A Refinement Calculus for the Development of Real-time Systems

Zhiqiang Chen, Antonio Cau, Hussein Zedan
Xiaodong Liu and Hongji Yang
Software Technology Research Laboratory
De Montfort University
Leicester LE1 9BH, UK
{zqc,cau,zedan,xdl,hjy}@dmu.ac.uk

Abstract

We present a calculus which can transfer specifications to objects for the development of real-time systems. The object model is based on a practical OO development technique—HRT-HOOD. A real-time logic is specified by extending a sound formal method for real-time systems—TAM, to formalise the object model. With integration of HRT-HOOD and TAM, the advantages of object-oriented structured methods with the stepwise refinement techniques are combined. The result is illustrated on a case study.

1. Introduction

HRT-HOOD [1] is an object-oriented development technique that extends the structured methodology HOOD [2] to provide objects with both functional and temporal requirements of real-time systems. Classification of objects in the HRT-HOOD characterises temporal properties of real-time systems. This domain-specific style, with the graphic representation and the object description skeleton, makes HRT-HOOD a concise, cohesive, and powerful set of capabilities. Moreover, HRT-HOOD provides a technique that systematically transforms the design to Ada code which presents a unifying support to the development of real-time systems.

We note that most practical structured and OO software development techniques follow the software engineering principles that can help developers to avoid errors occur in the development process (especially at early stages) as much as possible, but fail to eliminate them completely. However, complete elimination of errors in the development process has been one of the major aims in software engineering.

Formal methods are believed the most important means which probably achieve that aim (and others) of software engineering with their capabilities of precise descriptions and mathematical verifications. In the development of real-time systems, formal methods can ensure correctness of both functional and temporal requirements. By now, a large number of formal methods for the development of real-time systems have been put forward, including those based on logics, (process) algebra, net/graphics or model (for a more detail review, we refer the readers to [3]).

One of underlying formal models of our present work is TAM (Temporal Agent Model) [4,5,6,7] which is a realistic formal software development method for real-time systems. In TAM, the stepwise refinement development method is employed by means of executable constructs together with a specification statement. The TAM theory views a real-time system as a collection of concurrently executing agents. These agents communicate via shunts which are time-stamped with the time of the most recent write. The TAM real-time logic consists of first order predicate logic with a few extensions. A timing function is used to represent the value found in variables and shunts at a specific time and the projection functions are also used to refer to the time-stamp and value found in a shunt respectively. Specifications are therefore constraints on the relationship between time-stamps and values found in shunts during the lifetime of the system. A mechanism for specifying duration is provided by the release and termination times of the system or agents which may be predicated over in the usual way. TAM has a set of refinement laws which are a set of syntactic rewrite rules. These rules enable the software developers to transform a requirement specification into an executable program.

We also note that although immense benefits which may be brought by formal methods, turning them into sound practice has proved to be extremely difficult. Some “pure” formal methods may keep practically-oriented software engineers from employing their benefits. It has been believed that combining formal methods with practical development techniques, such as SSADM and OO approaches, can be
a fruitful approach when modelling and developing large-scaled and complex software systems. On the one hand, by formalising the constructs of a practical development technique, formal methods can force the meaning of each system component to be more rigorous. On the other hand, using practical structured and OO techniques with formal methods can make formal methods more acceptable for use by a large community. Much work has been undertaken in extending formal notations such as Z and VDM to include the SSADM and OO paradigms, such as [8,9,10].

In this paper, we present the work on combination of a practical OO development technique and a formal method mentioned above by formalising HRT-HOOD [1] with TAM [5] in which a refinement calculus is provided to transform requirements/specifications to objects for real-time systems mathematically. We extended TAM with the capability of describing behaviours of objects (a computational object model) and method invocations. The computational model is defined on HRT-HOOD, which focuses on specification and may be refined by corresponding HRT-HOOD objects. HRT-HOOD is used to decompose the system’s requirements. Each sub-requirement is formalised, using the TAM specification statement which is subsequently refined into objects by using a set of refinement laws.

