

An Extended TASM-Based Requirements Modeling Approach for Real-Time Embedded Software: An Industrial Case Study

Jin-Hui Shan¹(✉), Hai-Yan Zhao^{2,3}, Jin-Bo Wang¹, Rui-Xue Wang¹,
Cheng-Lin Ruan¹, and Zhe-Xi Yao¹

¹ Technology and Engineering Center for Space Utilization,
Chinese Academy of Sciences, Beijing 100094, China
shanjh@163.com,

{wangjinbo,snow.ruixue,ruanchenglin14,yaozheqian14}@csu.ac.cn

² Key Laboratory of High Confidence Software Technologies,
Ministry of Education, Beijing 100871, China

³ Institute of Software, School of EECS, Peking University, Beijing 100871, China
zhhy.sei@pku.edu.cn

Abstract. Formal models of software requirements facilitate automated analysis and verification of the models, and software testing as well. A challenging problem in requirements modeling for real-time embedded software is to devise an easy-to-use formal requirements modeling language and a corresponding requirements modeling approach. Timed Abstract State Machine, abbreviated as TASM, is an easy-to-use formal requirements modeling language, but it has its limitations. We have extended TASM. In this paper, we propose an extended TASM-based requirements modeling approach for real-time embedded software to identify the parallel and hierarchical components, and describe the functional and non-functional requirements of the software. We apply the proposed approach to model requirements of actual real-time embedded software to validate its effectiveness. Our study highlights several lessons for modeling requirements of real-time embedded software in the extended TASM.

1 Introduction

Embedded systems have been applied widely, and usually have to respect the real-time deadlines: the result of the computation should be correct, and the time taken to produce the result should conform to the requirement. Embedded software plays an important role in the embedded systems.

Software requirements are statements on the capabilities, features, or qualities of the software systems. Software requirements make the software systems usable or valuable to the customers [1]. And software requirements are the foundations of software design and software testing.

The requirement models for real-time embedded software fall into three categories: informal, semiformal and formal models.

One may start to specify software requirements in natural language. These informal requirement models have the advantage of lower cost of modeling, and the disadvantages of being prone to cause ambiguities and lose requirements.

Semiformal requirement models include data flow diagrams [2], Unified Modeling Language (abbreviated as UML) diagrams [3] and their extended models named MARTE (Modeling and Analysis of Real-Time and Embedded systems) [4], etc. Semiformal requirement models have the advantages that they are intuitive and easy-to-understand, and the disadvantage that it is difficult to be applied to formal analysis and verification [5].

Formal requirement models include Timed Automata (abbreviated as TA) [6] and its extended models named Uppaal TA (abbreviated as UTA) [7, 8], Abstract State Machine (abbreviated as ASM) [9] and its extended models named Timed ASM (abbreviated as TASM) [10–13], Petri Networks [9], SPARDL [14], and the extended State Transition Matrix (abbreviated as STM) [15], etc. Some time-dependent properties are expressed in Interval Temporal Logic (abbreviated as ITL) [16] formulas in [14]. And some time-dependent properties are expressed in Timed Computation Tree Logic (abbreviated as TCTL) [17] formulas in [15].

ASM is a kind of formal method for specifying software requirements [9]. ASM has the features of small size of syntax, simple semantics and ease of understanding and use [10]. TASM proposed in [10, 11] uses a subset of ASM and extends ASM with time and resource utilization. The safety and liveness properties of TASM specifications are expressed using a subset of TCTL in [10].

Formal requirement models have the advantages that they have precise mathematical specification, can be applied to formal analysis and verification directly, and facilitate automated generation of test cases [10]; on the other hand, they have the disadvantages that many of the formal requirement models can be used only by those who have extensive mathematical training. Many computer scientists, including Turing Award Winner Pnueli, consider that modeling and verification software in a formal way is one of the important techniques to assure the quality of software [18].

Although there are a variety of formal requirements modeling languages, how to construct requirement models is still an open problem [1]. Therefore, it is a challenge to devise an easy-to-use formal requirements modeling language and a corresponding requirements modeling approach.

Based on modeling requirements in terms of rules, TASM is an easy-to-use formal requirements modeling language. TASM provides a description for behavior of time, resources, synchronization, and concurrency, etc. However, TASM has some limitations: (1) Data types supported in TASM are limited, e.g., array is not supported in TASM. (2) The rule supporting iterative execution of actions is not supported in TASM [11]. And (3) operators supported in TASM are limited. For instance, “%” and some common binary operators in real-time embedded software are not supported in TASM. And no requirements modeling approach for real-time embedded software is proposed in [10, 11].

TASM is extended with event construct and observer construct, and a requirements modeling approach for real-time embedded software is proposed

in [12]. This approach comprises requirements preprocessing, components identification, connections identification, behavior specification and property annotation. The main machines identified only for the internal and external components are maybe not complete in [12]. Functional requirements are described in behavior specification, and property annotation adds timing and resource consumption annotations to the relevant TASM model [12]. In fact, functional requirements often involve time and resources in real-time embedded software.

