Refinement of Model Specifications in Embedded Systems Design

S. Schulz
Mobile Networks Laboratory
Nokia Research Center, P.O. Box 407
FIN-00045 NOKIA GROUP
Stephan.Schulz@nokia.com

J.W. Rozenblit
Dept. of Electrical and Computer Engineering
The University of Arizona
Tucson, Arizona 85721-0104, USA
jr@ece.arizona.edu

Abstract

Most current codesign approaches leverage from a complete specification of an application design at the implementation level. We pursue here an implementation independent system level design specification for real-time embedded systems using modular executable discrete event models. This paper introduces a formal abstraction for the specification of such design models. In addition, it defines a set of refinement steps which may be used to refine abstract design models into implementation level design specifications. Our approach is illustrated using a small embedded systems application.

1. Introduction

Many codesign approaches for embedded systems [1,2,3,4] focus on an integration of hardware and software components in the specification of design implementations. Given further advances in implementation technologies, increases in system complexity, and the demand for high performance embedded systems applications, the need for implementation independent, system level design approaches becomes apparent [5,6,7]. In [8,9,13], we have proposed one such design approach for embedded systems called model-based codesign which is heavily based on systems modeling concepts [11].

Executable system level modeling specifications have established themselves in industry over the past years as a tool for the conceptualization and analysis of system specifications for embedded systems applications. Although some system modeling tools offer the generation of software implementations [10], application models are rarely used in the actual implementation of embedded systems. One of the reasons for this reluctant acceptance are the processing requirements for the generated code, which often exceeds resources of available processing platforms for embedded systems applications. We believe that a well structured approach to a construction and transformation of design models into prototype implementations can result in an efficient design implementation. In this article, we propose a formal abstraction of computation which is used to specify abstract, system level design models for complex embedded systems applications. Furthermore we discuss how structure and behavior of these models can then be properly refined down to an implementation level design specification.

2. Modeling Embedded Computing Systems Behavior

Computation, which is performed by any kind of digital computing system, can be fundamentally decomposed into two phases: a computation on some specific data, and prior to that, waiting for these data to arrive. Similarly, we can view a more complex computation as a collection of either sequentially or concurrently executing computational segments (or a mixture of both), and their waiting for input data.

In this model of computing systems behavior, any computation can be eventually decomposed into such basic computational segments which wait for input data, perform some computation in some amount of time on it, and then generate outputs for other segments.

2.1 Basic Model Component Specification

We can encode this elementary behavior into a modular, deterministic event-based modeling specification with explicit timing as shown in Figure 1. Here, the model states S = {W, P} represent "waiting" and "processing" phases of computation. P is associated with the actual computational segment or some abstract specification of it. The initial state of this model specification is W. The maximum time spent in W can be infinite (the data never arrives), and is in P equivalent to the execution time for the computational segment once its required data has arrived.

The set of external input events X_W represents either data observed from the environment or sent from other model components, which is required by the computational segment encoded in state P. These input

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE

data are processed in state P within some required execution time which can be derived from application performance constraints. Data, which is to influence the environment or other model components, is generated after the completion of the computation in a set of external output events Y_P. Internal model state transitions lead from state W to P, and vice versa.

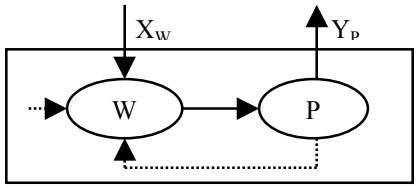

Figure 1. Basic Model Component Specification

From a general systems modeling perspective [11], the behavior defined by this basic model component can be described as follows: The basic model component stays in the waiting state W until all external input data for the computational segment, which are associated with P, have arrived. At this point an internal event is generated which changes the current model state to P where the data is processed. After some specified execution time the computation completes and an internal event as well as external output events are generated instantaneously. The model component then transitions back to the same waiting state.

More complex computations can be modeled is a variety of ways: by creating basic model components for each computational segment, by extending the set of basic model component states by other pairs of waiting and processing states, or most likely by using a mixture of the two approaches [13].

2.2 Derivation of Execution Times

Timing constraints are of primary importance in real-time embedded systems design. We can validate these constraints by executing our design model specification over time and to monitor the events involved in timing constraints by using simulation. In the previous definition of our basic model component, we have been assuming that the execution times of all computational segment are known for the design model. In practice, however, timing information is generally only available at best as required application response times.

