Rigorous Design of Real-Time Embedded Control Systems

R. Bhattacharya\(^{a}\), S. Mijanovic\(^{b}\), E. Scholte\(^{b}\), A. Ferrari\(^{d}\), M. Huzmezan\(^{b}\), M. Lelic\(^{c}\), M. Atalla\(^{c}\)

\(^{a}\) Texas A\&M University, Department of Aerospace Engineering, 727C H.R. Bright Building 3141 TAMU College Station, TX 77843-3141

\(^{b}\) Controls & Embedded Systems Group, Systems Department, United Technologies Research Center, East Hartford, CT, 06108

\(^{c}\) UTC Fire & Security Program Office, United Technologies Research Center, East Hartford, CT, 06108

\(^{d}\) PARADES, Palazzo Bonadies, Via San Pantaleo 66, 00186, Rome, Italy

Abstract

Two examples of the rigorous design of embedded control systems based on Platform Based Design (PBD) methodology, originally advanced by University of California at Berkeley and the PARADES GEIE research lab, are presented in this paper. Even though the included examples belong to different industrial applications, Finite State Machines (FSM) are used in both applications as the main model of computations for modeling the underlying control strategies. The overall design cycle including modeling, linking models with the requirements, automatic code generation, and finally testing and validation is presented. In one of the applications, the developed control models have been validated on the actual hardware and the same is expected to be done for the second application in this year.

1 Introduction

Embedded systems have become an integral part of an ever increasing number of advanced products, including transportation industry products, electrical and electronic appliances, process control, security, telecommunication products etc. It is even expected that within a relatively short period of time embedded systems will become a part of almost all equipment designed and manufactured in Europe, USA, and Asia [1]. As a result, the importance of embedded systems in today’s world and modern economies is already significant and considering above can reasonably be expected to become even more so in the future.

The functionality of embedded systems is rapidly increasing, from the one required by substituting simple electro-mechanical components (e.g. early 80’s automotive engine management systems), to very complex control algorithms capable of adapting in real time to changes in the controlled system and its environment. The paradigm for system design and implementation is also shifting from a centralized, single processor framework, to a decentralized, distributed processor implementation framework. Distribution and decentralization of services and components is driven by the falling cost of hardware, increasing computational power, increasingly complex control algorithms and development of new, low cost micro-sensors and actuators. A distributed, modular hardware architecture offers the potential benefit of being highly reconfigurable, fault tolerant and inexpensive. Modularity can also accelerate the development time of products, since groups can work in parallel on individual system components. These benefits, however, come with a price; the need for sophisticated, reliable software to manage the distributed collection of components and tasks.

The growing complexity of real-time systems suggests a need for a rigorous framework for embedded system design that will meet performance, quality, safety, cost and time-to-market constraints. This is pushing the embedded systems community towards formulating a framework for a rigorous embedded system design process that caters to these needs.
The most important issues related to the embedded systems design are outlined in Section 2. A methodology (Platform Based Design - PBD) for dealing with these issues is presented in Section 3. PBD methodology has been implemented for the design of two embedded control systems presented in Sections 4-5. Finally, conclusions are given in Section 6.

2 Complexity in Real-Time Embedded System Design

Embedded systems are electronic systems that in real-time interact with the physical world. The interaction requires the computation of a set of functions within the given time constraints, making embedded systems an important class of real-time systems. These systems must comply not only with the desired functionality and available time for computation, but also with other important constraints, such as reliability and power consumption, making their design a complex task. The complexity of embedded systems can be perceived on the nature of the environment it interacts with, the complexity of the control algorithms, the real-time aspects of the computation as strictness of deadlines, the robustness as fault-tolerance and reliability of the system, the decompositions of the system in terms of degree of communication and coordination among various computational elements.

Environment and Control Algorithm: The environment that interacts with the real-time system and the desired composed behavior plays a fundamental role in system design. Some environments and desired behaviors are well defined and deterministic and they give rise to real-time systems in which the overall system shows properties that can be guaranteed a priori. Unfortunately, this happens only for very few cases. In general, complex analysis must be performed to guarantee the behavior of the controlled system and the satisfaction of system properties.