We have extended TASM with the data type of arrays, a loop rule named **while**, and operators such as “%”, etc. On the basis of the extension, we propose an extended TASM-based requirements modeling approach which is applied to model requirements of actual real-time embedded software.

The main contributions of our research are as follows: (1) we propose an extended TASM-based requirements modeling approach for real-time embedded software to identify the parallel and hierarchical components, and describe the functional and non-functional requirements of the software. (2) We apply the proposed approach to actual real-time embedded software to demonstrate its effectiveness for requirements modeling. And (3) several lessons learned from our study for modeling requirements of real-time embedded software in the extended TASM are highlighted.

The remainder of this paper is organized as follows: TASM is introduced and extended in Sect. 2. An extended TASM-based requirements modeling approach is proposed in Sect. 3. The proposed approach is applied to model requirements of actual real-time embedded software in Sect. 4. Section 5 discusses the related work. Finally, Sect. 6 summarizes the paper.

2 Preliminaries

2.1 TASM

Definition 1. A *TASM specification* [10,11,13] is a pair $\langle E, ASM_s \rangle$ where:

- (1) $E = \langle EV, TU, ERs \rangle$ is the environment where EV is the set of environment variables whose type definitions are in TU including **Integer**, **Float**, **Boolean**, and user-defined types. ERs consist of zero or more ERs . Each ER is a definition of a resource variable of the form “ $rn := rs$,” where rn is the name of the resource, and rs is the limits of the resource. Examples of resources include processor, memory, communication bandwidth, power, etc.
- (2) ASM_s denote one or more machines. Each machine is a tuple $ASM = \langle McnName, MV, CV, IV, Rules \rangle$ where $McnName$ is the name of the machine. MV is a set of monitored variables which are the set of read-only environment variables that affect the execution of the machine. CV is a set of controlled variables which are the set of environment variables that the machine updates. IV is a set of internal variables whose scope is limited to the machine where they are defined. $Rules$ consist of one or more rules. Each rule is a tuple $Rule = \langle RuleName, T, RRs, r \rangle$ where $RuleName$ is the name of the rule. T is the duration of the rule execution. And T can be a

single value, an interval of the form $[tmin, tmax]$, or the keyword **next** which essentially states that the time elapses until an event of interest occurs. The T component can be simply absent also. RRs consist of zero or more RRs . Each RR is resource utilization during the rule execution with the form of “ $rn := RS;$ ” where RS is a constant. r is a concrete rule of the form “**if** $Condition$ **then** $Actions$ ” where $Condition$ is a conditional expression, and $Actions$ consist of one or more $Actions$. Each $Action$ is an update of a controlled variable, or the null operation “**skip**”; etc. We can also use the concrete rule “**else then** $Actions$ ”.

The concepts of parallel composition, hierarchical composition, and synchronous communication are supported in TASM. Accordingly, three kinds of machines are provided in TASM, i.e., main machine, sub-machine, and function-machine.

The main machines are executed in parallel. A main machine may contain sub-machines. A sub-machine may contain its sub-machines further. A sub-machine call can only be allowed in an $Action$. A function-machine cannot modify the environment and must derive its output solely from its inputs. The only side-effect of a function-machine is time and resource utilization. A function-machine call may appear in a $Condition$ or an $Action$.

The use of the keyword “**next**” is forbidden in both sub-machines and function-machines. Communication between different main machines is achieved only via shared variables in TASM [10].

2.2 Extensions to TASM

We have extended TASM with the data type of arrays, the loop rule named **while**, and some operators such as “%”, “&”, “|”, “^”, “>>”, “<<”, etc.

For example, there is a requirement in some embedded software: when the software receives a data packet, if the length of the data packet is less than desired (say 144 bytes), then the spare space within the data packet is padded with the fill character 0xBB. The snippets of a rule for padding fill characters are shown in Fig. 1.

The rule named **while** is used in the R1 rule in Fig. 1. The Equipment1 data packet is stored in an array in this rule. It is difficult to express data packet whose length is 144 bytes in TASM. More than one hundred rules are needed to express padding with different number of fill characters according to different length of the data packet received in this machine in TASM.

```
R1: Number of fill chars for Equipment1 data packet greater than 0 {
  while NO_Fill_Char_Equip1_Data_Pack > 0
    Equip1_Data_Pack[144-NO_Fill_Char_Equip1_Data_Pack+1]:=0xBB;
    NO_Fill_Char_Equip1_Data_Pack:=NO_Fill_Char_Equip1_Data_Pack-1;
}
```

Fig. 1. Snippets of sample rule named **while** and the data type of arrays

The syntax and semantics of TASM are defined in [10]. The main focuses of this paper are the requirements modeling approach for real-time embedded software and the industrial case study. And we will define the syntax and semantics of the extended TASM in another paper.

