ranging from project delays to design errors.

This paper introduces the concept of *cross-level simulation*, an approach to MDD aimed at bridging the gap between high- and low-level models, preserving coherence between them, and furthermore enabling formal verification of the control algorithm. A key point in this concept is that the implementation of the control algorithm is the same for both levels of simulation, and that the implementation is produced automatically from a formally verifiable model. Depending on application characteristics or project constraints, verification may be performed upfront on the formal model, or concurrently with simulation, the two activities cross-checking each other. This approach, summarised in Figure 1, is an extension to the common development flows based on Simulink-like tools, and relies on various tools for model construction, transformation, and simulation. More precisely, (i) a prototyping environment is used to create a high-level, automaton-based model and generate both a logic-based specification and C code; (ii) the specification is used to verify the control algorithm with a theorem-proving environment; (iii) high-level simulation executes the controller code together with a plant model, e.g., in Simulink; and (iv) low-level simulation executes the same code on simulated hardware, built in the SESAM/VPSim environment to account for timing behaviour.

In summary, this work extends the common MDD process by (i) starting with an abstract formal model; (ii) automatically generating an executable and a verifiable model; (iii) using formal verification side by side with simulation; (iv) relying on co-simulation to achieve modularity and flexibility of system models; and (v) using the same control code in high- and low-level simulation. The expected advantages are (i) a smoother and more controllable process and (ii) greater design dependability and accuracy with respect to basic MDD, relying on tools that enforce coherency among models at different levels of abstraction. In particular, the same code is used in both high-level and low-level co-simulations.

The rest of the paper is structured as follows: a selection of related works is presented in Section 2, the methods and tools for virtual prototyping/verification are introduced in Section 3, Section 4 illustrates the proposed approach, and Section 5 shows its application to a case study. Section 6 concludes the paper.

2 Related Work

Model-driven development relies mainly on simulation to analyse the system behaviour [25]. In cyber-physical systems, simulation often takes the form of co-simulation [11], which integrates simulation of heterogeneous sub-systems, modelled and simulated with the appropriate tools.

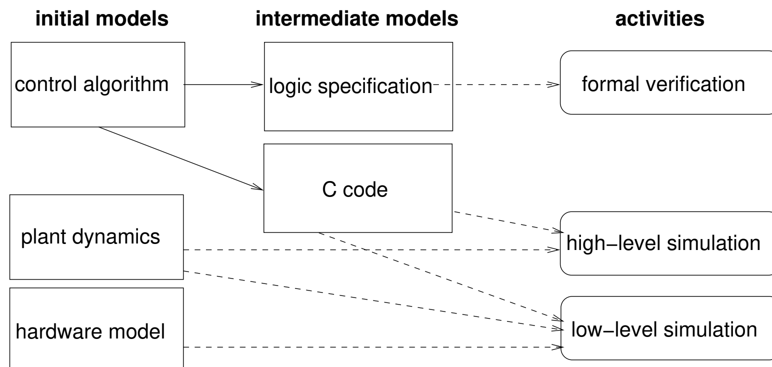

Figure 1: Overview of cross-modelling.

Due to the complexity of such systems, formal verification can help to assess compliance to safety requirements. Hybrid model checking, which relies on the formalism of Hybrid Automata [12], is used for the analysis of cyber-physical systems with a model-checking approach. One example of a model checking tool is HYCOMP [7] which also relies on Satisfiability Modulo Theories [8]. A complementary approach to model checking is theorem proving. Dynamic Logic [5] is used with the KeyMaeraX [22] theorem prover, which has been integrated with the SPIRAL environment [23] as reported by Franchetti et al. [10]. A framework that integrates simulation and theorem proving is PVSio-web [18], which uses higher-order logic as modelling language, as reported in [9, 20, 3].

None of these works integrates the processor emulation in the co-simulation.

In the field of electronic systems design, virtual prototypes are extensively used to simulate the behaviour of a system to be built. This allows hardware/software co-design to be better assessed and provides fast software development, reducing time to market. Many tools from academic work [14, 6] or electronic design automation vendors [26] address this need. The cited tools are all based on the IEEE SystemC standard [13] meant for model sharing. The standard defines a C++ library providing both a full discrete event simulation environment and design specific architectural constructs to enable hardware design at this level of abstraction. SystemC is further extended by the TLM 2.0 standard [1] which abstracts complex communication channels and protocols into simple function calls for faster simulation.

3 Background

This section provides details on the methods and tools used in this work.

3.1 PVS, Emucharts, and PVSio-web

Theorem proving consists in describing a system as a theory in some logic language, expressing its requirements as theorems, and verifying them with automatic or interactive theorem-proving software. The Prototype Verification System (PVS) [19] is an interactive theorem-proving environment based on a higher-order specification language whose variables can range over functions

and predicates. Theorems are proved by issuing commands to execute proof steps.

Users can create any theory by editing a text file, but the PVSio-web toolkit [18] can generate a PVS theory from an automaton created with the Emucharts [16] editor. An Emucharts graph is composed of modes linked by transitions. The graph is complemented by a set of variables whose values, together with the current mode, define the current state. Transitions are defined by a trigger (an event), a guard (an enabling condition on variables), and an action (updates on variables). The variables may range over discrete or continuous domains, they may represent time, state variables, values of time derivatives, and updated values in difference equations, so the Emucharts are a form of hybrid automata as defined in [12]. An example is shown in Section 5.1, Figure 5.

The PVSio-web toolkit can translate Emucharts into various specification and programming languages, including Misra C, a dependability-oriented version of C [17]. It is then possible to create a high-level automaton-based system model and from it generate a PVS theory to assert its properties, and use executable code automatically produced from the same model.

