Model-Integrated Development of Cyber-Physical Systems

Gabor Karsai and Janos Sztipanovits

Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37205, USA
gabor.karsai@vanderbilt.edu, janos.sztipanovits@vanderbilt.edu

Abstract. Cyber-physical systems represent a new class of systems that integrate physics with computation. Their correct design is frequently of great importance as they are applied in safety- or business-critical contexts. This paper introduces a model-integrated development approach that addresses the development needs of such systems through the pervasive use of models. A complete model-based view is proposed that covers all aspects of the hardware and software components, as well as their interactions. Early experiments and work in progress are also reported.

Keywords: model-driven development, model-integrated computing, cyber-physical systems, executable models, system integration.

1 Introduction

Cyber-physical systems (CPS) are systems that combine a physical system with an embedded information processing system such that the resulting system has novel capabilities that could not be achieved by either the physical or the computational entity alone[1]. To give examples for a cyber-physical system consider an unmanned aerial vehicle with active (fixed) wings. In such a UAV, an embedded controller monitors the airflow over the wing surface and modulates it through electromechanical actuators to ensure laminar flow such that the vehicle is capable of extreme maneuvers. Another example is a structural beam whose deflection is active monitored and modified through a piezoelectric actuator, resulting in a lighter, thinner structure whose resulting physical properties ('strength') is greater than that of the original beam without the embedded controller.

It is easy to see that the design of such systems cannot be accomplished following the classical strictly disciplinary approach – the design of the physical and computational aspects is an integrated activity. Design decisions made in one aspect (e.g. selecting the scheduling technique used in the embedded software) interacts with the physical component and has profound consequences on the dynamic properties of the entire system. We argue that the design of such systems could only be accomplished by taking an integrated view and co-designing the physical with the computational.

[1] This definition of cyber-physical systems is due to Janos Sztipanovits.

Model-driven development of embedded software systems [1] has gained acceptance during the past decade, and it is the de-facto approach used in systems industries (automotive and aerospace), and is well-supported by industrial-strength commercial tools, like Simulink/Stateflow [2] and Matrix-X [3]. Benefits of the model-driven approach are obvious, and the industry has built up a significant amount of knowledge and well-tested solutions.

The question naturally arises: are cyber-physical systems fundamentally different such that they need a different development approach, or the current approach is sufficient and no new research is necessary? In this paper we propose an answer to this question that is based on experience with the existing tools and practices, and the proposed answer is: we need new techniques, and a new view.

Our argument is as follows. The engineering of non-software artifacts is often based on models that typically have a computational manifestation (i.e. an executable form in some computational sense). The engineering of software using model-based techniques is an active area of research and it started to find its way into the overall software engineering practice. However, very little is being done with regard to an *integrated* approach, where both the 'physical artifacts' and the software would be engineered based on a set of coupled models. The closest practice comes to this ideal is the approach followed in Simulink/Stateflow and Matrix-X: 'plant models' and executable controller models are (co-)simulated in a shared simulation environment, under the control of a simulation engine. The approach increases the productivity of domain (in this case, control) engineers, because they don't have to deal with the accidental complexities of software engineering, and the tools (and the hardware platforms) are powerful enough such that code automatically generated could be immediately used in the application.

However, we believe this is not sufficient for the next generation of CPS-s. First, the approach does not consider the properties of the execution platforms (i.e. the properties and performance of processing units, the operating systems, the middleware, the QoS machinery, etc.). Although new tools like TrueTime [4] make progress in this direction, it is unclear how arbitrary platforms should be modeled and analyzed. Second, the Model of Computation (MoC) [5] used by the tools is rather limited: it is almost always some variant of the approach followed in the synchronous languages. Other approaches, like CDMA-style communication, or publish-subscribe approaches, or even priority based scheduling with potential priority inversion are rarely considered. Third, it is unclear how algorithms that apply search, do not have a guaranteed termination time, or are of the anytime variety could be considered in the systems. We simply don't have good models of the dynamics for such algorithms, and thus the analysis of the end-to-end system is very difficult to do.

In an engineering process for CPS-s we need to address the above and other issues related to para-functional properties like security, reliability, fault tolerance, etc. Modern development techniques, like extreme programming, test-based development, and continuous integration also need to be considered, as these represent the best practices in the industry today – and their track record is well-documented. In the paper we propose a fully model-integrated approach that allows the combined use of such techniques.