3 An Extended TASM-Based Requirements Modeling Approach

Requirements modeling approach tells us what to do and in what order to do it [2]. We propose an extended TASM-based requirements modeling approach for real-time embedded software, which constructs the corresponding extended TASM models by machines and environment according to the requirements described in natural language. As depicted in Fig. 2, this approach consists of the following steps: (1) identify the parallel components to establish the main machines; (2) identify the hierarchical components to establish the sub-machines and function-machines; (3) specify the functional requirements; (4) specify the non-functional requirements; and (5) execute the above four steps iteratively if needed. Environment is constructed in step (3), while machines are constructed in the first four steps.

(1) Identifying parallel components to establish main machines, which distinguishes into the following four cases: (a) Identifying component: Each component independent of the other components and being executed in parallel with the other components can be identified as a main machine. There are two kinds of components, namely, the external components that interact with the system and the internal components that compose of the system [12]. (b) Identifying complex interface: The processing function of an interface in a component can be identified as a main machine, if it is complex and can be executed in parallel with the other works of the component. (c) Identifying complex work in a component: If a piece of independent work in a component is complex and can be executed in parallel with the others, it can be identified as a main machine. And (d) identifying timer: Real-time embedded software often performs tasks at fixed time. These tasks can be fulfilled with the aid of timers. Each timer independent of the components and advancing in parallel with the time of the components can be identified as a main machine.

If all the function of a main machine obtained by component identification has been decomposed into several main machines identified by complex interface and

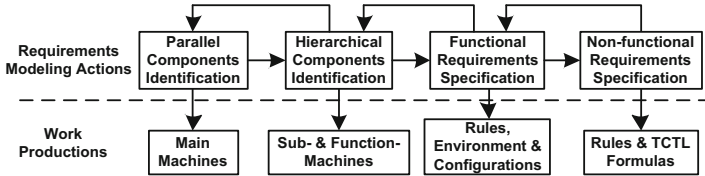


Fig. 2. An extended TASM-based requirements modeling approach

complex work in a component, then the main machine obtained by component identification can be removed.

Although some component may be independent of the other components, it is only used to save data, and never sends any message actively to the other components. If the interface processing function of this component is performed in the other main machines, then the main machine obtained by identifying this component can be removed.

(2) Identifying hierarchical components to establish sub-machines and function-machines.

Describing the requirements by a main machine should keep the function of the main machine simple and clear, and relatively complete. When the function of a main machine is relatively complicated, the “divide and conquer” and “stepwise refinement” approaches of requirements modeling should be taken to divide the main machine into sub-machines. Function-machines should be established as needed. If a function is frequently used by the other machines, and has only one return value, then a sub-machine for this function can be established.

The above principle is suitable for dividing a sub-machine into sub-machines and function-machines too.

(3) Specifying functional requirements.

Functional requirements capture requirements of system function and behavior [1]. Machine rules in the extended TASM can be used to model the functional requirements. Functional requirements related to time and resource utilization can also be modeled with machine rules in the extended TASM.

When the functional requirements are modeled with machine rules, controlled variables, etc., are used to represent the possible states of the system. The conditions of machine rules are given according to the corresponding values of the state variables and the transition conditions. The actions of machine rules are specified based on the behaviors of a component and the next possible state [12].

Environment and configurations are defined during modeling the functional requirements in the extended TASM. The environment comprises: (a) the definitions of the user-defined types; (b) the definitions of the resource variables; and (c) the type declarations and initializations of the environment variables. If there are several initial running states for a system, then a configuration is defined for each initial running state [10]. The configuration can contain initial values for the environment variables. The initial values specified in a configuration override the initial values of environment variables specified in the environment.

(4) Specifying non-functional requirements.

Non-functional requirements describe those requirements not belonging to functional requirements, typically including constraints and limitations attached to the system services [1]. The non-functional requirements of real-time embedded software include performance, safety properties, and liveness properties, etc. Performance requirements can be modeled with machine rules in the extended TASM with the help of time and resource utilization. Moreover, the other non-functional requirements can be described by timed logic formulas such as TCTL [17].

(5) Iterative execution of the above four steps if needed.

One can see from Fig. 2 that modeling requirements of real-time embedded software in the extended TASM is a procedure with information feedback, and should be iterated if needed.

4 An Experimental Study

In this section, we apply the extended TASM-based requirements modeling approach to model requirements of actual real-time embedded software to validate the effectiveness of the approach.

We adopt an actual real-time embedded software in the experiments, namely, the Experimental Handling Unit (abbreviated as EHU) main control software.

4.1 Introduction to the Experimental Project

EHU receives time code, data injection commands, and requests for digital telemetric data, etc., from the platform system, interprets and executes the data injection commands, is in charge of power distribution, observation and control, communication, and data management of the two pieces of equipment performing experiments, and packages the data packets transmitted to the platform system. Duplex pieces of equipment are adopted to improve the reliability.

