Co-Design of a Fault-Tolerant Communication Protocol—A Case Study*

Reinhard von Hanxleden
Responsive Systems (F3S/R)
Daimler-Benz AG
vhanx@DBresearch-berlin.de

Luciano Lavagno
Dipartimento di Elettronica
Politecnico di Torino
lavagno@polito.it

Alberto Sangiovanni-Vincentelli
Dept. of EECS
University of California, Berkeley
alberto@eecs.berkeley.edu

Abstract

Hardware/software co-design is a design technology that supports the integrated development of hardware and software components of a system. A special focus of the work described here is the application domain of safety-critical embedded systems. In addition to the raised abstraction level and shortened design time offered by co-design relative to conventional design methods, we are therefore particularly interested in paradigms based on formal models.

This paper gives a brief overview of a co-design case study based on an application typical for safety-critical applications. We compared two ways of designing embedded systems: STATEMATE, a commercial system design tool based on Statecharts, and POLIS/Ptolemy, with ESTEREL as a specification language. We describe our design experiences and give preliminary experimental results.

1. Introduction

Hardware/software co-design is a design methodology—or rather, a collection thereof—that has become increasingly popular in the design of embedded systems. A particular class of embedded system applications are *safety-critical applications*, which are the focus of the work described in this paper. Some requirements of this class of applications are shared with other applications' requirements, such as short design times or low costs of the final product. Other requirements appear in slightly different guise; e.g., performance is critical, but typically only in that certain deadlines have to be met, beyond which there is no benefit from improving performance further. Requirements that are particularly important whenever safety is an issue are for example simplicity and formality—both critical not only for a proper design itself but also for certification purposes. Finally, there are requirements that are very specific to safety-critical applications, such as the ability to specify fault-containment regions.

As a first step in assessing the state of the field with respect to safety-critical applications, we are performing a case study based on an application typical for this field. In this paper, we present a high-level comparison of the design methodology followed using STATEMATE, a commercial system design tool based on Statecharts, and POLIS/Ptolemy, with ESTEREL as a specification language. We describe our design experiences and give preliminary experimental results.

The POLIS model of computation is based on Co-design Finite-State Machines, i.e. FSMs that communicate asynchronously. The global asynchrony is stressed to be able to mix hardware and software components since they have widely different reaction times and the communication among different tasks does take a finite amount of time. The STATEMATE semantic is based on synchronously communicating FSMs and hence it does not allow to mix hardware and software components. To compare the methodologies we had to restrict the capabilities of POLIS and to explore only two extremes of the hardware-software tradeoff: fully software implementation and fully hardware implementation. Nevertheless this experiment allowed us to draw some interesting conclusions about the specification methods used by the two systems as well as their relative power.

2. The application: PROSA

PROSA (Protocol for Safety-Critical Applications) is a reliable communication protocol for safety-critical distributed systems currently developed at Daimler-Benz. Such communication systems are the backbone of applications such as fly-by-wire, steer-by-wire, or brake-by-wire, and naturally not only their performance but also their robustness has to meet particularly high standards. We chose PROSA as the application driving the co-design case study, motivated by the overall relevance of the application for Daimler-Benz, the suitable size and complexity, and the lo-

*University of California at Berkeley Memorandum No. UCB/ERL M97/13, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720

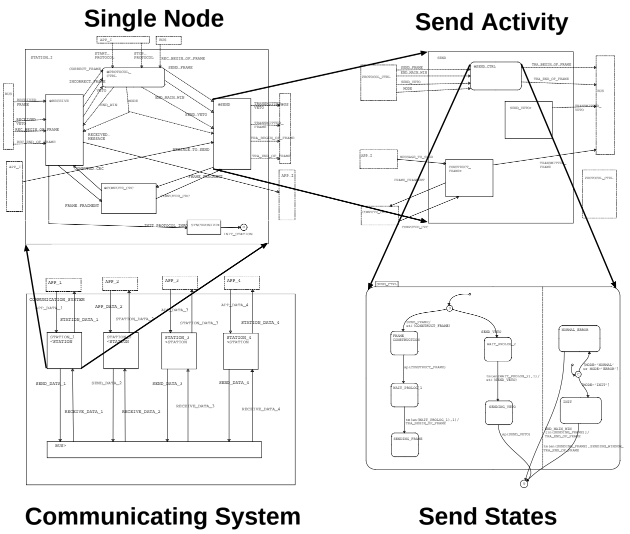

Figure 1. Traversing the hierarchy of the STATEMATE specification of PROSA.

cal availability of expertise. From a natural language description of the protocol we derived a simplified version of PROSA as a starting point for the case study.