Strictness of Deadlines: Computational tasks occurring in a real-time system have timing constraints, or deadlines, which need to be satisfied for the system to be functionally useful. The real-time systems can be classified in three categories based on the nature of the deadlines they face. They could be hard real-time systems, if the consequences of not executing a task before its deadline is catastrophic. The real-time systems are firm, if the consequences of missed deadlines are not severe. Finally, a real-time system is categorized as a soft real-time system if the utility of the system degrades with the time after the deadline expires. Some real-time systems, like flight control avionics, are hard real-time systems and failure to meet the deadlines of the constituting scheduling tasks, may result in catastrophic consequences. An off-line scheduling analysis [2, 3] is usually conducted to ensure that the deadlines of all the tasks are met.

Fault Tolerance and Reliability: Fault tolerance is defined as the system’s ability to deliver the expected service even in the presence of faults. A real-time embedded system may fail to function properly either because of error in its hardware, software, or both, or because it fails to respond in time to meet the timing requirements demanded by the environment it interacts with. Reliability it is a more general term and refers to the probability that, for a given time and in specified conditions, a system, hardware and software, will perform the services for which it has been designed or intended.

Degree of Communication and Coordination: Embedded systems are decomposed in several components, hardware and software, and the level of communication and coordination among them is an important aspect of the complexity of the system. If the coordination is low, the associated real-time tasks might be fairly independent of each other this makes analyses of such systems possible. However, with an increasing control algorithm complexity and role of information based systems (pg. 18 in [4]), the level of interaction and cooperation between the components is on the rise and the situations in which pervasive computing, sensing, and communication are becoming common.

The complexity of the design of these systems is expressed by the enormous solution space made available by the current silicon and software technology that must be explored satisfying the constraints and minimizing the cost of the entire system.

In the work presented in this paper, some of these design issues are dealt with using a Platform Based Design approach. The first application described in this paper is a safety critical system, therefore requiring a thorough analysis of the reliability. The second application is focused on the scalability of the process and deals with modularity and requirements tracing throughout the design. Before the applications and results are discussed, the methodology is summarized in the next section.

3 Platform Based Design - Paradigm for Embedded System Development

Platform Based Design (PBD) is a methodology for the design of embedded systems [5], originally proposed by the University of California at Berkeley and the PARADES GEIE reseach lab[6]. An essential component of the system design paradigm is the orthogonalization of concerns, i.e., the separation of the various aspects of design to allow more effective exploration of alternate solutions.

The basic tenets of the Platform-based Design Methodology, as exposed in [5], are:

- Regarding design as a “meeting-in-the-middle process” where successive refinements of specifications meet with abstractions of potential implementations;
- The identification of precisely defined layers where the refinement and abstraction process take place.
Such defined layers then support designs built upon them, isolating them from the lower-level details but letting enough information transpire about lower levels of abstraction to allow design space exploration with a fairly accurate prediction of the final implementation properties. The information should be incorporated in appropriate parameters that annotate design choices at the present layer of abstraction. These layers of abstraction are called Platforms. In this paper, a platform is defined to be an abstraction layer in the design flow that facilitates a number of possible refinements into a subsequent abstraction layer (platform) in the design flow. The abstraction layer contains several possible design solutions but limits the design exploration space. This is illustrated in Fig.1. During the design process, at every step we choose a platform instance in the platform space. For example, the solution $\text{Soln}_n$ in platform $P_k$ in Fig.1(d), assumes no knowledge of a particular available solution $\text{Soln}_{n-1}, \text{Soln}_j, \text{Soln}_{j+1}$ in the lower platform $P_{k+1}$. Every pair of platforms, the tools and methods that are used to map the upper layer of abstraction into the lower level one is a platform stack.

Key to the application of the design principle is the careful definition of the platform layers. Platforms can be defined at several point in the design process. Some levels of abstractions are more important than others in the overall design trade-off space. In particular, the articulation point between system definition and implementation is a critical one for design quality and time. Part of this articulation point is the mapping of functionality to architecture that becomes an essential step from conception to implementation. Fig.2 illustrates the platform based design paradigm by stressing the separation between:

- **Function** - defines what the system is supposed to do. The behavior of the system is expressed as the composition of functional processes communicating through media. The composition of functional processes is the *functional architecture*. The function layer, in general, only represents the system’s behavior, and does not express any of the hardware implementation decisions. It is however restricted by the constraints imposed by the architecture layer below. Examples of such constraints are maximum communication bandwidth, available memory, quantization of data, clock speed, etc.

