Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 215 (2008) 171–189

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Resource-Oriented Design Framework for Embedded System Components

Jin-Hyun Kim, Jae-Hwan Sim and Jin-Young Choi

Department of Computer and Communication Engineering
Korea University, Seoul, Korea
{jhkim,jhsim,choi}@formal.korea.ac.kr

Abstract

To implement functionally and timely correct embedded systems, it is essential to consider both hardware and software behavior simultaneously. This paper presents an embedded system design framework, called resource-oriented design, in which embedded system components are incrementally developed in two behavioral aspects; resource-independent model(RIM) and resource-oriented model(ROM). The former embedded system model describes embedded system behavior in terms of functionality, and the latter model specifies software behavior that is restricted by hardware resource constraints. The software behavior models in those two models are based on a formal and concise hardware behavior model so as to achieve software behavior model in compliance with hardware's behavior. The hardware and embedded software behavior we define is oriented to an interaction between hardware and software. The advantage of our framework is to gain two software behavior models, functional aspect and resource-constrained aspect, such that those two models are consistent in each other and they are in compliance with hardware behavior. For the specification and verification of resource-oriented models, we use ACSR(Algebra of Communicating Shared Resource) and VERSA(Verification Execution and Rewrite System for ACSR).

Keywords: Formal Methods, Embedded System, Resource-Based, Design Methodology

1 Introduction

An embedded system is a hardware and software interacting system such that behavior of them must be in accordance with each other. To construct embedded software component in accord with hardware's behavior, a correct and complete hardware requirement for software needs to be given in the view of software engineer, and such a hardware requirement originated from hardware engineers can be given in a formal description, and shows hardware's behavior that not only interacts with software but also restricts the behavior of

doi:10.1016/j.entcs.2008.06.027

software.

To create a formal hardware and software description that are in accord with each other, codesign[3,8] provides a design framework where an embedded system is captured in a unique system design without discriminating hardware and software. The system design is partitioned into respectively a specific detailed hardware and software design for implementation of them. However, such codesigned software models cannot be often implemented correctly and completely due to hardware and software interaction problems, such as communicating timing. The behavior of embedded software, in addition, generated from a codesigned system can often differ from the behavior of original software model because of resource constraints where processes are limited to the use of resources. Thus, it is necessary to capture and reason the behavior of embedded software that is restricted by not only communicating time but also resource constraints.

Our primary goal is to provide a design framework in which capturing a behavior of embedded software is based on a dynamic resource behavior. The behavior of resource is originated from hardware's behavior that interacts with software or restricts the behavior of software in an execution of embedded system. Thus, the behavior of resource is captured in two aspects; functionality and constraints. The functionality of resource consists of hardware's own functionality related to software's functionality and the interaction between hardware and software. The constraints of resource upon software's functionality is depicts in terms of time and the availability of resource. The timed behavior of embedded software often depends on not only communicating time but also the availability of shared resource in embedded software execution[6]. Thus, the behavior of resource describing hardware's constraints upon software's behavior consists of the communicating time of hardware and software and the availability of shared resource.

Based on such resource models, embedded software can gradually be captured in accord with the behavior of hardware. First, the behavior of embedded software can be captured in functional aspects by hardware and software's functions and interaction. The focus of hardware's behavior in the functional view is to capture hardware's actions that impact to embedded software's behavior. The interaction between hardware and software is composed of interrupts and addressable memory(we call it interface memory). The interrupts in our view are originated from hardware and used to synchronize software periodically or sporadically. The addressable memory, called interface memory, is accessible from both hardware and software and used to move data between hardware and software. The model of resource we present here is based on hardware and software's interaction in the execution of embedded system. Second, the functional model of embedded software can be extended with hardware timing constraint, such as communicating time. Moreover, it incorporates the availability of resource in software's execution by capturing the state of resource over software reaction to hardware stimulus.

With such concise resource models, an embedded system is captured in two system views; Resource-Independent Model(RIM) and Resource-Oriented Model(ROM)[5]. RIM is the composition of functional resource model and software behavior model while ROM is the composition of constrained resource model and software model. In RIM, software behavior model can be given without hardware constraints, such as time and availability. It is useful when a commonly used software over different hardware platforms should be designed into a behavioral model. Meanwhile, ROM can help us capture software's behavior that is oriented to a specific hardware platform because the behavior of software is rather restricted by communication timing and the availability of resource than the functionality of hardware when we have a hardware platform in mind. Thus, ROM is useful when software functional model needs to be extended with hardware specific properties, such as communication time and the availability of a hardware.

The rest of this paper is organized as follows: In next Section 2, some related works are discussed, and then, we define behavior models of embedded system components in Section 3. In Section 4, we present a resource model that captures a hardware's specific behavior and constrains. After that, we introduce two embedded system models that are composed of behavior models of embedded system components in Section 5. In Section 6, we explain our design framework by giving an example of embedded system. We discuss a formal verification in our design framework in Section 7, and we finally conclude this paper in Section 8.

2 Related Works

Embedded system models can be divided into structural model and behavioral model[2]. Software developers are usually even more concerned about how these structural elements behave dynamically as the system runs. Douglass[2] suggests a behavioral description using statechart[4]. However, He does not discuss how to define hardware properties, such as time, resource restriction, in the embedded system model.