3.2 INTO-CPS

Co-simulation [11] is a technique to couple different simulation units together. A complex system can then be divided in many simpler submodels, and each submodel can be simulated using its specific language and tools. The Functional Mock-up Interface (FMI) [4] is an emerging standard for co-simulation, in which different simulation units, called Functional Mock-up Units (FMU), are orchestrated by a master algorithm in charge of synchronisation and data exchange among the FMUs. The master algorithm adopted in this work is the Co-simulation Orchestration Engine (COE), developed by the INTO-CPS Association [15]. The COE requires as input the logical connections between FMUs, the parameter values and the constraints on the co-simulation time step size. The INTO-CPS application also collects and graphically displays data produced by the co-simulation experiments.

3.3 SESAM/VPSim Environment

Within the SESAM [27] CPS design framework, the VPSim [6] tool targets the fast assembly and simulation of SoC architecture for both design space exploration and hardware/software co-design and validation. VPSim uses Python scripts to define architectures composed of SystemC modules from an extensive library of simulated commercial components including CPUs, interconnects, peripherals, and external controllers from various vendors (Xilinx, Renesas, Cadence, etc.). VPSim relies on the QEMU [2] processor emulator to provide a rich and fast CPU library model. As it targets fast simulation, VPSim is based on a loosely-timed model in compliance with TLM 2.0

As all SESAM tools, VPSim supports FMI co-simulation for tool interoperability throughout the design stages. It is a fully automated solution for exporting a hardware/software virtual prototype as an FMU. This enables the co-simulation of a whole CPS as detailed in [24]. Therefore, it can easily interface with other FMI-compliant simulators. An FMU encapsulating the virtual platform can be automatically generated based on a high-level description of the hardware/software platform.

4 Cross-level Modelling, Co-simulation and Verification

Cross-level simulation, introduced in Section 1, is discussed below.

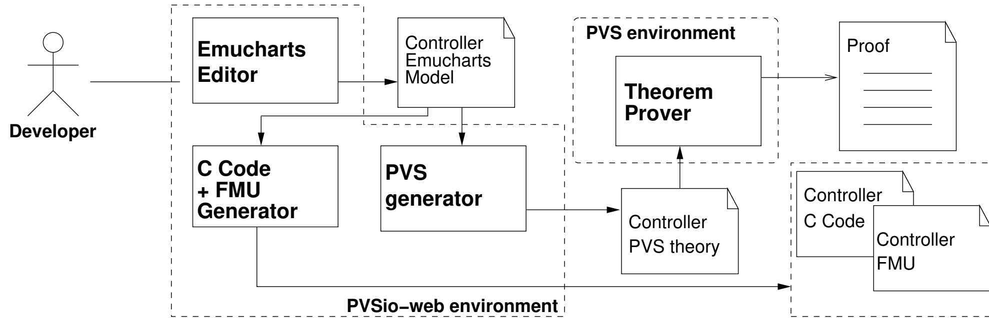

Figure 2: Model generation and formal verification.

4.1 Development Process

The initial steps of the development process are depicted in Figure 2: first, the developer uses the PVSio-web environment to generate the Emucharts model of the algorithm under analysis and then the developer uses the PVS and C code generators to generate the PVS theory and the Misra C code.

In the verification activity, the theory is used for two forms of verification: First, the well-formedness of the system model is assessed with the PVS type checker [21], then its compliance to requirements is checked with the theorem prover. The type checker may generate *type checking conditions* (TCC), i.e., statements that must be proved to ensure type correctness. Many TCCs are discharged automatically, others can be proved by the user with one or few commands, but unprovable TCC reveal incompleteness or inconsistency in a theory. The specification of requirements involves translating the desired property from natural or mathematical language to a PVS theorem.

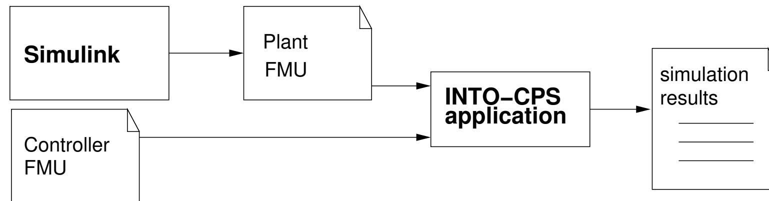

Figure 3: High-level co-simulation.

In the high-level simulation phase, the controller is co-simulated to validate it in connection with the plant model (Figure 3). The latter, built with a tool such as Simulink, is packaged in an FMU.

In the low-level simulation phase, the controller implementation is compiled and executed on a simulated platform including accurate models of real hardware, such as processors, memories, and controllers. In this phase, performance-related properties are assessed, such as execution time, latencies, or cache misses, possibly evaluating alternative choices of hardware components (Figure 4).

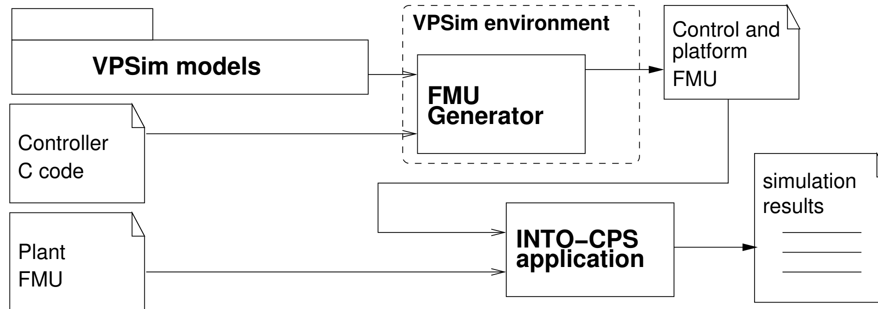

Figure 4: Low-level co-simulation.

