MODEL-BASED DEPLOYMENT WITH AUTOFOCUS: A FIRST CUT

Jan Romberg

Institut für Informatik
TU München
D-80290 München, Germany
romberg@in.tum.de

ABSTRACT

Embedded control applications have to meet detailed performance requirements on their interactions with the external environment. Examples for such requirements include response times, execution rates, and low-level interactions with hardware. Current approaches to embedded real-time software development are often code-centric and do not allow predictions about the satisfiability of performance constraints. The underlying platform is an intrinsic part of the design, yielding poor portability.

We describe model based deployment, an approach to extend the capabilities of the AutoFOCUS software development method towards the design and implementation of distributed real-time systems. We show how the technical architecture can be expressed in terms of our modeling language, and how performance predictions can be drawn from the system model. Our goal is to define an integrated approach to design, verification, and testing for distributed reactive systems with special consideration of the platform properties.

1. INTRODUCTION

In the widely expanding field of embedded real-time systems, existing design practices are largely code-based with little or no capabilities for predictions of actual performance characteristics. Portability among different hardware platforms and operating systems frequently has to be sacrificed for the conveniences of a target-specific development process using specialized development tools for deployment. Formal verification of the logical and real-time properties of a system, as well as weaker notions of verification such as automatic test case generation, can usually only be performed with the aid of domain-specific abstractions, *models*, of the system

Unfortunately, many of the analysis capabilities available for "abstract" models of the system, that is, models that assume a fixed scheduling, communication and partitioning scheme, yield little or no assertions about the behavior of the actual program running on the platform if the real-time properties of the target are unknown or cannot be incorporated into the model. Consequently, one goal of a model-based approach should be the integration of appropriate abstractions of processors, channels, and scheduling policies into the system model.

An important benefit of model-based software engineering is that the abstract logical specification can, in principle, be kept separate from those parts of the system determined by the underlying hardware and operating system. As a consequence, the logical model is *retargetable*, i.e. the implementation can be deployed on a variety of technical architectures. While the definition of the *logical architecture* and the *technical architecture* is left to the developer, the structure of the generated software should be implicit in our modeling paradigm so code generation can be fully automated. Current practice dictates that at some level, there certainly is no alternative to the integration of hand-written or third-party code into the application; this should be kept to a minimum, however, as the technical and logical models cannot accurately reflect the behavior of the integrated extensions.

In the remainder of this paper, we extend the graphical description language of AutoFOCUS, previously used for purely logical descriptions of globally synchronized systems, by a notation for the description of technical architectures. We enrich AutoFOCUS's basic semantics by an extended interpretation using stereotypes on channels. We then show how the logical and technical model can be combined to a system model in a partitioning step and how the AutoFOCUS tool can be used for decision support for partitioning decisions. Finally, we give an outlook discussing further extensions of our method.

Related work. Most of the related work stems from the domain of hardware/software co-design. Gupta [1] defines the Vulcan approach to analysis and synthesis of hardware/software based on the specification language HardwareC, a hierarchical flow graph model with non-deterministic delays for synchronized read/write operations. As opposed to our approach, the target architecture in Vulcan is restricted to a single-processor, single-bus architecture. Vulcan synthesises both schedules from timing constraints and hardware (ASIC) components from the logical description, which is outside the scope of our work.

Giotto [2] is a language specifically designed for real-time implementation of switched-mode control systems. Giotto uses the "embedded machine" as an abstract operating system (middleware) which controls and surveys the execution of tasks in real time. Essentially, Giotto specifies a set of properties for fixed sets of tasks, so-called *modes*, which roughly correspond to discrete control states of a high-level control component in the AutoFOCUS model. A mode is a set of tasks active at a given time. Because of its entirely time-triggered and not event-triggered nature, Giotto puts a heavy restriction on the kind of implementable system. Nevertheless, we perceive the application domain of Giotto as an important benchmark for our work, e.g. in the context of flight control systems [3].

