

Transforming Structural Model to Runtime Model of Embedded Software with Real-time Constraints *

Sharath Kodase, Shige Wang, Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122
email: {skodase,wangsg,kgshin}@eecs.umich.edu

Abstract

The model-based methodology has proven to be effective for fast and low-cost development of embedded software. In the model-based development process, transforming a software structural model that describes the underlying application, to an implementable runtime model is a critical issue. Since the designed software will finally run on the target platform, non-functional issues like schedulability, timing constraints and resource requirements have to be considered during the transformation. In this paper, we propose a generic runtime model architecture that can best satisfy the non-functional requirements of the system, and a generic transformation method to convert a structural model to a runtime model in such an architecture. The transformation approach is based on the notion of end-to-end computations performed by the system in response to external stimuli. We demonstrate the advantages and effectiveness of the proposed method by constructing a software runtime model for a combined electronic throttle and air-fuel ratio control system.

1 Introduction

The model-based software development methodology is known to accelerate embedded software (ESW) development and lower the development cost. In the model-based methodology, multiple models are used to specify the ESW behavior, structure, and implementation, and verify the system correctness. The process of ESW development can therefore be viewed as a multi-stage process of transforming a model at one stage to a model at another stage. Differ-

ent stages in the ESW design can be partitioned into behavior design, structure design, runtime system generation, and code generation, with behavioral model, structural model, runtime model, and programming model used at each stage, respectively. In each stage, the model is refined with further implementation details to generate a model for the next stage.

One of the major hurdles in current model-based ESW development practice is the transformation from a structural model to a runtime model. A structural model specifies components and their interactions in ESW, and is usually represented with data flow diagrams, context flow diagrams, and object collaboration diagrams. A runtime model, on other hand, specifies the organization of entities in a structural model into runtime tasks,¹ inter-task communications, and task allocations on the execution platform. During the transformation from a structural model to a runtime model, complex non-functional issues like schedulability of the task set, message delays, and response-time constraints must be considered to ensure the correctness of the resultant runtime system. This makes the transformation complex for real applications with a large number of components and component-interactions.

Although there are methods proposed for systematically transforming a structural model to a runtime model [2, 4, 9], non-functional constraints are usually either ignored or considered as non-critical requirements during the transformation. This makes it uncertain for the generated runtime model to meet system timing constraints. Hence, the ESW designer is driven to perform the transformation by expensive ad-hoc trial-and-error.

In this paper, we propose a novel transformation method to convert an ESW structural model to a runtime model.

*The work reported in this paper was supported in part by DARPA under the US AFRL Contract No. F33615-00-C-1706.

¹Tasks in our model can be implemented as processes/threads. We use the terms “task” and “process/thread” interchangeably.

Our transformation process is based on a component-based structural model and a task-based runtime model. The objective of runtime model generation is to obtain an implementation with high processor utilization and low runtime overheads. The transformation process obtains *transactions* or end-to-end computations performed by the system in response to external stimuli, by analyzing the structural model. Then, it derives real-time properties of these transactions according to system timing constraints and available resources in the platform configuration. Finally, the runtime model is obtained using the derived properties of transactions.

The rest of the paper is organized as follows. Section 2 presents the structural model and the runtime model that we use to develop the transformation method. Section 3 describes the algorithm to transform a structural model to a runtime model with timing and scheduling constraints. Section 4 illustrates the transformation method and its advantages through a real world example of automotive electrical throttle control (ETC) with air-fuel ratio (AFR) control system. Section 5 states the related work. The paper concludes with Section 6.

2 Structural and Runtime Models

Structural and runtime models describe the same ESW but with different views and are used during different development phases. Structural models are generally used to model the ESW structure for functional design, while runtime models are normally used later for ESW implementation on a given target platform.

Structural model. The structural model we use here is component-based, in which ESW design is modeled as a set of software components and their interactions. Each component in the model contains a set of external interfaces, a behavior controller, and controlled actions as shown in Figure 1. The external interfaces are represented as events accepted by the component through dedicated ports, and define the functionality that can be invoked by other components. The behavior controller implements a subset of functions defined in the behavioral model, and controls the actions performed in response to the input and output events generated. With such a component model, the system can be designed by connecting cooperating components through their ports, and the system execution can be done by having external events like timer interrupts and/or sensor I/O trigger a sequence of actions in components. We assume asynchronous event communication between components. Such a component structure [13] has been shown beneficial for software reuse and model-based integration.

