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 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.

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 synchronous. Recursive invocations between methods are prohibited, neither directly nor indirectly.

An object consists of a declaration and 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 $\mathsf{ProvidedMethods}(o)$ 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 transfered between m and its clients; (3) **used methods** — We use $\mathsf{UsedMethods}(o)$ to denote the used method set of an object o. The elements of the set $\mathsf{UsedMethods}(o)$ take the form of (o', m'), where m' is a method to be invoked by o and is defined in o'. $\mathsf{UsedMethods}(o)$ defines *use* relationships between o and objects in $\mathsf{UsedMethods}(o)$. 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 $\mathsf{ChildObjects}(o)$ to denote the child object set of o if o is an active object. $\mathsf{ChildObjects}(o)$ 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 $\mathsf{ObjEnv}(o)$ 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 $\mathsf{Methods}(o)$ 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 $\mathsf{MthEnv}(m)$ to denote the local environment of the method m. If $in(m) \neq \emptyset$ and/or $out(m) \neq \emptyset$, then they are defined in $\mathsf{MthEnv}(m)$. A method m is defined in the form of

$$m([in, out]) =_{def} \ [\mathsf{MthEnv}(m)] \ \ \mathcal{A} \ \ \mathsf{end}$$

where $\mathcal{A}$ is an agent, called the body of the method m. We use $\mathcal{A}_m$ 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, ..., o_n and methods $m'_1(in_1, out_1)$, ..., $m'_k(in_k, out_k)$ which are defined in its child objects o_{i_1}, ..., o_{i_k} can be represented as:

Object o A
include $\{o_1, o_2, ..., o_n\}$
provide $\{m'_1(in_1, out_1), ..., m'_k(in_k, out_k)\}$
methods
$m'_1(in_1, out_1) : o_{i_1}.m_1(in_1, out_1)$
:
$m'_k(in_k, out_k) : o_{i_k}.m_k(in_k, out_k)$
end o

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

Object o C
used *pair_list*
period P
constants *constant_list*
variables *variable_list*
shunts *shunt_list*
thread
$\mathcal{A}$
end o

Object o S
used *pair_list*
interval T
constants *constant_list*
variables *variable_list*
shunts *shunt_list*
thread
$\mathcal{A}$
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:

Object o Pr
provide
$\{m_1(in_1, out_1), \ldots, m_n(in_n, out_n)\}$
used *pair_list*
constants *constant_list*
variables *variable_list*
shunts *shunt_list*
methods
$m_1(in_1, out_1)$
variables *variable_list*
$\mathcal{A}_1$
end
:
$m_n(in_n, out_n)$
$\mathcal{A}_n$
end
end o

Object o P
provide
$\{m_1(in_1, out_1), \ldots, m_n(in_n, out_n)\}$
used *pair_list*
constants *constant_list*
variables *variable_list*
shunts *shunt_list*
methods
$m_1(in_1, out_1)$
variables *variable_list*
$\mathcal{A}_1$
end
:
$m_n(in_n, out_n)$
$\mathcal{A}_n$
end
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. $o'.m'([\overline{in}, \overline{out}])$ (**Invocation**)—o' is the name of an object, m' is the method provided by o' and $\overline{in}$, $\overline{out}$ 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 $\overline{in}$ and/or $\overline{out}$.

2. $m([in, out]) : o'.m'([in', out'])$ (**Encapsulation**)—o' is a child object of the object o and m' is a method provided by o'. The definition of *in*, *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

$$\mathsf{InvStatus} = \{\mathsf{REQ}, \mathsf{ACT}, \mathsf{TER}\}$$

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 Inv_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_α and t_ω 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 ⓐ, ⓑ and ⓒ 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:

$$\begin{aligned} @&: \quad \mathit{Var} \times \mathsf{Time} \to \mathsf{Value} \\ ⓑ&: \quad \mathit{Shunts} \times \mathsf{Time} \to \mathsf{Value} \times \mathsf{Time} \\ ⓒ&: \quad \bigcup_{\text{all } m} \mathsf{Inv}_m \times \mathsf{Time} \to \mathsf{InvStatuses} \times \mathsf{Time} \end{aligned}$$

where *Var* and *Shunts* are sets of variables and shunts respectively.

