PROFILING SPECIFICATION PEARL DESIGNS

"Try before build"

Roman Gumzej[1], **Matjaž Colnarič**[1] and Wolfgang A. Halang[2]
[1]University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova 17, SI-2000 Maribor, Slovenia, Email: roman.gumzej@uni-mb.si;
[2]FernUniversität in Hagen, Faculty of Electrical and Computer Engineering, 58084 Hagen, Germany, Email: wolfgang. halang @ fernuni-hagen. de

Abstract: An approach to holistic system modelling is presented, based on the Specification PEARL hardware/software co-design methodology, having its origin in the standard Multiprocessor PEARL specification language. Specification PEARL specifications and models represent prototypes of systems and programs, based on the PEARL program model. The system models built are checked for timely execution by co-simulation. The resulting information shall be used for fine-tuning the designs. Through the analysis of simulation traces a profiling process takes place. An integral CASE tool facilitating this holistic approach has also been devised.

After the profiling process has produced a feasible system model, the program prototypes may be enhanced to their full functionality. As long as their timing properties are not changed by this, utilising this methodology should minimise the possibility of implementing an infeasible system.

Key words: Real-time systems, co-design, co-simulation, verification, PEARL.

1. INTRODUCTION

With the ever increasing complexity of embedded control systems, the traditional development process of manual coding followed by extensive and lengthy testing is becoming inadequate. The main design concern, which first moved from low to high level program languages, recently moved to a higher abstraction level, which relies on automatic or semi-automatic code generators to produce code in traditional programming languages. Examples of these include the Unified Modelling Language (UML) [17], Model-Driven Architecture (MDA) [16] and Model-Integrated Computing (MIC) [14]. For real-time systems, timeliness and safety issues are just as important as functional correctness. Hence, to avoid exhaustive testing, they should be designed holistically, taking all their temporal and functional properties into consideration as early as possible, with their subsequent verification in mind.

To enable verification, often formal languages and/or mathematical notations are used, for which subsequently a proof can be worked out (e.g., formalisms supported by differential equations, which describe a system's operation in time and space [4], formal languages and timed automata [1], combinations of conventional CASE methods and state charts [15], graphical techniques with the expressive power of their formal language counterparts [3]). While enabling formal verification, most of these methods lack the versatility of basic constructs and user friendliness. Therefore, graphical formalisms with a greater set of basic constructs have been defined (e.g., CSR/CCSR [7], TTM/RTTL [9]), while keeping enough "strictness" to enable verification. Dedicated state transition automata like CRSM [9] are often used as basic internal computation model (e.g., POLIS [2]).

A wide variety of different verification methods has been combined with VHDL based design tools, ranging from formal methods to simulation with fault insertion and combinations thereof (e.g., [6]).

For pragmatic reasons, simulation is often used to check the correctness of a system designed or parts thereof. Hence, verifying systems with time limitations led to the introduction of real-time scheduling algorithms into their co-design and simulation (e.g., [8]). This approach is also used to check our designs for timeliness. The design and profiling processes are described in the forthcoming sections followed by a concluding summary.

2. SPECIFICATION PEARL METHODOLOGY

Our co-design method [5] is based on the notation of the standardised Multiprocessor PEARL [12] specification language. It enables the construction of a conceptual system model, whereby its hardware and software architectures may be designed in parallel. The system model is built as a result of running the associated CAD tool. For each hardware and software component its timing information is specified for the later timeliness checking by co-simulation.

The main motives for using the mentioned method with the mentioned profile are:

1. the concepts of the modelling (specification) language and of the PEARL real-time programming language are syntactically and

conceptually closely related,

2. the ability to use software specifications as program prototypes as well as to extend them to fully functional programs in a straightforward manner,
3. the ability for early reasoning on system integration,
4. the ability to apply timing parameters to hardware and software constructs, and
5. the ability to simulate a modelled solution and to check its feasibility before implementation.

3. VERIFICATION METHOD

Our verification method is based on co-simulation with earliest-deadline-first (EDF) scheduling and time boundaries [5]. It is primarily meant to profile the timing properties of designs in order to make them feasible. A design is transformed into an internal representation for simulation, whose primary result is a successful execution or a failure, whereas the secondary result is an execution trace, from which additional profiling information is extracted. This is used to discover bottlenecks and unreachable states, as well as to fine-tune the resource parameters and to balance the load on the designed prototypes.

For successful verification, it is assumed that the designed system model is consistent. Intermediate checks on the following points may be performed during the design of the system architecture, and a final check has to be performed prior to verification to ensure this:

1. completeness check (all components are present and fully described),
2. range and compatibility check (parameter compatibility), and
3. software to hardware mapping check (complete coverage and consideration of resource limitations).