The names of the external interfaces in the EHU main control software include 1394, RS422, program-controlled instruction, Static Random Access Memory (SRAM, for short), analog/digital (A/D, for short), Electrically Erasable Programmable Read-Only Memory (EEPROM, for short), reset, etc. The interface relation diagram of the EHU main control software is depicted in Fig. 3.

Time code is broadcasted periodically for EHU by the platform system to calibrate the time of EHU. And the platform system sends requests for digital telemetric data to EHU periodically, and sends data injection commands to EHU nonregularly. EHU forwards some data injection commands to the two pieces of equipment. Time code is broadcasted periodically for the two pieces of equipment by EHU. And EHU sends requests for data packet to the two

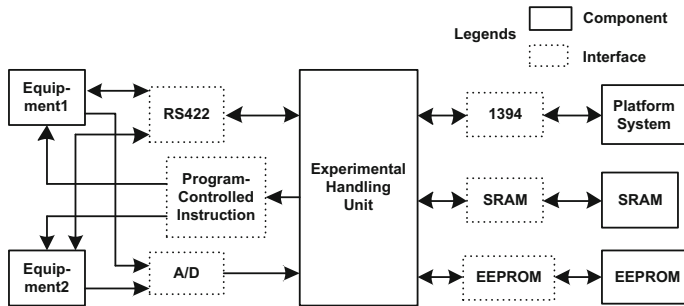


Fig. 3. Interface relation diagram of the EHU main control software

pieces of equipment periodically. When a piece of equipment receives a request for data packet, a data packet is transmitted to EHU in the specified time. And EHU receives data packets from the two pieces of equipment, which are processed then to create 1394 isochronous packets to be saved in the SRAM; collects analogue signals from itself and the two pieces of equipment, which are converted into digital data then to create digital telemetric packets; returns the 1394 isochronous packets and digital telemetric packets to the platform system; sends program-controlled instructions to the two pieces of equipment through the program-controlled instruction interface; saves main/spare piece of equipment status in the EEPROM; and sends signals of feeding dog, which are pulses of positive and negative alternating, constantly to the reset chip. If the reset chip has not received signal of feeding dog for a certain time, it sends a reset signal to EHU, which is called reset by watchdog. In addition, there are two more reset modes for EHU, namely, power on reset and reset by the command sent by the platform system. When EHU is reset, the main/spare piece of equipment status is restored according to the status saved in the EEPROM.

4.2 Identifying Parallel Components to Establish Main Machines

Main machines are established by identifying the parallel components according to the four cases in the proposed requirements modeling approach.

For the EHU main control software, the names of the main machines obtained by component identification include: EHU, Equipment1, Equipment2, the platform system, SRAM, EEPROM, and the reset chip; the names of the main machines obtained by identifying complex interface include: “1394 interface processing”, “RS422 interface processing”, and “reset interface processing”; the name of the main machine obtained by identifying complex work in a component is “event execution”; and the names of the main machines obtained by timer identification include: “broadcasting time code for equipment timer”, “sending request for data packet to Equipment1 timer”, “receiving data packet from Equipment1 timer”, “sending request for data packet to Equipment2 timer”, “receiving data packet from Equipment2 timer”, “broadcasting time code for EHU timer”, and “sending request for telemetering digital data to EHU timer”.

EHU per se is of complex function which has been decomposed into the main machines named “1394 interface processing”, “RS422 interface processing”, and “event execution”. Therefore the main machine named EHU can be removed.

The processing function of the EEPROM interface is performed in the main machines named “event execution” and “reset interface processing”. And the processing function of the SRAM interface is performed in the main machines named “1394 interface processing” and “RS422 interface processing”. Thus the main machines named SRAM and EEPROM can be removed.

The processing function of the A/D interface is performed in the main machine named “1394 interface processing”. And the processing function of the program-controlled instruction interface is performed in the main machine named “event execution”.

4.3 Identifying Hierarchical Components to Establish Sub-machines and Function-Machines

Sub-machines and function-machines are established by identifying the hierarchical components.

The names of the sub-machines contained *directly* by the main machine named “RS422 interface processing” include “judging and sending request for data packet to Equipment1”, “judging and sending request for data packet to Equipment2”, “judging and receiving data packet from Equipment1”, “judging and receiving data packet from Equipment2”, “saving EHU isochronous packets”, “forwarding data injection command to Equipment1”, “forwarding data injection command to Equipment2”, and “judging and broadcasting time code for equipment” in the EHU main control software.

The names of the sub-machines contained *directly* by the main machine named “1394 interface processing” include “judging and sending 1394 isochronous packets” and “receiving 1394 asynchronous packets”. The sub-machine named “receiving 1394 asynchronous packets” contains the function-machine named “judging validity of the checksum of data injection command”.

The other main machines and sub-machines in the EHU main control software can be divided into sub-machines and function-machines further as needed.

We do not list all the names of the sub-machines and function-machines in the EHU main control software here due to the space limitation.