In the next Section, we introduce the computational object model and its syntax. A corresponding real-time logic is given briefly in Section 3, which is used to describe abstract specification and define the semantics. Based on the real-time logic, a refinement calculus is specified in Section 4. We demonstrate the application of the calculus with a case study in Section 5. Some conclusions are presented in the final section.

2. Computational Object Model

The computational model we used is an extension of that adapted in TAM [5], by introducing objects defined in HRT-HOOD [1]. In the model, a real-time system is viewed as a collection of concurrent activities which are initiated either periodically or sporadically with services which can be requested by the execution of the activities. The operations of the activities and services, as threads and methods, are allocated to the corresponding objects (an encapsulated operation environment for the thread or methods) according to their functional and temporal requirements and the relationships between them.

Like HRT-HOOD, five types of objects are defined in our model: sporadic objects, cyclic objects, protected objects, passive objects and active objects. Threads are defined in both sporadic and cyclic objects which activate and terminate with the corresponding objects and are concurrent with each other. Methods are defined in protected, passive, or active objects, and they are activated by invocations and their executions may be either concurrent or sequential. Invocations of methods can be either asynchronous or sequential. Recursive invocations between methods are prohibited, neither directly nor indirectly.

An object consists of a declaration and a method(s) in a structure. The declaration presents the definitions of attributes and/or an execution environment for methods defined in the object. The attributes of an object include: (1) object type — We use A, S, C, P and Pr to represent, respectively, that the object is either active, sporadic, cyclic, passive, or protected; (2) provided methods — We use ProvidedMethods(o) to denote the provided method set of an object o. They are declared in the form of m(in, out), where m is a method name which is free in the object. in and out are sets which present parameters transferred between m and its clients; (3) used methods — We use UsedMethods(o) to denote the used method set of an object o. The elements of the set UsedMethods(o) take the form of (d′, m′), where m′ is a method to be invoked by o and is defined in d′. UsedMethods(o) defines use relationships between o and objects in UsedMethods(o). Such relationships specify control flows between objects and together with in(m) and out(m), data flows are also specified. Other attributes vary with the type of objects: (1) the activation interval of the thread for a cyclic object; (2) the minimum activation interval of the thread for a sporadic object; (3) the child object set for an active object. We use ChildObjects(o) to denote the child object set of o if o is an active object. ChildObjects(o) specifies an include relationship between o and its child objects based on which the decomposition process is achieved; (4) the environment of a non-active object is a set of data over which the methods of the object execute for computations and communications. The data include constants, variables and shunts. For cyclic and sporadic objects, an activation period and a minimum activation interval are specified in the environment declaration respectively. We use ObjEnv(o) to denote the environment set of an object o.

A method consists of a head and a body. The head specifies a method name and a local environment (if necessary) of the method. The body specifies operations over either the object environment or the method environment, or both. We use Methods(o) to denote the set of method defined by the object o.

The operations are described by means of agents which may be either abstract or concrete. A method can define its local execution environment. We use MthEnv(m) to denote the local environment of the method m. If in(m) ≠ ∅ and/or out(m) ≠ ∅, then they are defined in MthEnv(m). A method m is defined in the form of

\[ m([\text{in, out}], \text{MthEnv}(m)) = \alpha \] where \( \alpha \) is an agent, called the body of the method m. We use \( \text{A}_m \) to denote the body of a method m.
An active object defines a system or subsystem which consists of a number of related objects as its child objects, optionally with a number of methods which are implemented by its child objects. An active object \( o \) with child objects \( o_1, o_2, \ldots, o_n \) and methods \( m'_1(\text{in}_1, \text{out}_1), \ldots, m'_k(\text{in}_k, \text{out}_k) \) which are defined in its child objects \( o_1, \ldots, o_n \), can be represented as:

\[
\text{Object } o \ A \\
\text{include } \{o_1, o_2, \ldots, o_n\} \\
\text{provide } \{m'_1(\text{in}_1, \text{out}_1), \ldots, m'_k(\text{in}_k, \text{out}_k)\} \\
\text{methods} \\
\quad m'_1(\text{in}_1, \text{out}_1) : o_1, m_1(\text{in}_1, \text{out}_1) \\
\quad \vdots \\
\quad m'_k(\text{in}_k, \text{out}_k) : o_k, m_k(\text{in}_k, \text{out}_k) \\
\text{end } o
\]