The verification or simulation activity (or both) may fail, e.g., because the results of the co-simulation are not the expected ones or because the discharge of some TCC failed; In this case, the Emucharts model should be refined, using the results of the failed activities, then a new Misra C code and a new PVS theory should be generated, and the two activities iterated until both succeed.

When type checking and co-simulation succeed, it is then possible to specify and prove safety properties of the submodel under analysis.

4.2 Emulation of Processors with VPSim

Any architecture can be simulated by VPSim using the CPU models provided by the QEMU, an open-source hardware emulation solution, although it also allows the integration of model providers that have a SystemC/TLM interface, such as ARM Fast Models. By using QEMU for CPU modelling, we can obtain a very high simulation speed. Such a high performance is achieved mainly by abstracting the architectural aspects of CPUs while maintaining the functional accuracy in the execution. To provide the essential performance statistics to users, the QEMU models is enriched by VPSim in the SystemC simulation domain to model architectural aspects. To that end, all models that are backed by QEMU, including the VirtIOs, are encapsulated in SystemC modules and executed in the context of SystemC threads. Accordingly, QEMU models are controlled by the SystemC kernel like any native module and are transparently exposed to the user like any other VPSim component. For CPS validation purposes, the VPSim virtual platform can be packaged as an FMU by adding the definition of necessary FMI interfaces used in the Python front-end interface. VPSim supports models such as CAN controllers in its hardware library.

5 Automatic Transmission Control Case Study

The case study of this work is based on the Automatic Transmission Controller example from the Matlab documentation[1]. This example is a Simulink/Stateflow model composed of five high-level blocks: the Engine, the ShiftLogic, the Transmission, the Vehicle, and the ManeuvresGUI, which drives the simulation by producing the throttle and brake signals for a passing manoeuvre. The ShiftLogic block is a hybrid automaton, defined in Stateflow, that produces the upshift and downshift commands to the Transmission, according to a *shift schedule* that takes into account the current gear, the vehicle speed, and the throttle position.

The ShiftLogic controller is in a steady state if the vehicle is driving at an intermediate speed between the upshift and downshift thresholds for the current gear and throttle. If the throttle or speed cross a threshold, the controller moves to either of two waiting states (for upshift or downshift). If the speed remains beyond the threshold for a given time, the corresponding shift command is issued and the controller moves to the steady state of the new gear.

5.1 High-Level Virtual Prototyping

The ShiftLogic block has been re-designed as outlined in Section 4.1 and packed in an FMU. Another FMU, generated by Simulink, contained the other four blocks. The two FMUs were then co-simulated in INTO-CPS.

5.1.1 Emucharts Model for ATC

The ATC behaviour is specified by the shift schedule. Using the data from the cited example, the shift schedule is defined by the functions represented in Tables 1 and 2. Each row labelled as $n - m$ shows the threshold speed value (in miles per hour) for a shift from gear n to gear m in consecutive intervals of throttle position $t_{\%}$(in percent).

Table 1: Shift schedule, speed thresholds for upshifts.

shift	$t_{\%} \leq 25$	$25 < t_{\%} \leq 35$	$35 < t_{\%} \leq 50$	$50 < t_{\%} \leq 90$	$90 < t_{\%} \leq 100$
1-2	10	$0.5t_{\%} - 2.5$	$0.53333t_{\%} - 3.6667$	$0.425t_{\%} + 1.75$	40
2-3	30	30	$0.73333t_{\%} + 4.3333$	$0.725t_{\%} + 4.75$	70
3-4	50	50	$0.66667t_{\%} + 26.6667$	$t_{\%} + 10$	100

Table 2: Shift schedule, speed thresholds for downshifts.

shift	$t_{\%} \leq 5$	$5 < t_{\%} \leq 40$	$40 < t_{\%} \leq 50$	$50 < t_{\%} \leq 90$	$90 < t_{\%} \leq 100$
4-3	35	$0.1429t_{\%} + 34.28571429$	$t_{\%}$	$0.75t_{\%} + 12.5$	80
3-2	20	$0.1429t_{\%} + 19.2857$	$0.5t_{\%} + 5$	$0.5t_{\%} + 5$	50
2-1	5	5	5	$0.625t_{\%} - 26.25$	30

The shift schedule has been modelled as an Emucharts diagram. Figure 5 shows the diagram fragment relative to the lower two gears, while the complete diagram (Figure 13), drawn in

[1] `https://www.mathworks.com/help/simulink/slref/modeling-an-automatic-transmission-controller.html.`

a more compact form, is in the Appendix with the transition definitions. Each transition is identified with a label followed by a guard in square brackets and possibly an action in braces.

In the diagram, *stdy* modes represent steady conditions of the ATC, while *up* and *down* modes represent the waiting phases before the ATC is going to issue an upshift or downshift command, respectively, if the speed remains beyond the corresponding threshold for a long enough time.

The transitions depend on variables: the discrete variables *clock* and *gear* and the continuous variables *tht* (throttle), *up_th* (upshift threshold), *dw_th* (downshift threshold), and *speed*. Variable *clock* is a timer that can be incremented by one step or reset, and *gear* is the controller output. The flow conditions [12] for the input variables *tht* and *speed* are defined externally by the Vehicle and ManeuvresGUI models, while the flow conditions for *up_th* and *dw_th* are given by the shift schedule in Tables 1 and 2, respectively.

