Integrating Structured OO Approaches with Formal Techniques for the Development of Real-time Systems

Z. Chen, A. Cau, H. Zedan [1] and H. Yang

Software Technology Research Laboratory,
SERCentre,
De Montfort University,
The Gateway, Leicester LE1 9BH, England,
http://www.cms.dmu.ac.uk/STRL/

Abstract

The use of formal methods in the development of time-critical applications is essential if we want to achieve a high level of assurance in them. However, these methods have not yet been widely accepted in industry as compared to the more established structured and informal techniques. A reliable linkage between these two techniques will provide the developer with a powerful tool for developing a provably correct system. In this paper, we explore the issue of integrating a real-time formal technique, TAM (Temporal Agent Model), with an industry-strength structured methodology known as HRT-HOOD. TAM is a systematic formal approach for the development of real-time systems based on the refinement calculus. Within TAM, a formal specification can be written (in a *logic-based* formalism), analysed and then refined to concrete representation through successive applications of sound refinement laws. Both *abstract* specification and *concrete* implementation are allowed to freely intermix. HRT-HOOD is an extension to the Hierarchical Object-Oriented Design (HOOD) technique for the development of Hard Real-Time systems. It is a two-phase design technique dealing with the logical and physical architecture designs of the system which can handle both *functional* and *non-functional* requirement, respectively. The integrated technique is illustrated on a version of the mine control system.

Key words: object-oriented, refinement calculus, Temporal Agent Model, semantics, HRT-HOOD

[1] The author wishes to acknowledge the funding received from the U.K. Engineering and Physical Sciences Research Council (EPSRC) through the Research Grant GR/M/02583

1 Introduction

The reliability of real-time safety critical systems that must meet not only functional but also stringent timing requirements has been a major and important research issue in recent years. As a result, it has been widely acknowledged that formal methods, with their mathematically rigorous notations and their associated verification techniques, are essential approaches to ensure the correctness of the development and hence increase our dependability on them. For this purpose, a large number of techniques have been developed which can be generally classified as assertional methods such as VDM [12], Z [33], and B-Method [1], temporal logic such as RTTL [24], MTL [13], XCTL [9] and ITL [23], process algebra such as CSP [10], CCS [22], ACP [2] and LOTOS [34], and Petri nets [25] such as time Petri Nets [21] and timed Petri Nets [27]. However, these techniques, with the exception of a few, have not been widely accepted by industry. Even those which are accepted (e.g., Z and VDM), their use have remained within limited scope of the whole development process. The requirement of adequate mathematical skills to both read/write formal specifications and perform formal proofs of correctness has often been attributed to the limited use of formal techniques.

At the other end of the spectrum, *structured* methods are well established and are heavily used by industry. Examples include Structured System Analysis and Design Methodology (SSADM) [20], Yourdon [35] and Jackson [11] for non-real-time systems. In addition, ROOM [6] and HRT-HOOD [3] are examples used for real-time applications.

Although they are well defined and permit the precise specification of systems structure, *structured* methods do not, in general, have a sound mathematical basis which is an essential criterion for the development of provably correct systems.

Various attempts have been made on integrating structured and formal techniques. In [19,32], both SSADM and Yourdon were integrated with the formal notation Z respectively. An attempt to incorporate data flow diagrams into the formal specification notation VDM was done in [8,26]. Recently, Liu, et al, provided a method that integrates both formal techniques, structure methodologies and Object-Oriented paradigm [17].

The main objective of the present work is to explore approaches that link between formal and structured methodologies with the aim that such a linkage should result in a method that enjoys the advantages of both techniques and, in addition, it should provide us with a well integrated software development process. This will certainly help to increase the applicability of formal techniques to large scale industrial applications and also provides the structured methodologies with the required mathematical underpinning.

To achieve our goal, we chose Hard Real-Time Hierarchical Object-Oriented De-

sign (HRT-HOOD) [3] as a representative of an industry-strength structured methodology which is being widely used within the space industry (e.g., European Space Agency). As for formal techniques we decided to use an extension to the Temporal Agent Model (TAM) [31,30,29,18] which is an example of a well developed technique based on formal refinement. The extension to TAM is rather conservative to cater for the object model present in HRT-HOOD.