A cyclic or sporadic object defines a unique thread that operates periodically or sporadically:

\[
\text{Object } o \ C \\
\text{used pair_list} \\
\text{period } P \\
\text{constants constant_list} \\
\text{variables variable_list} \\
\text{shunts shunt_list} \\
\text{thread } A \\
\text{end } o
\]

where \( P \) and \( T \) are the activation period and the minimum activation interval of the corresponding objects.

Both protected and passive objects are used to define methods which can be requested by other objects. The difference between them is that the methods defined in a protected object can be executed exclusively while those defined in a passive object can be executed immediately when being requested:

\[
\text{Object } o \ P \\
\text{provide} \\
\quad \{m_1(\text{in}_1, \text{out}_1), \ldots, m_n(\text{in}_n, \text{out}_n)\} \\
\text{used pair_list} \\
\text{constants constant_list} \\
\text{variables variable_list} \\
\text{methods} \\
\quad m_1(\text{in}_1, \text{out}_1) \\
\quad \vdots \\
\quad m_n(\text{in}_n, \text{out}_n) \\
\quad A, \\
\text{end } o
\]

An agent describes a set of operations with explicit or implicit timing constraints. We directly use agents defined in [5] in the context of our object model. Two new agents are introduced:

1. \( d \cdot m[[\text{in}, \text{out}]] \) (Invocation)—\( d \) is the name of an object, \( m' \) is the method provided by \( d \) and \( \text{in}, \text{out} \) are optional parameters to be passed to the method \( m' \) as a substitution. This agent causes that an invocation to the method \( m' \) optionally with \( \text{in} \) and/or \( \text{out} \).

2. \( m[[\text{in}, \text{out}]] : o \cdot m[[\text{in}', \text{out}']] \) (Encapsulation)—\( o \) is a child object of the object \( o \) and \( m' \) is a method provided by \( o \). The definition of \( \text{in}, \text{out} \) must be in accordance with that of \( m' \). This agent serves as the body of a method of an active object. It transfers the invocation of \( m \) to that of \( m' \).

3. The Real-Time Logic

Like TAM, a discrete, linear time domain is used which is modelled by the natural numbers and denoted by Time. The current time is denoted by the free time variable now which is global and can be referred to by any agent.

Variables are used for computation and shared by the methods within an object or the agents within a method. Their values can be referred to with times. Shunts are used for communication with external environments, such as a sensor in hardware or an object in software. A shunt consists of two fields: one holds a (set of) value(s) and another holds a time at which the value(s) is (are) written. Shunts are also referred to with times. The difference between variables and shunts is that shunts are time-stamped. Shunts are assumed to be non-blocking on reading and writing. An event occurrence can be modelled by one or more shunts: it occurs if and only if the shunt is written. To restrict the complexity, we do not classify the domains of values found in variables and shunts while they are simply defined by Value.

An invocation is viewed as a special shunt whose value and timestamp represent its status and occurrence time. The invocation status includes: (1) Request—an invocation is in Request status if and only if it occurs but has not yet been served; (2) Activation—an invocation is in Activation status if and only if it has been served, i.e., the requested method is being executed for it, but has not yet terminated; (3) Termination—an invocation is in Termination status if and only if the execution of the method for it has terminated.

We define

\[
\text{InvStatus} = \{\text{REQ, ACT, TER}\}
\]

to represent the domain of the values found in an invocation. Like a shunt, an invocation is assumed to be non-blocking on reading and writing. We use Inv\( m \) to denote the sequence of all occurring invocations of a method \( m \). The order of elements in Inv\( m \) is that of requests to the method \( m \) and an element uniquely identifies an invocation. It is assumed that
an invocation of the method \( m \) is put into to \( \text{ln}_m \) if and only if it occurs.