The notion of a resource provides a convenient abstraction mechanism[7] that can describes the behavior of hardware in the view of software. A resource can be captured in two aspects; preemptive [6] and quantitative [7,10]. Lee and Ben-Abdallah suggests a description language, called ACSR(Algebra of

Communicating Shared Resource), in which a real-time system is designed with focusing on a set of communicating processes that use shared resource for execution and synchronize with one another. The nature of resource in ACSR is preemptive by a higher priority process during an execution of processes. Thus, a system is captured in terms of the behavior of process that is restricted by limit resources and a scheduling of processes for shared resources, then such processes do not care for the function or the action of resource. The nature of resource that Lee and Sokolsky present in PACSR(Probability ACSR)[7,10] differs from the resource of ACSR in that a process can acquire a resource with a given probability so that PACSR allow to reason quantitatively about a system's behavior. In this paper, we present a more dynamic resource that can changes its state according to stimulus from environment and embedded software, including time. A resource we present here captures hardware's behavior in terms of software, and is modeled by the interaction between hardware and embedded software in their execution and communication.

3 Embedded System Behavior Models

In this section, we present three behavior models that can constitute an embedded system; software behavior model, software-oriented hardware model, and resource model. **Software Behavior Model(**M_S**)** defines the functionality of software including its reaction to hardware events, such as hardware interrupts and time tick. **Software-Oriented Hardware Model(**M_{SH}**)** defines a hardware functional behavior over events in embedded software engineer's view. The hardware functional behavior is oriented to hardware and software interaction. **Resource Model(**M_R**)** captures the behavior of hardware over events and time in the execution of embedded system. In this resource model, we focus on hardware constraints upon embedded software, such as hardware and software's communication time. Thus, the behavior of hardware in a resource model describes by not only the functionality of hardware but also timing and availability constraints. Resource Model differs from Software-Oriented Hardware Model in that Resource Model is captured after the decision of hardware platform. Embedded software is often implemented after hardware is developed and a software functional requirement needs to be delivered before hardware's development is finished. Then, software engineers can design embedded software in aspect of functionality using Software-Oriented Hardware Model because Software-Oriented Hardware Model presents the aspect of hardware functionality.

In this paper, the behavior of hardware and embedded software in Fig. 1 is oriented to the interaction of them. We assume that an embedded system is a tuple (M_S, S_Ω, M_{SH}), where M_S is a software behavior model, S_Ω is interface memory, and M_{SH} is a hardware behavior model. In this embedded system model, embedded software interfaces directly with hardware through interface memory(we call it interface memory because it is shared by hardware and software, such as CPU register and addressable memory), The interaction between hardware and software is achieved as follows; A hardware unit(M_{SH}) interrupts CPU running software system when it wants to order a software routine to process hardware data and completes its data writing on interface memory. The interrupted CPU finds out a corresponding interrupt handler in its interrupt vector table(IVT), and then, it initiates interrupt service routine(ISR) based on an address from IVT. ISR just sends a message to the real-time OS in order to initiate the corresponding task. The initiated task reads hardware input data from interface memory, processes them, and writes the processed data into interface memory in a specific time. Hardware reads the software-processed data from interface memory in a specify time and computes another operations according to the software-processed data and its own state. In firmware, an ISR directly calls a software function or procedure when hardware interrupts CPU running a idling function. Thus, the first software routine initiated by a hardware interrupt is an interrupt handler. Other software routines, task, processes, function, and procedure are initiated by such interrupt handlers. These software routines process hardware data from interface memory based on their own state, and hardware units also perform their functions based on data in interface memory and their own state recoded in register.

3.1 Embedded Software Behavior Model

Software Behavior Model defines the functionality of software including its reaction to hardware events. If a hardware event, hardware interrupt or time

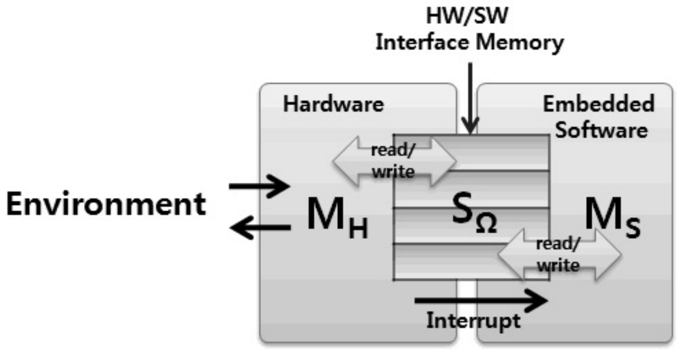


Fig. 1. Embedded System Model

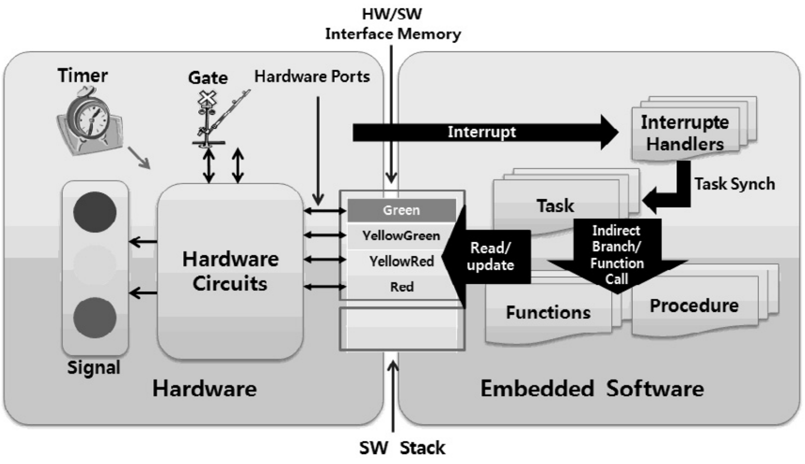


Fig. 2. The Overview of Embedded System Behavior

tick, initiates an interrupt service routine(ISR), the ISR calls software routines that would handle a hardware input data. The software routine transforms the configuration of interface memory by processing data or calls other software routines. Finally, a software routine calls a special task, called **IDLE**, to wait for a next hardware event. Software Behavior Model is defined as follows:

Definition 3.1 Software Behavior Model (M_S) is a tuple $(S_S, E^I_S, F_S, Q_S, \theta)$. S_S is a set of software states where $S_\Omega \subseteq S_S$, and E^I_S is a set of interrupts from hardware. $F_S : S_S \to S_S \times F_S$ is a set of software routines, and $\theta : E^I_S \to F_S$ is a function from hardware events to a software routine, called interrupt service routines, where $\forall q \in F_S . \forall s \in S_S . \exists f \in F_S . q(s) = f$.

In F_S, IDLE is a special function such that

$$\forall s \in S_S.\ I(s) = (s, I),$$

The software behavior model defines software's reaction to hardware interrupts, software routine calls, and interface memory read and write. The embedded software routines can be task, process, function, procedure, or interrupt handler. When a hardware unit interrupts CPU running embedded software, CPU looks at an interrupt vector table($E^I_S \to Q$) in order to find out a corresponding interrupt handler. An interrupt handler calls another software routine($q(s) = f$), such as tasks. A software routine initiated by interrupt handler starts its data process using the state of interface memory and another software routine($F_S : S_S \to S_S \times F_S$). In a completion of embedded software execution, a special function, IDLE, is executed for a next hardware event.

Fig. 2 shows a behavioral view of embedded system behavior. Hardware units and embedded software components in an embedded system transmit their data via addressable memory(we call it interface memory). A hardware port connected to interface memory moves the software processed data into hardware registers to allow a hardware circuit to control the signal system and the gate system in Fig. 2. In the execution of embedded software, hardware interrupts CPU that runs embedded software, and an interrupt handler calls a corresponding task. The task can directly modify interface memory or indirectly modify it via another function and procedure, utilizing its local memory area to process data.

3.2 Embedded Hardware Behavior Model

To capture hardware's behavior in the running of embedded system, a hardware and software's interaction using hardware's state, communication data, and event would be focused on. The information transmitted through interface memory are about hardware's state and data to be processed by embedded software. **Software-Oriented Hardware Model**, an abstract hardware behavior model, is defined as follows:

Definition 3.2 Software-Oriented Hardware Model(M_{SH}) is a tuple $(S_{SH}, E_H^I, E_H^O, E_S^I, \delta_{SH}, I_{SH})$, where S_{SH} is a set of abstract hardware states, and E_H^I is a set of events from the environment. E_H^O is a set of events to the environment, E_S^I is a set of interrupt to embedded software, $\delta_{SH} : S_{SH} \times E_H^I \rightarrow S_{SH} \times E_H^O \times E_S^I$ is a transition function, and I_{SH} is a set of initial hardware states.

In Software-Oriented Hardware Model, an event from its environment initiate a hardware module so as to execute a sequence of functions based on both its own state and the value of data from embedded software in interface memory. The software-processed data is written in interface memory($S_\Omega \in S_{SH}$) with the state of software(if software records its state in interface memory). A hardware module reads such software-processed data via port and bus connected to interface memory. During running of hardware unit, it can change its state and the value of interface memory, generate events to the environment, and occur a hardware interrupt for software's initiation($\delta_{SH} : S_{SH} \times S_\Omega \times E_H^I \rightarrow S_{SH} \times E_H^O \times E_S^I$).

Hardware's behavior is often subject to physical time because it physically controls devices. Therefore, we define a time-related function, called duration function(D).

Definition 3.3 Duration Function $D : E \rightarrow N$, where $E \subseteq E_S^I$

Most of hardware modules makes a synchronization of each hardware modules by a specific signal. In running of embedded system, a hardware module interacting with embedded software often requires embedded software to finish all of its operations in a given time. A deadline is a time by which a task must complete[11]. However, Duration we define here is a particular time interval from the start of software operation to the completion of all of software operations. Duration(D) is from a hardware event, interrupt, to a duration time.

4 Resource-Based Embedded System Models

Resources in computer system are physical and logical elements that application software needs in processing data. For instance, some embedded software needs physical elements, power and bandwidth, to meet QoS and CPU, memory, and I/O devices, to process its data. Some embedded software needs a logical element, data structure, semaphore, message queue, mailbox, network data packet served by operating systems for accessing to such hardware devices.