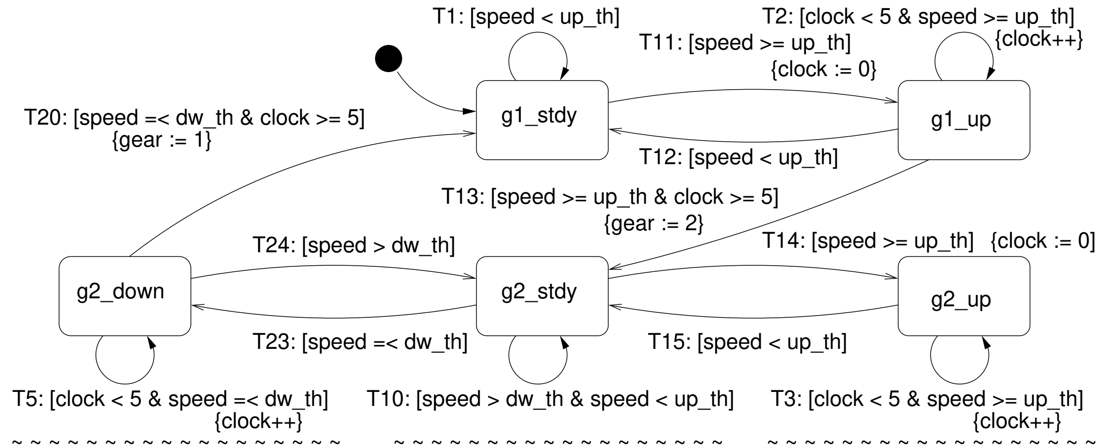

Figure 5: Fragment of the Emucharts diagram for the shift logic automaton.

5.1.2 Discharging the TCCs for the ATC

The Emucharts model of the ATC was derived from the Stateflow machine in the cited Matlab example. Typechecking the PVS theory generated from the first Emucharts version produced unprovable TCCs. One is a *coverage* TCC, stating that the disjunction of the guards of all transitions is identically true, i.e., at least one transition is enabled. The complementary *mutual exclusion* TCC requires that at most one transition is enabled. The problem was that some transitions were implicit in the Stateflow model, and was fixed by explicitly adding the needed transitions to the Emucharts model. These transitions are labelled in boldface as T1, T8, T9, and T10 in Figure 13. This is an example of how a sophisticated type system, together with automatic checking, helps spotting hidden assumptions that are often sources of errors.

5.1.3 Co-simulation for ATC

Figure 6 shows the co-simulation architecture for the high-level simulation. The co-simulations are executed using a fixed time step of 0.1 seconds and last 80 seconds of simulated time. Figure 7

shows the results of a co-simulation run. These results are consistent with those obtained with the original Simulink model, i.e., the shapes of the throttle and speed curves between the first and third upshifts match those of the plot in the MATLAB documentation, except for the initial speed value and the initial transient as discussed in Section 5.2 below.

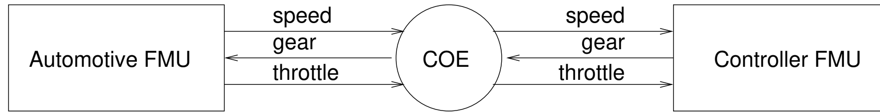

Figure 6: High-level Co-simulation architecture.

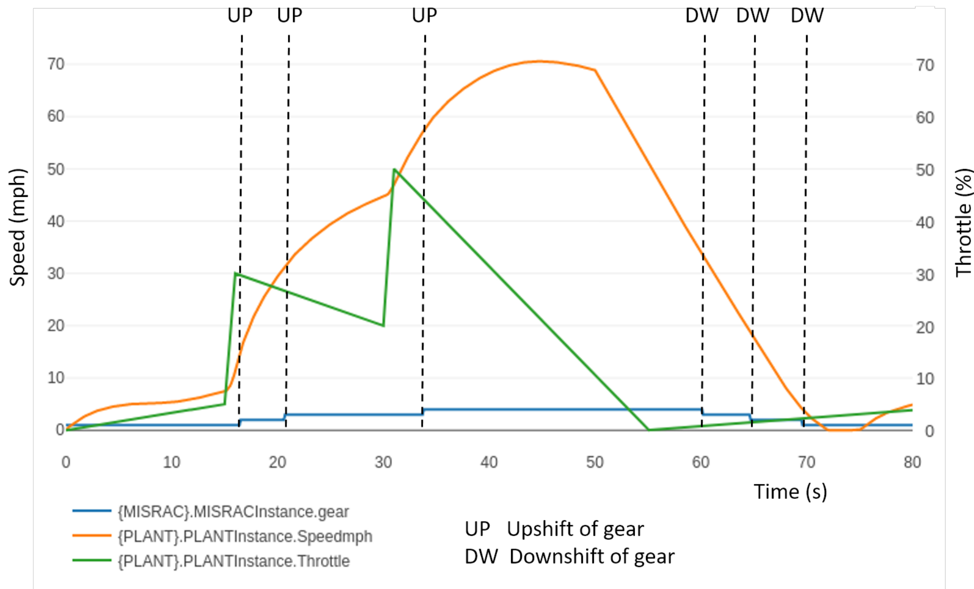

Figure 7: High-level control co-simulations.

5.1.4 Verification Process for ATC

As an example, one of the safety properties of the ATC algorithm is "*it is never possible to move in one step from a state where gear equals g to a state where gear equals $g \pm 2$* ". This natural language statement can be translated in PVS with the *main_th* theorem, where `abs` is the absolute value function:

```
gear_T: TYPE = {x: posnat | x<=4}

main_th: THEOREM
FORALL (N:nat, g:gear_T):
  kth_step(N)`gear = g => abs(kth_step(N+1)`gear - g) < 2
```

where `kth_step(N)` is a data structure containing the values of all variables at step N, and `‘gear` selects the value of *gear*. The proof has been done by induction on the number of steps and by analysing separately the different values of gear. The proof relies on a few lemmas. For example, it must be proved that in any gear, the computed threshold speed for an upshift (`compute_UP_TH(s)‘up_th`)is greater than the one for a downshift (`compute_DW_TH(s)‘dw_th`):

```
UPgtDW: LEMMA
FORALL (s:State):
  compute_UP_TH(s)‘up_th > compute_DW_TH(s)‘dw_th
```

Another lemma excludes direct transitions between two *steady* states:

```
gear1: LEMMA
FORALL(N:nat):
  kth_step(N)‘mode = g1_steady => kth_step(N+1)‘mode /= g2_steady
```

Verifying this and similar theorems guarantees the functional correctness of the control algorithm.