4.4 Specifying Functional Requirements

Modeling Timer and the Elapse of Time. A single timer is modeled in TASM in Fig. 8 in [19]. We can model the functional requirement “EHU broadcasts time

```

R1: broadcasting time code for equipment timer
  timeout {
  if (BRDCST_time_code equip_Timer >= 59*1000) and
  (BRDCST_time_code equip_Timeout_Flag = 0) then
    BRDCST_time_code equip_Timeout_Flag := 1;
    BRDCST_time_code equip_Timer := 0;
  }
R2: broadcasting time code for equipment timer
  not timeout {
  t := 1; // the time duration is 1ms
  if (BRDCST_time_code equip_Timer < 59*1000) and
  (BRDCST_time_code equip_Timeout_Flag = 0) then
    BRDCST_time_code equip_Timer :=
    BRDCST_time_code equip_Timer + 1;
  }
R3: broadcasting time code for equipment timer
  reset {
  t := next;
  else then
    BRDCST_time_code equip_Timer := 0;
  }

```

Fig. 4. Snippets of the rules in the main machine named “broadcasting time code for equipment timer”

```

R1: broadcasting time code for equipment
  if BRDCST_time_code equip_Timeout_Flag = 1
  then
    Broadcast_Time_Code_for_Equipment1();
    Broadcast_Time_Code_for_Equipment2();
    BRDCST_time_code equip_Timeout_Flag := 0;
  }
R2: not broadcasting time code for
  equipment
  else then
    skip;
  }

```

Fig. 5. Snippets of the rules in the sub-machine named “judging and broadcasting time code for equipment”

code for the two pieces of equipment every $1\text{ min} \pm 1\text{ s}$ in the extended TASM with the aid of timer. And the snippets of the rules in the main machine named “broadcasting time code for equipment timer” are shown in Fig. 4.

The values of the `BRDCST_time_code_equip_Timer` and `BRDCST_time_code_equip_Timeout_Flag` variables are set to 0 at system startup time. Therefore the R2 rule in Fig. 4 is enabled continually, and the value of this timer begins to increase progressively.

When the value of this timer is not less than 59000, the R1 rule is enabled, the value of the `BRDCST_time_code_equip_Timeout_Flag` variable is set to 1, and the value of this timer is cleared. Since then the R3 rule keeps on being enabled to keep the main machine named “broadcasting time code for equipment timer” “live” until the state is updated; for instance, the value of the `BRDCST_time_code_equip_Timeout_Flag` variable is cleared by one rule in the sub-machine named “judging and broadcasting time code for equipment” in the main machine named “RS422 interface processing” in Fig. 5. Then the value of this timer is cleared, and the R2 rule in Fig. 4 is enabled continually again, and the value of this timer begins to increase progressively again.

This sample demonstrates that TASM is capable of modeling timer and the elapse of time.

The main machine named “broadcasting time code for equipment timer” and the sub-machine named “judging and broadcasting time code for equipment” work in concert in Figs. 4 and 5. Moreover, the principle in Figs. 4 and 5 is suitable for modeling two timers working in concert. For example, EHU sends a request for data packet to a piece of equipment every 600 ms, and receives one data packet from the piece of equipment 200 ms later than the request. Thus one timer for sending request for data packet and one timer for receiving data packet are set for every piece of equipment. Both of these two timers are main machines. When a timeout of a timer for sending request for data packet occurs, EHU sends a request for data packet, set the flag of allowing for receiving the data packet, and starts the timing of the timer for receiving the data packet.

Defining Environment and Configurations. EHU saves 1394 isochronous packets in the SRAM, and the size of each packet is 160 bytes. Thus the capability of the SRAM is a kind of resource in this software. The resource variable named `SRAMUsed` is defined in the environment as “`SRAMUsed:= [0, 2*1024*1024];`”.

A configuration is defined for each of the three reset modes. For example, the `Reset_by_Watchdog_Flag` variable is set to 1 in the configuration for reset by watchdog. And the initializing actions for reset by watchdog will be taken when the *Condition* of “`Reset_by_Watchdog_Flag = 1`” in the rule in the main machine named “reset interface processing” is true.

Modeling Resource Utilization. 160 bytes SRAM is consumed every time one packet is saved in the SRAM. The snippets of one rule in the sub-machine named “Equipment1 saving packets” in the main machine named “RS422 interface processing” are shown in Fig. 6. The “%” operator is used in this rule.

```

R1: SRAM not overflow after saving one
    Equipment1 isochronous packet {
SRAMUsed := 160; // resource consumption
if ( Comm_Status_Equip1 = 1 ) and
  ( SRAM_Overflow_Status = 0 ) and
  (( SRAM_Write_Ptr + 1 ) % SRAM_Length
   != SRAM_Read_Ptr ) then
  SRAM_Isochronous_Packet_Buf[SRAM_Write_Ptr]
  := Current_Isochronous_Packet_Equip1;
  SRAM_Write_Ptr :=
    ( SRAM_Write_Ptr + 1 ) % SRAM_Length;
}