Runtime model. The generic runtime model we are assuming consists of tasks, whose invocations are triggered by messages including interrupt signals, data, and events.

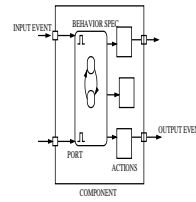


Figure 1. Component structure.

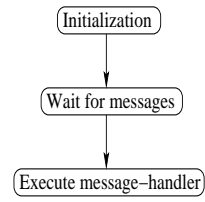


Figure 2. Task structure.

The structure of a task is shown in Figure 2.

Each task may perform a sequence of actions from one or more components. This requires a task to have references to all components that own the actions to be performed. Such references are constructed during task initialization. Also, during the initialization, all messages that a task can receive and actions that must be performed in response to a message must be registered.

After initialization, each task performs blocking-wait for messages. Upon arrival of a message, the task executes the registered components' actions in a predefined order to process the message, and returns to the blocking-wait. In each round of a task execution, exactly one message is processed. Therefore, a queue is required for each task to buffer incoming messages during the task execution. The task is not re-entrant, meaning that the arrival of a new message has to wait in the queue until the task finishes processing the current message.

We further assume that all tasks execute with statically-assigned priorities in the runtime model. A task with high priority preempts lower-priority tasks. Actions within a task are executed at the same priority assigned to the task itself. This distinguishes our model from other models used in automatic runtime model generation which require dynamic priority assignment to tasks according to the processed messages [6, 11]. Compared to these models requiring dynamic priority assignment to tasks and two-level scheduling (at message and task levels), our approach has the advantages of low runtime overhead, lesser complexity and better support from current commercial real-time operating systems.

Given such structural and runtime models, the problem of transformation from a structural model to a runtime model is to map components' actions in the structural model to tasks in a runtime model in a way that (a) the execution sequence of actions defined in the structural model to achieve functional objectives is preserved, and (b) both the timing and scheduling constraints of the system are met.

3 Model Transformation

Our transformation method relies on the notion of *transaction* to generate a runtime model from a structural model.

A *transaction* is defined as a sequence of actions of components performed in the end-to-end processing of an input signal. A *transaction* can be identified by tracing the actions in a structural model along the data processing path from input to output. Multiple transactions may exist in a structural model and each transaction can be represented as a directed acyclic graph.

Our method for generating a runtime model from a structural model consists of three steps: (1) identify transactions in the structural model, (2) assign priorities to component actions in each transaction to meet timing and scheduling constraints, and (3) allocate actions in transactions to threads. This process is outlined in Figure 3.

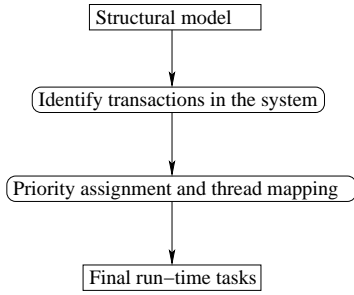


Figure 3. Process of transforming structural model to runtime model.

Transaction identification: The first step of the transformation process is to identify all transactions in the structural model with action sequences and components. Starting from each input signal to the system, transactions can be found by recording actions performed by components in response to the input signal and following the output events generated. Following an event involves recording actions that are triggered by the event and following other generated output events. This following is terminated when no further output events are generated, or the outputs leave the system. For the structural models represented as UML collaboration diagram or sequence diagram, for example, the transactions can be obtained through tracing events along the links, and determining the sequence of actions by following the sequence number.

The number of transactions in a structural model can be large as the model has to consider transactions performed in all possible system modes/states. However, not all the transactions in the system are essential in runtime model generation.

Given the fact that some transactions will not be active simultaneously (for example, transactions to control ETC without cruise control and with cruise control), these transactions can be treated as a group, and the one with the

longest end-to-end computation time can be used as the representative in runtime model generation. Such grouping reduces the number of transactions significantly, and makes the approach scalable for large applications.

We further assume that some performance information, like worst case execution times (WCETs), of component actions are available. Such information can be obtained during unit-testing of a component, and stored with the components.

Priority assignment and thread mapping: After identifying the transactions, we assign priorities to component actions before constructing tasks in the runtime model. The priority of an action is assigned so as to meet the timing constraints of the transaction containing the action. Such assignment works on finer-grained actions rather than tasks. Therefore, it is more flexible than traditional approaches where task construction is done first, and priority assignment to the tasks is done later.