5.2 Co-simulation with VPSim

Figure 8 shows the architecture for the low-level simulation with VPSim. This architecture is very similar to the high-level scenario, the only difference is that the controller FMU has been replaced with the FMU generated from VPSim. The FMU generated by VPSim emulates a cluster of ARMv8 64-bit architectures. The cluster contains 1-core processors with private L1 and L2 caches, which is connected to the on-chip interconnect and peripheral devices. More cores and clusters can be added in future works for more complex applications. The architecture executes a Linux OS which supports the ShiftLogic application.

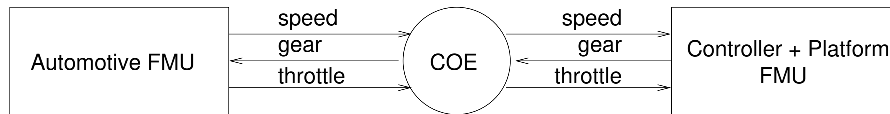

Figure 8: Low-level co-simulation architecture.

Hence, the VPSim FMU requires an initial time to boot the operating system before executing the application of the ShiftLogic algorithm, while the MisraC FMU executes the algorithm since the beginning of the co-simulation. For sake of comparison between the two architectures, the value of throttle is always kept close to zero for the first seconds of the co-simulations.

VPSim enables the timing behaviour of a system to be captured. Hence the duration of the applicative code has a direct impact on the evolution of the FMU outputs that it may change. Simulations have been performed with different execution times of the ATC, obtained by adding delay loops to the original code. When the ATC executes faster than the FMI simulation step demanded by the FMI master (Figure 9), the behaviour is similar to what is achieved with high-level simulation. (Functions fmi2set and fmi2get are write and read operations, while function fmi2DoStep triggers the execution of one simulation step.) In that case, Figure 11 shows the behaviour of the co-simulation with the VPSim FMU: The resulting behaviour is

consistent with the one previously obtained with the high-level co-simulation. On the contrary (Figure 10), when the ATC executes slower than the interval between invocations of fmi2DoStep, output value changes are differed to after future simulation steps.

Having both simulation levels together allows inadequate execution speed of the code under scrutiny to be better underlined. Indeed, when the application execution speed is appropriate, the behaviour is consistent throughout the validation levels as expected. However, it is worth noting that, even if one may set the co-simulation step to an arbitrarily small value, this will create new discrepancies due to processing delay in the low-level simulation compared to the high-level one. If achievable, users should be advised to keep a simulation step duration in line with the expected control decision deadline.

To further improve the alignment of models without this constraint, it could also be beneficial in future work to model the target execution time during high-level simulation by delaying control decisions accordingly. This would render the high-level model behaviour invariant to simulation step choices. Similarly the low-level simulation results shall not be made available too early, even if the control code execute too fast. This would likely be taken into account in real-time code where control decision would never be output before the target deadline.

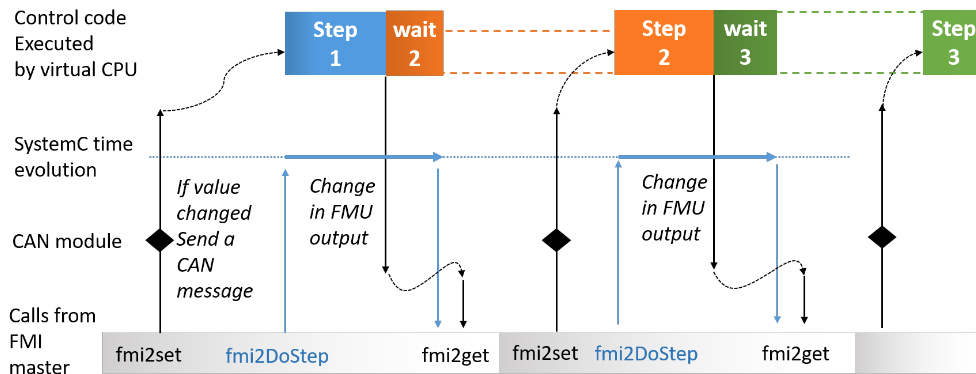

Figure 9: Co-simulation of VPSim-generated FMU with fast ATC execution.

5.3 Results and Discussion

It is possible to compare Figure 11 with Figure 7: excluding the first seconds, in which gear equals zero because the processor is still booting, the variable of gear has the same behaviour for both cases. This result implies that the time required to execute the ATC on the emulated processor is lower than the step-size chosen for the co-simulation (0.1 seconds).