2 Models and Cyber-Physical Systems

If one needs to consider a full spectrum modeling for CPS-s, a scheme shown on the figure below could be used as a starting point. On the left we show the 'model elements', on the right their 'real world' counterparts are shown that exists in the implementation.

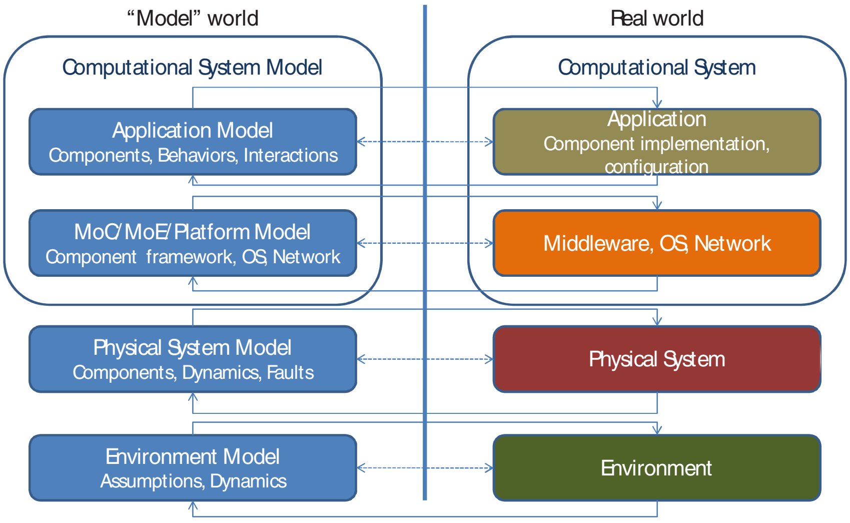

Fig. 1. Models and the real world in Cyber-Physical Systems

Note that the 'real world' includes the 'application software' as implemented on some computation platform (that includes the component middleware, operating system, compute engines, network), that is layered upon and interacts with a physical system (the 'hardware' of the CPS), which then interacts with the physical environment. What is envisioned here is a complete model-integrated approach across all levels of the hierarchy. In short, one needs models for the environment (that is outside of the CPS), for the physical system (that is part of the CPS but is not computational), for the computational platform (that includes all hardware and software elements that are reusable across different CPS-s), and for the application (i.e. the software that implements all the functions of the CPS).

As we assume model-driven development, orthogonal to the models we find the tools that support modeling (i.e. model creation and editing), model analysis (i.e. verification, validation, etc.) and synthesis (i.e. implementation generation). There are various tools in existence today that address some of the problems here (e.g. dedicated modeling environments, code generators, code verification tools), but they are often difficult to use together, in an integrated manner. For CPS-s better tool integration is needed that is based on the semantic integration of the models used across the layers. After all, we need to model a physical system's dynamics, and study how it interacts with the dynamics of the implementation of a particular MoC on a specific hardware and software platform.

2.1 Challenges in CPS

Developers of CPSs face several challenges, many of which are well-known from the work on embedded systems. Here we would like to highlight a few challenging aspects that arguably have received less attention in the past.

First, CPSs imply a major *integration problem*. Both the system that is being built, as well as the process used to build it are highly heterogeneous and unforeseen interactions often arise. There are two major perspectives on integration: model integration and system integration.

Model integration problems arise when we want to simulate, for instance, the entire CPS, including all implementation layers. One needs simulators that (1) either follow the same, shared execution semantics (which seems to be the approach used in Simulink where all simulations are executed in continuous time), or (2) they are federated and can run under the control of a coordinating authority (which is the approach followed in the HLA model). The situation is further complicated by the fact that models are often on different levels of abstraction, and one needs different models of the same system for different work (e.g. transaction level models vs. register transfer level models for hardware). Another problem in model integration is the decoupling between models used in design, the model verification tools, and the final executable system. When subjecting design models to analysis (e.g. model checking), we need to carry over the results to the final system, i.e. the system as implemented by executable code running on a real, physical software and hardware platform. Often design languages (e.g. UML activity diagrams) and analysis languages (e.g. SMV model checker's language) are different, and we need to use translators. However these translators must be correct for the analysis results be valid. We need this triangle of design models / analysis models / executable models 'verified' such that analysis results are provably true for the executable system.