The logic is basically first-order logic with conservative extension to deal with objects, methods, and invocations, with which temporal and functional requirements of the system and objects can be described succinctly.

A free time variable pair \( t_o \) and \( t_a \) is defined to represent the activation and termination times of agents, objects, and systems. The activation and termination times of objects, methods and agents must be within those of the system defining the objects, the objects defining the methods and the method defining the agents respectively. However, the activation and termination times of threads are viewed as those of the sporadic and cyclic objects which define the threads respectively.

The timing functions \( @, \varnothing \) and \( \tau \) are defined over pairs \( (x, t) \), where \( x \) is a variable, shunt, or invocation, and \( t \) is a time, resulting in the value of the variable, the value and the timestamp of the shunt, or an invocation status and its occurrence time at the given time respectively:

\[ @ : \text{Var} \times \text{Time} \rightarrow \text{Value} \]
\[ \varnothing : \text{Shunts} \times \text{Time} \rightarrow \text{Value} \times \text{Time} \]
\[ \tau : \bigcup_{m \in \text{All}} \text{ln}_m \times \text{Time} \rightarrow \text{ln}_m \times \text{Time} \]

where \( \text{Var} \) and \( \text{Shunts} \) are sets of variables and shunts respectively.

Projection functions of \( \varnothing \) and \( \tau \) for shunts and invocations are also defined:

1. \( v\varnothing \) and \( t\varnothing \) result in the value and the timestamp of the shunt respectively.
2. \( v\tau \) and \( t\tau \) result in an invocation status and its occurrence time respectively.

These functions are used as infix functions.

We define that \( \text{req} \) and \( \text{act} \) are defined as:

\[ \text{req} \rightarrow (\text{ln}_m(i) \cap t = (\text{req}, t)) \Rightarrow \text{req}\_\text{time}(\text{ln}_m(i)) = t \]
\[ \text{act} \rightarrow (\text{ln}_m(i) \cap t = (\text{act}, t)) \Rightarrow \text{act}\_\text{time}(\text{ln}_m(i)) = t \]
\[ \text{ter} \rightarrow (\text{ln}_m(i) \cap t = (\text{ter}, t)) \Rightarrow \text{ter}\_\text{time}(\text{ln}_m(i)) = t \]

It is clear that a normal invocation begins with request, activation and then termination. However, in some cases, such as that in passive objects, an invocation starts with activation and ends with termination. Moreover, the null operation in some implementation may have zero duration. This is a useful mechanism, especially in fault-tolerant systems in which a fail-stop mechanism is adopted. In such case, a request may terminate immediately.

A sporadic activity in which whenever a shunt \( s_1 \) is written to, another shunt \( s_2 \) must be written to within 5 time units can be described by the formula:

\[ \forall i \in [a, b] \cdot s(i) \cdot n(i) t = t \Rightarrow \exists f \in [t + 5] \cdot f \leq s + s_2 \cdot \text{act} \rightarrow f = f \]

A predicate can be defined to describe that shunts, variables or constants remain stable during a range of time:

\[ \text{stable}(X, [t_1, t_2]) = \text{def} \left\{ \begin{array}{ll} \forall x \in X, f' \in [t_1, t_2], \text{inv} \rightarrow f = f' & \text{if } X \text{ is a variable set} \\
\forall x \in X, f \in [t_1, t_2] \wedge t \leq t_1 & \text{if } X \text{ is a shunt set} \\
\text{true} & \text{if } X \text{ is a constant set} \\
\end{array} \right. \]

4. Refinement Calculus

The refinement relation \( \subset \) is defined on a component (agent, method and object) in a similar fashion to that of TAM. A component \( X \) is refined by the corresponding component \( Y \) (\( X \subseteq Y \)) if and only if \( \mathcal{F}[X] \Rightarrow \mathcal{F}[Y] \).