- **Architecture** - defines how it is realized. The architecture is made up of elements that will physically realize the functionality. These elements can be thought of as containers that support a variety of behaviors for given costs (e.g. time, power, area). The architectural elements support different types of functionality, but do not specify which ones will be used or when. The information necessary to realize the functionality of the system is provided by the function layer as specifications. The specifications contain information on the run-time characteristics of the algorithms that realize the functionality, such as sequence of execution, execution deadlines, priority, etc. Examples of architectural elements include hardware, microprocessors, digital signal processors, memories, buses, and reconfigurable logic.

The mapping of one platform to another is facilitated if *formal descriptions* in the form of *models* are used to capture functions and architecture. For example, functional processes of a control system can be modeled in Simulink, a graphical programming environment for developing control systems developed by Mathworks [7]. Once functionality has been defined as Simulink models, code generation tools such as TargetLink by dSPACE [8] or Real-Time Workshop by Mathworks, can be used to map functionality to specific software platforms. Fig.(2) illustrates the mapping of functionality to architecture in more detail for a generic system $\text{mySystem}$. In PBD approach, the non-idealities of the implementation must
be captured as architectural models during the refinement of the system from the top level abstraction down to implementation. In the functional/architecture mapping the non idealties of computation and communication must be taken into account as platform abstractions. These abstractions, limited to the interesting aspects for the desired mapping, can be seen as a virtual prototyping of the embedded system.

3.1 Rigorous Design Process for Embedded Systems

The traditional V cycle, as shown in Fig. 3.1, is purely top-down. It is good for new functionality but does not take into account available alternative solutions. The bottom-up approach is good for strong component reuse, but it becomes difficult to guarantee the functional requirements of the integrated system. In PBD framework, embedded system design is shifting from physical prototyping and integration to virtual prototyping and integration. Virtual prototyping is enabled by the modeling framework discussed above. A key feature of PBD compared to the traditional V-cycle is the continuous verification throughout the design part of the cycle. At each step, the models are verified to ensure that the virtual prototype will exactly represent the functionality described by the requirements and that the non-functional constraints are met by the estimations based on platform models.

The PBD methodology mapped to the classical design V is represented in Fig. 4. For the sake of this paper, the design V can be partitioned into three activities as described next.

3.1.1 Capturing Requirements using Models

The main steps involved in this process are mapping requirements to models. Since given functional requirements can be implemented in many ways, it is necessary to define the functional architecture. Obviously there is a degree of freedom in selecting the desired functional architecture and model of computation thus making it a platform.

Once the models have been developed, the functional requirements are linked to the models, which are subsequently validated against the requirements. It is important to stress that the advocated methodology would ask for expressing the requirements in terms of formal properties. Unfortunately, not a single solution today is capable of formally describe the entire set of functional and non functional requirements of a real-life system. The Unified Modeling Language (UML) [9] semi-formal language and its derivatives, such as SysML, try to provide answers to this problem, unfortunately not quite satisfactory yet. More formal approaches called assertion based verification, based on temporal logic, are becoming familiar in silicon design methodology, but again they cover partially only timing constraints. To overcome this limitation, and only when these methods are not applicable, a simulation based verification approach can be used. Input traces are created that define the test environment for a given requirement. The models executed in simulation map the input traces to output traces. The output traces are visually inspected to verify correct implementation of the requirements, i.e. functional correctness of the models. Once the models are validated, the output traces are used to design the platform.
satisfy the requirements, the input and output traces are qualified as "golden traces", and are further used in the regression testing downstream in the design flow. Model validation process is represented in Fig.4 by the dotted arrow going from "Models" to "REQ". The text "Manual Test Vectors" on the arrow, represents the manually generated input traces that define the test environment. The models now represent the requirements without any ambiguity and are strongly coupled with the functional architecture chosen. This refines the models one step closer to the real implementation. It is important to point out that the described method is only one among several possible solutions on how to verify the requirements provided as non-formal properties by simulation.