Projection functions of ⓑ, ⓒ for shunts and invocations are also defined:

1. $.v$ⓑ and $.ts$ⓑ result in the value and the timestamp of the shunt respectively.
2. $.v$ⓒ and $.ts$ⓒ result in an invocation status and its occurrence time respectively.

These functions are used as infix functions.

We define that $\mathsf{req\_time}(q)$, $\mathsf{act\_time}(q)$ and $\mathsf{ter\_time}(q)$ represent the request, activation and termination times of an invocation q to a method m respectively, and

1. $\forall i \in [1, \mid \mathsf{Inv}_m \mid], t : \mathsf{Time} \cdot \mathsf{Inv}_m(i)$ⓒ$t = (\mathsf{REQ}, t) \Rightarrow \mathsf{req\_time}(\mathsf{Inv}_m(i)) = t$
2. $\forall i \in [1, \mid \mathsf{Inv}_m \mid], t : \mathsf{Time} \cdot \mathsf{Inv}_m(i)$ⓒ$t = (\mathsf{ACT}, t) \Rightarrow \mathsf{act\_time}(\mathsf{Inv}_m(i)) = t$
3. $\forall i \in [1, \mid \mathsf{Inv}_m \mid], t : \mathsf{Time} \cdot \mathsf{Inv}_m(i)$ⓒ$t = (\mathsf{TER}, t) \Rightarrow \mathsf{ter\_time}(\mathsf{Inv}_m(i)) = t$

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, specially 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:

$$\forall t \in [t_\alpha, t_\omega] \cdot s_1 .ts\text{ⓑ}\ t = t \Rightarrow \exists t' \in [t, t+5] \cdot t' \le t_\omega \wedge s_2 .ts\text{ⓑ}\ t' = t'$$

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

$$\mathsf{stable}(X, [t_1, t_2]) =_{def} \begin{cases} \forall v \in X, t', t'' \in [t_1, t_2] \cdot v @ t' = v @ t'' & \text{if X is a variable set} \\ \forall s \in X, t' \in [t_1, t_2] \cdot s .ts\text{ⓑ}\ t' \le t_1 & \text{if X is a shunt set} \\ \mathsf{true} & \text{if X is a constant set} \end{cases}$$

A shunt is written to exactly once during a range of time can be specified:

$$\begin{aligned} &\mathsf{write\_sh}(x, s, [t_1, t_2]) =_{def} \\ &\quad \exists t \in [t_1, t_2] \cdot s\text{ⓑ}\ t = (x, t) \wedge \mathsf{stable}(\{s\}, [t_1, t \Leftrightarrow 1]) \wedge \mathsf{stable}(\{s\}, [t, t_2]) \end{aligned}$$

If M is a set of methods provided by a protected object o, then the following predicate must be held for the object o:

$$\begin{aligned} &\mathsf{exclusive}(M) =_{def} \forall m, n \in M, i_1 \in [1, \mid \mathsf{Inv}_m \mid], i_2 \in [1, \mid \mathsf{Inv}_n \mid] \cdot \\ &\quad \exists t \in [t_\alpha, t_\omega] \cdot t \ge \mathsf{Inv}_m(i_1) .req\text{ⓒ}\ t \Rightarrow \\ &\qquad \mathsf{Inv}_m(i_1) .act\text{ⓒ}\ t \le \mathsf{Inv}_n(i_2) .act\text{ⓒ}\ t \le \mathsf{Inv}_m(i_1) .ter\text{ⓒ}\ t \Leftrightarrow m = n \wedge i_1 = i_2 \end{aligned}$$

This predicate asserts that executions of all methods in a protected object must be mutually exclusive.