3. The baseline: STATEMATE

STATEMATE, the “official” implementation of Statecharts [5], is a commercial tool that allows high-level system design and offers synthesis paths to both hardware and software. STATEMATE uses Statecharts to describe the behavior of a component within the system to be specified. In addition, Activitycharts describe which components a system consists of and what the signal and data paths are between the components. Finally, Module Charts describe the structure of the system (as opposed to the functionality—Activitycharts—and behavior—Statecharts).

Figure 1 shows a subset of the total of 15 State- and Activitycharts that describe PROSA. The abstract timing model uses time-outs as provided by STATEMATE. A watchdog simulates the bus traffic and randomly sends frames and veto signals.

3.1. Positive Aspects

Overall the implementation of STATEMATE appears to be quite complete and robust; even the novice can specify and simulate simple applications fairly quickly. During the simulation the current states and transitions are highlighted in the Statecharts with different colors. The animation of input and output signals and conditions with graphical “panels” is simple. One can check properties such as potential deadlock and reachability of certain states. (An example of this within a traffic signal application would be to check whether a state with the simultaneous sub-states “pedestrian signal green” and “car signal green” is reachable.)

3.2. Drawbacks

A weak point of STATEMATE was the archaic and not always consistent graphical user interface—this has somewhat improved with the Motif-based GUI of STATEMATE MAGNUM. Another problem is replicating and instantiating modules, which is difficult at best. It is possible to synthesize C or VHDL; however, due to the so far prevailing inefficiency of the resulting code, this feature seems better suited for just testing or prototyping, rather than actual development. Related issues are the openness and compatibility to other tools, which are limited. Other aspects of the tool that are of potential interest are the possibility to incorporate external C-code and the generation of detailed reports with the aid of a Document Generation Language.

A well-recognized problem that hampers not only efficient automatic code synthesis but also the general clarity of a design is the sheer richness of the Statecharts dialect implemented by STATEMATE. For example, the undisciplined use of event broadcast and Statechart-parallelism at lower levels can make it arbitrarily difficult to understand the flow of causality in a design. History connectors and real-time time-outs can introduce non-determinism that is not statically detectable.

Furthermore, STATEMATE is a powerful tool, but does not come with a methodology. As a consequence, “style guides” have been developed by the user community to guide the designers towards clear specifications—one such guide has also been developed at Daimler-Benz. It would be desirable to get tool-support for this as well, for example by automatically checking whether a design conforms to certain rules.

3.3. Summary of STATEMATE

STATEMATE can be a very useful aid in describing and specifying control-driven applications, if used in a disciplined manner. However, depending on the actual requirements on the end result, it may be better to view STATEMATE as just the beginning of an engineering chain, where for example a separate synthesis step turns the STATEMATE-specification into a quality prototype. STATEMATE is not a full co-design tool in that it does not allow a mixing of hardware and software components—it is either all C or all VHDL. However, it is an interesting candidate as a front-end embedded into a co-design environment.

4. The POLIS System

The POLIS co-design environment is targeted towards real-time control systems composed of software on a micro-controller and semi-custom hardware [3, 4]. The POLIS system translates a formal specification of a reactive real-time system written as an *asynchronous interconnection of synchronous modules*. Asynchronicity is necessary in order to accommodate heterogeneous implementations from a relatively "neutral" extended FSM specification.

Since the semantics of STATEMATE do not allow to mix hardware with software components, we used POLIS to synthesize two different versions from an ESTEREL specification of PROSA, a pure software version (C) and a pure hardware version (VHDL), and simulated both of them within the POLIS system. Due to the different targets, the semantics of the different versions vary slightly at the low level—which turned out to be rarely a problem in practice, but is still something that an engineer should be aware of when designing a system. As described above, the model of synchronicity postulates reactions to be instantaneous. This hypothesis is fairly expensive to implement in a homogeneous architecture (e.g., all in software or all in hardware), and almost impossible in a heterogeneous architecture. Moreover, it is valid only insofar as the embedded controller can react much faster than the minimum time constant in the environment to *all* inputs, while it may be often more efficient to check the speed of such reaction times only on an input-by-input basis.