The POLIS [4] approach to hardware/software co-design uses CFSMs (Codesign Finite State Machines) for system specification. CFSMs are similar to AutoFOCUS components in some re-

This work was supported with funds of the Deutsche Forschungsgemeinschaft under reference number Br 887/9 within the priority program *Design and design methodology of embedded systems*.

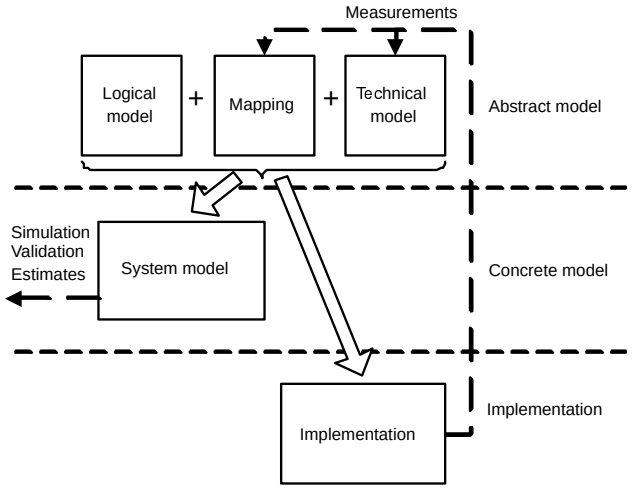

Figure 1: Design flow

spects: like components, CFSM are based on atomic execution of transition operations, and events between CFSMs are buffered. Send/receive operations, however, are not bound to global clock ticks and may take an indefinite amount of time. Communication is broadcast, with mutually disjoint sets of output events. POLIS covers all the typical codesign steps like translation from high-level languages (e.g. ESTEREL [5]), simulation, partitioning and scheduling, hardware synthesis, and software synthesis. The generated C code is restricted to a limited subset which is in close correspondance with the internal representation of software in POLIS.

2. DESIGN FLOW

2.1. Process

Our design flow for software synthesis is illustrated in Fig. 1.

The designer starts by specifying the intended behavior of the software with the *logical model*. The logical model specifies both the structure of the application software and the behavior for each component. Other properties such as communication mechanisms and scheduling properties are specified abstractly. The characteristics of one or more technical platforms are specified within the *technical model*. The technical model will usually be supported by libraries.

The synthesis step uses a specified *mapping* from the logical to the physical model to synthesize both the *system model* and the *implementation*. The system model is the internal representation of the combined logical and technical models. It is used to obtain estimates, e.g. for timing properties, and to perform analysis through simulation, verification, and generation of test cases. The implementation denotes the actual software that is deployed to the target platform.

2.2. Implementation and system model

We make some simplifying assumptions about the target platform. The technical architecture is assumed to be built up from a number of nodes, each equipped with one processor, memory, and possibly sensors and actuators. Processors are assumed to communicate through I/O links (channels) exclusively; if shared memory communication is available between two processors, some kind of flow control is assumed. For hardware-to-software communication, we assume a polling scheme; the opposite direction is simply handled by writing to a specific memory location.

If our model-based analysis and simulation methods on firm ground, the following two equivalences between the system model and the implementation are crucial:

- *functional equivalence*, the correspondance of the behavioral semantics of the logical model — the *simulation semantics* — with the behavior of the actual distributed implementation — the *technical semantics*.
- *non-functional equivalence*, the correspondance of estimates for non-functional values (such as timing properties) drawn from the model with actual properties of the implementation.

For distributed systems based on general-purpose hardware and operating systems, establishing either one of the above correspondances in a precise way proves to be a futile effort. For instance, it is extremely difficult to derive accurate models for modern processor microarchitectures [6]. Consequently, our approach weakens the above requirements and requires *some* correspondance between the model and the implementation. The development of sufficient metrics to define the permitted degree of "looseness" allowed by a model is left to future research.

3. AutoFOCUS

3.1. Introduction

AutoFOCUS [7] is a CASE-tool for distributed and embedded systems design and was developed at Technische Universität München. Using a number of dedicated graphical description languages, the AutoFOCUS tool allows the specification of views of a common underlying system model.