Thus specification-oriented semantics of TAM in object context can be derived. We define $\mathcal{F}$ is a semantic function, such that $\mathcal{F}[\![X]\!]$ gives the semantics of a component X, such as an object, method, or agent. For example, if a method invocation $o'.m'(\overline{in}, \overline{out})$ is used in a method m in an object o, and

$$\begin{aligned} &\exists o_1 \in \mathsf{Ancestor}(o), o_2 \in \mathsf{ChildObjects}(o_1), \\ &\qquad m_2 \in \mathsf{ProvidedMethods}(o_2) \cdot o' = o_2 \wedge m' = m_2 \end{aligned}$$

where $\mathsf{Ancestor}(o)$ is the set of ancestors of the object o, and the method m' is defined as in the object o':

$$m'(in, out) =_{def} \mathcal{A}\ \mathsf{end}$$

then the semantics of the invocation is defined as:

$$\begin{aligned} &\mathcal{F}[\![o'.m'(\overline{in}, \overline{out})]\!] =_{def} \\ &\mathsf{stable}((\mathsf{ObjEnv}(o) \cup \mathsf{MthEnv}(m)) \setminus \overline{out}, [t_\alpha, t_\omega]) \wedge \exists t' \in [t_\alpha, t_\omega], i \in [1, \mid \mathsf{Inv}'_m \mid] \cdot \\ &\mathsf{Inv}'_m(i)\ \text{©}\ t' = (\mathsf{REQ}, t') \wedge \exists t_1, t_2 \ge t' \cdot t_1 \le t_2 \wedge \mathcal{F}[\![\mathcal{A}[t_1/t_\alpha, t_2/t_\omega, \overline{in}/in, \overline{out}/out]]\!] \wedge \\ &(t_\omega = t' \Leftrightarrow out(\mathsf{Inv}'_m(i)) = \emptyset) \vee (t_\omega = t_2 \Leftrightarrow out(\mathsf{Inv}'_m(i)) \ne \emptyset) \end{aligned}$$

If o_C is a cyclic object in the form defined in the last section, then the semantics of o_C can be described by:

$$\begin{aligned} &\mathcal{F}[\![o_\mathsf{C}]\!] =_{def} \mathsf{ChildObjects}(o_\mathsf{C}) = \emptyset \wedge \mathsf{ProvidedMethods}(o_\mathsf{C}) = \emptyset \wedge \\ &\forall o' \in \Sigma, (o'', m'') \in \mathsf{UsedMethods}(o'), m \in \mathsf{Methods}(o_\mathsf{C}) \cdot o_\mathsf{C} \ne o'' \wedge m \ne m'' \wedge \\ &\forall n \in \mathcal{N}, \exists t_1 \in [t_\alpha, t_\omega] \cdot t_1 = T \times n \Rightarrow \exists t_2 \in [t_\alpha, t_\omega] \cdot \\ &t_1 \le t_2 \le t_1 + T \wedge \mathcal{F}[\![\mathcal{A}[t_1/t_\alpha, t_2/t_\omega]]\!] \end{aligned}$$

4. Refinement Calculus

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

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 $\forall t \in [0,\infty), \exists n \in \mathcal{N}, T \in \mathsf{Time} \cdot t = T \times n \wedge \Phi' \Rightarrow \Phi$, then there is an object o with type of C, such that $w : \Phi \sqsubseteq o$, 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
$w : \Phi'$
end o

Object o S
used *pair_list*
interval T
constants *constant_list*
variables *variable_list*
shunts *shunt_list*
thread
$w : \Phi'$
end o

Law. 2 (Sporadic object)
If $\forall t \in [0,\infty), \exists s_1, s_2, \ldots, s_n \in E \cdot$, such that $\forall i \in [1..n], t' < t \cdot t \Leftrightarrow s_i.ts\textcircled{b}\, t' \geq T \wedge s_i.ts\textcircled{b}\, t = t \wedge \Phi' \Rightarrow \Phi$, then there is an object o with type of S, such that $w : \Phi \sqsubseteq o$, where o is in the form shown as above (right).