In POLIS, hence, only single FSM transitions are considered atomic operations, while communication (including the manifestation of the results of a transition) takes an a priori non-deterministic time. This non-determinism can cause some problems, especially when a signal propagates along two distinct, reconvergent paths, and those paths can have widely different timings under different implementation choices. While it would theoretically be possible to detect when these delays can critically affect the overall behavior of the system, the POLIS design environment currently provides only means to *estimate and simulate* this timed behavior. This high-level timed co-simulation can be carried out once a set of choices about the implementation of each FSM, about the scheduling of software FSMs, and about the CPU type and speed has been made.

4.1. The ESTEREL specification language

A designer working with the POLIS co-design system has the choice between a graphical input language, which is basically a Finite State Machine (FSM) editor, and a text-based one, which is ESTEREL [1]. The graphical input language is similar to Statecharts but without the capabilities for hierarchy and parallelism; we therefore chose the ESTEREL front-end.

ESTEREL and STATEMATE use approximately the same underlying model of reactivity and synchronicity; this and the interest in a comparison of the languages suggested to attempt a direct mapping of the STATEMATE-specification of PROSA into an ESTEREL version. However, this approach turned out to be sensible only up to a certain point; there are some concepts in both languages that are unique to either and whose direct emulation in the other would result in a rather clumsy specification. Furthermore, finding a satisfying, natural decomposition and description of the requirements on the protocol engine turned out to be fairly demanding. However, this reiteration also gave a chance to simplify the existing specification. Overall, the process of understanding and summarizing the existing specification, decomposing it again, and filling in the modules one by one was still quite straightforward. Another nice facet of working with ESTEREL is the availability of an Emacs ESTEREL-mode, which highlights keywords and comments, indents properly, closes blocks, etc. A simulation script of about 1300 lines stepped the system successfully through various scenarios of stations initializing and integrating themselves with and without transmission errors and disconnecting themselves again; a fragment of the script is shown in Figure 2. After about two weeks the specification reached a steady state, consisting of 18 modules with about 1100 lines total.

At first glance, ESTEREL appears to be at a somewhat lower level and "pickier" than STATEMATE. Unlike STATEMATE, which accepts incomplete and even non-deterministic specifications and allows simulation and synthesis with only dynamic checks, an ESTEREL specification has—by default—to be statically proven to be sound ("causal") before it can be further processed. This approach has to be inherently conservative; that is, certain programs may be rejected even though they would never result in problems at run time.

There is also a temptation to write ESTEREL-programs with a sequential, data-driven, von-Neumann machine model in mind. It does require some re-thinking to produce an ESTEREL program that is truly in the spirit of synchronous languages—and does not look like a C-program. For example, states in ESTEREL are preferably represented as halting points in the program, rather than by explicit variables; this concept is not completely obvious, but quite intuitive once understood.

4.2. The POLIS-Ptolemy interaction

The Ptolemy system is a software environment for simulation and prototyping of heterogeneous systems [2, 6]. It uses object-oriented software technology to model each subsystem in a natural and efficient manner, and it has

```
MY_TICK ;
%emitted:   GATE_OPEN(true)
MY_TICK ;
%emitted:   IN_FRAME_WINDOW(true)
MY_TICK ;
%emitted:
%%
%% The beginning of a frame is encoded as "100".
%%
BUS_IN(99) ;
%emitted:
%%
%% Receive the beginning of a frame!
%%
BUS_IN(100) ;
%emitted:   RECV_BEGIN_FRAME_OK START_RECV_INIT \
  IN_ACK_WINDOW(false) IN_FRAME_WINDOW (true) \
  START_RECV_FRAME(false) START_WINDOW(false) \
  TRAP_BEGIN_FRAME
%%
%% The initialization information is "1" (ie, we
%% are listening to a signal from station 1).
%%
BUS_IN(1) ;
%emitted:   INIT_INFO_IN(1)
```

Figure 2. Fragment of an ESTEREL simulation script. Comments are prefixed with "%%", reactions of the system are prefixed with "%".