Structural view. In a *System Structure Diagram (SSD)*, the architecture of a distributed system can be defined. The visual syntax of SSDs consists of *components*, *ports* of components, and *channels* between ports, analogous to other notations such as UML-RT/ROOM [8]. Components may either be specified as a network of sub-components or through their behavior.

Behavioral view. The *behavior* of a leaf component is described by *State Transition Diagrams (STD)*. An STD is an extended hierarchical state machine with a set of control states, transitions, and local variables. AutoFOCUS offers extensive support for pre- and user-defined data types with its *Data Type Description (DTD)* language.

Interaction view. Interactions are specified by *Extended Event Traces*, which are a simplified version of ITU's Message Sequence Charts (MSCs) or UML's Sequence Diagrams. EETs are adapted to the clock-synchronous semantics of AutoFOCUS by the introduction of *tick marks*.

3.2. Analysis and synthesis

In addition to being an editor for system models, the AutoFOCUS toolset offers detailed analysis of system models through formal verification. Using the Quest framework, a range of different model checkers (e.g. SPIN, μcke, SMV), propositional solvers (SATO), and theorem provers can be employed. An interactive simulation

environment is also incorporated in the tool. In addition to its analysis capabilities, AutoFOCUS supports a semi-automated approach to test sequence synthesis using a Constraint Logic Programming framework. For single-processor, single-task implementations, code generators for C, Java, and high-integrity Ada are available.

3.3. Basic semantics

The behavioral semantics of AutoFOCUS components is similar to the synchronous-reactive style as characterized by the ESTEREL and Statecharts languages. Synchronous here refers to the model of computation, where transitions are assumed to be atomic and synchronized through a global clock. While communication in Esterel and Statecharts is instantaneous and broadcast, AutoFOCUS restricts communication to *channels* between components; the exchange of events between components consumes one clock tick, i.e. channels are buffered. An event is consumed by the receiving component if the message matches one of the *input patterns* of the current control state, and is lost otherwise (non-blocking communication).

3.4. Extended semantics

Similar to other approaches from the hardware/software codesign domain, we extend our simple global-clock semantics by an globally asynchronous, locally synchronous interpretation of our models. Because we do not assume a global clock in our technical architecture, the global clock synchronization between components in the logical model has to be removed whereever a logical channel is mapped to a technical channel. In the system model, which is based on AutoFOCUS's basic semantics, the de-synchronization is achieved by inserting *buffer components* and global *scheduler components* for channels. The specified mapping can be checked for correctness by specifying `stereotypes` on channels in the logical model which reflect the desired communication properties of the channel. A stereotype $f_i \in \mathcal{F}$ is a tuple $(p_{i1}, p_{i2}, \ldots, p_{in})$ of *partial stereotypes* $p_{ij} \in \mathcal{P}_j$. The two sets of partial stereotypes currently implemented are {«`next`», «`finally`»}, meaning the message will be consumed at the next step, or at some future step, and {«`volatile`», «`safe`»}, meaning that delivery of the message is not guaranteed or guaranteed, respectively. We define a partial *implementation order* on P_j such that if $p_{1j} \prec_{P_j} p_{2j}$, then p_{1j} is said to *implement* p_{2j}. The implementation order on the stereotypes $\mathcal{F} = \mathcal{P}_1 \times \mathcal{P}_2 \times \cdots \times \mathcal{P}_n$ can now be derived as

$$f_1 \prec_F f_2 \text{ iff } \bigwedge_{j=1}^{n} p_{1j} \prec p_{2j}$$

For instance, if «`safe`» implements «`volatile`», then «`finally`»«`safe`» implements «`finally`»«`volatile`». We allow a channel from the logical model to be mapped to a channel in the system model precisely if the system channel implements the logical channel. Scheduler and buffer components may then be immediately derived from the system channel stereotype.