In order to highlight the advantages of considering low-level co-simulation, the algorithm of the ATC has been artificially extended with a redundant code that increases its computation time so that the execution time of the ATC becomes greater than the co-simulation step-size. Figure 12, shows the comparison between the low- and high-level behaviour of the gear variable with the extended code. It is possible to notice a small time delay in the behaviour of the variable which is due to the different time management: The high-level co-simulation always

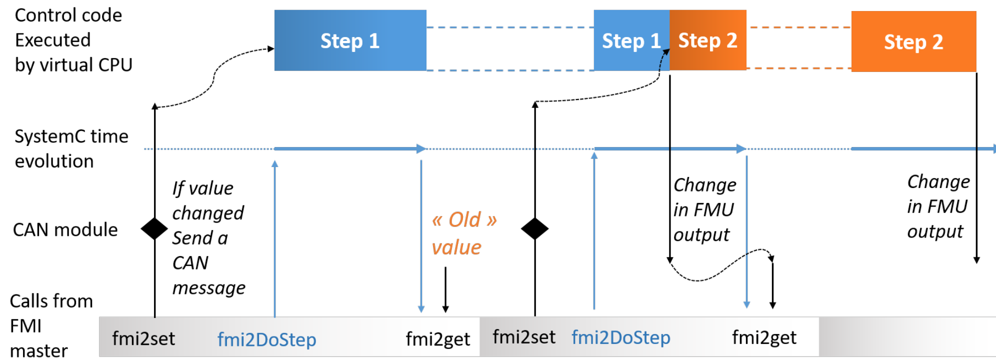

Figure 10: Co-simulation of VPSim-generated FMU with slow ATC execution.

executes the whole ATC algorithm within a co-simulation step while the low-level now requires more co-simulation steps. Please notice that the delay in the first two gear transitions of the low-level co-simulation has affected the value of speed, increasing it, in such a way that the next gear transition occurs earlier with respect to the high-level simulation and, apparently, with no delay.

The co-simulation results show that the underlying hardware performance (e.g., computation speed), must be taken into account to ensure that the plant can be controlled within the step-size. High-level simulation hides system performance issues that the virtual prototype can highlight.

In all the co-simulation runs, both high-level and low-level, the results obtained with PVS hold, as it is never the case that two gear transitions are executed in two adjacent steps, thus validating the results obtained in Section 5.1.4. Of course, time-related properties will be affected by the different time management and so require an additional step in the verification process, i.e., the specification of the processor in PVS, but this will be subject of future work.

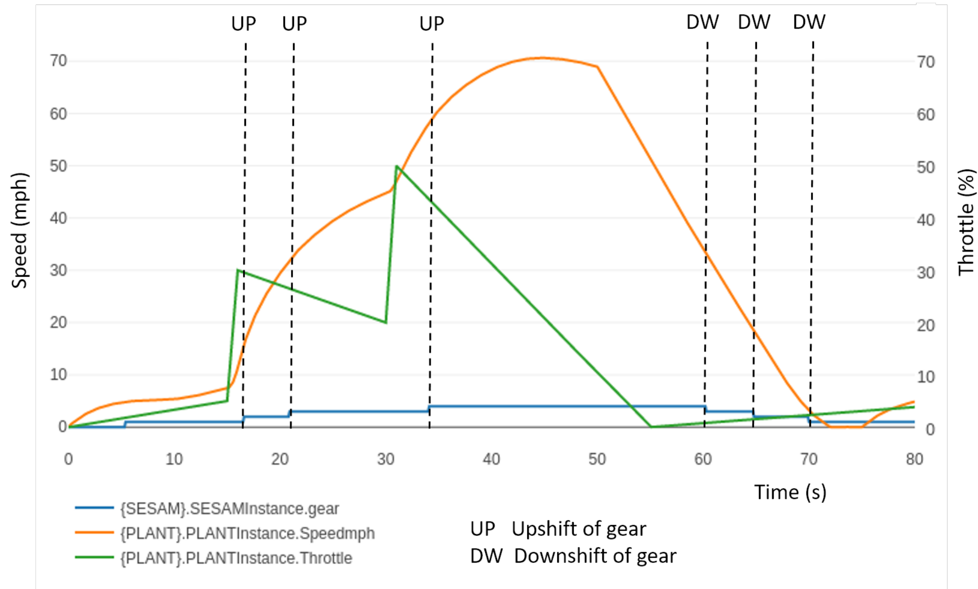

Figure 11: Low-level control co-simulations.

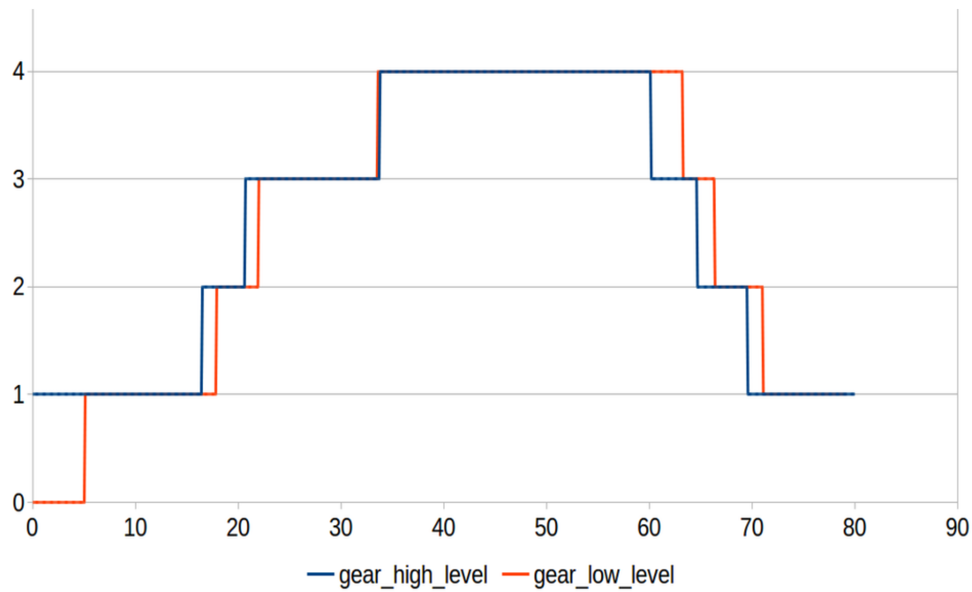

Figure 12: Gear shifts in high- and low-level simulations.