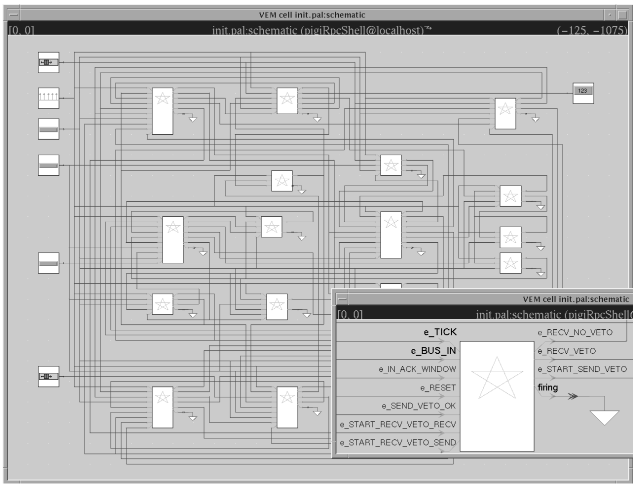

Figure 3. Ptolemy schematic of PROSA. The "stars" represent different modules, one of which (RECV_VETO) is enlarged on the lower right. There are additional inputs (on the left) and outputs (right).

mechanisms to integrate subsystems hierarchically. This versatility together with their nice graphical user interface and rich library of pre-defined components makes it a good candidate for a host system to POLIS. A collection of POLIS modules can be incorporated into a Ptolemy schematic, shown in Figure 3, that links the individual modules together. From this schematic Ptolemy generates the topology that POLIS in turn can use for synthesis.

5. Experimental results

5.1. Software synthesis—Accuracy of size and speed estimates

A critical component of a co-design environment is performance prediction, which can guide the developer in deriving a good partitioning between hardware and software. Furthermore, tight bounds on the worst case execution time are of particular interest to safety-crittical applications. To measure the prediction qualities of POLIS, we estimated both code size and speeds and compared these data with the actual results.

The total actual number of clock cycles on a DECstation 5000 executed by the scheduler, the I/O drivers and the user tasks was 1106325. The total actual number of clock cycles executed by the user tasks is 344185. The POLIS timing estimator reported an estimated 345614 clock cycles for the user tasks only. (The Real-Time Operating System synthesized by POLIS has not yet been characterized for the R3000, so no estimates for the other cycles are available.)

Table 1 compares the estimated and actual code speeds and sizes for each task. The speed comparison is based on the 510 simulation cycles that have also been used for functional validation. The estimates are from a dynamic simulation, the actual values are obtained with `pixie`. The estimation parameters were obtained without compiler optimizations (`-O0`), so the fairest comparison is between the columns labeled "Estimated avg." and "Actual average -O0". The estimator tries to take into account the cost of some RTOS operations, and that explains part of the discrepancy between the columns. A quantitative comparison (column "Δ") lists the ratio between the differences between actual and estimated timings and the actual timing. This gives an estimate of the error due to ignoring the effects of an aggressive optimizing compiler (the MIPS `cc`). Such effects are impossible to take into account by using the current estimation technique in POLIS, because it uses only local information. Note, however, that at least for modest effort (`-O0`), the optimizations decrease the *accuracy* of the estimator (summarized in the line labeled "Average estimation error"), but are not hampering *predictability* (given by the variance). In other words, the effect of optimization can be statistically estimated with good accuracy by uniformly dividing by 1.4 all the figures obtained without taking op-

Task	Number of calls	Code Speed (clock cycles)							Code Size (Bytes)				
		Estimated			Actual average				Estimated	Actual			
		min	max	avg.	-O0	Δ (%)	-O2	Δ (%)		-O0	Δ (%)	-O2	Δ (%)
CRC	18	44	90	80	50	60.0	46	73.9	295	304	-3.0	272	8.5
INIT	35	48	105	79	60	31.7	48	64.6	1056	1040	1.5	784	34.7
MODE	88	56	125	95	71	33.8	56	69.6	958	960	-0.2	736	30.2
NORMAL_ERROR	46	52	90	79	58	36.2	45	75.6	512	528	-3.0	352	45.5
RECV	128	46	97	81	64	26.6	48	68.8	1256	928	35.3	640	96.2
RECV_BEGIN_FRAME	513	39	117	70	52	34.6	41	70.7	628	608	3.3	432	45.4
RECV_FRAME	113	79	139	106	79	34.2	57	86.0	1256	1120	12.1	752	67.0
RECV_VETO	137	49	129	83	63	31.7	44	88.6	825	816	1.1	560	47.3
SEND	43	39	90	78	58	34.5	42	85.7	763	752	1.5	496	53.8
SEND_FRAME	634	71	146	86	69	24.6	60	43.3	1295	1232	5.1	800	61.9
SEND_VETO	620	55	106	90	70	28.6	45	100.0	666	656	1.5	400	66.5
SLICE	21	45	136	113	70	61.4	55	105.5	473	464	1.9	368	28.5
STATE_VECTOR	31	96	218	170	99	71.7	76	123.7	2841	2240	26.8	1712	65.9
STATION	21	54	75	74	50	48.0	42	76.2	208	256	-18.8	208	0.0
SYNCH	31	55	96	68	52	30.8	43	58.1	379	400	-5.2	320	18.4
TSC	486	95	160	143	95	50.5	70	104.3	2041	1664	22.7	1136	79.7
WINDOW	466	82	166	137	92	48.9	67	104.5	1901	1664	14.2	1104	72.2
Average estimation error						40.5		82.3			5.7		48.3
Variance						252.2		785.4			157.8		679.0