In order to derive execution times for computational segments, we may use simulation results to distribute global performance constraints onto the computational segments of a model component specification. Given a application specification with unknown execution times but with a timing constraint, we can first run a controlled simulation where we specify all processing state execution times in our design model to be some common value, e.g., ”1”. The purpose of this latter value is merely to establish an ordering in the processing of computational segments. By monitoring the processing states of design model component during a simulation run a design model state execution trace can be recorded for each simulation experiment. This trace allows us to identify executed computational segments as well as their execution frequency over the given period of time. This information alone, however, does not suffice to derive execution times of individual segments. So far the most common approach [3,5] in codesign has been to assign execution times based on "educated guesses".

We have approached this issue by investigating a common property which affects the execution time of all computational segments, i.e., computational complexity. We selected and adapted the well known formula for the unbiased computation of CPU performance by Patterson and Hennessy [12] in order to estimate unknown execution times as shown in equation (1).

$$\textit{Segment Execution Time} = \textit{Estimated Model IC} \bullet \textit{Model Component Cycle Time} \quad (1)$$

This formula uses the model instruction count (IC) as a measure for the computational complexity of a segment, assumes a cycle per instruction (CPI) count of one, and introduces the new concept of a model component cycle time. The model component cycle time will be considered a model component property and is computed from the evaluation of a model state execution trace, i.e., the execution times of all computational segments, which are part of a model component specification, are always computed from its model component cycle time.

2.3 Revisiting the Basic Modeling Specification

In our abstraction of computation, we may interpret internal events or internal model state transitions as means to express data dependencies between computational segments or represent changes in the internal model component data structures. Therefore, we define an internal event to be a pair of internal inputs I_s and outputs O_s (for some model state $s \in S$) which are "propagated" in internal model state transitions [13]. This new interpretation expands our previous definition of general basic model component specification as shown in Figure 2.

By definition, the waiting model state W is either the initial model state or entered via some internal transition, i.e., from another previous processing model state pP. The purpose of W is to collect external inputs $X_W \subseteq X$ which are required by the computational segment associated with processing model state P (where X is the set of all accepted inputs by the model component). Entering W also implies access to the part of the component data

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE

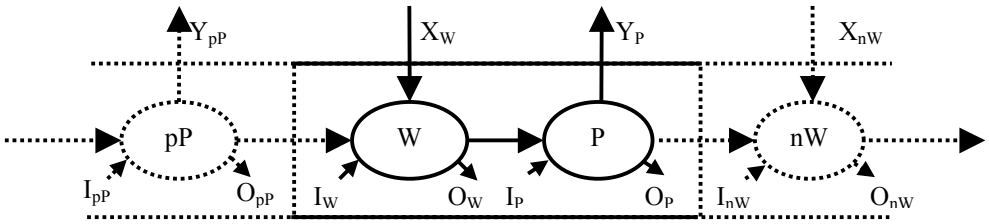

Figure 2. Revised Diagram of a Basic Model Component Specification

structure which will store the values of arriving input events. Current values of these variables are represented by the set of internal inputs $I_W \subseteq V$ (where V represents the values of all shared component data structures). This set of values is then overwritten in W by the values specified by X_W. The time spent in W is likely to be a positive rational number but may be infinite. After the required set of external inputs X_W arrives (for full discussion of multiple event arrivals we refer the reader to [13]), an internal transition to P triggers the generation of internal outputs O_W (which is basically an update of the previously accessed data structure values I_W), so that new $V = V' = (V - I_W) \cup O_W$.

The execution time of P is computed as previously discussed from instruction count of the its associated computational segment and the cycle time of the model component specification. Its set of internal inputs I_p consists of some internal outputs of W and possibly other internal data, i.e., $I_P \subseteq V'$ and $O_W \subseteq I_P$. Finally, P also generates in its internal transition to a next waiting model state nW a set of external outputs Y_P as well as a set of internal outputs O_P, so that afterwards the new $V = V'' = (V' - (I_p \cap O_p)) \cup O_P$ since $O_P \subseteq I_P$. Modifications of the component data structure may then enter the next basic model behavior again as internal inputs, i.e., $I_{nW} \subseteq V''$.

3. Design Model Refinement

The objective of design model development is to generate a detailed, valid design specification from abstract application models. In iterative design model refinement cycles, we gradually increase fidelity of the structural and behavioral aspects of a design model specification based on the introduction of application requirements and information obtained in an extensive design model analysis [13]. The end result of design model refinement is a design model prototype which incorporates all requirements stated in its system specification where each component is specified at the appropriate level of resolution.