Law. 3 (Passive object)
If $m_1(in_1, out_1) =_{def} \Phi_1, m_2(in_2, out_2) =_{def} \Phi_2, \ldots, m_n(in_n, out_n) =_{def} \Phi_n$, and $\forall t \in [0,\infty) \cdot (\exists d : \mathsf{Time} \cdot \mathsf{stable}(E, [t, t+d])) \vee \bigvee_{i \in [1..n]} m_i(in_i, out_i) \Rightarrow \Phi$,
then there is an object o with type of P, such that $w : \Phi \sqsubseteq o$, where o is in the form as shown below (left).

Object o P
provide
$\{m_1(in_1, out_1), \ldots, m_n(in_n, out_n)\}$
used *pair_list*
constants *constant_list*
variables *variable_list*
shunts *shunt_list*
methods
$m_1(in_1, out_1)$
variables *variable_list*
$w : \Phi_1$
end
$\vdots$
$m_n(in_n, out_n)$
$w : \Phi_n$
end
end o

Object o Pr
provide
$\{m_1(in_1, out_1), \ldots, m_n(in_n, out_n)\}$
used *pair_list*
constants *constant_list*
variables *variable_list*
shunts *shunt_list*
methods
$m_1(in_1, out_1)$
variables *variable_list*
$w : \Phi_1$
end
$\vdots$
$m_n(in_n, out_n)$
$w : \Phi_n$
end
end o

Law. 4 (Protected object)
If $m_1(in_1, out_1) =_{def} \Phi_1, m_2(in_2, out_2) =_{def} \Phi_2, \ldots, m_n(in_n, out_n) =_{def} \Phi_n$, and $\forall t \in [0,\infty), \exists d : \mathsf{Time}$, such that

$$\mathsf{stable}(E, [t, t+d]) \vee (\forall t' \in [t, t+d], i \in [1..n] \cdot m_i(in_i, out_i)\ @\ t \Rightarrow \neg \bigwedge_{j \in [1..n], j \neq i} m_j(in_j, out_j)) \Rightarrow \Phi$$

then there is an object o with type of Pr, such that $w : \Phi \sqsubseteq o$, where o is in the form as shown above (right).

Law. 5 (Active object)
If o_1, o_2, ..., o_{i_1}, ..., o_{i_k}, ..., o_n are related objects, then an object o can be defined to include them in the form given in Section 2, and $(o_1, o_2, \ldots, o_n) \sqsubseteq o$.

Law. 6 (Method)
Suppose o_1, o_2 are two objects with the same type.
If $\forall m' \in \mathsf{ProvidedMethods}(o_1), \exists m'' \in \mathsf{ProvidedMethods}(o_2) \cdot \mathcal{A}_{m'} \sqsubseteq \mathcal{A}_{m''}$, then $o_1 \sqsubseteq o_2$.

Law. 7 (Invocation)
If a method $m'(in, out) =_{def} \mathsf{MthEnv}(m)\ \mathcal{A}\ \mathsf{end}$ is defined in an object o', $\mathcal{B}$ is an agent in an object o, and $\mathcal{B} \sqsubseteq \mathcal{A}[\overline{in}/in, \overline{out}/out]$, then $\mathcal{B} \sqsubseteq o'.m'(\overline{in}, \overline{out})$ and if $w_{\mathcal{B}} \subseteq w_{o'}$, then $w_o = w_o \backslash w_{\mathcal{B}}$, where $w_{\mathcal{B}}$, w_o and $w_{o'}$ are frames of $\mathcal{B}$, o and o' respectively, and $\mathsf{UsedMethods}(o) = \mathsf{UsedMethods}(o) \cup \{(o', m'(in, out)\}$

Law. 8 (Child Objects)
Suppose o_1, o_2 are two active objects.
If $\forall o' \in \mathsf{ChildObjects}(o_1), \exists o'' \in \mathsf{ChildObjects}(o_2) \cdot o' \sqsubseteq o''$, then $o_1 \sqsubseteq o_2$

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.