The problem of priority assignment to actions in transactions is similar to the problem of priority assignment to tasks in a real-time system. As the problem of assigning task priorities to make a system schedulable is an NP-hard problem, assigning priorities to actions with transaction timing constraints met is also NP-hard, and hence requires heuristic techniques to obtain a solution.

In this paper, we use an approach based on simulated annealing [7, 8] to generate action priority assignments. Given any priority assignment s , simulated annealing randomly searches for an optimal solution among the neighbors of s . When it finds a better assignment n , it jumps to n and continues search from n until it finds the optimal solution. Therefore, important elements in the simulated annealing algorithm are the energy function used to describe the desirability of a candidate assignment, neighbor-selection function and a temperature annealing schedule.

The energy function is designed such that the solution with the lowest energy will be the optimal solution. In our approach, the energy function is designed such that the optimal solution satisfies timing and schedulability constraints and has low runtime overheads, in terms of threads and inter-thread communications. Particularly, the energy function for priority assignment s in our method is defined as:

$$\begin{aligned}
 E(s) = & k_1 * \max\{r(R_{e_i}) - d(R_{e_i}) : R_{e_i} \text{ is a transaction}\} \\
 & + k_2 * \text{number of distinct priority levels} \\
 & + k_3 * \text{number of communicating} \\
 & \quad \text{component-actions of different priority levels}
 \end{aligned}$$

where $E(s)$ is the energy value of the assignment s ; $d(R_{e_i})$ is the deadline associated with R_{e_i} ; and $r(R_{e_i})$ is the response time of R_{e_i} . Values of k_1 and k_2 are chosen such that the first two components of $E(s)$ return similar values. Constant k_3 is designed to be smaller than k_2 for minimizing the number

of priority levels used in a transaction. Selection of these constants depends on the problem under consideration.

As can be seen from the above equation, the factors we consider in action priority assignment include timing and scheduling constraints, the number of priority levels used in the whole system, and the number of priority levels used in a transaction. While timing and scheduling constraints must be met as the system requires, minimizing the number of priority levels used both in a system and within a transaction reduces runtime overhead.

To reduce the search space for neighbor selection, we apply the result from the [5], which states that the end-to-end system response time is not affected if priorities assigned to actions are changed to canonical form. In a canonical priority assignment to a transaction, succeeding actions have higher or equal priority than preceding ones. Our algorithm uses this result and restricts the search to only those points that are in the canonical form. The neighborhood of a given point s is found by either of the following:

1. Raise the priority $p(a_{e;k})$ of an action $a_{e;k}$, which is not at the highest priority level, to a new level p' , such that $p(a_{e;k}) < p' \leq p(a_{e;k+1})$, where $a_{e;k+1}$ is the succeeding action. If $a_{e;k}$ is the last action in the system response, then raise $a_{e;k}$ to level greater than $p(a_{e;k})$.
2. Lower the priority of an action $a_{e;k}$, which is not at the lowest priority level, to a new level, p' such that $p(a_{e;k-1}) \leq p' < p(a_{e;k})$, where $a_{e;k-1}$ is the preceding action. If $a_{e;k}$ is the first action in the system response, then lower the priority of $a_{e;k}$ to level lower than $p(a_{e;k})$.

The action $a_{e;k}$ is picked randomly among the set of transactions.

For the annealing schedule, the temperature is decreased linearly according to the equation, $T_{\text{next}} = k_4 * T$. The values of k_4 is usually between 0.95 to 0.99. The algorithm terminates when a feasible solution is found and neither upward nor downward energy jumps have been made for the last $E_{\text{equilibrium}}$ moves. In our implementation, we set $E_{\text{equilibrium}}$ to be 3 times the neighborhood (space) of a point in the search space.

We use deadline-monotonic priority assignment to actions as the starting point for the simulated annealing algorithm. For the schedulability analysis, we modify the approach in [5] to include the blocking delays due to synchronization between actions, overheads due to context switch and scheduling, and overheads due to communicating actions of differing priorities.

The task set is constructed only after priorities of all actions are assigned. We then construct the tasks by mapping all actions with the same priority to one task. The advantage of this mapping is that the results of timing and

schedulability analysis in the priority assignment step remain same and valid, even after task mapping. The reason for this is that we consider the worst-case response times for transactions in the energy function and context switch, scheduling overheads in the schedulability analysis. Since non-functional issues like schedulability and system timing constraints, are satisfied in the priority assignment step, the runtime tasks thus constructed are also guaranteed to satisfy the non-functional requirements. This approach can also be viewed as a generic approach to task construction. On one end, transaction-based task construction² can be achieved by assigning the same priority to actions in a transaction. On the other end, the approach becomes component-based or object-based task construction³ if all actions of a component are assigned the same priority.