Resource in embedded system can be defined as a hardware-controlled object that interacts with embedded software. The hardware-control object consists of interface memory for the communication between hardware and software and interrupts from hardware. Interface memory is dedicated to data incorporating the hardware and software's behavior as well as communicating information. The interrupt initiating the activity of software is originated from hardware and interface memory has the value of data controlled by hardware and software. The reason why the hardware-controlled object is regarded as a resource is as follows: The configuration of interface memory at one time describes, in terms of software, the behavior of hardware at that time, and it can also include the value of data to be processed by software. The configuration of interface memory changes along with hardware behavior, and embedded software reacts to interrupt from hardware and executes its operations according to the configuration of interface memory. That is, interface memory is controlled by hardware and moves hardware's state to software. Moreover, it also delivers data of hardware to embedded software to process them. Thus, interface memory is no doubt regarded as a resource. Moreover, the behavior of hardware shown by interface memory can be regarded as one of resource properties. In short, we add a behavioral aspect shown on interface memory to resource properties. The resource of embedded system we consider here can be classified into hardware resource and software resource. A hardware resource is a hardware object that interacts with software or supports software execution. For instance, CPU, memory, I/O ports, motors, sensors are hardware resource. Software resource is a software object that provides hardware management, such as read/write external devices. A software resource also provides application software with data to be processed and a bunch of functions supporting software's execution. The software resource differs from application software in that it continuously reacts to its environment and keeps reacting to software's requests and hardware's interrupts. Thus, it can be regarded as a hardware component in a view of application software.

The nature of resource in embedded systems can characterized by three properties. First, a resource in embedded system has a specific interactive behavior with software. The behavior of resource reflects in a software engineer view hardware behavior that interacts with software. Using interface memory, embedded software can grasp hardware behavior and input data that it needs in its computation. In addition to interface memory, a hardware interrupt and time tick is often used for software to acknowledge hardware behavior in the sense that it can know when to start its operation in hardware and software interaction. Thus, the behavior of resource can be consist of hardware state over hardware events and input data to software. Second, the behavior of resource in embedded system is often timed-constrained so that it sometimes requires software to finish its computation in a given time. Hardware interpreted by resource to software controls external and physical devices, such as sensor, motor, and so on. Such hardware-controlled devices often put real-time constraints upon the behavior of embedded software, and such timing-constraints is also interpreted to software via interface memory. Thus, timing-constraints interpreted by interface memory is incorporated into a property of resource. Finally, the availability of resource also restricts the behavior of embedded software. An execution of software routine can be delayed by not acquiring its necessary resource in a specific time, and resources that embedded software needs in its execution are often hardware resources, such as network packet and ports, including data from hardware. Thus, The software execution may be dependent on the availability of a resource.

To define the availability of hardware resource, we define a resource access function as follows:

Definition 4.1 Availability Function Λ :
$(S_S \cup S_{SH}) \rightarrow \lambda \times \{ACCESS, DENIED\}$, where $\lambda \in S_\Omega$

Λ is a function that is used to find out a resource's state, whether it is accessible or not by a software routine. That is, an embedded software routine can never control a resource including interface memory when the resource is locked or preempted by another software routine.

We define the behavior of resource as resource model, incorporating our du-

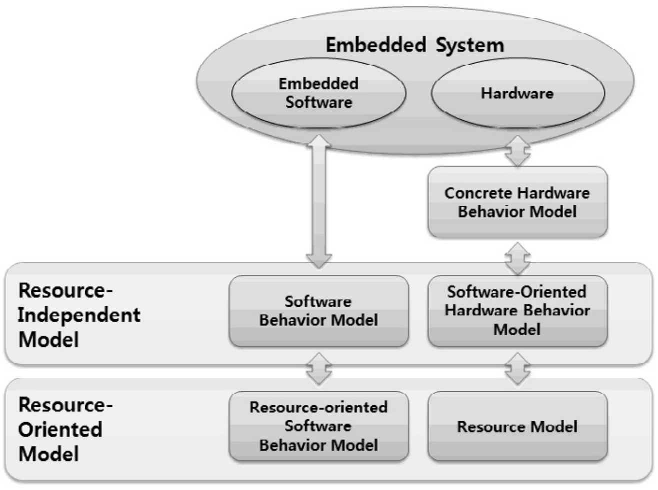

Embedded System
Embedded Software
Hardware
Concrete Hardware Behavior Model
Resource-Independent Model
Software Behavior Model
Software-Oriented Hardware Behavior Model
Resource-Oriented Model
Resource-oriented Software Behavior Model
Resource Model

Fig. 3. Resource-Based Embedded System Model

ration function and resource availability function(Λ) into our software-oriented hardware behavior model(M_{SH}).

Definition 4.2 Resource Model(M_R) is a tuple (M_{SH}, D, Λ)

In Resource Model, we can figure out the following hardware's properties related to software's functionality.

- Hardware's events that initiate embedded software routine.
- Time that is a duration when an embedded software routine can run.
- Hardware state and data when an embedded software routine can access to hardware.

5 Resource-Based Models for Embedded Software

Fig. 3 presents two embedded software models; **Resource-Independent Model(RIM)** and **Resource-Oriented Model(ROM)**[5] that are based on two hardware behavior model; software-oriented hardware model and resource model. The hardware models can become software requirements by which embedded software's behavior is designed and restricted in functionality, timing and availability aspect.

RIM is a software behavior model that is based on hardware functionality, consisting of software behavior model and software-oriented hardware model. RIM is defined as follows:

Definition 5.1 Resource-Independent Model(RIM) is a tuple (M_S, M_{SH}).