3.1.2 Automatic Code Generation from Abstract Models

Once the requirements have been captured as abstract models, code generation technology is used to map functionality, defined as models, to the computational platform (see Figure 4), typically expressed as ANSI ‘C’ language. The mapping of models to code allows various flexibilities such as data type abstraction, polymorphic operators, float-to-fixed point translations, target data-type refinement, set of API (Application Program Interface) to implement hardware related or communication services. It is possible to synthesize entire applications by mapping sub-systems to real-time tasks. The platform is modeled for synthesis by abstracting all these aspects and providing them to the code generator as a separate model. By formally defining the interface of the models with the platform API and platform non idealities, the number of expected errors for the integration of the automatically generated code with the software platform is drastically reduced. Moreover, the generated code is verified automatically, as unit testing, by running the simulation performed in the previous design step with the generated code in the loop and comparing the model and code input/output traces.

3.1.3 Integration with Physical System

Given the automatically generated code and the software platform, system wide testing is then conducted to exercise hardware-software interactions (integration with API, in Figure 4). This could be conducted by using hardware platform models, if available, or with proper hardware-in-the-loop testing facility. The same input/output traces, used to validate the models and the mapping during the previous design phases, can be used to automatically validate the integration with the hardware platform, hence drastically reducing the number of errors not discovered in this design phase.

4 UTRC Application Example 1

This application was a safety critical application, which had previously been implemented using a traditional approach, i.e. using a standard software engineering approach. The original implementation was largely done by directly writing C-code based upon a description of the requirements in DOORS [10], and full verification was only done on the actual hardware at a later stage. At high level, the overall system can be described by the following sequential steps:

1. Read bytes from sensors.
2. Convert bytes to actual data and flag any errors.
3. Based on the data and error information turn on the safety system if deemed necessary.

A novel way of designing the above embedded system, based on PBD methodology detailed in Section 3, is presented below. Also, a full verification and testing was eventually completed on the actual prototype hardware. For this project, the tool of choice was Simulink/Stateflow. This is largely because of the existing knowledge of use, maturity and support for elements of PBD such as model based testing, code generation, etc.

4.1 Mapping of Requirements to Models

The requirements for the system were provided in natural language using tools such as Microsoft Word and DOORS. Simulink models were developed from the requirements. The V&V toolbox of Simulink was then used to link the models to the requirements document for traceability.

The realization of the system from text descriptions involved an intermediate step where the functional architecture of the system had to be decided, as the requirements document did not specify this architecture. Thus, there is a degree of freedom in selecting the functional architecture. The selected functional architecture of the system is shown in Fig.5.

The definition of the functional architecture enabled mapping of text based requirements to formal models. The three subsystems of the system were modeled in Simulink and Stateflow, which resulted in a library of components available for re-use in future projects. As an
example, a subsystem of this library is shown in Figure 6. The system was realized as a virtual prototype by interconnecting these library elements. The requirements were validated and verified in desktop simulation environment.

One of the key benefits of modeling is the reduced time for testing the functionality and higher confidence in the validation results. The modeling environment allows control of the input values and the sequence in which they are applied to the system. This allows very precise design of experiments to test the functionality and verify requirements. Traditionally, the testing is done on the physical prototype with actual communication hardware where it is difficult to control timing and sequence of inputs. This leads to lower confidence in the testing procedure and its results. An additional benefit of modeling the communication protocol as a hierarchical finite state machine is a great amount of clarity provided this way.

4.2 Model Testing and Validation

For successful mapping of requirements to models, it is necessary to validate the models against requirements. For this purpose, test vectors were generated manually to validate each requirement. Input conditions were described as traces or time trajectories and were fed into the models. The output trajectories obtained from the models were visually qualified to be correct. Once accurate outputs were obtained, the pair of input and output trajectories were qualified as golden traces which enabled automatic regression testing. Any subsequent modification done to the models such as alternate implementations can be automatically verified by feeding the input golden trace into the system and comparing the outputs of the modified model with the golden output trace. This comparison can be done programmatically. With such a capability, features can be added incrementally and reliably, and the entire system can be validated against all previous requirements by means of regression testing.

The next step taken was to determine whether all the parts of the model are exercised and whether there are any errors, deadlocks or other issues to be resolved. The Simulink V&V toolbox was used to compute the coverage. The model coverage tool determines the extent to which a model test case exercises simulation pathways through a model. The tool performs any or all of the following types of coverage analysis, depending on which coverage options is selected. The coverage options provided by Simulink V&V toolbox are cyclomatic complexity, decision coverage, condition coverage, MC/DC and lookup table coverage.