A set of refinement laws are specified, based on the real-time logic specified in the last section, to transform an abstract specification into concrete objects. Here we give some useful refinement laws.
Law 1 (Cyclic object)  
If \( \forall t \in [0, \infty), \exists n \in N, T \in \text{Time} \cdot t = T \times n \wedge d' = \Phi \), then there is an object \( o \) with type of \( G \), such that \( w : \Phi \subseteq o \), where \( o \) is in the form shown as below (left).

| Object \( o \) C  
|--------------------------------------------------|
| used pair_list  
| period \( P \)  
| constants constant_list  
| variables variable_list  
| shunts shunt_list  
| thread \( w : d' \)  
| end o |

Law 2 (Sporadic object)  
If \( \forall t \in [0, \infty), \exists s, x_2, \ldots, x_k \in E, \) such that \( \forall t \in [1, n], t < t \cdot t \Rightarrow s, x_2, \ldots, x_k \in T \wedge x_i = t \wedge d' = \Phi \), then there is an object \( o \) with type of \( S \), such that \( w : \Phi \subseteq o \), where \( o \) is in the form shown as above (right).

| Object \( o \) S  
|--------------------------------------------------|
| used pair_list  
| period \( P \)  
| constants constant_list  
| variables variable_list  
| shunts shunt_list  
| thread \( w : d' \)  
| end o |

Law 3 (Passive object)  
If \( m_1(m_1, \ldots, m_n) = \Phi_1, m_2(m_2, \ldots, m_n) = \Phi_2, \ldots, m_n(m_n, \ldots, m_1) = \Phi_n \), and \( \forall t \in [0, \infty) \cdot (\exists d : \text{Time} \cdot \\mbox{stable}(E, t + d) \wedge \bigvee_{i \in [1, n]} m_i(m_i, \ldots, m_n)) = \Phi \), then there is an object \( o \) with type of \( P \), such that \( w : \Phi \subseteq o \), where \( o \) is in the form shown below (left).

| Object \( o \) P  
|--------------------------------------------------|
| provide \{ \( m_1(m_1, \ldots, m_n) \), \ldots, \( m_n(m_n, \ldots, m_1) \}  
| used pair_list  
| constants constant_list  
| variables variable_list  
| shunts shunt_list  
| methods \( m_1, m_2, \ldots, m_n \)  
| variables variable_list  
| w : \Phi  
| end |

Law 4 (Protected object)  
If \( m_1(m_1, \ldots, m_n) = \Phi_1, m_2(m_2, \ldots, m_n) = \Phi_2, \ldots, m_n(m_n, \ldots, m_1) = \Phi_n \), and \( \forall t \in [0, \infty) \cdot \text{Time} \), such that \( \text{stable}(E, t + d) \wedge \bigvee_{i \in [1, n]} m_i(m_i, \ldots, m_n) \wedge t = 0 \), then there is an object \( o \) with type of \( Pr \), such that \( w : \Phi \subseteq o \), where \( o \) is in the form shown above (right).

Law 5 (Active object)  
If \( o_1, o_2, \ldots, o_n \) are related objects, then an object \( o \) can be defined to include them in the form given in Section 2, and \( o_1, o_2, \ldots, o_n \subseteq o \).

Law 6 (Method)  
Suppose \( o_1, o_2 \) are two objects with the same type.  
If \( \forall m \in \text{ProvidedMethods}(o_1), \exists m' \in \text{ProvidedMethods}(o_2) : A_{m'} \subseteq A_{m} \), then \( o_1 \subseteq o_2 \).

Law 7 (Invocation)  
If a method \( m'(in, out) = \text{MthEnv}(m) A \) end is defined in an object \( o', B \) is an agent in an object \( o \), and \( B \subseteq \text{A}[m'(in, out)] \), then \( B \subseteq o' \). \( m'(in, out) \) and if \( w_n \subseteq w_o \), then \( w_n = w_o \wedge \text{frame} \), where \( w_n, w_o, \) and \( w_o' \) are frames of \( B, o \) and \( o' \) respectively, and UsedMethods\( (o) = \text{UsedMethods}(o) \cup \{ (o', m'(in, out)) \} \).