3.1 Correct Refinement

Both, structural and behavioral refinement, require a proper adaptation of the behavioral aspect in the generated model component specifications in order to constitute a correct refinement. A correct refinement is achieved when the external interface (especially its timely behavior) as well as former internal data dependencies of the original model component specification are preserved.

A refinement of a basic model specification leads either to the decomposition of the original processing state P into two processing states P1 and P2 (e.g., in the increase of model resolution) or the addition of a new processing state where P1 = P and P2 = new P (e.g., when extending behavior). Waiting states for each processing state are generated in the course of this decomposition. Each set of external inputs can be specified as $X_{W1} \subseteq X_W$, $X_{W2} \subseteq X_W$ and each set of external outputs as $Y_{P1} \subseteq Y_p$, $Y_{P2} \subseteq Y_P$, or $X_{W1} = X_W$, $X_{W2} = X_{new\ W}$, $Y_{P1} = Y_P$, and $Y_{P2} = Y_{new\ P}$, respectively.

3.2 Types of Model Component Specifications

Based on the temporal relationship of the external event arrival for the two derived processing states in their original model specification we can distinguish between three different scenarios prior to a refinement:

a) Type I: Both external events arrive at the same point in time
b) Type II: The last event of the set of external inputs for W1 arrives prior to the one for W2
c) Type III: The last event of the set of external inputs for W1 arrives after to the one for W2

Any refinement of a model specification is accompanied by a revision of computational segments and instruction counts. These modifications effect the execution times for P1 and P2. In addition, the external outputs originally produced by P may now be generated at an earlier point in time i.e., either by P1 or P2. Since we are interested in the preservation of the external interface in a refinement step these issues need to be properly addressed.

In order to simplify the following discussion we will use "ta(s)" to denote the execution time of the processing model state *s*, "t_s" as the point in time that the model state *s* is entered, and "et(s, x)" as the total elapsed time in a model state *s* since the arrival of the last external input *x* of the set X_s.

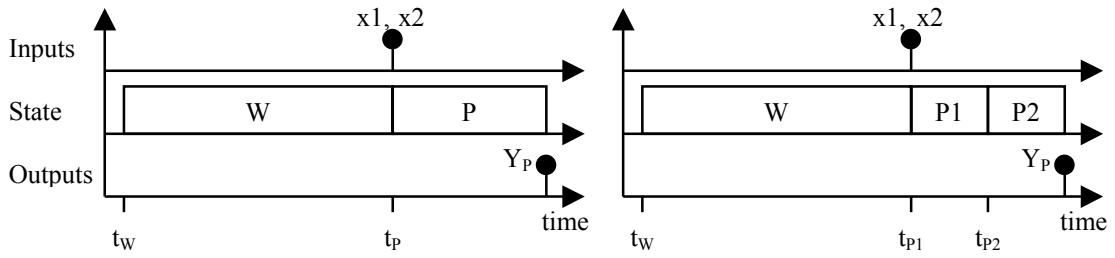

Figure 3. Original Behavior and Result from Behavioral Refinement in Type I Scenario

3.2.1 **Type I Scenarios**

A Type I scenario is depicted in Figure 3. Here, the temporal relationship between the external inputs which are to be received in waiting states W1 and W2 of the refined model specification can been observed in the original model specification as et(W, x1) = et(W, x2), i.e., both of these external events arrive at time t_P.

In a pure behavioral refinement, the same waiting state W is used to receive both sets of external input events instead of specifying two separate waiting W1 and W2 states as shown in the second part of Figure 3. This is due to our single state condition which requires a model component to be only in one model state at a time. The ordering of the processing states is either based on the data dependency between their computational segments or can be arbitrary if there is none (i.e., $O_{P1} \cap I_{P2} = \{\}$). In the latter case an ordering may be established based on the generation of the former external outputs Y_P.

A correct behavioral refinement is achieved when the execution times of the processing P1 and P2 are at most equal the original execution time of P, i.e., ta(P1) + ta(P2) ≤ ta(P). If the computed execution time of both processing states exceed the original, i.e., ta(P1) + ta(P2) > ta(P) cycle time of the model component specification has to be decreased so that ta(P1) + ta(P2) = ta(P). Notice that such a change affects the execution times of all other computational segments within that model component specification.