In the forthcoming sections the structure of system models is described, followed by the explanation of the co-simulation (verification) method.

3.1 System Model

The *hardware model* is represented by STATIONs, being the processing nodes of a system. Their properties are determined by their components (e.g., processors, memories, interfaces). There are four different types of processing nodes in a system architecture: BASIC (program and operating system), TASK (program), KERNEL (operating system) and COMPOSITE (multi-station node). A processing node may have one or more communication lines attached to it, each one connecting it to another node.

The components of the *software model* are COLLECTIONs of tasks, which are mapped to the stations of the *hardware model.* They are composed of sub-layers of nodes representing program tasks. The tasks themselves are represented by Timed State Transition Diagrams (TSTD), cp. [8]. For inter-task co-operation, collections communicate via PORTs, which represent references to "physical" communication lines between stations of the hardware model.

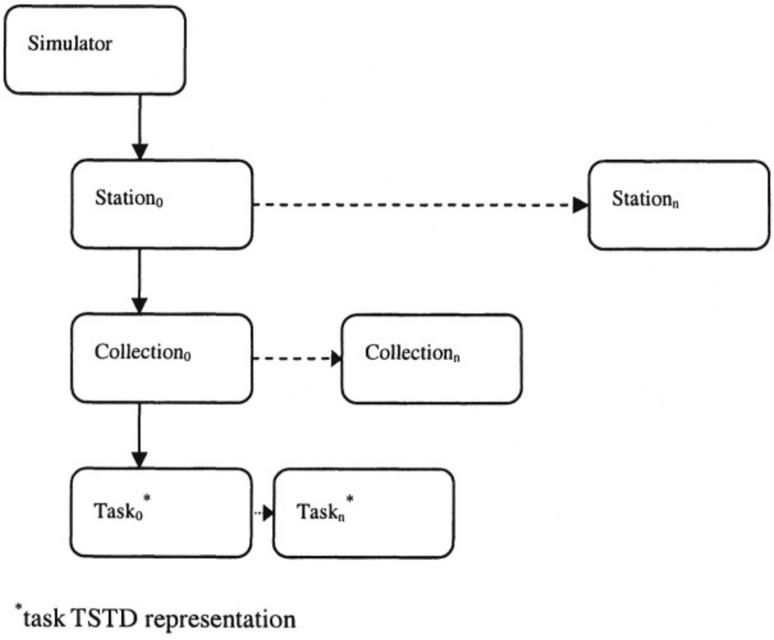

Figure 1: Structure of simulation units

While being designed on separate layers, the mapping of collections to stations is made explicit for co-simulation. The structure of the simulation units is shown in Fig. 1.

The structure of a task is represented by the start/end and action states of the TSTDs (see Fig. 3):

- start states (representing task trigger conditions),
- working states (having continuation pre-conditions, timeout condition, on-timeout action, and actions to be performed within the current state),
- super-states (representing a working state's decomposition into a sub-diagram), and
- final states (representing finalisation actions).

Since the state transition conditions relate to the (operating) system's internal states, they are formed around system calls, changing these states. Naturally, they have to outline the task's structure together with its control structures and conditions.

For co-simulation, the execution times of individual states need to be estimated. A task's execution time is calculated based on the longest path in the corresponding TSTD.

The *hardware* and *software models* are glued together by a *Configuration Manager* (CM) including a real-time operating system (RTOS), which supports the tasking model and system services of the PEARL language [10, 11] chosen as most suitable for our tasks.

Functionally, the CM module has the same role in co-simulation (Fig. 3) as in execution on the target platform. The main difference lies in the global real-time clock, which is maintained by the simulation environment, and the context switches, which are performed virtually in the case of simulation (the context refers to task states - not processor registers). Pre-emption points remain the same in both cases - the atomic execution of task states is maintained in both cases.

The CM also represents the hardware abstraction layer for the executing application. The hardware abstraction layer, as configured by the hardware architecture model, is mainly used to define the properties and interfaces of stations. It is the only visible hardware simulation unit, and is also considered as a whole, with the properties specified in the target platform implementation. The resource access functions and interface device drivers of stations perform virtual functions in case of simulation, and concrete functions in case of target platform implementations.

Our RTOS supports earliest-deadline-first scheduling (later enhancements for other strategies are foreseen). Its resources are parameterised (e.g., number of tasks, synchronisers, signals, events, queued events) by setting the parameters of the KERNEL station or BASIC station, which are RTOS host nodes for any TASK station nodes.

Each RTOS processing node maintains a real-time clock. In a simulation environment, all these clocks need to be synchronised with the global simulation time.