Law 8 (Child Objects)  
Suppose \( o_1, o_2 \) are two active objects.  
If \( \forall o \in \text{ChildObjects}(o_1), \exists o'' \in \text{ChildObjects}(o_2) : o \subseteq o'' \), then \( o_1 \subseteq o_2 \).

5. A Case Study

In this section we illustrate the formal development of a real-time system using our model by a case study. The case study is designed based on “The Mine Control System” [1], as depicted in Fig.1.

![Motor Control System](image)

5.1. Requirements of the Motor Control System

We can express the system requirements (denoted by \( \text{REQ} \)) as:

\[
\text{REQ} \quad \text{Motor Control System}
\]

Whenever a command from the operator and 10 time units have elapsed since the last command if the command is “START,” the motor is off, and the gas level is not higher than 40, then the motor is switched on within 5 time units. If the command is “STOP,” the motor is on, then the motor is switched off within 5 time units. Every 20 time units, the gas level is checked, and if the gas level is higher than 40 and the motor is on, then the motor is switched off within 5 time units.

\( \text{REQ} \) is decomposed into three sub-requirements \( \text{req}_1, \text{req}_2, \) and \( \text{req}_3 \), corresponding to gas monitor, operator, and motor respectively:

\[
\begin{align*}
\text{req}_1 : & \quad \text{gas sensor} \rightarrow \text{motor} \quad \text{if gas level} > 40 \\
\text{req}_2 : & \quad \text{operator} \rightarrow \text{motor} \quad \text{if command} = \text{START} \\
\text{req}_3 : & \quad \text{motor} \rightarrow \text{operator} \quad \text{if gas level} > 40 \text{ and motor is on}
\end{align*}
\]
5.2. Specification of the System

In this Section, we define specifications of the subsystems and objects to represent corresponding requirements. This is achieved by identifying the system observables. The system operates the motor according to commands from the operator console and gas level sampled by the gas sensor. We use shunts cmd, level, and ctrl to model command, gas level and motor control interface respectively:

\[
\begin{align*}
\text{cmd:} & \quad \{ \text{START, STOP} \} \\
\text{level:} & \quad \mathbb{N} \\
\text{ctrl:} & \quad \{ \text{ON, OFF} \}
\end{align*}
\]

Sub-specifications \( \text{spec}_1 \), \( \text{spec}_2 \), and \( \text{spec}_3 \) corresponding to \( \text{req}_1 \), \( \text{req}_2 \), and \( \text{req}_3 \) can be specified by means of the real-time logic:

- **Operator**
  \[ \text{spec}_1 = \begin{cases} 
  \{ \text{ctrl} \} & \forall \ t \in [0, \infty) \quad \exists \text{cmd} \cdot \text{cmd} \cdot t \geq 10 \\
  \{ \text{cmd} \} & \quad (\text{cmd} = \{ \text{START, STOP} \} \land \text{level} \cdot \text{level} \cdot t \geq 40 \land \text{ctrl} \cdot \text{ctrl} \cdot t = \text{OFF} \Rightarrow \\
  \exists \ d : \text{Time} \cdot \text{write-sh}(\text{OFF}, \text{ctrl}, [t, t + d]) \land d \leq 5) \\
  \{ \text{cmd} \} & \quad (\text{cmd} = \{ \text{STOP} \} \land \text{ctrl} \cdot \text{ctrl} \cdot t = \text{ON} \Rightarrow \\
  \exists \ d : \text{Time} \cdot \text{write-sh}(\text{OFF}, \text{ctrl}, [t, t + d]) \land d \leq 5)
\end{cases} \]

- **Gas Check**
  \[ \text{spec}_2 = \begin{cases} 
  \{ \text{ctrl} \} & \forall \ t \in [0, \infty) \quad \exists n \in \mathbb{N} \cdot t = 20 \ast n \land \\
  \{ \text{ctrl} \} & \quad (\text{ctrl} \cdot \text{ctrl} \cdot t > 40 \Rightarrow \\
  \exists \ d : \text{Time} \cdot \text{write-sh}(\text{OFF}, \text{ctrl}, [t, t + d]) \land d \leq 5)
\end{cases} \]