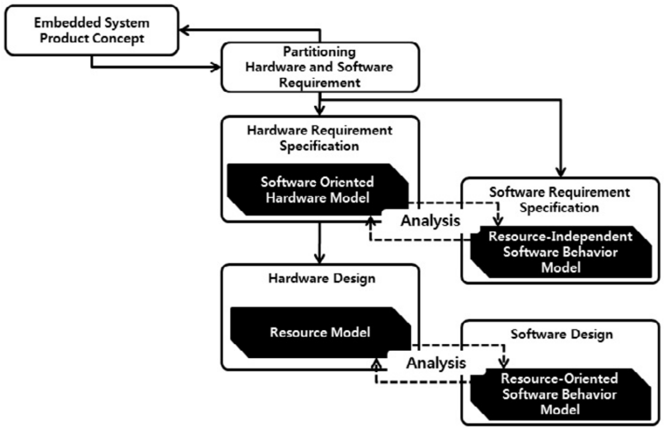

Embedded System Product Concept
Partitioning Hardware and Software Requirement
Hardware Requirement Specification
Software Oriented Hardware Model
Software Requirement Specification
Analysis
Resource-Independent Software Behavior Model
Hardware Design
Resource Model
Software Design
Analysis
Resource-Oriented Software Behavior Model

Fig. 4. Resource-Oriented Design

RIM defines the interaction and communication between hardware and software, the functionality of software, and the behavior of hardware based on the interaction. RIM does not consider the resource constraints in respect of timing and availability. The purpose of RIM is to overview embedded system behavioral requirement by capturing hardware and software interaction in their running. In embedded system development, a software-oriented hardware model is provided by hardware engineers after the hardware functional verification. And then, software engineers can construct its necessary operations into RIM with guided by the software-oriented hardware behavior model and system requirement that says software requirement. The constructed RIM is used to verify software behavior in their software behavior model in respect of functional aspects of an embedded system.

ROM is a software behavior model that is based on the resource model, and it consists of a software behavior model and a resource model for requiring the property of timing and availability for software behavior. ROM is defined as follows:

Definition 5.2 Resource-Oriented Model(ROM) is a tuple (M_S, M_R).

ROM defines software behavior restricted by hardware constraints, such as timing and availability, as a property of resource model. The software behavior in ROM can be verified against such hardware constraints. The resource model in ROM is originated from hardware engineers. They prove its timing properties using hardware timing verification techniques. And then, the hardware model can be abstracted in a software engineer view for providing hardware constraints to software engineers, and software engineers can build their software behavior model based on the software-oriented hardware behavior model and verify it against the hardware model.

Fig. 4 shows the resource-oriented development for embedded system. In this development, the requirement of embedded system is firstly partitioned into respectively hardware and software requirement. A software-oriented hardware model abstracted from hardware's requirement is constructed so as to be delivered for software requirement analysis of software engineer, in which embedded software's functionality is validated in companion with hardware's functionality for checking if embedded software's behavior is in accord with hardware's in the view of execution of their interaction and communication. In software requirement specification, there are two requirement specifications about software's behavior; software's functionality and its interaction with hardware. The requirement specification with respect to software's interaction behavior describes pre-conditions and post-conditions of software to hold true, and actions to be performed before and after the execution of hardware and software's interaction. After a hardware design completion through timing verification, a resource model abstracted from hardware design is build, and then, an embedded software model modeled in a resource-oriented software behavior model is configured to be in accord with hardware's timing and availability shown in a resource model.

6 Modeling Embedded System Components with ACSR

To show a way how to apply resource-oriented design models to the embedded system design, a construction of embedded system model using ACSR(Algebra of Communicating Shared Resource)[6] is illustrated in this section. ACSR is a timed process algebra based on the synchronization model of CCS that includes features for representing synchronization, time, temporal scopes, resource requirements, and priorities.

In resource-oriented models, there are software behavior model, software-oriented hardware model, resource model, Resource-Independent Model, and Resource-Oriented Model. The first three models are embedded system component models, and the last two models are embedded system models. To explain each property of resource-oriented models, we present an example of a signal and gate control system.

Fig.2 shows a signal and gate control system, in which the color of signal is determined by a software process and the gate is controlled by a hardware circuit. The color of signal is determined by software in a way that it starts its computation when receiving an periodic interrupt from hardware, and it cyclically turns on one of signals of red, yellow, or green. The gate is controlled by a hardware circuit according to the state of signal. The gate closes down when the red signal is turned on, and it opens up when the green signal is turned on. In an exceptional case, if an emergency signal is occurred from the environment, the red signal is forced to be turned on, and the gate must shut down regardless of whether this system is in any states.

6.1 Software Behavior Model

A software behavior model captures the behavior of embedded software consisting of software's reaction to a hardware interrupt and a sequence of software function calls during its computation.