```

Fig. 6. Snippets of one rule in the sub-machine named “Equipment1 saving packets”

```

R1: 1394 Isochronous channel active and some
    1394 isochronous packets in SRAM {
SRAMUsed := -160; // resource returning
if ( Isochronous_Chnl_Active = 1 ) and
  ( SRAM_Read_Ptr != SRAM_Write_Ptr ) and
  ( Isochronous_Buf_Full = 0 ) then
  Isochronous_Buf :=
    SRAM_Isochronous_Packet_Buf[SRAM_Read_Ptr];
  Isochronous_Buf_Full := 1;
  SRAM_Read_Ptr :=
    ( SRAM_Read_Ptr + 1 ) % SRAM_Length;
  Send_1394_Isochronous_Packet();
}

```

Fig. 7. Snippets of one rule in the sub-machine named “judging and sending 1394 isochronous packets”

When the 1394 isochronous channel is active, and there are some 1394 isochronous packets in the SRAM, EHU picks up one 1394 isochronous packet and sends it to the platform system. The snippets of one rule in the sub-machine named “judging and sending 1394 isochronous packets” are shown in Fig. 7.

The value of the resource variable named `SRAMUsed` is `-160` in Fig. 7, which means that when the R1 rule in Fig. 7 is enabled, 160 bytes SRAM is returned back. And Figs. 6 and 7 demonstrate the modeling method of resource utilization in the extended TASM.

The other functional requirements of the EHU main control software can be modeled with machine rules in the extended TASM.

4.5 Specifying Non-functional Requirements

The modeling method of the performance requirement that “the cycle of broadcasting time code for the two pieces of equipment by EHU shall be $1\text{ min} \pm 1\text{ s}$ ” is as follows: (1) Define the user-defined data type in the environment as “`Error_Cycle := {none, invalid_cycle};`” where `invalid_cycle` denotes that the cycle of broadcasting time code for the two pieces of equipment is invalid. (2) Whenever time code is broadcasted for the two pieces of equipment, record the time of broadcasting this time and the last time with two variables, respectively. (3) The following rule is added to the sub-machine named “broadcasting time code for the two pieces of equipment”:

```

if (Current_Time_BRDCST_Time_Equip-Last_Time_BRDCST_Time_Equip < 59*1000) or
  (Current_Time_BRDCST_Time_Equip-Last_Time_BRDCST_Time_Equip > 61*1000) then
  Error_Time_Code_Cycle := invalid_cycle;

```

where the data type of the `Error_Time_Code_Cycle` variable is `Error_Cycle`. And (4) this performance requirement is expressed in a temporal logic formula: $AG(\text{Error_Time_Code_Cycle} \neq \text{invalid_cycle})$ which states that the `Error_Time_Code_Cycle` variable is never set to `invalid_cycle`, i.e., the performance requirement is met by the extended TASM model. This performance requirement is also a kind of safety property.

```

R1: time interval for feeding dog greater than 1.6 seconds leading to reset by watchdog {
if This_Time_Feed_Dog - Last_Time_Feed_Dog > 1600 then
  Reset_by_Watchdog_Flag := 1;
}
R2: time interval for feeding dog not greater than 1.6 seconds and succeed in feeding dog {
if ( This_Time_Feed_Dog - Last_Time_Feed_Dog <= 1600 ) and
  ( This_Feed_Dog_State != Last_Feed_Dog_State ) then
  Last_Time_Feed_Dog := This_Time_Feed_Dog;
  Last_Feed_Dog_State := This_Feed_Dog_State;
}

```

Fig. 8. Snippets of the rules in the main machine named “the reset chip”

The timeout for reset by watchdog is 1.6s, i.e., if the reset chip has not received signal of feeding dog for 1.6s, it sends a reset signal to EHU. The snippets of the rules in the main machine named “the reset chip” are shown in Fig. 8.

Therefore, the safety property that “the time interval for feeding dog shall not be more than 1.6s” can be expressed in a temporal logic formula: $AG(Reset_by_Watchdog_Flag \neq 1)$. If this formula is not valid, then the safety property is not met by the extended TASM model. And the liveness property that “once the time interval for feeding dog is greater than 1.6s, then the reset chip sends a reset signal to EHU” can be expressed in a temporal logic formula: $AG((This_Time_Feed_Dog - Last_Time_Feed_Dog > 1600) \rightarrow Reset_by_Watchdog_Flag = 1)$.

The modeling method of the performance requirement “the time for caching engineering data packets of the two pieces of equipment shall not be less than 20min” is as follows: (1) Record the time when the SRAM is empty using the keyword **now** that returns a value denoting the current time [10] as “SRAM_empty_time:=now;” at system startup time. (2) When a 1394 isochronous packet is saved in the SRAM, add resource consumption as “SRAMUsed:=160;”, and judge whether the SRAM is full. If so, record the time when the SRAM is full as “SRAM_full_time:=now;”. (3) When a 1394 isochronous packet is removed from the SRAM, add resource returning as “SRAMUsed:=-160;”, and judge whether the SRAM is empty. If so, record the time when the SRAM is empty as “SRAM_empty_time:=now;”. And (4) the performance requirement that “the time for caching engineering data packets of the two pieces of equipment shall not be less than 20min” is expressed in a temporal logic formula: $AG((SRAM_full_time - SRAM_empty_time) \geq 20 * 60 * 1000)$. If this formula is not valid, then the performance requirement is not met by the extended TASM model.