- **Motor**
  \[ \text{spec}_3 = \begin{cases} 
  \{ \text{ctrl} \} & \forall \ t \in [0, \infty) \quad \exists (\text{ctrl} \cdot \text{ctrl} \cdot t = \text{OFF} \Rightarrow \\
  \text{write-sh}(\text{OFF}, \text{ctrl}, [t, t + d]) \land d \leq 5) \\
  \{ \text{ctrl} \} & \quad (\text{ctrl} \cdot \text{ctrl} \cdot t = \text{ON} \Rightarrow \\
  \text{write-sh}(\text{OFF}, \text{ctrl}, [t, t + d]) \land d \leq 5) \end{cases} \]

Because the gas level is also accessed by the object **Motor** to check if the gas is safe, a corresponding method should be provided. However, because the level is sampled periodically, it cannot be accessed sporadically. We introduce an operation **Gas Status** to maintain the gas status according to the gas level. A variable is introduced to represent the gas status:

\[ \text{gas} \cdot \text{stat} : \{ \text{SAFE, UNSAFE} \} \]

A corresponding specification \( \text{spec}_4 \) is defined as:

\[ \text{Gas Status} \]

\[ \begin{align*}
\text{spec}_4 & \quad \begin{cases} 
  \{ \text{gas} \cdot \text{stat} \} & \forall \ t \in [0, \infty) \quad (\exists \ i \in [t, t + d] \cdot \text{gas} \cdot \text{stat} \cdot i) \land (\exists \ i \in [t, t + d] \cdot \text{gas} \cdot \text{stat} \cdot i) \\
  & \quad \forall \ d : \text{Time} \cdot \text{stable}(\text{gas} \cdot \text{stat}, [t, t + d])
\end{cases} \\
\end{align*} \]

Correspondingly, \( \text{spec}_2 \) is adapted, denoted by \( \text{spec}_{2}' \):

\[ \text{Gas Check} \]

\[ \begin{align*}
\text{spec}_{2}' & \quad \begin{cases} 
  \{ \text{ctrl} \} & \forall \ t \in [0, \infty) \quad \exists n \in \mathbb{N} \cdot t = 20 \ast n \land \\
  & \quad (\text{level} \cdot \text{level} \cdot t \geq 40 \Rightarrow \exists \ d : \text{Time} \cdot \text{write-sh}(\text{OFF}, \text{ctrl}, [t, t + d]) \land d \leq 5) \\
  & \quad \exists \ d : \text{Time} \cdot \text{gas} \cdot \text{stat}(\text{gas} \cdot \text{stat}, [t, t + d]) \\
  & \quad \exists \ d : \text{Time} \cdot \text{gas} \cdot \text{stat}(\text{gas} \cdot \text{stat}, [t, t + d]) \land d \leq 5)
\end{cases} \]

Thus we have

\[ \text{SPEC} = \begin{cases} 
  \text{spec}_1 \land \text{spec}_{2}' \land \text{spec}_3 \land \text{spec}_4
\end{cases} \]

At this stage, if there is some \( i \), such that \( \text{spec}_i = \text{false} \), then \( \text{req}_i \) must be re-formalised.

5.3. Refinement

In this Section, we use refinement laws listed in the Appendix A to refine sub-specifications derived in the last Section. At the concrete level, some time parameters are removed.

5.3.1 From specifications to Objects

The sub-specification \( \text{spec}_1 \) can be refined by a sporadic object **Operator** (Law 2.2 producing):

\[
\text{Object Operator S}
\begin{align*}
\text{interval} & \quad 10 \\
\text{shunts} & \quad \text{cmd:} \quad \{ \text{START, STOP} \} \\
& \quad \text{ctrl:} \quad \{ \text{ON, OFF} \}
\end{align*}
\]

\[
\text{thread}
\begin{align*}
\{ \text{ctrl} \} & \quad \text{cmd:} \quad \{ \text{ON, OFF} \} \\
& \quad \text{level:} \quad \mathbb{N} \times \text{Time}
\end{align*}
\]