3.2 Verification Presumptions and Criteria Function

Verification is based on the following presumptions:

1. there is only one global simulation clock in the system and all real-time clocks (timers) relate to it;
2. the time events relate to the corresponding station's real-time clock;
3. tasks are assigned deadlines for their execution (the only exception are short initialisation tasks);

4. task states (TSTDs) have a time frame for the activity being performed within the state (in real-time clock time units).

The time required to execute the operating system itself (schedule and dispatch cycle) is assumed constant. This time is considered to be a part of the system call service time and is, therefore, not modelled separately. The time needed to service a system call is considered to be included in the time frame of the calling task's state. Its sole function is to change the system state and to trigger task states, whose trigger conditions relate to the internal data structures of the (operating) system.

Every verification method requires the definition of a *criteria function,* which tells when a system fails, i.e., what the limits of the "normal" execution of the system being checked are.

The *concept of correctness* had been defined as follows: "The system fails if it holds true during co-simulation: the system reaches an undefined state, or its pre-defined time frame is violated and no timeout-action is defined."

By trying the shortest and taking the longest transition times through the task states (TSTD) of the system, it is assumed that enough of the time domain can be covered to be able to generalise the results to an arbitrary transition time (within these time limits) of every state and herewith also of the system as a whole.

3.3 Co-simulation with EDF Scheduling

The station clock rate is translated into the step size of the station in the simulation, and is used when the next event time is being calculated. For feasibility profiling, next critical event simulation and (EDF) scheduling are used. The time instant of the next critical moment is always determined by the simulation unit whose activation time is the closest. This time is forwarded to all its parent units and, finally, it becomes the next global critical moment.

On each step it is checked, whether timing or synchronisation errors have occurred. A "timeout-action" (performed upon violation of the state's timeout condition) represents a controlled program fault. Herewith, a transition is performed into a final state, from which there is no further transition. If, upon the same error, this action is not defined for the current state, the system fails and the error is logged. Otherwise, the transition into the next state is always tried in the minimum and performed in the maximum time variant, if the pre-conditions for the transition are fulfilled. The transitions through the task states executing at their stations are performed first for all nodes, which share the current critical moment, and the previous state is remembered on every transition. The execution protocol is logged instantly for all simulation nodes.

3.3.1 EDF Next Event Scheduling

Earliest-deadline-first next event scheduling is based on the following timing information (see Fig. 2):

A: task activation time,

R: accumulated task run time (updated with the next critical event),

E: task end time (the time when the normal task end is expected based on its maximum run time; upon a context switch the current time t1 needs to be remembered, because this parameter needs to be reset based on the current time t2 and the formula E'=E+(t2-t1) when the task is re-run), and

D: task deadline (set, when A is known).

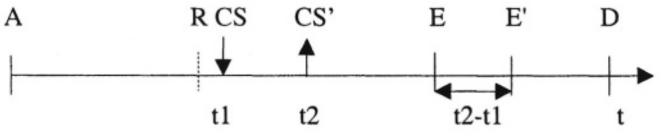

Figure 2: Task run with a single context switch

Re-scheduling takes place when a task is activated due to a scheduled event or on request. The task with the earliest deadline is chosen for execution, and its current state is assigned the current time t as its next critical moment.

While re-scheduling, the following criteria (failure conditions) need to be checked for all tasks:

$t < Z=D-(E-(A+R))$, where Z represents the latest time when the task needs to start/continue in order to meet its deadline; and

$t < E \leq D$ must be true for all active tasks, since they would have missed their deadlines, otherwise.

Tasks can be scheduled to be executed on events. For simulation purposes, they are assigned occurrence times regardless if they represent timers or external interrupts. They represent a special simulation unit, which takes its next critical moments' data from an occurrence table. When these events occur, they are fed into the stations' CM interfaces, and appropriate tasks are woken up through the RTOS.

3.4 Course of Simulation

During co-simulation (Fig. 3), the time of progression to the next state is calculated in two variants for each state:

1. RTC + minT (to check the pre-conditions), and
2. RTC + maxT (for transition to a new state).

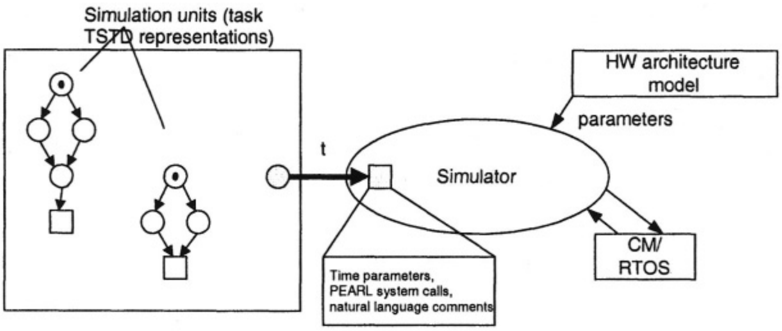
Simulation units (task TSTD representations)
t
Simulator
HW architecture model
parameters
CM/ RTOS
Time parameters, PEARL system calls, natural language comments

Figure 3: The course of simulation

If in critical moment (2) the pre-condition for the transition to any further state is not fulfilled, the on-timeout action is executed. If it is not defined, the system fails.