Example 6.1 The signal system controlled by embedded software is captured with software behavior model. To simply define states of system, we use an abbreviated form, $(Var \mapsto 1)$, meaning that the only variable(Var) in a tuple is value 1 and any other variables are value 0. The software behavior model of signal system is is a tuple $(S_S, E_S^I, F_S, Q_S, \theta)$, where

· $S_S = \{(Green \mapsto 1), (YellowGreen \mapsto 1), (YellowRed \mapsto 1), (Red \mapsto 1)\}$,

· $E_S^I = \{Emergency,\ Go\}$,

· $F_S = \{ChangeYellowToGreen, ChangeGreenToYellow$
$, ChangeYellowToRed, ChangeGreenToRed$
$, ChangeRedToYellow, ErrorHandle\}$,

  · $ChangeYellowToGreen = \{(YellowGreen \mapsto 1) \rightarrow ((Green \mapsto 1), IDLE),$
  $(otherwise) \rightarrow ((otherwise), ErrorHandle)\}$,

  · $ChangeGreenToYellow = \{(Green \mapsto 1) \rightarrow ((YellowRed \mapsto 1), IDLE),$
  $(otherwise) \rightarrow ((otherwise), ErrorHandle)\}$,

  · $ChangeYellowToRed = \{(YellowRed \mapsto 1) \rightarrow ((Red \mapsto 1), IDLE),$
  $(otherwise) \rightarrow ((otherwise), ErrorHandle)\}$,

  · $ChangeRedToYellow = \{(Red \mapsto 1) \rightarrow ((YellowGreen \mapsto 1), IDLE),$
  $(otherwise) \rightarrow ((otherwise), ErrorHandle)\}$,

  · $ChangeGreenToRed = \{(Green \mapsto 1) \rightarrow ((Red \mapsto 1), IDLE),$
  $(otherwise) \rightarrow ((otherwise), ErrorHandle)\}$,

  · $ErrorHandle = \{(otherwise) \rightarrow ((Red \mapsto 1), -)\}$

· $Q_S = \{HNormalMode, HEmergencyMode, NormalMode, EmergencyMode\}$,

  · $NormalMode = \{(YellowRed \mapsto 1) \rightarrow ChangeYellowToRed,$
  $(YellowGreen \mapsto 1) \rightarrow ChangeYellowToGreen,$
  $(Red \mapsto 1) \rightarrow ChangeRedToYellow,$
  $(Green \mapsto 1) \rightarrow ChangeGreenToYellow,$
  $(otherwise) \rightarrow ErrorHandle\}$,

  · $EmergencyMode = \{(Green \mapsto 1) \rightarrow ChangeGreenToRed,$
  $(YellowGreen \mapsto 1) \rightarrow ChangeYellowToRed,$
  $(otherwise) \rightarrow ErrorHandle\}$

  · $HNormalMode \rightarrow NormalMode$,

  · $HEmergencyMode \rightarrow EmergencyMode$

· $\theta = \{Go \rightarrow HNormalMode, Emergency \rightarrow HEmergencyMode\}$

A state of software can be captured by a composition of values of software variables at a specific time. The state transition can be enabled by assigning a set of new values to software variables during its computation. However, ACSR includes no explicit syntax that is able to capture software's state and computation. By this reason, we define several event types for specifying software's state and operations.

In Table 1, the receiving of event captures a hardware interrupt occurrence. The operations of read and write of a device describe software's access to external hardware devices, such as motor, and sensor. Those operations need a particular consideration, such as hardware's timing and availability constraints, so we would capture those constraints in resource model.

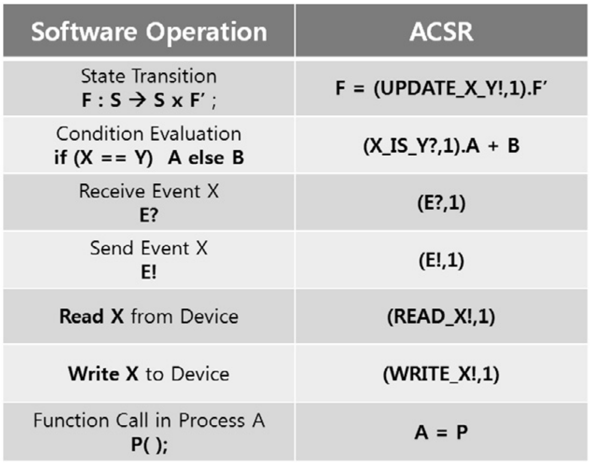
Example 6.2 Signal Control Embedded Software in ACSR

```
SBM_TCS          = (Go,1).HNormalMode
                 + (Emergency,1).HEmergencyMode;

HNormalMode      =  NormalMode;

HEmergencyMode   = EmergencyMode;

NormalMode       = (Green_IS_0,1).(YellowGreen_IS_0,1).(Red_IS_0,1).(YellowRed_IS_1,1)
                 .ChangeYellowToRed
                 + (Green_IS_0,1).(YellowGreen_IS_1,1).(Red_IS_0,1).(YellowRed_IS_0,1)
                 .ChangeYellowToGreen
                 + (Green_IS_0,1).(YellowGreen_IS_0,1).(Red_IS_1,1).(YellowRed_IS_0,1)
                 .ChangeRedToYellow
                 + (Green_IS_1,1).(YellowGreen_IS_0,1).(Red_IS_0,1).(YellowRed_IS_0,1)
                 .ChangeGreenToYellow
                 + SW_ERROR_A;
```



Software Operation	ACSR
State Transition **F : S → S x F′ ;**	**F = (UPDATE_X_Y!,1).F′**
Condition Evaluation **if (X == Y) A else B**	**(X_IS_Y?,1).A + B**
Receive Event X **E?**	**(E?,1)**
Send Event X **E!**	**(E!,1)**
Read X from Device	**(READ_X!,1)**
Write X to Device	**(WRITE_X!,1)**
Function Call in Process A **P( );**	**A = P**