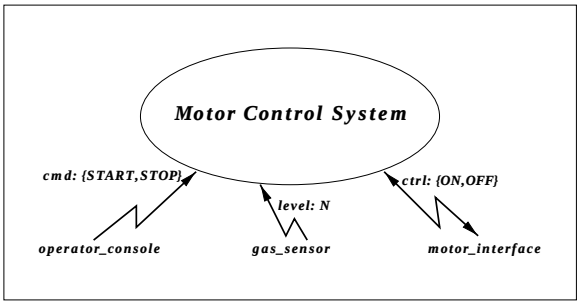


Figure 1: Motor Control System

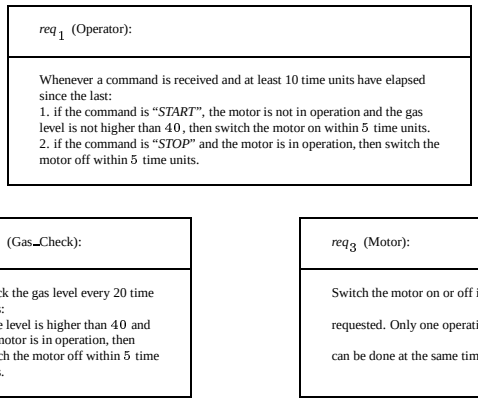
5.1. Requirements of the Motor Control System

We can express the system requirements (denoted by *REQ*) as:

REQ (Motor_Control_System):

Whenever receiving 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.

REQ is decomposed into three sub-requirements req_1, req_2 and req_3 corresponding to gas monitor, operator, and motor respectively:



In general, if *REQ* is inconsistent, then further requirement analysis is needed and if there is some *i*, such that req_i is not proper, then the decomposition must be redone.

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:

cmd: $\{START, STOP\}$
level: $\mathcal{N}$
ctrl: $\{ON, OFF\}$

Sub-specifications $spec_1$, $spec_2$ and $spec_3$ corresponding to req_1, req_2 and req_3 can be specified by means of the real-time logic:

- *Operator*
  $spec_1 =_{def}$
  $\{ctrl\}$:
  $\forall\, t \in [0,\infty) \cdot t \Leftrightarrow cmd\ .ts(\textcircled{b})\ t \geq 10 \wedge$
  $((cmd\ (\textcircled{b})\ t = (START, t) \wedge level\ .v(\textcircled{b})\ t \leq 40 \wedge ctrl\ .v(\textcircled{b})\ t = OFF \Rightarrow$
  $\exists\, d : \mathsf{Time} \cdot \mathsf{write\_sh}(ON, ctrl, [t, t+d]) \wedge d \leq 5) \vee$
  $(cmd\ (\textcircled{b})\ t = (STOP, t) \wedge ctrl\ .v(\textcircled{b})\ t = ON \Rightarrow$
  $\exists\, d : \mathsf{Time} \cdot \mathsf{write\_sh}(OFF, ctrl, [t, t+d]) \wedge d \leq 5))$
- *Gas_Check*
  $spec_2 =_{def}$
  $\{ctrl\}$:
  $\forall\, t \in [0,\infty) \cdot \exists\, n \in \mathcal{N} \cdot t = 20 * n \wedge level\ .v(\textcircled{b})\ t > 40 \Rightarrow$
  $(ctrl.v(\textcircled{b})\ t = ON \Rightarrow \exists\, d : \mathsf{Time} \cdot \mathsf{write\_sh}(OFF, ctrl, [t, t+d]) \wedge d \leq 5)$
- *Motor*
  $spec_3 =_{def}$
  $\{ctrl\}$:
  $\forall\, t \in [0,\infty) \cdot$
  $\neg((ctrl\ .v(\textcircled{b})\ t_1 = OFF \Rightarrow \mathsf{write\_sh}(ON, ctrl, [t, t+d]) \wedge d \leq 5) \wedge$
  $(ctrl\ .v(\textcircled{b})\ t_1 = ON \Rightarrow \mathsf{write\_sh}(OFF, ctrl, [t, t+d]) \wedge d \leq 5)) \vee$
  $\exists\, d : \mathsf{Time} \cdot \mathsf{stable}(ctrl, [t, t+d])$

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 can not be accessed sporadically. We introduce an operation *Gas_Status* to maintain a gas status according to the gas level. A variable is introduced to represent the gas status:

gas_st: $\{SAFE, UNSAFE\}$

A corresponding specification $spec_4$ is defined as:

- *Gas_Status*
  $spec_4 =_{def}$
  $\{gas\_st, s\}$:
  $\forall\, t \in [0,\infty) \cdot \neg((\exists\, t' \in [t, t+d] \cdot s\ @\ t' = gas_s t\ @\ t) \wedge (\exists\, t' \in [t, t+d] \cdot gas\_st\ @\ t' = s\ @\ t)) \vee$
  $\exists\, d : \mathsf{Time} \cdot \mathsf{stable}(\{gas_s t, s\}, [t, t+d])$

Correspondingly, $spec_2$ is adapted, denoted by $spec'_2$:

- *Gas_Check*
  $spec'_2 =_{def}$
  $\{ctrl, gas\_st\}$:
  $\forall\, t \in [0,\infty) \cdot \exists\, n \in \mathcal{N} \cdot t = 20 * n \wedge$
  $(level\ .v(\textcircled{b})\ t_\alpha > 40 \Rightarrow (ctrl\ .v(\textcircled{b})\ t_\alpha = ON \Rightarrow$
  $\exists\, d : \mathsf{Time} \cdot \mathsf{write\_sh}(OFF, ctrl, [t_\alpha, t_\omega]) \wedge d \leq 5) \wedge$
  $(gas\_st\ @\ t_\alpha = SAFE \Rightarrow$
  $\exists\, d' : \mathsf{Time} \cdot gas\_st\ @\ (t_\alpha + d') = UNSAFE)) \vee$
  $(level\ .v(\textcircled{b})\ t_\alpha \leq 40 \wedge gas\_st\ @\ t_\alpha = UNSAFE \Rightarrow$
  $\exists\, d' : \mathsf{Time} \cdot gas\_st\ @\ (t_\alpha + d') = SAFE)$

Thus we have

$$SPEC =_{def} spec_1 \wedge spec'_2 \wedge spec_3 \wedge spec_4$$

At this stage, if there is some *i*, such that $spec_i = \mathsf{false}$, then 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 $spec_1$ can be refined by a sporadic object *Operator* (**Law. 2**) producing:

Object *Operator* S
interval 10
shunts
  $cmd : \{START, STOP\} \times \mathsf{Time}$
  $ctrl : \{ON, OFF\} \times \mathsf{Time}$
  $level : \mathcal{N} \times \mathsf{Time}$
thread
  $\{ctrl\}$:
  $cmd\ .v(\textcircled{b})\ t_\alpha = START \wedge level\ .v(\textcircled{b})\ t \leq 40 \wedge ctrl\ .v(\textcircled{b})\ t = OFF \Rightarrow$
  $\mathsf{write\_sh}(ON, ctrl, [t_\alpha, t_\omega]) \wedge t_\omega \Leftrightarrow t_\alpha \leq 5 \vee$
  $cmd\ .v(\textcircled{b})\ t_\alpha = STOP \wedge ctrl\ .v(\textcircled{b})\ t = ON \Rightarrow$
  $\mathsf{write\_sh}(OFF, ctrl, [t_\alpha, t_\omega]) \wedge t_\omega \Leftrightarrow t_\alpha \leq 5$
end *Operator*

$spec'_2$ can be refined by a cyclic object *Gas_Check* (**Law. 1**) producing:

Object *Gas_Check* C
period 20
shunts
  $ctrl : \{ON, OFF\} \times \mathsf{Time}$
  $level : \mathcal{N} \times \mathsf{Time}$
variables
  $gas\_st : \{SAFE, UNSAFE\}$
thread
  $\{ctrl, gas\_st\}$:
  $(level\ .v(\textcircled{b})\ t_\alpha > 40 \Rightarrow (ctrl\ .v(\textcircled{b})\ t_\alpha = ON \Rightarrow$
  $\exists\, d : \mathsf{Time} \cdot \mathsf{write\_sh}(OFF, ctrl, [t_\alpha, t_\omega]) \wedge d \leq 5) \wedge$
  $(gas\_st\ @\ t_\alpha = SAFE \Rightarrow$
  $\exists\, d' : \mathsf{Time} \cdot gas\_st\ @\ (t_\alpha + d') = UNSAFE)) \vee$
  $(level\ .v(\textcircled{b})\ t_\alpha \leq 40 \wedge gas\_st\ @\ t_\alpha = UNSAFE \Rightarrow$
  $\exists\, d' : \mathsf{Time} \cdot gas\_st\ @\ (t_\alpha + d') = SAFE)$
end *Gas_Check*