The other non-functional requirements of the EHU main control software can be modeled with machine rules in the extended TASM and temporal logic formulas.

4.6 Modeling Requirements of Another Software

The extended TASM-based requirements modeling approach is also applied to model requirements of another actual real-time embedded software, namely, the Communication and Control Unit (abbreviated as CCU) main control software.

CCU fulfils the tasks of communication between CCU and the Payload Data Handling Unit (abbreviated as PDHU) through the 1553B interface, communication between CCU and two peripherals through the RS422 interface, telemetering, remote control and temperature control of Peripheral1, calibration of the calibrating unit, interpretation and execution of the data injection commands, and packaging the data packets that are transmitted to PDHU, etc. The other external interfaces in the CCU main control software include telemetric interface, remote control interface, temperature control interface, calibrating interface, reset interface, etc.

The study on the CCU main control software shows that the proposed approach is effective in requirements modeling for this software.

4.7 Summary of Observations and Lessons Learned

We now summarize our main observations, and also highlight several lessons for modeling requirements of real-time embedded software in the extended TASM.

Snippets of the rules in Figs. 4 and 5 and the example about two timers working in concert shown in Subsect. 4.4 suggest that the elapse of time, one single timer, and two timers working in concert can be modeled in the extended TASM. Snippets of the rules in Figs. 6 and 7 suggest that resource utilization can be modeled in the extended TASM.

Some samples for functional requirements modeling are given in Subsect. 4.4, which exemplifies that functional requirements involving time and resource utilization can be modeled with machine rules in the extended TASM.

Some samples for non-functional requirements modeling are given in Subsect. 4.5, which exemplifies that non-functional requirements of real-time embedded software such as performance, safety properties, and liveness properties can be modeled with machine rules in the extended TASM and temporal logic formulas.

We have learned the following lessons from the case study: (1) The main machines are the central section of the TASM models, and should be identified not only for the internal and external components, but also for complex interface, complex work in a component, and timers, e.g., the following main machines are identified in Subsect. 4.2: EHU, the reset chip, “1394 interface processing”, “event execution”, and “broadcasting time code for equipment timer”. (2) Each main machine and sub-machine should not be complicated. A main machine of complicated function should be divided into multiple main machines according to the parallel relation, e.g., the function of EHU is decomposed into the main machines named “1394 interface processing”, “RS422 interface processing”, and “event execution” in Subsect. 4.2. And a main machine and sub-machine of

complicated function should be divided into multiple sub-machines or function-machines according to the hierarchical relation, e.g., the main machine named “1394 interface processing” is divided into the sub-machines named “judging and sending 1394 isochronous packets” and “receiving 1394 asynchronous packets” in Subsect. 4.3. And (3) some time-dependent requirements can be described in machine rules in TASM by using the keyword **now** and the variables recording time, and some properties that must be met by the software can be described by using the variables recording time explicitly in TCTL formulas, e.g., we can see this from the modeling method of the performance requirement “the time for caching engineering data packets of the two pieces of equipment shall not be less than 20 min” in Subsect. 4.5.

5 Related Work

The requirement models for real-time embedded software include informal, semi-formal and formal models.

UML can support iterative processing when interaction frames are used in sequence diagrams [3]. The **Resource** stereotype is defined in MARTE to model resources of processing, storage, communications, concurrency, device, and timing, etc. in [4].

SPARDL is used to model aerospace embedded software, and some time-dependent properties are expressed in ITL [16] formulas in [14]. The extended STM-based modeling method is proposed in [15], which is attained by adding time semantics and constraint for each cell in STM. And some time-dependent properties are expressed in TCTL [17] formulas in [15]. However, neither SPARDL nor the extended STM-based modeling method considers resource description.

UTA does not support float data type except for time value [10]. Petri nets are a specific class of multi-agent ASMs [9]. ASM avoids extraneous mathematical symbols, and is understandable and usable without extensive mathematical training [10]. ASM notion is used to define a ground model of situation to be modeled, to which appropriate refinement steps can be applied in order to obtain the desired system [9]. ASM supports iterative processing. Nevertheless, ASM does not provide a description for time and resource utilization.

TASM uses a subset of ASM with extensions for time and resource utilization. The notion of configuration is presented in [10]. And TASM is extended with event construct and observer construct, and a requirements modeling approach for real-time embedded software is proposed in [12]. This approach comprises requirements preprocessing, components identification, connections identification, behavior specification and property annotation. However, the main machines identified only for the internal and external components are maybe not complete in [12]. Functional requirements are described in behavior specification, and property annotation adds timing and resource consumption annotations to the relevant TASM model [12].