6 Conclusions

This work proposes an approach to the analysis of control algorithms deployed on automotive systems. The approach uses models with different levels of abstraction: a more abstract, high-level, model composed of the executable code of the control algorithm, and a more accurate, low-level, model that also includes the emulation of the hardware executing the code. The high-level analysis provides information on the functional correctness of the model by exploiting both formal reasoning tools such as the PVS theorem prover and simulation tools such as INTO-CPS, while the low-level analysis provides information on the execution performances related to the chosen hardware for the low-level model by exploiting VPSim. The proposed approach also uses FMI co-simulation to include the physical components of the car in the analysis.

Both levels of analysis are needed in the development of CPSs, especially safety- or mission-critical ones.

It would not make sense to jump to software/hardware integration before validating and possibly verifying the controller design, as it would not make sense to choose a hardware platform without assessing its adequacy with respect to timing constraints and evaluating its performance. This work strives to provide a framework to maintain as much coherence as possible to the three key aspects of development, i.e., formal verification, high-level, and low-level modelling.

A case study of an automatic transmission controller algorithm is used as a proof of concept for the methods and tools involved in a safety-critical area like automotive applications. The results highlight that it is possible to assess the performance of the chosen hardware: If the emulated processor is fast enough to accommodate the execution of the algorithm within a co-simulation time step, then the behaviour of the low-level co-simulation is the same as the one of high-level co-simulation, otherwise the co-simulation shows a different behaviour. The proposed methodology was applied to an ARMv8 64-bit single-core processor with private L1 and L2 caches, used in many application automotive processors (e.g. NXP S32V). More cores will be added in future work to consider high-end processors (like the Rhea1, expected as output of the European Processor Initiative project) and for more complex safety-critical applications like autonomous driving and model predictive control of vehicle dynamics.

Acknowledgements

The authors would like to thank the reviewers for their useful comments and suggestions.

References

[1] Accelera: TLM-2.0 Language Reference Manual. `https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf` (2009)

[2] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. p. 41. ATEC '05, USENIX Association, USA (2005)

[3] Bernardeschi, C., Domenici, A., Masci, P.: A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems. IEEE Transactions on Software Engineering **44**(6), 512–533 (2018)

[4] Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauß, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Proceedings of the 9th International MODELICA Conference. pp. 173–184. No. 76 in Linköping Electronic Conference Proceedings (2012)

[5] Bohrer, B., Rahli, V., Vukotic, I., Völp, M., Platzer, A.: Formally verified differential dynamic logic. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs. pp. 208–221. CPP 2017, ACM (2017). https://doi.org/10.1145/3018610.3018616

[6] Charif, A., Busnot, G., Mameesh, R.H., Sassolas, T., Ventroux, N.: Fast virtual prototyping for embedded computing systems design and exploration. In: Chillet, D. (ed.) Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO 2019, Valencia, Spain, January 21-23, 2019. pp. 3:1–3:8. ACM (2019). https://doi.org/10.1145/3300189.3300192

[7] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 52–67. Springer (2015)

[8] De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)

[9] Domenici, A., Fagiolini, A., Palmieri, M.: Integrated simulation and formal verification of a simple autonomous vehicle. In: International Conference on Software Engineering and Formal Methods. SEFM 2017. Lecture Notes in Computer Science, vol 10729. pp. 300–314. Springer (2017)

[10] Franchetti, F., Low, T.M., Mitsch, S., Mendoza, J.P., Gui, L., Phaosawasdi, A., Padua, D., Kar, S., Moura, J.M.F., Franusich, M., Johnson, J., Platzer, A., Veloso, M.M.: High-assurance spiral: End-to-end guarantees for robot and car control. IEEE Control Systems **37**(2), 82–103 (April 2017). https://doi.org/10.1109/MCS.2016.2643244

[11] Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a survey. ACM Computing Surveys (CSUR) **51**(3), 1–33 (2018)

[12] Henzinger, T.A.: The theory of hybrid automata. In: M.K., I., R.P, K. (eds.) Verification of Digital and Hybrid Systems, NATO ASI Series (Series F: Computer and Systems Sciences), vol. 170, pp. 265–292. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/978-3-642-59615-5_13

[13] IEEE: IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) pp. 1–638 (2012)

[14] Imperas Ltd.: Open Virtual Platforms. `http://www.ovpworld.org/` (2020)

[15] Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In: 2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data). pp. 1–6 (April 2016). https://doi.org/10.1109/CPSData.2016.7496424

[16] Masci, P., Zhang, Y., Jones, P., Oladimeji, P., D'Urso, E., Bernardeschi, C., Curzon, P., Thimbleby, H.: Combining PVSio with Stateflow. In: Proceedings of the 6th NASA Formal Methods Symposium (NFM2014). pp. 209–214. Springer-Verlag (2014)

[17] Mauro, G., Thimbleby, H., Domenici, A., Bernardeschi, C.: Extending a user interface prototyping tool with automatic MISRA C code generation. In: Dubois, C., Masci, P., Méry, D. (eds.) Third Workshop on Formal Integrated Development Environments. Electronic Proceedings in Theoretical Computer Science, vol. 240, pp. 53–66. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.240.4

[18] Oladimeji, P., Masci, P., Curzon, P., Thimbleby, H.: PVSio-web: a tool for rapid prototyping device user interfaces in PVS. In: FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems, London, UK, June 24, 2013 (2013)

[19] Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) Automated Deduction — CADE-11, Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer Berlin Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217

[20] Palmieri, M., Bernardeschi, C., Masci, P.: A framework for FMI-based co-simulation of human–machine interfaces. Software and Systems Modeling **19**(3), 601–6023 (2020)