If ta(P1) + ta(P2) < ta(P) or one of the processing states produce external outputs part of the original set (i.e.,; $Y_{P1} \subseteq Y_P$ or $Y_{P2} \subseteq Y_P$ given that $Y_P \neq \{\}$) we introduce special kind of model component for each such set of external outputs, called communication component [13], which delays the propagation of these events by a fixed amount of time ta(Pd). These delay parameters are computed or updated using the execution times of the original processing state P and a new state P1 or P2, e.g., ta(P1d) = ta(Pd) + (ta(P)-ta(P1)). Notice that delays of communication components are also affected by changes in the cycle time of the model component that they are attached to.

In a structural refinement, a new basic model component specification B is created with an identical model cycle time as in the original model component specification. Waiting model states are specified for each component, i.e., W1 and W2. The required adaptation of each component behavior as depicted in Figure 4. Here, the original model component A exhibits the behavior specified by W1 and P1, and component B the behavior specified by W2 and P2. Figure 4 illustrates that two cases may arise from a possible data dependency of P2 on P1, i.e., $O_{P1} \cap I_{P2} \neq \{\}$ or $O_{P1} \cap I_{P2} = \{\}$, respectively.

In a first case, a structural decomposition does not seem advisable since the behavior does not exhibit any concurrent concurrency. In the second case, however, such a decision is advisable if there is no or a low degree of data dependency of P2 on data structures which are modified by previous processing states in the original model component A, i.e., the set of internal inputs $I_{P2}^* = O_{pP} \cap I_{P2}$ should be empty or small. I_{P2}^* should be a small set since its values have to be communicated properly explicitly between the model components A and B.

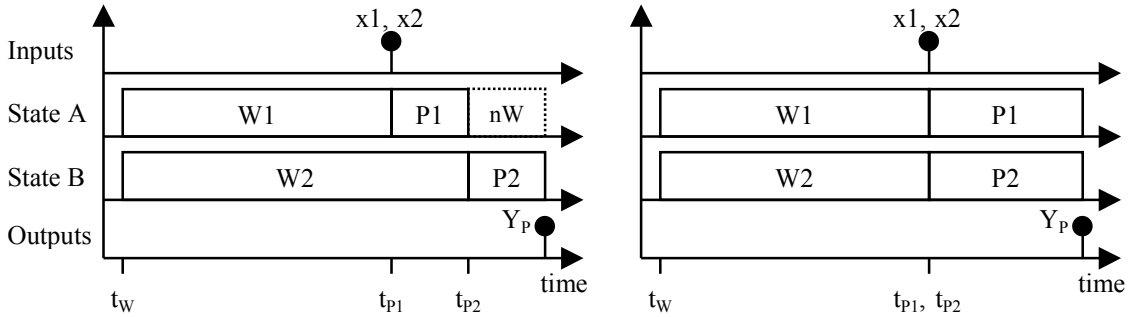

Figure 4. Results from Structural Refinement based on data dependency in Type I Scenario

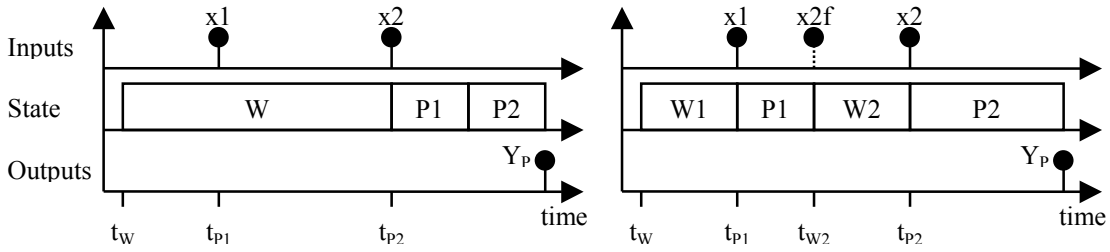

Figure 5. General and Special Case of Type II Scenario Results after a Behavioral Refinement

In the latter case the execution times for each processing state should be at most equal to the original processing time, i.e., ta(P1) ≤ ta(P) and ta(P2) ≤ ta(P). The same techniques are applied as in a correct behavioral refinement in Type I scenarios to adjust revised execution times for each component to fit that constraint. For example, if ta(P1) < ta(P) we have to adjust its communication component if $Y_{P1} \subseteq Y_P$. Notice that if the execution time of the isolated behavior does not equal the original execution time, i.e., ta(P2) ≠ ta(P) when $O_{P1} \cap I_{P2} = \{\}$, we can increase the cycle time of component B directly to $MCT_B = ta(P)/IC_{P2}$ since component B is by definition a basic model specification.