The proposed linkage may be summarised as follows. HRT-HOOD is used to decompose original system's requirements into some manageable sub-requirements. Each sub-requirement is formalised, using our extended TAM Specification statement which is subsequently refined into concrete code in an object-oriented style. This can then be transformed into an equivalent, industrially accepted, programming language, such as Ada. Provision for resource allocation and scheduling issues may also be addressed in a similar fashion as discussed in our earlier work [4,5,18].

The paper is organised as follows. A brief overview of HRT-HOOD and TAM is given in Section 2. Section 3 introduces the computational object model for extended TAM. The syntax and semantics of extended TAM are given in Section 4 and Section 5, respectively. The refinement calculus of TAM is given in Section 6. Section 7 describes in some detail our integrated method and we demonstrate its application through a case study in the Section 8.

2 HRT-HOOD and The Temporal Agent Model

In this section we briefly outline both HRT-HOOD and TAM. For more details the reader should refer to the published materials [3,31,30,29,18].

2.1 HRT-HOOD

HRT-HOOD (Hard Real-Time HOOD) [3] is a design method for real-time systems in general, and hard real-time systems in particular. It is based on the HOOD (Hierarchical Object-Oriented Design) [28] method.

HRT-HOOD development process is divided into two phases: a *logical* followed by a *physical* architecture phase. The former supports hierarchical decomposition of the functional requirement of the system. This results in a collection of objects of various types and properties. The later addresses system's non-functional requirements and the constraints of the underlying execution environment.

There are five types of objects in HRT-HOOD. These are:

Active Objects may control the invocations of their operations, and may spontaneously invoke operations in other objects. Active objects are the most general class of objects and have no restriction placed on them.

Passive Objects have no control over the invocations of their operations, and do not spontaneously invoke operations in other objects.

Protected Objects may control the invocations of their operations, and do not spontaneously invoke operations in other objects. In general, protected objects may not have arbitrary synchronisation constraints and must be analysable for the blocking times they impose on their callers.

Cyclic Objects represent periodic activities; they may spontaneously invoke operations in other objects, but the only operations they have are requests which demand immediate attention.

Sporadic Objects represent sporadic activities; they may spontaneously invoke operations in other objects. Each sporadic object has a single operation which is called to invoke the sporadic activities, and one or more operations which are requests which demand immediate attention.

In the logical design, a collection of terminal objects (any objects except the *active* ones) is derived through some form of functional decomposition which implies timing requirements. The decomposition is represented by the *include* relationship. A parent object at a high abstraction level includes its child objects at lower levels. The control flow between objects is represented by the *use* relationship. An object can use methods defined in other objects, however, used methods can only be visible outside the parent object of the used objects.

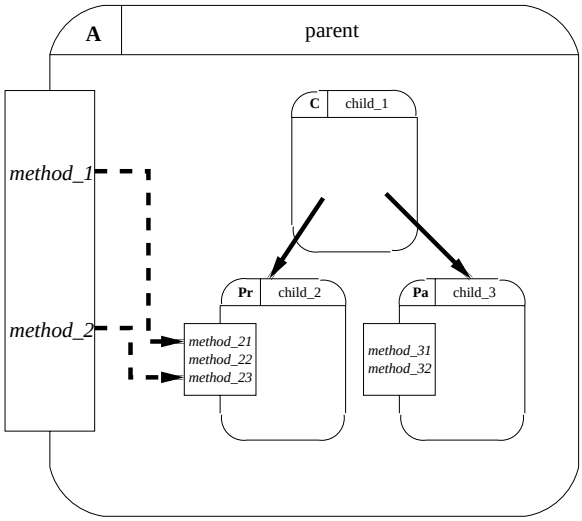


Fig. 1. HRT-HOOD Objects

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, provides the developer with a concise, cohesive, and powerful set of capabilities. Figure 1 illustrates the graphical representation of objects in HRT-HOOD.

2.2 TAM

The Temporal Agent Model(TAM) [31,30,29,18] was developed to be a realistic formal software development method for real-time systems. The method is based on refinement calculus and consists of a logic, a wide-spectrum language and a refinement calculus.

2.2.1 Computational Model

A real-time system in TAM is taken to be a finite collection of possibly concurrently executing computation agents which communicate asynchronously via time-stamped shared data areas called shunts. Shunts are passive shared memory spaces that contain two values: the first gives the time at which the most recent write took place, and the second gives the value that was most recently written. Systems themselves can be viewed as single agents and composed into larger systems.