System integration is perhaps the most challenging aspect of CPS engineering, but arguably, this is the area where models are extremely beneficial. The physical and the computational parts of the CPS have to be designed together, and should be modeled and analyzed together. Note that this is notably different from hardware-software codesign, where functions are designed in a common framework, and where the partitioning is decided late in the process. In CPSs the 'hardware' is not computational and thus it is fixed early on, such that the computational part has to be designed accordingly. Naturally, codesign techniques are highly applicable to the design of the 'cyber' part. As discussed in more detail below, for the system integration we envision an incremental, simulation-based development and integration approach. The concept is that initially the entire system is executed in a simulated environment, and later simulated parts are incrementally replaced by real implementations and real hardware.

The second major challenge is the *support for certification* of CPSs that are used in critical environments (e.g. vehicles, medical systems, etc.). Note that we need end-to-end certification, according to current practices followed in the aerospace industry ('we certify the airplane, not the software'). However, this approach becomes very hard to sustain, and a modular approach is more desirable. One can consider three methods for providing arguments for certification: simulation-based, verification-based, and hardware-based testing. In the first, a high-fidelity simulation of the physical system and environment is created, that is independently validated. Next, the computational ‘stack’ is subjected to exhaustive testing in the ‘context’ provided by the simulation. Here, the simulation must be ‘interface-compatible’ with the real physical system, i.e. the interfaces that the computational system interacts with must be the same as in the real implementation. For the second, verification-based approach we build assurances via checking the models of computational system and/or the code itself. Obviously, this necessitates robust and verified translations on the models, as discussed above. For the third, the computational system is tested in the context of a hardware test setup, and arguments for certification are collected through exhaustive testing again. In summary, when certification is needed for CPSs the development process and tools should incorporate various elements to produce the arguments to be used in the certification process. These steps and tools need to become part of the toolchain.

The third group of challenges includes *mode changes* and *fault management*. CPSs often have a large number of operational modes, where their dynamics and behavior are radically different. For instance an aircraft flies very differently when landing than in cruising mode. The CPS should be prepared to handle and manage these different modes and changes between modes. Often we cannot simply reinitialize software components upon mode changes as this would lead to intolerable transients.

The ultimate test for modal systems is the management of faults. Faults can happen in the physical system, in the platform, as well as in the application software, and the application needs to be prepared for handling them. Obviously, fault needs to be detected, their primary cause isolated, and then a corrective action needs to be taken. This process is traditionally known under the name ‘Fault Detection, Isolation, and Recovery’, FDIR. A good CPS design is not only a functional design, but it also anticipates faults and has provisions for managing them, through the steps described above. Whenever the CPS is in a critical application, such fault management is unavoidable, and it has to be ‘designed in’ to the system from the beginning. Fault management may include simply redundancy management (which involves complex mode changes), but could also be as complex that a full FDIR approach is needed. In complex physical systems, continuous on-line testing and verification is often used for FDIR. For CPSs these techniques need to be applied to the software ingredients as well.

2.2 An Approach to Development and Integration

As it was emphasized in the previous section, integration is of utmost importance in CPSs: in fact the definition of this category refers to it. Hence, the integration of the physical and the computational should be *the* key design activity; in fact, it should possibly drive the entire design process.

Here we propose a continuous integration process that establishes the interfaces between the physical and computational from the beginning, and the integration of the system is performed continuously. This approach is not new for software developers: the concept of nightly build and continuous integration is a well-known practice today. Here, we extend this idea in the context of CPSs.

The approach requires some assumptions about the CPS design as follows. We assume that the system is constructed in layers (as shown on the figure below), and for each layer we have models that are executable (perhaps with the help of simulation engine).

The general layers of a CPS include the environment, the physical systems, the coputational platform, and the application. The computational platform interacts with the physical system via sensors and actuators, and the application interacts with the platform via APIs. Note that this is the same organization discussed earlier.

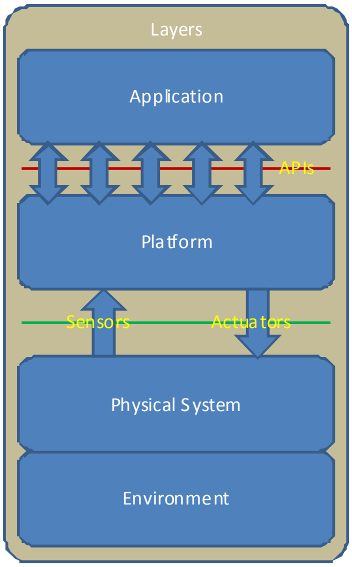

Fig. 2. Layers of CPS design

In the proposed continuous integration approach we assume that models are available for each layer, and these models could be used in an executable form. Initially, these could be low-fidelity, approximate models, that are incrementally replaced by high-fidelity models, and finally with implementations.