For the above subsystem, condition and decision coverage analysis were conducted. From the analysis it was observed that the test cases achieved 100% coverage for decision and 75% coverage for condition.

For the entire system approximately 85% coverage of the models was achieved using manually generated traces. It is important to achieve high percentage of coverage for a reliable system. The remaining 15% is due to non-feasible scenarios and redundancy in the model for reconfigurability that was not in the requirements. Often models include features that capture a superset of the requirements and test cases have to be built to test beyond the requirements.

Automatic test vectors can be generated from Simulink/Stateflow models that can exercise the system in all possible combinations. It is, however, difficult to associate requirement to such test vectors. They are useful to prove absence of deadlocks and other modeling errors. For this application, this was not done. However, they were used to establish equivalence of the models and the code generated from them. Once the models have been verified, they are transformed into target models. Target models are such that the code generated from these models interfaces directly with the variables in the target platform. Target models are one step closer to non-feasible scenarios and redundancy in the model to implementation and contain information about the input and output variables. In this application, interprocess communication is achieved by means of global variables. The target variables are defined to be global variables by header files and are included in the header file auto-generated by Realtime Workshop Embedded Coder (RTW), the code generator from Mathworks. The code generator generates the Simulink/Stateflow Subsystem as a function Subsystem() and was embedded in the existing code.

The auto-generated code was embedded into Simulink using S-functions to enable desktop simulation of the generated code. The golden traces were used to perform regression tests on the code. All models using the embedded code produced the exact same output traces as the Simulink/Stateflow models, or functional models, from which code was generated.

Another tool: Safety Test Builder [13], was used to automatically generate test vectors to maximize coverage. Recall that the golden traces, based on the requirements,
Table 1
Performance metrics of the original code (LC) and autogen-
erated code (AGC).

<table>
<thead>
<tr>
<th>Metric</th>
<th>LC</th>
<th>AGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>11.70</td>
<td>11.75</td>
</tr>
<tr>
<td>Lines of code</td>
<td>4293</td>
<td>7180</td>
</tr>
<tr>
<td>Stack Usage</td>
<td>916</td>
<td>1344</td>
</tr>
</tbody>
</table>

did not achieve 100% coverage of the models developed. With the automatically generated test vectors, coverage increased to 95% compared to the original 85%. The 5% uncovered can be attributed to either limitations of the tool used to generate test vectors or the presence of unreachable elements in the model.

4.3 Hardware Integration and Performance

Once the code has been verified it was implemented in a Hardware-In-the-Loop (HIL) system. The hardware setup used for this application is shown in Figure 7, showing two communicating boards. The HIL system was tested following the same steps as for the desktop simulation models to verify that the embedded application was functionally correct. Additionally, the performance of the application was analyzed. Table 1 shows the performance of the automatically generated code (AGC) data and code size is greater than the legacy code (LC). This is expected. The AGC implementation largely contains finite state machines (FSM), which result in larger footprint. In terms of execution time, the difference between the AGC and LC is negligible. The footprint of the code generated from the models are expected to be larger than those developed manually. Finite state machines will result in larger code because of features like bookkeeping of states and repetition of conditions for state transitions. The execution time of the finite state machines are expected to be smaller since the software remembers its current state and only evaluates logic relevant to the current state, instead of executing the entire code every time.

The next step of the process was embedding the application in the actual system. This was done and the system was thoroughly tested against the requirements for approximately one week. As expected, the testing now was straightforward, since most of the testing had already been done on the models earlier in the design cycle. The impact of PBD on engineering effectiveness is the greatest in the verification phase of the development cycle.

5 UTRC Application Example 2

In this section, another example of implementing PBD methodology for the design of an embedded control system is presented. In this case, PBD methodology has been implemented for designing an embedded control system for a different application. In addition to implementing PBD methodology, another similarity between this and the example presented in the previous section is that Finite State Machines (FSM) [11,12] are used as a primary model of computation for implementing underlying control strategy.