4. THE MINE PUMP EXAMPLE

4.1. Logical model

Fig. 4.1 shows the top-level structure view of the logical model for a benchmark example of a mine pump taken from [9]. The

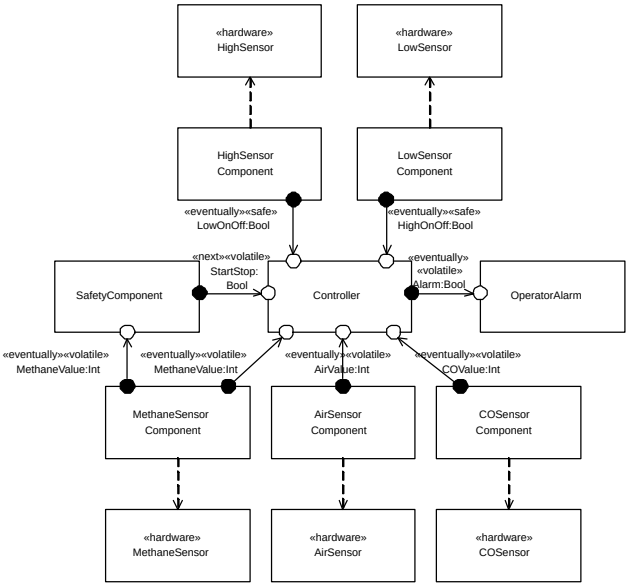

Figure 2: Mine pump — Logical model, SSD

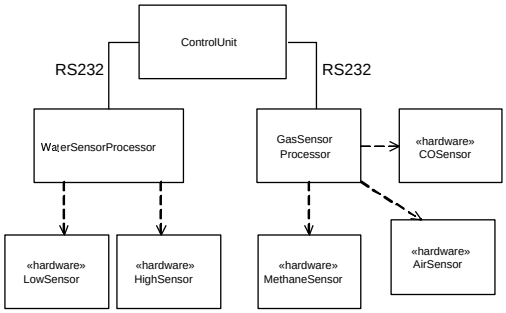

Figure 3: Mine pump — Technical model

mine pump is used to pump water out of the bottom of a mine. It is equipped with water level (high, low) and gas sensors. A pump controller switches the pump on when the water reaches the high water level and off when it goes below the low water level.

In the System Structure Diagram for the logical model, the components tagged as «`hardware`» are merely used as an annotation to define the context of adjacent components in the logical and nodes in the physical model. Hardware components are empty with no behavior or sub-structure. Hardware and software components are linked through *dependencies*. In the logical model, a dependency defines the set of local variables with read and/or write access by the hardware.

4.2. Technical model

A possible technical model for the example is shown in Fig. 4.2.

Technical models in AutoFOCUS consist of a network of *nodes*, *channels*, and hardware components. A node is a physical computation unit with dedicated I/O interfaces. Channels can be either directed (unidirectional communication) or undirected (bidirectional communication). Similar to the logical model, hardware components define the context of nodes in the technical model. Depen-

Description	Model	Variable	Value
t_{GP} invocation time	Map	$\tau_{inv(\mathrm{GP})}$	6ms
Execution time: Write to OS buffer	Techn.	$\tau_{ex(\mathrm{BW})}$	0.2ms
Transit delay, serial bus	Techn.	$\tau_{d(\mathrm{RS232})}$	2ms
t_{CU} invocation time	Map	$\tau_{inv(\mathrm{CU})}$	4ms

Table 1: Mine pump: Some real-time parameters

dencies in the technical model may include specific parameters such as memory addresses for data exchange or update frequencies.

Real-time parameters. Nodes are associated with parameter sets which describe typical delays for basic processor operations like arithmetic operations, get and fetch, and control flow operations, as well as some operating system specifics like scheduling policy, execution times for I/O operations, and multitasking primitives (for code generation). Each channel, in turn, is characterized by several real-time and quality of service parameters such as transit delay and data rate. Parameters sets may be average, worst-case, or both, depending on the analysis to be performed.

Execution model. Our current model for execution demands that all logical components running on a given node are compiled into a single periodic task; the period is specified with the mapping. Future extensions to our approach may involve the generation of multiple tasks for a given component.