Table 1
State and Operations in ACSR

```
EmergencyMode    = (Green_IS_1,1).(Yellow_IS_0,1).(Red_IS_0,1).(Pass_IS_0,1)
                 .ChangeGreenToRed
                 + SW_ERROR_A;

ChangeYellowToGreen     = ('UPDATE_Green_1,1).('UPDATE_YellowGreen_0,1)
                        .('UPDATE_Red_0,1).('UPDATE_YellowRed_0,1)
                        .SBM_TCS
                        + SW_ERROR_A;

ChangeGreenToYellow     = ('UPDATE_Green_0,1).('UPDATE_YellowGreen_1,1)
                        .('UPDATE_Red_0,1).('UPDATE_YellowRed_0,1)
                        .SBM_TCS
                        + SW_ERROR_A;
...
SW_ERROR_A       = ('SW_ERROR,1).SW_ERROR_A;
```

6.2 Software-Oriented Hardware Model

A software-oriented hardware model captures hardware's behavior in interaction with software. Therefore, it includes only information that software needs in the execution of software. A hardware component receives an event from the environment, changes its state based on states of both hardware and interface memory, and generates events to the environment and to its relevant embedded software component. The state of hardware is inherently related to the execution of embedded software. To specify a hardware component in ACSR, both states(S_{SH} and S_{Ω}) are interpreted in ACSR as follows:

First, states in the type S_{SH} are captured by the process of ACSR. For instance, states of gate can consist of opened, opening, closed, closing, and warning closing, and these states are captured with the process of ACSR. Second, the state of interface memory (S_{Ω}) is captured in the early defined software operations with the event of ACSR.

Example 6.3 Gate Control Hardware System in ACSR

```
HBM_GC       = OPENED ;

OPENED       = (tick,1).('closing,1)
             .(Green_IS_1,1).(Yellow_IS_0,1).(Red_IS_0,1).(YellowRed_IS_0,1)
             .('Close,1).('Go,1).CLOSING
             + ('HW_ERROR,1).HW_ERROR_A
             + (Emergency,1).('Close,1).('Go,1).CLOSING
             + OPENED;

CLOSING      = (down,1)
             .(Green_IS_0,1).(YellowGreen_IS_0,1).(Red_IS_0,1).(YellowRed_IS_1,1)
             .('Go,1).CLOSED
             + ('HW_ERROR,1).HW_ERROR_A
             + CLOSING;

CLOSED       = (tick,1).('opening,1)
             .(Green_IS_0,1).(YellowGreen_IS_0,1).(Red_IS_1,1).(YellowRed_IS_0,1)
             .('Open,1).('Go,1).OPENING
             + ('HW_ERROR,1).HW_ERROR_A
             + CLOSED;

OPENING      = (up,1)
```

```
              .(Green_IS_0,1).(YellowGreen_IS_1,1).(Red_IS_0,1).(YellowRed_IS_0,1)
              .('Go,1).OPENED
              + ('HW_ERROR,1).HW_ERROR_A
              + (Emergency,1).('Close,1).('Go,1).CLOSING
              + OPENING;

GATE_SENSOR = (opening,1).('up,1).GATE_SENSOR
              + (closing,1).('down,1).GATE_SENSOR
              + GATE_SENSOR;
```

In Example 6.3, The functionality of gate control system is captured in ACSR. The gate starts its closing operation when it is opened($OPENED$) and receives a signal tick from a timer. If the signal of green is on and the others are off, it continues its actual closing operation with sending a signal GO to the signal control software($('Go, 1).CLOSING$).

6.3 Resource Model

A resource model in resource-oriented design captures a hardware behavior including hardware constraints put upon embedded software components. The constraints we include into a resource model are timing and availability in the use of hardware.

The model of ACSR corresponding to a resource model would capture not only the behavior of a hardware component but also explicitly aspects of time and availability of it. An action in ACSR represents the consumption of named resource for one time unit because the execution of an action is subject to the availability of the named resources and the contention for resource is arbitrated according to the priorities of competing actions[6]. The action $\emptyset$ represents the passage of one time unit without consuming any resources, that is, idling for one time unit. However, ACSR does not provide a specific way to describe an exclusive use of a resource without consuming time. A time behavior in ACSR is a sequence of actions, in which a sequence of events may appear between any two actions, but event's actions consume no time. In short, ACSR provides no way to capture an action of resource without consuming time because ACSR is designed to describe an exclusive use of a resource that must be in companion with a timing consuming. **The resource-oriented design we discuss here pursues a modeling of a resource including more explicit and specific behavior of a hardware component.** That is, only an exclusive use of a resource in modeling a resource can be captured in the specification of ACSR. Meanwhile, the resource-oriented design pursues capturing not only the exclusive use of a resource but also a specific impact of resource behavior on the behavior of software.

Example 6.4 A Resource Model for a Gate System in ACSR

```
TIMER   = ('tick,1).{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:TIMER + {}:TIMER;
```

```
GATE_SENSOR
        = (closing,1).{}:{}:{}:('down,1).GATE_SENSOR
        + (opening,1).{}:{}:{}:('up,1).GATE_SENSOR
        + {}:GATE_SENSOR;
...
...

Green_0 = ('ACCESS_Green,1).('Green_IS_0,1).Green_0
        + ('ACCESS_Green,1).(UPDATE_Green_1,1).Green_1 + {}:Green_0 ;

Green_1 = ('ACCESS_Green,1).('Green_IS_1,1).Green_1
        + ('ACCESS_Green,1).(UPDATE_Green_0,1).Green_0 + {}:Green_1 ;
```

In Example 6.4, two aspects of a hardware component are highlighted by a resource model; a time consuming of an hardware's activity and an access to a hardware component. For instance, $TIMER$ occurs a signal tick every ten time-unit. The gate takes three time-unit to complete its opening or closing after its operation starts. The shared memory indicating the color of signal is protected by an access control using a signal $ACCESS\_Green$, so an software component can refer or update the variable of signal color only when a shared hardware component permits its accessibility to be given to the software component.