If there are any data dependencies between A and B, i.e., $I_{P2}^* \neq \{\}$, both model component specifications have to be modified to reflect these dependencies properly. In component A, P1 has to include the required subset of internal data for component B in its external outputs Y_{P1}, i.e., new $Y_{P1} = Y'_{P1} = I_{P2}^* \cup Y_{P1}$. The waiting state W2 in component B needs to be modified in the same manner to also receive these former internal inputs as external inputs (i.e., $I'_{W2} = I_{W2} \cup I_{P2}^*$ and $X'_{W2} = X_{W2} \cup I_{P2}^*$), and to include revise its set of internal outputs O_{W2} correspondingly. If component B changes data structures pertinent to component A, i.e., $O_{P2} \subseteq I_{nP}$, then these values have to be also communicated explicitly, requiring similar changes to Y_{P2}, X_{nW}, I_{nP}, and O_{nW}.

3.2.2 Type II Scenarios

The results of a behavioral refinement (of the original behavior shown previously in Figure 3) in a Type II scenario is illustrated in Figure 5. Here, the temporal relationship between the external inputs, which are to be received in the waiting state W1 and W2 of the refined model specification, are observed in the original model specification as et(W, x1) < et(W, x2), i.e., in the original model specification the last external input for W2 arrives after the one for W1 at time t_P (see Figure 3). Generally we can perform the same adjustments as during a behavioral refinement in a Type I scenario which is depicted as the general case in Figure 5.

A different refinement is possible (but not required) if the first input event of the set X_{W2} arrives after the last external event of the set X_{W1} and there is no data dependency between P1 on P2, i.e., $O_{P1} \cap I_{P2} \neq \{\}$. This special case is shown in the second part of Figure 5 is subject to an additional constraint based on x2f which represents the first event of set X_{W2}. Here, the revised execution times of the processing P1 has to be within the arrival of the x1 and x2f, i.e., ta(P1) ≤ et(W, x2) - et(W, x2f) and ta(P2) ≤ ta(P).

If any of recomputed execution times should exceed these values, i.e., ta(P1) + ta(P2) > ta(P) or ta(P1) > et(W, x2) - et(W, x2f) and ta(P2) > ta(P), the model component cycle time is adjusted accordingly. If the revised execution times should be less than these values or produce external outputs part of the original set (i.e., ta(P1) < et(W, x2) - et(W, x2f) or ta(P2) < ta(P); $Y_{P1} \subseteq Y_P$ or $Y_{P2} \subseteq Y_P$ given that $Y_P \neq \{\}$) requires again a modification of communication components. Otherwise no further modifications are necessary. Model specifications which exhibit the behavior of the special case are better resolved by performing a structural refinement.

In a structural refinement, a new basic model component specification B is created similarly as in a Type I scenario with an identical cycle time as the original model component. The results of such a refinement are depicted in Figure 6. In the case of a data dependency a correct structural refinement is achieved when revised execution time for processing state A is within the arrival times of x1 and x2 while the execution time for P2 should be at most equal to the original processing time, i.e., ta(P1) ≤ et(W, x2) - et(W, x1) and ta(P2) ≤ ta(P). Again the same techniques are applied to adjust either the cycle time or respective communication components in the event that revised execution times exceed this constraint.

In the case of no data dependency a correct structural refinement is achieved when the execution time of P1 is within the arrival times of x1 and x2, and the original execution time while the execution time for P2 should be at most equal to the original processing time, i.e., ta(P1) ≤ et(W, x2) - et(W, x1) + ta(P) and ta(P2) ≤ ta(P). From a theoretical perspective, this second case is the most attractive for a structural refinement since there is concurrency inherent to this computation. The communication of required data structures between components A and B is achieved in the same manner as described in the Type I scenario.

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE

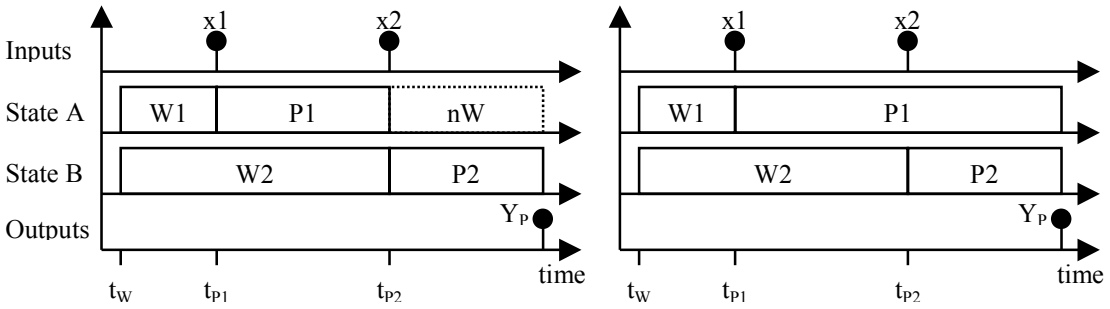

Figure 6. Results from Structural Refinement based on data dependencies in Type II Scenario