The key observation about this approach is that interface and architecture design are primary activities. In fact, architecture modeling and analysis is done early in the design process. Furthermore, interfaces are designed early. As the basic tenet of systems engineering, interfaces are designed first, well-before the system is implemented. In the scheme above this involves at least two essential interfaces: the one between the computational and physical world, and the other one between the application and the platform. Architecture is a primary driver, and it needs to be designed and refined, before the component implementation happens. Architecture models should be preserved and used throughout the development process.

We envision that eventually high-fidelity models of the platform and the physical system are available. While for physical devices this is a well-established practice, for software systems (platforms) this is not always possible, as it could be too expensive to develop. In this situation, the (software) models could be low-fidelity, and they need to be replaced with real implementations as soon as feasible.

The key process element in the above approach is the continuous existence of an executable system, with a concrete architecture, well-defined interfaces, and an executable form. This can give the designers an early feedback about their work, and for the customers the opportunity for early evaluation. The design and implementation evolves from a fully simulated version to a fully implemented version as shown on the figure below.

The development starts with a fully simulated system, then the real computational platform is introduced (as this is the hardest to model and simulate). Note that the real platform should have timing-accurate interfaces towards a (real-time) simulation engine, and functional interfaces towards the simulated application. This step is followed by a step where the real application is run on the real platform, with a simulated physical system and environment, and the final step is the full realization of the real system.

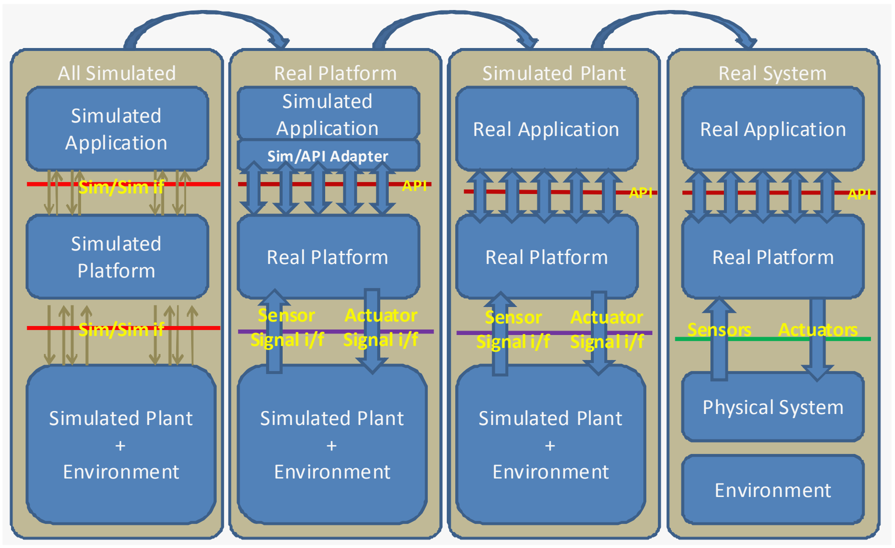

Fig. 3. Continuous concurrent integration

3 Related Work

The approach described above has grown out from the well-known development practices of model-driven development [6,7]. In the various MDD approaches the use of models is pervasive, models are used for 'higher-order', domain-specific programming, for code generation, and for analysis and verification. Our approach uses these techniques and concepts, but it also considers the effects of and the integration with the physical system and the environment.

The use of simulation in developing complex embedded systems is a well-known practice as well [8]. The use of simulations to approximate the behavior of software systems has been proposed in [9]. A key concept for carrying over results from simulations to implementations is 'model continuity' has been proposed in [12]. These techniques provide valuable insights into the simulation-based integration of systems, and technology (e.g. the DEVS-based approach for simulation) for actually time-synchronization and coordination. The proposed approach builds on these foundations, but extends and integrates them with the model-based development framework.