\[
\text{write-sh}(\text{OFF}, \text{ctrl}, [t, t + d]) \land d \leq 5
\]

\[
\text{spec}_{2}' \text{ can be refined by a cyclic object **Gas Check** (Law 1.1 producing):}
\]

\[
\text{Object Gas Check C}
\begin{align*}
\text{period} & \quad 20 \\
\text{shunts} & \quad \{ \text{ON, OFF} \}
\end{align*}
\]

\[
\text{level:} \quad \mathbb{N} \times \text{Time}
\]

\[
\text{variables}
\begin{align*}
\text{gas} \cdot \text{stat}: & \quad \{ \text{SAFE, UNSAFE} \}
\end{align*}
\]

\[
\{ \text{ctrl} \} & \quad \text{level:} \quad \{ \text{SAFE, UNSAFE} \}
\end{align*}
\]

\[
\{ \text{level} \} & \quad \text{gas} \cdot \text{stat}(\text{gas} \cdot \text{stat}, [t, t + d]) \land d \leq 5
\]

\[
\{ \text{gas} \cdot \text{stat} \} & \quad \text{gas} \cdot \text{stat}(\text{gas} \cdot \text{stat}, [t, t + d]) \land d \leq 5
\]

\[
\text{end Gas Check}
\]

Authorized licensed use limited to: Edinburgh Napier University. Downloaded on July 20,2010 at 13:40:22 UTC from IEEE Xplore. Restrictions apply.
5.3.2 From Objects to the Final System

1. **Motor** (Law 6) and laws on agents defined in TAM [5], shown in Fig. 2.
2. **Gas_Status** (Law 6) and laws on agents, shown in Fig. 3.
3. **Gas_Check** (Law 6, Law 7 and laws on agents), shown in Fig. 4.
4. We can encapsulate **Gas_Check** and **Gas_Status** with a new object **Gas_Monitor** which refines them (Law 5), shown in Fig. 5.
5. **Operator** (Law 6, Law 7 and laws on agents), shown in Fig. 6.
6. Fig. 7 shows the final system.

We omit time obligations because of their observiveness.
6. Conclusions

In this paper, we present an approach to integrate a practical OO development technique, HRT-HOOD, and a sound formal method, TAM, in which a refinement calculus is provided to transform requirements/specifications to objects for real-time systems. We extended TAM with the capability of describing behaviours of objects (a computational object model) and method invocations. The computational model is defined based on HRT-HOOD, which focuses on specification and may be refined by corresponding HRT-HOOD objects. HRT-HOOD is used to decompose the system’s requirements. Each sub-requirement is formalised, using the TAM specification statement which is subsequently refined into objects by using a set of refinement laws.

With the combination, a framework is specified which supports the development of real-time systems from informal requirements to formal specification, concrete design, possibly until executable code. Based on the refinement calculus, a development method is suggested:

1. Use HRT-HOOD to decompose the system requirement, namely \( \text{REQ} \), to produce sub-requirements: \( \text{req}_1, \text{req}_2, ..., \text{req}_n \).

2. Formalise each sub-requirement \( \text{req}_i \) using the specification statement of TAM to produce \( \text{spec}_1, \text{spec}_2, ..., \text{spec}_n \). Note that the formal specification, \( \text{SPEC} \), which corresponds to \( \text{REQ}\), is given by

\[
\text{SPEC} = \text{def} \bigwedge_{i \in [1,n]} \text{spec}_i
\]

3. Use the refinement calculus to refine \( \text{spec}_i \) into a concrete object \( \text{obj}_i \), i.e.,

\[
\text{spec}_i \subseteq \text{obj}_i
\]

4. The collection of resulting objects are then composed to produce the final concrete system.

5. Use HRT-HOOD to map the resulting concrete code to an equivalent Ada code.

It is clear that some of the timing characteristics may be left as ‘variables’ to be determined at a later stage of development. These variables are constraints by the obligations themselves. For example, in the case study presented, the duration and worst case execution time of the operations of concrete thread in the object Gas_Check may be left as a variable \( d \) which can be determined at the implementation phase.

Reference