3.2.3 Type III Scenarios

In a Type III scenario the temporal relationship between the external inputs, which are to be received in the waiting state W1 and W2 of the refined model specification, are observed in the original model specification as et(W, x1) > et(W, x2), i.e., in the original model specification the last external input for W1 arrives after the one for W2 at time t_P. A pure behavioral refinement in a Type III scenario follows a Type I scenario for the general case and a Type II scenario for the special case with indexes interchanged. Any adjustments of model parameters are done in the same manner.

Structural refinement also follows the cases of Type I and Type II scenarios, respectively. Again, a structural refinement is again most advisable if there is no data dependency between P1 and P2 (i.e., $O_{P1} \cap I_{P2} = \{\}$).

4. Application Example

A system specification for an alarm clock outlines the application of our modeling and refinement concepts in practice. This example has also been implemented and tested using pDEVS - a Java-based implementation of the parallel Discrete EVent System Specification formalism [11, 13].

The functional requirements of this device are based on three modes of operation which encompass the display of the current time, the setting of the current and alarm time, and the triggering of the alarm. The user has a *time*, *alarm*, *hrs*, *min*, and *alarm-off* buttons to control the alarm clock.

In its initial operating mode, the device should display the current time. In the display mode a digital digit display is to be updated within the second that the current time (hours or minutes) changes. If the *time* button is asserted the alarm clock enters the set-time mode. Here, the current time can be modified by asserting the *min* and *hrs* buttons. A second assertion of the *time* button returns the alarm clock the display mode. The same behavior is exhibited when the *alarm* button is asserted with the difference that the alarm time can be displayed and modified. When the alarm time is reached the device enters asserts an alarm which has to be disabled by the user using the *alarm-off* button. Any other input scenarios may be ignored. The current time should be kept during any mode of operation.

We assume that alarm clock display and response in set-alarm and set-time operation modes should occur within 0.01 seconds - "instantaneous" - to the user. Furthermore clock pulses arrive every minute from a timer peripheral device, e.g., on the microcontroller chip.

In the following discussion, model state diagrams illustrate the refinement of our design model. Arrows indicate transitions between model states where double-headed arrows are used to simplify the illustration of state transitions which return to their original state.