$spec_3$ and $spec_4$ can be refined by protected objects *Motor* and *Gas_Status* (**Law. 4**) producing respectively:

Object *Motor* Pr
provide $set\_on(d)$, $set\_off(d)$
shunts
  $ctrl : \{ON, OFF\} \times \mathsf{Time}$
methods
  $set\_on(d)$
    variables $d : \mathsf{Time}$
  $\{ctrl\}$:
  $ctrl.v@t_\alpha = OFF \Rightarrow$
    $\mathsf{write\_sh}(ON, ctrl, [t_\alpha, t_\omega]) \wedge$
    $t_\omega \Leftrightarrow t_\alpha \leq d$
  end
  $set\_off(d)$
    variables $d : \mathsf{Time}$
  $\{ctrl\}$:
  $ctrl.v@t_\alpha = ON \Rightarrow$
    $\mathsf{write\_sh}(OFF, ctrl, [t_\alpha, t_\omega]) \wedge$
    $t_\omega \Leftrightarrow t_\alpha \leq d$
  end
end *Motor*

Object *Gas_Status* Pr
provide $read(s,d)$, $write(s,d)$
variables
  $gas\_st : \{SAFE, UNSAFE\}$
methods
  $read(s,d)$
    variables
      $s : \{SAFE, UNSAFE\}$
      $d : \mathsf{Time}$
  $\{s\}$:
  $s @ t_\omega = gas\_st @ t_\alpha \wedge$
    $t_\omega \Leftrightarrow t_\alpha \leq d$
  end
  $write(s,d)$
    variables
      $s : \{SAFE, UNSAFE\}$
      $d : \mathsf{Time}$
  $\{gas\_st\}$:
  $gas\_st @ t_\omega = s @ t_\alpha \wedge$
    $t_\omega \Leftrightarrow t_\alpha \leq d$
  end
end *Gas_Status*

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.

Object *Motor* Pr
provide $set\_on()$, $set\_off()$
shunts
  $ctrl : \{ON, OFF\} \times \mathsf{Time}$
variables
  $x : \{ON, OFF\}$
  $t : \mathsf{Time}$
methods
  $set\_on()$
    $(x,t) \leftarrow ctrl$;
    if $x = OFF$ then
      $(ON, \mathsf{now}) \rightarrow ctrl$
    endif
  end
  $set\_off()$
    $(x,t) \leftarrow ctrl$;
    if $x = ON$ then
      $(OFF, \mathsf{now}) \rightarrow ctrl$
    endif
  end
end *Motor*

Figure 2: Object *Motor*

Object *Gas_Status* Pr
provide $\{read(s), write(s)\}$
variables
  $gas\_st : \{SAFE, UNSAFE\}$
methods
  $read(s)$
    $s : \{SAFE, UNSAFE\}$
    $s := gas\_st$
  end
  $write(s)$
    $s : \{SAFE, UNSAFE\}$
    $gas\_st := s$
  end
end *Gas_Status*

Figure 3: Object *Gas_Status*

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 observativeness.