Table 1. Comparison of estimated and actual code speeds and sizes for each task.

timization into account. With full optimization (`-O2`), predictability suffers as well.

Table 1 also reports a comparison between estimated and actual code size (in bytes) for each task on the MIPS R3000. The estimation parameters were obtained without compiler optimizations (`-O0`), so the fairest comparison is between the first two columns, as for the code speed.

A first conclusion that can be drawn from these figures is that the overhead due to the synthesized RTOS is fairly high. This is mostly due to the fact that the RTOS is optimized for *size* rather than speed. It requires 19 Kbytes out of a total of 45952 Kbytes for the complete synthesized system on a MIPS R3000.

For the sake of comparison, the total code size (including a simple simulation interface) was 63.3Kbytes for POLIS and 86.2Kbytes for ESTEREL (version 5.0). The total execution times were 866K clock cycles and 2421K clock cycles, on a DECstation 5000 respectively.

5.2. Hardware synthesis

An all-hardware synthesis of PROSA resulted in 392 latches and about 88k literals for a plain RTL synthesis. An optimization step had little effect on the number of latches (391), but could significantly decrease the number of literals (6.6k). We are estimating that on an actual FPGA, this translates to about 1200 CLBs (without merging). We are currently in the process of validating these data experimentally.

6. Conclusion and outlook

We chose hardware/software co-design as a guiding principle for prototyping a controller implementing PROSA, a communication protocol specifically designed for safety-critical applications. We have completed the specification of a first version of PROSA in ESTEREL; the process of specifying and simulating the protocol with co-design tools has already been very valuable for understanding and refining it. We are now in the process of building a first physical prototype based on the Motorola 68332 in the context of a Brake-by-Wire demonstrator.

Overall it appears that the field of co-design has progressed fairly well in general, in particular as far as process-integration and automation are concerned; it is possible to start at a high level of specification and arrive at an efficient prototype design in a largely automated fashion. For safety-critical applications particularly beneficial are easy accessibility and understandability of the specification, offered by a tool such as STATEMATE, and the formality, simplicity, and efficiency provided by ESTEREL/POLIS. What appears to be lacking so far, however, is a combination of these properties. Other aspects of a co-design environment that would be desirable from a safety-critical application's point of view are for example the ability to specify fault-containment regions, within for example a hardware partition, or the ability to annotate the specification with non-functional requirements, such as a desired degree of reliability.

Acknowledgments

Jürgen Bohne has made the PROSA-application available and has assisted in translating the natural-language specification into a formal model. Slavomir Kupczyk and Carsten Sühl have written the STATEMATE specification of PROSA. Ralph Sasse has been providing his expertise and help with regard to the prototyping hardware and the Brake-by-Wire demonstrator. Throughout the project, members of the POLIS and the ESTEREL groups have been very helpful, in particular Gerard Berry, Xavier Fornari, Harry Hsieh, Attila Jurecska, Claudio Passerone, and Ellen Sentovich.

References

[1] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, pages 1293–1304, Sept. 1991.

[2] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[3] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli. A case study in computer-aided codesign of embedded controllers. *Design Automation for Embedded Systems*, Nov. 1995.

[4] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, Aug. 1994.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[6] A. Kalavade and E. A. Lee. Hardware/software co-design using Ptolemy—A case study. In *Proceedings of the International Workshop on Hardware-Software Codesign*, Sept. 1992.