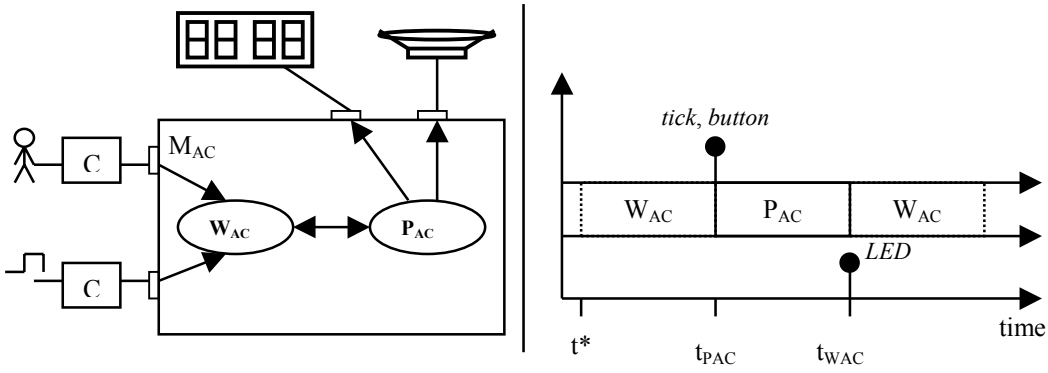

Figure 7. Abstract Initial Design Model and Type I Input Arrival Scenario

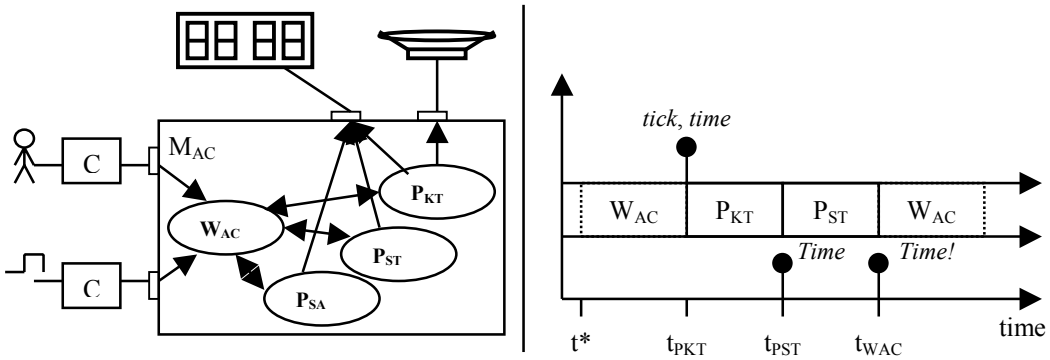

Figure 8. Design Model Specification and Trace after First Refinement Step

4.1 Abstract Design Model

We start the development with a single abstract design model component M_{AC} as shown in the first part of Figure 7. This abstract model specification waits first for either the arrival of a button from the user or a clock pulse from the timer in W_{AC}. After the arrival of each clock tick M_{AC} transitions to the processing state P_{AC} and then generates either some external outputs for the display. An initial processing state which initializes model component data structures has been omitted from this illustration.

Since the button inputs may arrive aperiodically, i.e., randomly at any point in time, we derive a worst case testing scenario, i.e., a stress test, which is depicted as a model execution trace in the second part of Figure 7. In this scenario a *button* and *tick* event arrive at the same time, i.e., a Type I input arrival scenario. Notice that the external inputs from the environment to M_{AC}, i.e., the user and the timer, are buffered with communication components to allow a sequential processing in such a scenario for P_{AC}. From the requirements the execution time P_{AC} is directly derived from the stated response time, i.e., $ta(P_{AC}) = 0.01$ seconds. Based on an average estimated instruction count (IC) for P_{AC} of $IC_{AC} = 10$ we compute a first model component cycle time (MCT) of $MCT_{AC} = 0.001$.

4.2 Refinement of the Alarm Clock Design Model

The results of initial design model refinement are shown in the first part of Figure 8. Here, we have first introduced the functional requirement of three operation modes for the alarm clock. Here, P_{SA} resembles the setting of the alarm time, P_{ST} that of the current time, and P_{KT} keeps track of time from each arrival of clock pulses. Now we may specify the behavior of the state P_{KT} in more detail; i.e., specify in its computational segment the manipulation of time based on the external *tick* input. Lets assume this specification results in a revision of its instruction count from our original value to $IC_{KT} = 9$, whereas instruction counts for P_{ST} and P_{SA} remain abstractly specified at $IC_{ST} = IC_{SA} = 10$. Since there is a data dependency between P_{KT} and P_{ST} (they both modify or access information about the current time) we need to decrease model component cycle time of M_{AC} to preserve our external interface (i.e., our required response time of 0.01 seconds) so that $MCT_{AC} = 0.01/(IC_{KT}+IC_{ST}) = 1/1900$.

The second part of Figure 8 shows the model execution trace of the refined model specification using the previous Type I test scenario. Notice that the generic *button* event has now also been refined to the actual buttons. Our worst case scenario can now be more accurately described as the *tick* event arriving just before a *time* or *alarm* event. The system response in the reverse case (a button prior to a pulse) remains well within the specified bounds, e.g., $ta(P_{SA}) + ta(P_{KT}) = 0.01 < 1$.

In our second refinement step, we perform a structural refinement triggered by introducing the requirement that time is always to be kept in parallel at any point of the alarm clock operation. The resulting composed design model specification is shown in the first part of in . In the structural refinement, we isolate state P_{KT} in a new design model component specification M_{KT}.

Due to the decomposition into two modular model component specifications, each modification of the current time has to be now communicated explicitly in P_{KT} and P_{ST} from M_{KT} to M_{AC}. Since the *time* event may arrive at any point in time we introduce some extra model states to address this data dependency explicitly: The states P_{SCT} and P_{UCT} handle send or update the current time in M_{TK}, whereas P_{GCT} and W_{CT} request and wait for this information in M_{AC}. Notice that state P_{ST} triggers in M_{AC} the update of the current time in M_{TK}. We simplified our illustration of this refinement by omitting P_{SA} from the diagram. Communication components are introduced to buffer and delay the events between model component ports.