4 Status

We have started work on an integrated toolchain [10][13] that supports the development paradigm outlined above. The toolchain uses Simulink/Stateflow as the primary simulation environment (for fully simulated implementations). The platform modeling aspect is handled with a modified version of the TrueTime package, which allows the co-simulation of controller models, platform models, and physical plant models. The controller models are then imported into our modeling tool that supports a modeling language called EsMoL, which is then used to specify (1) hardware platform models, (2) software component models (whose implementation comes from the Simulink controller models), and (3) deployment models that connect the two. A set of integrated code generator tools produces executable code from the models that could be run either in the Simulink environment, or on a target platform. If the code is run in Simulink, physical plant models and TrueTime platform models could be used to study how the 'real code' runs against a simulated platform and plant. We have two target platforms: one is a TTP/C cluster from TTTech Inc.: a time-triggered platform of four controllers connected via a TTP/C bus running periodically scheduled components; the other one is a software emulation of the TTP/C cluster using Linux nodes connected via an isolated TCP/UDP network. For the latter, we have built a scheduler tool to compute time-triggered scheduled. The code generators produce all the 'wrapping code' needed to run controller code on the platform. The toolsuite also includes interfaces towards verification tools: the code generators produce the code executable code first in an abstract form that could be used to 'print' imperative code. This way the executable code could be subjected to analysis via using a tool like the Java Path Finder (JPF) [14]. Currently we are testing the toolchain on various applications following the development paradigm described.

5 Conclusions

In this paper we have introduced a framework for the design of cyber-physical system that is model-based and places great emphasis on early integration, based on the models. Some elements of the framework are already available (e.g. modeling languages and generators for embedded systems), and technology is available [11] for constructing the rest. Currently we are working on realizing and trying out a toolchain that implements the concepts and architecture described above, and which also integrates code verification tools.

Acknowledgements

This work was sponsored (in part) by the Air Force Office of Scientific Research, USAF, under grant/contract number FA9550-06-0312. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

References

1. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE 91(1), 145–164 (2003)
2. Mathworks, Inc., `http://www.mathworks.com`
3. National Instruments, `http://www.ni.com`
4. Cervin, A., Henriksson, D., Lincoln, B., Eker, J., Årzén, K.-E.: How Does Control Timing Affect Performance? IEEE Control Systems Magazine 23(3), 16–30 (2003)

5. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A denotational framework for comparing models of computation. Technical Report UCB/ERL M97/11, EECS Department, University of California, Berkeley (1997)
6. Model-Driven Architecture, http://www.omg.org/mda
7. Model-Integrated Computing, http://www.isis.vanderbilt.edu/research/MIC
8. Papp, Z., Dorrepaal, M., Verburg, D.J.: Distributed Hardware-in-the-Loop Simulator for Autonomous Continuous Dynamical Systems with Spatially Constrained Interactions. In: Proceedings of the 17th international Symposium on Parallel and Distributed Processing. IPDPS, April 22 - 26, 2003, vol. 119, p. 1. IEEE Computer Society, Washington (2003)
9. Huang, D., Sarjoughian, H.S.: Software and Simulation Modeling for Real-time Software-intensive System. In: The 8th IEEE International Symposium on Distributed Simulation and Real Time Applications, Budapest, Hungary, October, pp. 196–203.
10. Sztipanovits, J., Karsai, G., Neema, S., Nine, H., Porter, J., Thibodeaux, R., Volgyesi, P.: Towards a Model-based Toolchain for the High-Confidence Design of Embedded Systems. In: Work-in-Progress Workshop at the Real-Time Application Systems conference (2008)
11. Karsai, G., Ledeczi, A., Neema, S., Sztipanovits, J.: The Model-Integrated Computing Toolsuite: Metaprogrammable Tools for Embedded Control System Design. In: IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany (2006)
12. Hu, X., Zeigler, B.P.: Model continuity in the design of dynamic distributed real-time systems. IEEE Transactions on Systems, Man, and Cybernetics, Part A 35(6), 867–878 (2005)
13. Porter, J., Karsai, G., Volgyesi, P., Nine, H., Humke, P., Hemingway, G., Thibodeaux, R., Sztipanovits, J.: Towards Model-Based Integration of Tools and Techniques for Embedded Control System Design, Verification, and Implementation. In: The Models 2008 workshop on Model Based Architecting and Construction of Embedded Systems (submitted, 2008)
14. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal 10(2) (April 2003)