We have extended TASM with the data type of arrays, a loop rule named **while**, and operators such as “%”, etc. On the basis of the extension, an extended

TASM-based requirements modeling approach is proposed in this paper. In our approach, the main machines should be identified not only for the internal and external components, but also for complex interface, complex work in a component, and timers. The refinement method of main machines is also proposed. And we believe that time and resources have something to do with both non-functional requirements and functional requirements. As a result, the approach proposed in this paper considers time and resources when both functional requirements and non-functional requirements are modeled.

6 Conclusion

Software requirements serve important roles in real-time embedded software. It is a challenge to devise an easy-to-use formal requirements modeling language and a corresponding requirements modeling approach. TASM is an easy-to-use formal requirements modeling language. And TASM provides a description for behavior of time, resources, synchronization, and concurrency, etc. However, TASM has its limitations.

We have extended TASM. In this paper, an extended TASM-based requirements modeling approach is proposed. This approach identifies the parallel and hierarchical components, describes the functional and non-functional requirements of the software, and is applied to model requirements of actual real-time embedded software. The effectiveness of the proposed approach for requirements modeling is validated preliminarily. And several lessons learned from the case study are highlighted.

In the future, we will apply the proposed approach to model requirements of more actual real-time embedded software, and improve and perfect our approach. And we will investigate automated generation of test requirements and test cases, and automated verification of the safety and liveness properties, etc., in the extended TASM models.

Acknowledgments. This work is supported by the National Natural Science Foundation of China under Grant No. 61272163.

References

1. Jin, Z., Liu, L., Jin, Y.: Software Requirements Engineering: Principles and Methods. Science Press, Beijing (2008). in Chinese
2. Hull, E., Jackson, K., Dick, J.: Requirements Engineering, 3rd edn. Springer, London (2011)
3. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Addison Wesley Longman Inc., Reading (2003)
4. Selić, B., Gérard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. Elsevier, San Francisco (2014)
5. Yang, Z.B., Pi, L., Hu, K., Gu, Z.H., Ma, D.F.: AADL: an architecture design and analysis language for complex embedded real-time systems. Ruan Jian Xue Bao/J. Softw. **21**(5), 899–915 (2010). in Chinese

6. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). doi:[10.1007/978-3-540-27755-2_3](https://doi.org/10.1007/978-3-540-27755-2_3)
7. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). doi:[10.1007/978-3-540-30080-9_7](https://doi.org/10.1007/978-3-540-30080-9_7)
8. AbouTraba, M.S., Brockway, M., Counsell, S., Hierons, R.M.: Testing real-time embedded systems using timed automata based approaches. *J. Syst. Softw.* **86**(5), 1209–1223 (2013)
9. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
10. Ouimet, M.: A formal framework for specification-based embedded real-time system engineering. Ph.D. thesis, MIT (2008)
11. Ouimet, M., Lundqvist, K.: The timed abstract state machine language: abstract state machines for real-time system engineering. *J. Univers. Comput. Sci.* **14**(12), 2007–2033 (2008)
12. Zhou, J., Lu, Y., Lundqvist, K.: A TASM-based requirements validation approach for safety-critical embedded systems. In: George, L., Vardanega, T. (eds.) Ada-Europe 2014. LNCS, vol. 8454, pp. 43–57. Springer, Heidelberg (2014). doi:[10.1007/978-3-319-08311-7_5](https://doi.org/10.1007/978-3-319-08311-7_5)
13. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From AADL to timed abstract state machines: a verified model transformation. *J. Syst. Softw.* **93**, 42–68 (2014)
14. Gu, B., Dong, Y.W., Wang, Z.: Formal modeling approach for aerospace embedded software. *Ruan Jian Xue Bao/J. Softw.* **26**(2), 321–331 (2015). in Chinese
15. Hou, G., Zhou, K.J., Chang, J.W., Wang, J., Li, M.C.: Software formal modeling and verification method based on time STM. *Ruan Jian Xue Bao/J. Softw.* **26**(2), 223–238 (2015). in Chinese
16. Dutertre, B.: Complete proof systems for first order interval temporal logic. In: Proceedings of IEEE LICS, pp. 36–43 (1995)
17. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proceedings of IEEE LICS, pp. 414–425 (1990)
18. Pnueli, A.: Verification engineering: a future profession. In: Proceedings of ACM PODC, p. 7 (1997)
19. Zhou, J., Lu, Y., Lundqvist, K.: The observer-based technique for requirements validation in embedded real-time systems. In: Proceedings of IEEE RET, pp. 47–54 (2014)

Software Engineering and Methodology for Emerging
Domains

15th National Software Application Conference, NASAC
2016, Kunming, Yunnan, November 3-5, 2016,
Proceedings

Zhang, L.; Xu, C. (Eds.)

2016, XII, 227 p. 74 illus., Softcover

ISBN: 978-981-10-3481-7