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE

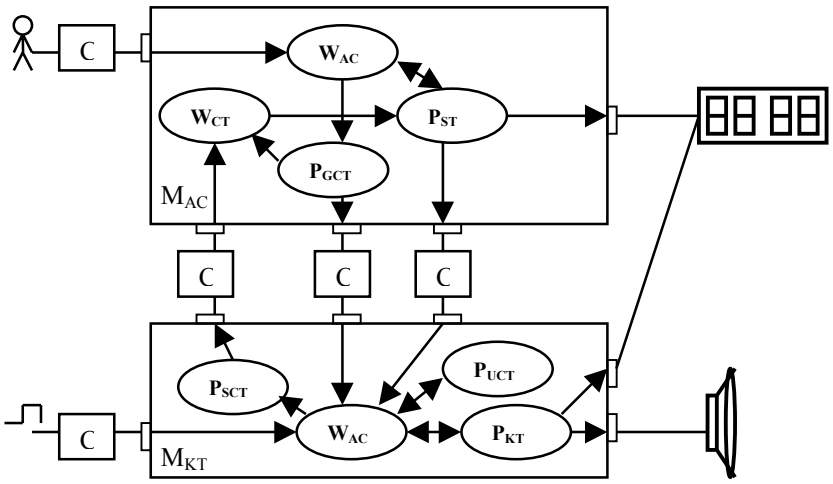

Figure 9. Design Model Specification after Second Refinement Step

5. Conclusions

We presented in this paper a formal abstraction of computation for the model-based codesign of real-time embedded systems. We introduced the specification of basic model components based on this abstraction as well as their correct structural and behavioral refinement, i.e., with a preservation of internal data dependencies and the external interface of the original modeling specification. Finally, we illustrated the abstract model specification for a simple example application and its stepwise refinement by introducing system requirements.

An emerging trend in embedded computing systems is the design for adaptable application implementations. Model-based design offers a great potential for the design of such systems since it allows an implementation independent design specification. Future work could introduce more flexibility to the concept of a model component cycle time in order to simulate changes in the implementation of system components on the fly. Similarly, the delay time parameter in communication components could be made more flexible to reflect a change of the relative location of a design component in respect to others.

References

[1] G. DeMicheli and R.K. Gupta, "Hardware/Software Co-Design", *Proceedings of the IEEE*, 85(3), 349-65, 1997.

[2] J. Fleischmann, K. Buchenrieder, and R. Kress, "A Hardware/Software Prototyping Environment for Dynamically Reconfigurable Embedded Systems", *Proceedings of International Workshop on Hardware/Software Codesign*, Seattle, WA, 105-10, March 1998.

[3] D. Gajski, S. Narayan, F. Vahid, and J. Gong, *Specification and Design of Embedded Systems*, Englewood Cliffs, Prentice-Hall, New Jersey, 1994.

[4] F. Fischer, A. Muth, A. Kirschbaum, and G. Färber, "Towards Interprocess Communication and Interface Synthesis for a Heterogeneous Real-Time Rapid Prototyping Environment", *Proceedings of International Workshop on Hardware/Software Codesign*, Seattle, WA, 35-9, March 1998.

[5] J. Axelsson, *Analysis and Synthesis of Heterogeneous Real-Time Systems*, Dissertation No. 502, Linköping University, Sweden, November 1997.

[6] S. Kumar, *A Unified Representation for Hardware/Software Codesign*, Ph.D. Dissertation, University of Virginia, UMI Number 9600485, Ann Arbor, 1995.

[7] A. Kalavade and E.A. Lee, "A Hardware-Software Codesign Methodology for DSP Applications", *IEEE Design and Test Computers*, 10(3), 16-28, 1993.

[8] J.W. Rozenblit and K. Buchenrieder, *Codesign: Computer-Aided Software/ Hardware Engineering*, IEEE Press, Piscataway, 1994.

[9] S. Schulz, J.W. Rozenblit, M. Mrva, and K. Buchenrieder, "Model-Based Codesign", *IEEE Computer*, 31(8), 60-7, August 1998.

[10] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998.

[11] B.P. Zeigler, H. Praehofer, and T.G. Kim, *Theory of Modeling and Simulation*, 2nd Edition, Academic Press, Burlington, MA, 2000.

[12] J.L. Hennessey and D.A. Patterson, *Computer Architecture - A Quantitive Approach*, Kaufmann Publishers, 1993.

[13] S. Schulz, *Model-Based Codesign for Real-Time Embedded Systems*, Ph.D. Dissertation (UMI#: 3002539), The University of Arizona, Ann Arbor, 2001.