In this project, application software models, implementing control strategy using FSM, have been developed based on Unified Modeling Language (UML) 2.0 [9]. The UML is a family of graphical notations, backed by a single meta-model, that help in describing and designing software systems [14]. In contrast to the example presented in the previous section, a software development suite used for developing the models, linking them with the requirements, as well as validating them was Rhapsody of I-Logix [15]. In particular, Rhapsody-in-C has been used for model development and automatic C-code generation, Rhapsody Gateway for requirements management, and Test Conductor for model validation.

The models developed for this project are expected to be validated and tested on the actual hardware in this year.

5.1 Requirements Management

In this example, functional requirements (i.e. what the system is supposed to do - see orthogonalization of concerns detailed in 3) have been developed and managed using UML use cases [14,16]. The overall functional requirements have been grouped in 14 use cases. The relationships between these use cases (mainly “include” and “extend” relationships as defined by UML 2.0 [9]), as well as the links between the use cases and external actors have been defined with two use case diagrams. One of the diagrams is illustrated in Figure 8.

Some of the benefits of applying UML and use cases for requirements management are: (1) They provide multiple views (including graphical) of the functional requirements to make sure they are accurately defined and easy to understand; (2) Use case diagrams define the relationships between the use cases so that they can easily be prioritized; (3) Sequence diagrams are used to define the scenarios of each use case to help the developers to model the system correctly. (Sequence diagrams essentially represent instantiations of the requirements and,
as illustrated below in Section 5.3, are also further used for model testing/validation; (4) Use case requirements are defined as allocated functional requirements to ensure better traceability later in the development and validation process.

5.2 Functional Architecture and Modeling

One of the most widely used types of UML diagrams is class diagram (in the Rhapsody tool, it is also known as Object Model Diagram - OMD). Object model diagram defines objects and their relationships within the system, in other words it gives the static view of the system. The actual control strategy is further developed defining statecharts (Hierarchical Finite State Machines - FSM) within defined classes of the system. The overall application software architecture in this project has been defined with two object model diagrams. Some of the main considerations taken into account when defining object model diagrams were: (1) Include abstractions of physical entities (e.g. detector, bus, panel, display, etc.) of the system; (2) Include logical entities (e.g. detection, evacuation, and protection zone, LED group, etc.) of the system; (3) OMD’s need to be as generic as possible to ensure model reusability; (4) Dynamic generation of class instances based on configuration files to ensure model flexibility.

One of the defined object model diagrams is illustrated in Figure 9a. In this figure, both physical and logical entities, as well as the relationships between them are illustrated. As an illustration, statechart (i.e., FSM) of one of the defined classes is illustrated in Figure 10. FSM models the actual control strategy, i.e. actions/computations that need to be carried out depending on a particular state of the system, configuration, and external outputs.

The above object model diagrams along with the finite state machines modeling control strategies of their corresponding classes are used as a basis for automatic C-code generation by Rhapsody-in-C development tool. Even though certain parts of the code still have to be manually written, much greater portion of the overall code is automatically generated by the tool.

5.3 Model Testing and Validation

Model testing, required for ensuring correct model functionality, has been carried out using Test Conductor - an add-on software package provided by I-Logix [15]. Testing procedure was based on sequence diagrams, used for specifying scenarios within various use cases as detailed in Section 5.1 and the developed UML models. Namely, sequence diagrams used for specifying scenarios within different use cases are used as "inputs" into Test Conductor tool which simulates the model behavior using C code generated as explained in Section 5.2. The tool subsequently generates all required exogenous signals and verifies whether the actual model behavior is identical to the desired behavior as prescribed by the given sequence
6 Conclusions

Embedded systems have become an integral part of the majority of advanced products in the modern world. Along with their ever increasing presence and importance, their complexity has also been steadily growing. As a result, a rigorous and efficient design of such systems has become of great importance in today’s world and modern economies.

Two examples of rigorous design of embedded control systems based on Platform Based Design (PBD) methodology have been presented in this paper. The included examples belong to different industrial application areas of interest to United Technologies Corporation. Two different software development suites have been used in the presented examples, but in both cases, the overall design cycle including requirements management, modeling, automatic code generation, validation and testing has been presented. The developed models have been validated on the actual hardware in case of one of the presented examples, and the same is expected to be done for the second example in this year.

7 Acknowledgement

We would like to thank Weiying Cheng, Leonardo Mangeruca, Julien Marquet, Timoor Sediq, Marc Soboloff, and other people from UTC for their contributions to the projects described in this paper.

References