4.3. Performance analysis

We illustrate how AutoFocus can support design decisions in early development phases with a simple example. One of the requirements for the mine pump may be stated as follows:

> *A high methane value reading by* `MethaneSensor` *shall cause an emergency shut down of the mine pump within 30 milliseconds.*

In order to evaluate the two deployed alternatives, we first take a look at the technical model shown in Fig. 4.2. The first alternative will map `MethaneSensorComponent` onto node `GasSensorProcessor` (task t_{GP}, period $T_{\mathrm{GP}} = 12$ms), and both `Controller` and `OperatorAlarm` onto `ControlUnit` (task t_{CU}, period $T_{\mathrm{CU}} = 8$ms). The second alternative maps both `MethaneSensorComponent` and `OperatorAlarm` to `GasSensorProcessor`, leaving `OperatorAlarm` to be implemented on `ControlUnit`. We assume the average values shown in 1 for the real-time parameters of the system. Note that events between two components running in the same task are buffered for one task period.

The overall propagation delay for the event "methane level too high" can now be estimated to be

$$\tau_{inv(\mathrm{GP})} + \tau_{ex(\mathrm{BW})} + \tau_{d(\mathrm{RS232})} + \tau_{inv(\mathrm{CU})} + T_{\mathrm{CU}} = 20.2\mathrm{ms}$$

for the first alternative. For the second alternative, the propagation delay is

$$\tau_{inv(\mathrm{GP})} + T_{\mathrm{GP}} + \tau_{ex(\mathrm{BW})} + \tau_{d(\mathrm{RS232})} + \tau_{inv(\mathrm{CU})} = 24.2\mathrm{ms}.$$

In a realistic setting, we may now prefer alternative 1 because it offers a higher safety margin.

5. DISCUSSION AND FUTURE WORK

Like other authors before, we recognize that there are numerous challenges to the introduction of models and high-level abstractions in real-time development [8]. First of all, abstraction may result in efficiency problems. Abstractions usually encourage loose coupling between system components, which in turn implies a higher communication overhead. The introduction of specialized execution models like Time-Triggered Architectures [10] may improve analyzability through improved functional and non-functional equivalence, but also results in performance overheads.

On the other hand, the larger and more software-heavy a system becomes, the more the use of model-based development is warranted. Through further exploitation of optimization capabilities, we aim to make model-based approaches competitive with traditional real-time software development.

In the future, we are planning to extend our AutoFOCUS framework with timing and size constraints in the form of annotations to model entities. Contraints could either be formally or informally validated with respect to a fixed logical/technical mapping, or used for automated partitioning leading to a synthesis approach.

Acknowledgements. Thanks to Jan Philipps for fruitful discussions on the use of AutoFOCUS for deployment.

6. REFERENCES

[1] R. K. Gupta, *Co-synthesis of hardware and software for digital embedded systems*, Kluwer, Boston, 1995.

[2] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch, "Giotto: A time-triggered language for embedded programming," in *Proceedings of EMSOFT 2001*. 2001, Springer-Verlag.

[3] A. Blotz, F. Huber, H. Lötzbeyer, A. Pretschner, O. Slotosch, and H.-P. Zängerl, "Model-based software engineering and ada: Synergy for the development of safety-critical systems," in *Proc. Ada Deutschland Tagung*, Jena, 2002.

[4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer, Boston, 1997.

[5] G. Berry, "The foundations of esterel," *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000.

[6] Raimund Kirner and Peter Puschner, "International workshop on WCET analysis - summary," Research Report 12/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[7] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch, "Tool supported specification and simulation of distributed systems," in *Proc. Intl. Symp. on Software Engineering for Parallel and Distributed Systems*, B. Krämer, N. Uchihira, P. Croll, and S. Russo, Eds. 1998, pp. 155–164, IEEE.

[8] B. Selic, G. Gullekson, and T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.

[9] Mathai Joseph, Ed., *Real-time Systems: Specification, Verification and Analysis*, Prentice Hall, 1996.

[10] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer, Boston, 1997.