The last two embedded system models, RIM and ROM, are constructed by a composing basic models embedded system component. A software behavior model and a software-oriented hardware model compose a RIM. After that, the software-oriented hardware model is configured in order to include hardware constraints, such as timing and availability of a hardware component interacting with software components. The configured software-oriented hardware model is a resource model we present here, and such a resource model is used for software engineers to construct a detailed design of embedded software.

7 Formal Verification in ACSR

The model of ACSR can be verified with VERSA[1]. VERSA is a tool that assists in the algebraic analysis of real-time specification written in ACSR. VERSA can also to be used to analyze ACSR. An important technique that VERSA supports is an equivalence checking of ACSR process. In ACSR, if a design specification is equivalent to its requirement specification, then the design specification can be considered correct. The property to be checked in RIM is whether there is no contradiction in the interaction between hardware and software in terms of functionality. ROM, in addition, is verified in aspects of a timed behavior restricted in using a limited resource.

RIM can be formally verified with the following verification property.

$$SYSTEM \approx_\pi \tau^\infty$$

The τ^{∞} means that it consumes no time and resources, so the question means that the system constantly is bisimilar[9] to an infinite sequence of idle action τ^{∞} without the consuming of time and the use of resource. If VERSA says "yes", a modeled system always succeeds synchronization correctly using their communication events. In other words, the functionality of a system in an interaction of hardware and software is correct.

ROM can be formally verified with the following verification property.

$$SYSTEM \approx_{\pi} \emptyset^{\infty}$$

The $\emptyset^{\infty}$ indicates that a system consumes time in using a resource. Models in a ROM are captured with timing and availability constraints so a verification property for a ROM also asks a question if there is no contradiction in an interaction of hardware and software not only consuming time but also using a limited resource.

8 Conclusions

A resource in computer system is a classical notion to abstract a component used in software computation. An embedded system is often limited in using a resource so that a system requirement includes a specification with respect to such limited resources.

In this paper, we present an embedded system design framework, called resource-oriented design, in which an embedded system is captured in the view of hardware behavior and constraints. This paper presents three embedded system component models, software behavior model, software-oriented model, and resource model. The software behavior model capture the behavior of a software component in respect of its functionality in an interaction of hardware and software software. The software-oriented model abstracts hardware's behavior in the view of software' behavior, and the resource model incorporates hardware resource constraints, such as timing and availability. For a system view, such embedded system component models can be composed into RIM and ROM. Using RIM, each component of embedded system is verified against the functionality of embedded system. Using ROM, aspects of timing and availability can be verified in the execution of embedded system.

Using resource-oriented design, an embedded system can be developed gradually. In embedded system industry, hardware is first developed, and embedded software is implemented on the hardware. Moreover, software components are necessarily reused in other hardware platforms. Thus, the reused embedded software components need to be designed independently from hardware platform. By means of a software behavior model that resource-oriented design provides, an embedded software component in requirement can be analyzed without a complete hardware model and can be captured to be reused over various hardware platforms. And the software behavior model is formally analyzed in companion with hardware behavior model by RIM in the view of functionality.

A requirement of embedded software needs to be developed into a detailed software component design. The resource-oriented design provides a detailed software component model that is originated and extended directly from a software requirement models by evolving a software behavior model based on an hardware-constraint model, called resource model. Moreover, a detailed embedded system model, ROM, provides a framework in which a detailed software design can be analyzed in companion with hardware constraints captured in a resource model.

Currently, we have an investigation on a formal language that is suited for the resource-oriented design.

References

[1] D. Clarke, I. Lee, and H. Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems, 1995.

[2] Bruce Powel Douglass. *Real-time UML (2nd ed.): developing efficient objects for embedded systems*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.

[3] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proc. of the IEEE*, 85(3), 1997.

[4] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[5] Jin Hyun Kim and Jin-Young Choi. Embedded system modeling based on resource-oriented model. In *Proceddings of IEEE Engineering of Computer Based Systems,*, 2007.

[6] Insup Lee, Hanene Ben-Abdallah, and Jin-Young Choi. *FORMAL METHODS FOR REAL-TIME COMPUTING*, chapter 7:A Process Algebraic Method for Real-Time Systems. Trends in Software. John Wiley & Sons Ltd, 1996.

[7] Insup Lee, Anna Philippou, and Oleg Sokolsky. Resources in process algebra. *Journal of Logic and Algebraic Programming*, 72(1):98–122, May 2007.

[8] Alberto Sangiovanni-Vincentelli Marco Sgroi, Luciano Lavagno. Formal models for embedded system design. *IEEE Design and Test of Computers*, 17(2):14–27, June 2000.

[9] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[10] Oleg Sokolsky. Resource modeling for embedded systems design. In *WSTFEUS*, pages 99–103, 2004.

[11] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C.Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms.* Pearson, Prentice Hall, 1988.