Object *Gas_Check* C
used
  $(Gas\_Status, write(s))$,
  $(Gas\_Status, read(s)$
period 20
shunts
  $level : \mathcal{N} \times \mathsf{Time}$
variables
  $x : \mathcal{N}$
  $t : \mathsf{Time}$
  $s : \{SAFE, UNSAFE\}$
thread
  $(x,t) \leftarrow level$;
  if $x > 40$ then
    duration d
    $Motor.set\_off()$;
    $Gas\_Status.read(s)$;
    if $s = SAFE$ then
      $Gas\_Status.write(UNSAFE)$;
    endif
  end
  orif $x \leq 40$ then
    duration d
    $Motor.set\_on()$;
    $Gas\_Status.read(s)$;
    if $s = UNSAFE$ then
      $Gas\_Status.write(SAFE)$;
    endif
  end
  endif
end *Gas_Check*

Figure 4: Object *Gas_Check*

Object *Operator* S
used
  $(Gas\_Monitor, check(s))$,
  $(Motor.set\_on())$
interval 10
shunts
  $cmd : \{START, STOP\} \times \mathsf{Time}$
variables
  $x : \{START, STOP\}$
  $t : \mathsf{Time}$
  $s : \{SAFE, UNSAFE\}$
thread
  duration 5
    $(x,t) \leftarrow cmd$;
    $Gas\_Monitor.check(s)$;
    if $x = START \wedge s = SAFE$ then
      $Motor.set\_on()$
    orif $x = STOP$ then
      $Motor.set\_on()$
    endif
  endif
endif *Operator*

Figure 6: Object *Operator*

Object *Gas_Monitor* A
include
  Gas_Check, *Gas_Status*
provide $check(s)$
methods
  $check(s)$: $Gas\_Status.read(s)$
end *Gas_Monitor*

Figure 5: Object *Gas_Monitor*

Object *Motor_Control_System* A
include
  Operator, *Gas_Monitor*, *Motor*
end *Motor_Control_System*

Figure 7: Design of the System

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 *REQ*, to produce sub-requirements: req_1, req_2, ..., req_n.
2. Formalise each sub-requirement req_i using the specification statement of TAM to produce $spec_1$, $spec_2$, ..., $spec_n$. Note that the formal specification, *SPEC*, which corresponds to *REQ*, is given by

$$SPEC =_{def} \bigwedge_{i \in [1,n]} spec_i$$

3. Use the refinement calculus to refine $spec_i$ into a concrete object obj_i, i.e.,

$$spec_i \sqsubseteq obj_i$$

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.

A characteristic of our approach is that during the refinement stages, all necessary timing information may be gathered in the form of 'proof-obligations'. These obligations are obviously proved correct (as a result of the soundness of the refinement laws) and are vital to scheduling theorists. Once these obligations are available, various scheduling tests and analysis may be applied. In fact these tests could also be applied after each refinement step; if the test is not valid then the step is repeated until the obligation is satisfied.

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

[1]. Burns, A. and Wellings, A., HRT-HOOD: A Structured Design Method for Hard Real-Time Systems, Elsevier, 1995.

[2]. European Space Agency, HOOD Reference Manual Issue 3.0, WME/89-353/JB, December 1989.

[3]. Chen, Z., Formal Methods for Object-Oriented Paradigm Applied to the Engineering of Real-Time Systems: A Review, TechReport, De Montfort University, 1997.

[4]. Scholefield, D. and Zedan, H., TAM: A Formal Framework for the Development of Distributed Real-Time Systems. Symposiums on Formal Techniques in Real-Time and Fault Tolerant Systems, Nijmegen, Netherlands, January 1992.

[5]. Scholefield, D. and Zedan, H., A Collection of Papers on Temporal Agent Model, FSR Group, University of York, 1993.

[6]. Scholefield, D., Zedan, H. and He, J., 'A Specification Oriented Semantics for the Refinement of Real-Time Systems' , Journal of Theoretical Computer Science, Vol. 129, No. 1, pp. 219-241 (1994).

[7]. Lowe, G. and Zedan, H., "Refinement of Complex Systems: A Case Study", The Computer Journal, Vol. 38, No. 10 (1995).

[8]. Lano, K., Distributed System Specification in VDM++, FORTE'95 Proceedings, Chapman and Hall, 1995.

[9]. Polack, F., Whiston, M. and Mander, C., The SAZ Method, TechReport, YCS207, University of York, 1993.

[10]. Semmens, L. and Allen, P., Using Yourdon and Z: an Approach to Formal Specification, Proceedings of 5th Annual Z User Meeting, Oxford, Springer-Verlag, 1991.