Since our approach maps actions to tasks, it is an action-based mapping. The unit of mapping is of finer granularity than transactions or objects. Consequently, this approach leads to lower blocking delays and results in higher processor utilization than either transaction-based or object-based approaches [1]. Also, in some cases, action-based mapping yields a successful runtime [5] that cannot be obtained using other approaches.

4 Runtime Model Generation of an Automotive Application

To demonstrate the effectiveness of our approach in obtaining a runtime model, we apply the proposed method to an application in the automotive domain, consisting of Electronic Throttle Control (ETC) and Air-Fuel Ratio (AFR) control. ETC is a subsystem in the Powertrain engine control system, which regulates throttle and controls the volume of air in the engine cylinders. AFR is another part of engine control, which controls the ratio of air-to-fuel mixture in the cylinders. The high level structural model of the ETC and AFR system is shown in Figure 4.

In this model, all components of Manager, Monitor, AFR and Servo Control are triggered by periodic timing signals. The behavior specification for these components are represented as statecharts.

Using the statecharts, we obtain transactions present in the system. Table 1 lists the end-to-end transactions, timing characteristics of actions, and the deadlines of the transactions in the example application. The worst-case execution time (WCET) values in the table are arbitrarily chosen to demonstrate the ability of our approach to runtime mapping. However, the periods and the deadlines are from a real ETC/AFR application.

²In transaction-based mapping, all actions in a transaction are mapped to one task.

³In object-based mapping, all actions of a component are assigned to one task.

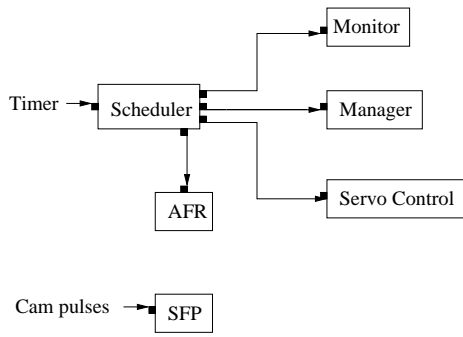


Figure 4. Electronic throttle control and air fuel ratio control.

Since ETC and AFR components are essential during the engine operation, these transactions are always active in the system in all engine operation modes. For the aperiodic SFP component, we assume that it is invoked at the highest rate. The actions of *afr* and *sfp* access a shared data variable, and hence must be synchronized. A 0.05 ms blocking time is assumed for any synchronization in this example.

An extra 0.05 ms is added to the execution time for any inter-thread communication. We apply our priority-assignment algorithm for these transactions. Table 2 shows the obtained priority assignments, and the response times for these transactions under such assignment, and the final task set generated. Priorities obtained using simulated annealing with the discussed energy function are given as the numbers next to component actions.

The processor utilization of the constructed system for the above example is as high as 0.933. We also notice that other transformation methods like object-based (making scheduler, monitor, and other components a separate task) and transaction-based (making each transaction a separate task) mapping cannot yield a schedulable system.

5 Related Work

There have been a number of approaches proposed for real-time software development and for runtime mapping from different structural models. CODARTS [9] and COMET [3] are two examples that support systematic generation of runtime models based on functional properties of components. The structural model used in these tools is similar to UML-RT. The components and their interactions are classified into several “stereotypes” according to their functionality. This classification is used for runtime mapping. They do not use schedulability to guide the mapping process.

In [2, 12], object-based mapping is proposed to guide the transformation from a structural model to an implementa-

Action	WCET (ms)	Period (ms)	Deadline (ms)
sch.monitor	0.20	-	-
sch.manager	0.20	-	-
sch.servo	0.20	-	-
sch.afr	0.20	-	-
monitor	1.25	20	20
manager	3.25	20	20
servo	1.25	5	5
afr.get_speed	4.00	20	-
afr	6.25	20	20
sfp	0.25	10	10

Transactions
sch.monitor → monitor
sch.manager → manager
sch.servo → servo
sch.afr → afr.get_speed → afr
sfp

Table 1. Real-time characteristics of component-actions and transactions in ETC and AFR.