During simulation, the E and D parameters are set for each task when it is activated (the A parameter is set). When a critical moment is reached, it is checked if herewith the time frame given for the task has been violated, which results in the following consequences: (1) subtraction of the overhead from the task's slack time, or (2) the system fails as the task deadline is missed.

The simulation results are logged during the execution of every simulation unit, and each step is accounted for also within all its parent simulation units.

This means that every task state logs its actions into the task log, whereas a task logs its beginning and end into the module/collection log. The collection logs the time when it was first allocated at the station, possible subsequent re-loads, and the changes of states which triggered them into the station log. The stations also log the times when they were communicating among each other.

3.5 Interpretation of Results

The simulation logs are checked manually for irregularities, which could represent faults in the original design, or timing/synchronisation errors that might have occurred during the virtual "execution" of the system model.

Busy and idle times are considered for each station and, if necessary and possible, load balancing actions are taken.

The process of analysing and fine-tuning, also known as profiling process, cannot be unified due to the great diversity of possible designs. For this reason, it must be carried out manually and remains the responsibility of the designer.

4. CONCLUSION

Some "design for verification" methodologies and formalisms, used to design and verify real-time systems, have been mentioned. In particular, the part to verify temporal feasibility of the Specification PEARL co-design methodology for real-time systems has been presented. Our goal is to be able to determine a priori the feasibility of a program part on a specified hardware architecture without the need to implement it first.

A feasible design model, which is produced by the presented modelling and profiling process, retains its value if the foreseen execution time frames were chosen correctly (i.e., the circumstances on the time scale shall not change when the program part is extended to a fully functional program and run on the target platform).

ACKNOWLEDGEMENTS

This article presents results of the research project "Holistic Embedded Control Systems Design" (Z2-3493), financed by the Slovenian Ministry of Education, Science and Sport.

REFERENCES

1 G. Agha. The Structure and Semantics of Actor Languages. J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, Foundations of Object-Oriented Languages, pp. 1-59, Springer-Verlag, 1991.

2 F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki and B. Tabarra. Hardware-Software Co-Design of Embedded Systems: The POLIS Approach. Kluwer Academic Publishers, 1997.

3 C. Dietz. Action Diagrams. Proc. 22nd IFAC/IFIP Workshop on Real-Time Programming, Lyon, 1997.

4 T. J. Eriksen, S. T. Heilmann, M. Holdgaard and A.P. Ravn. Hybrid Systems: A Real-Time Interface to Control Engineering. Proc. 8th Euromicro Workshop on Real-Time Systems, pp. 114-120, 1996.

5 R. Gumzej. Embedded System Architecture Co-Design and its Validation, Doctoral thesis, University of Maribor, Slovenia, 1999.

6 M. Khalil, Y. Le Traon and C. Robach. Control-flow System Diagnosis: An Evolutive Method. Proc. 24th Euromicro Conf., Västerås, 1998.

7 I. Lee, S. Davidson, and R. Gerber. Communicating Shared Resources: A Paradigm for Integrating Real-Time Specification and Implementation. Foundations of Real-Time Computing: Formal Specifications and Methods. Kluwer Academic Publishers, 1991.

8 V. J. Mooney III. Hardware/Software Co-Design of Run-Time Sytems. PhD thesis.

9 J. S. Ostroff. A Visual Toolset for the Design of Real-Time Discrete Event Systems. IEEE Trans. Control Systems Technology, May 1997.

10 Basic PEARL, DIN 66253, Part 1.

11 Full PEARL, DIN 66253, Part 2.

12 Multiprocessor PEARL, DIN 66253, Part 3.

13 A.C.Shaw. Communicating Real-Time State Machines. IEEE Trans. Software Engineering, Vol. 18, No. 9, pp. 805-816.

14 Janos Stipanovits and Gabor Karsai. Model-integrated computing. IEEE Computer 30(4): 110-111, April 1997.

15 I. Traore and Abd-el-Kader Sahraoui. A Multiformalism Specification Framework with Statecharts and VDM. Proc. 22nd IFAC/IFIP Workshop on Real-Time Programming , Lyon, 1997.

16 OMG MDA website. www.omg.org/mda.

17 OMG UML website. www.omg.org/uml.