[21] Palmieri, M., Macedo, H.D.: Automatic Generation of Functional Mock-up Units from Formal Specifications. In: 3rd Workshop on Formal Co-Simulation of Cyber-Physical Systems — A satellite event of SEFM 2019 (2019), in press

[22] Platzer, A., Quesel, J.D.: KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, Lecture Notes in Computer Science, vol. 5195, pp. 171–178. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_15

[23] Püschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE **93**(2), 232–275 (Feb 2005). https://doi.org/10.1109/JPROC.2004.840306

[24] Saidi, S.E., Charif, A., Sassolas, T., Le Guay, P.G., Souza, H.V., Ventroux, N.: Fast Virtual Prototyping of Cyber-Physical Systems using SystemC and FMI: ADAS Use Case. In: Proceedings of the 30th International Workshop on Rapid System Prototyping (RSP'19). pp. 43–49 (2019)

[25] Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software **20**(5), 19–25 (2003). https://doi.org/10.1109/MS.2003.1231146

[26] Synopsys: Virtualizer. `https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html` (2020)

[27] Ventroux, N., Guerre, A., Sassolas, T., Moutaoukil, L., Blanc, G., Bechara, C., David, R.: SESAM: An MPSoC Simulation Environment for Dynamic Application Processing. In: 2010 10th IEEE International Conference on Computer and Information Technology. pp. 1880–1886 (2010). https://doi.org/10.1109/CIT.2010.322

Appendix

Table 3 below shows the transition definitions of the simplified Emucharts diagram shown in Figure 13. Functions *up_th* and *dw_th* implement the shift schedule specifications from Tables 1 and 2, respectively.

Table 3: Shift logic transitions.

Transition	source	target	guard	action
T1	g1_stdy		speed < up_th(1, $t_{\%}$)	
T2	g1_up		clock < 5 AND speed ≥ up_th(1, $t_{\%}$)	clock++
T3	g2_up		clock < 5 AND speed ≥ up_th(2, $t_{\%}$)	clock++
T4	g3_up		clock < 5 AND speed ≥ up_th(3, $t_{\%}$)	clock++
T5	g2_down		clock < 5 AND speed ≤ dw_th(2, $t_{\%}$)	clock++
T6	g3_down		clock < 5 AND speed ≤ dw_th(3, $t_{\%}$)	clock++
T7	g4_down		clock < 5 AND speed ≤ dw_th(4, $t_{\%}$)	clock++
T8	g4_stdy		speed > dw_th(4, $t_{\%}$)	
T9	g3_stdy		speed > dw_th AND speed < up_th(3, $t_{\%}$)	
T10	g2_stdy		speed > dw_th AND speed < up_th(2, $t_{\%}$)	
T11	g1_stdy	g1_up	speed ≥ up_th(1, $t_{\%}$)	clock := 0
T12	g1_up	g1_stdy	speed < up_th(1, $t_{\%}$)	
T13	g1_up	g2_stdy	speed ≥ up_th(1, $t_{\%}$) AND clock ≥ 5	gear := 2
T14	g2_stdy	g2_up	speed ≥ up_th(2, $t_{\%}$)	clock := 0
T15	g2_up	g2_stdy	speed < up_th(2, $t_{\%}$)	
T16	g2_up	g3_stdy	speed ≥ up_th(2, $t_{\%}$) AND clock ≥ 5	gear := 3
T17	g3_stdy	g3_up	speed ≥ up_th(3, $t_{\%}$)	clock := 0
T18	g3_up	g3_stdy	speed < up_th(3, $t_{\%}$)	
T19	g3_up	g4_stdy	speed ≥ up_th(3, $t_{\%}$) AND clock ≥ 5	gear := 4
T20	g2_down	g1_stdy	speed ≤ dw_th(2, $t_{\%}$) AND clock ≥ 5	gear := 1
T21	g3_down	g2_stdy	speed ≤ dw_th(3, $t_{\%}$) AND clock ≥ 5	gear := 2
T22	g4_down	g3_stdy	speed ≤ dw_th(4, $t_{\%}$) AND clock ≥ 5	gear := 3
T23	g2_stdy	g2_down	speed ≤ dw_th(2, $t_{\%}$)	clock := 0
T24	g2_down	g2_stdy	speed > dw_th(2, $t_{\%}$)	
T25	g3_stdy	g3_down	speed ≤ dw_th(3, $t_{\%}$)	clock := 0
T26	g3_down	g3_stdy	speed > dw_th(3, $t_{\%}$)	
T27	g4_stdy	g4_down	speed ≤ dw_th(4, $t_{\%}$)	clock := 0
T28	g4_down	g4_stdy	speed > dw_th(4, $t_{\%}$)	

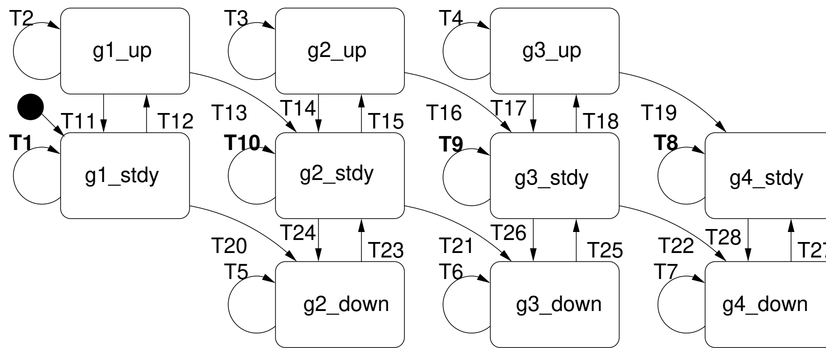

Figure 13: Simplified Emucharts diagram for the shift logic automaton.