Transaction	Response time (ms)	Deadline (ms)
sch.monitor [3] → monitor [3]	17.75	20.0
sch.manager [2] → manager [2]	17.75	20.0
sch.servo [3] → servo [3]	3.25	5.0
sch.afr [1] → afr.get_speed [1] → afr [2]	27.2.0	30.0
sfp [3]	3.75	10.0

Thread	Priority	Actions
Thread_1	1	{ sch.afr → afr.get_speed }
Thread_2	2	{ {sch.manager → manager}, afr }
Thread_3	3	{ {sch.servo → servo}, {sch.monitor → monitor}, sfp }

Table 2. Priority assignment and response times of transaction in ETC and AFR, and runtime threads in the implementation model.

tion model. First, threads are identified first based on properties like periodicity and functionality. Then, objects are mapped to threads. One of the main difficulties with using object-based mapping in static priority threading architectures is that every component-action executes with the same fixed priority, irrespective of the criticality or the importance of the transaction it is involved in. As a result, some actions that could be run at lower priorities get executed at higher priorities, increasing the blocking times and decreasing the processor utilization. It also becomes difficult to assign priorities to threads when a component is involved in multiple transactions. But the advantage of this approach is that it is easier than other approaches to do the transformation.

In [10,11] a mapping methodology is provided for ROOM [12] models considering schedulability. The runtime architecture they assume is a dynamic priority thread architecture where threads change their priority based on messages they handle. In [6] a mapping process is presented that extends this work for UML-RT models. The runtime architecture they assume is also a dynamic priority threading architecture where threads have pre-emptive thresholds that specify what other threads can pre-empt them. They adopt a combination of transaction-based and priority-based mappings.

The structural model we assume for our mapping process is from [13], which is the model proposed for embedded software architectures, designed for re-usability of components. The implementation model for our approach is inspired by practical applications like avionics mission control and automotive control applications which use very similar models.

6 Conclusion

In this paper, we propose a methodology of transforming structural models to runtime models with real-time constraints considered. Our method makes use of results from real-time scheduling theory and tries to obtain a runtime model that not only satisfies timing constraints but also yields high processor utilization and low overheads. The proposed approach is based on identifying transactions, and assigning priorities to fine-granularity actions using the simulated annealing technique. As opposed to the traditional object-based or transaction-based approach, our approach yields higher processor utilization, lower implementation overheads, and timing constraint satisfaction.

Our approach departs from the usual practice of determining priorities for transactions in the system first and then determining the threads present in the system by integrating these two steps into one. Priorities are assigned to component-actions considering the system timing constraints, platform configuration, and overheads due

to scheduling, context-switch, and inter-thread communication. The simulated annealing heuristic tries to obtain a solution that satisfies timing constraints and yields minimal implementation overheads. Using an example motivated by a automotive power-train application, we have shown that our method can produce successful implementation models when other approaches fail.

References

- [1] A. Burns and A. J. Wellings. Dual priority assignment: A practical method for increasing processor utilization. In *5th Euromicro workshop on Real-time systems*, pages 48–53. IEEE Computer Society Press, June 1993.
- [2] B. P. Douglass. *Doing Hard Time*. Addison Wesley, 1999.
- [3] H. Gomaa. Designing real-time applications with the comet/uml method. citeseer.nj.nec.com/470515.html.
- [4] H. Gomaa. *Designing Concurrent Distributed, and Real time Applications with UML*. Addison-wesley, 2000.
- [5] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. *Real-Time Systems Symposium*, pages 116–128, December 1991.
- [6] S. Kim, Sukjae-Cho, and Seongsoo-Hong. Schedulability-aware mapping of real-time object-oriented models to multi-threaded implementations. *Real-Time Computing Systems and Applications*, pages 7–14, December 2000.
- [7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, (220):671–680, 1983.
- [8] P. G. M. Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. D. Reidel Publishing, 1987.
- [9] K. Mills and H. Gomaa. Knowledge-based automation of a design method for concurrent systems.
- [10] M. Saksena. Towards automatic synthesis of qos preserving implementations from object-oriented design models. *Workshop on Object-Oriented Real-time Dependable Systems*, (93–99), November 2000.
- [11] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-time object-oriented models. *International Symposium on Object-Oriented Real-Time Distributed Computing*, March 2000.
- [12] B. Selic, G. Gullickson, and P. Ward. *Real-time object oriented modeling*. John wiley and sons, 1994.
- [13] S. Wang and K. G. Shin. An architecture for embedded software integration using reusable components. *International conference on compilers, architectures, and synthesis for embedded systems*, 2000.