At any time, a system can be thought of having a unique state, defined by the values in the shunts and local variables. An agent is described by a set of computations, which may transform a local data space and may read and write shunts during execution. The computation may be nondeterministic. In particular:

- Time is global, i.e., a single clock is available to every agent and shunt. The time domain is discrete, linear, and modelled naturally by the natural numbers.
- No state change may be instantaneous.
- An agent may start execution either as a result of a write event on a specific shunt, or as the result of some condition on the current time: these two conditions model sporadic and periodic tasks respectively.
- An agent may have deadlines on computations and communication. Deadlines are considered to be hard, i.e., there is no concept of deadline priority, and all deadlines must me met by the run-time system. We are currently investigating the inclusion of prioritised deadlines into the language.
- A data space is created when an agent starts execution, with nondeterministic initial values; the data space is destroyed when the agent terminates. No agent may read or write another agent's local data space.
- A system has a static configuration, i.e., the shunt connection topology remains fixed throughout the lifetime of the system.
- An agent's output shunts are owned by that agent, i.e., no other agent may write to those shunts, although many other agents may read them.
- Shunt writing is destructive, but shunt reading is not.

2.2.2 TAM Syntax

Agents in TAM are described as follows.

$$
\begin{aligned}
A ::= {} & w : \Phi \mid \mathit{Skip} \mid \Delta t \mid x := e \mid x \leftarrow s \mid e \rightarrow s \mid A;\, A' \mid \\
& \mathsf{var}\, x : T \,\mathsf{in}\, A \mid \mathsf{shunt}\, s : T \,\mathsf{in}\, A \mid [t]\, A \mid \mathsf{if}_t \;\; \Box_{i \in I}\, g_i \,\mathsf{then}\, A_i \;\; \mathsf{fi} \mid \\
& A \sqcap A' \mid A \rhd_t^s A' \mid A \parallel A' \mid \mathsf{loop\,for}\, n \,\mathsf{period}\, t \;\; A .
\end{aligned}
$$

where w is a set of computation variables and shunts; Φ is a predicate in the *TAM Logic Language*, which we define below; t is a time; x is a variable of type T; e is an expression on variables; s is a shunt of type $\mathit{Time} \times T$; I is some finite indexing set; g_i is a boolean expression; and n is a natural number.

Informally:

- $w : \Phi$ is a specification statement. It specifies that only the variables in the *frame* w may be changed, and the execution must satisfy Φ. Φ is a formula expressed in the TAM logic (see below).
- The agent *Skip* may terminate after any delay.
- The agent Δt terminates after t time units.
- $x := e$ evaluates the expression e, using the values found in variables at the start time of the agent, and assigns it to x. The expression e may not include the values held in shunts: it may only use the values held in variables.
- $x \leftarrow s$ performs an input from shunt s, storing the value in x; the type of x must be a time–value pair.
- $e \rightarrow s$ writes the current value of expression e to shunt s, time-stamping it with the time of the write.
- $A;\, A'$ performs a sequential composition of A and A'.
- $\mathsf{var}\; x : T \,\mathsf{in}\, A$ defines x to be a new local variable of type T within A; its initial value is chosen nondeterministically.
- $\mathsf{shunt}\; s : T \,\mathsf{in}\, A$ defines s to be a new local shunt of type $\mathit{Time} \times T$ within A; its initial value is chosen nondeterministically, but it is time-stamped with the time of its declaration.
- $[t]\, A$ gives agent A a duration of t: if the agent terminates before t seconds have elapsed, then the agent should idle to fill this interval; if the agent does not terminate within t seconds, then it is considered to have failed.
- $\mathsf{if}_t \;\; \Box_{i \in I}\, g_i \,\mathsf{then}\, A_i \;\; \mathsf{fi}$ evaluates all the boolean guards g_i, and executes an A_i corresponding to a true guard; if all the guards evaluate to false, then the agent terminates correctly. The evaluation of the guards should take precisely t time units; if necessary, the agent should idle to fill this interval. We shall sometimes omit the parameter t if we do not want to specify it. We shall sometimes write this construct as $\mathsf{if}_t\; g_1 \,\mathsf{then}\, A_1 \,\Box\, g_2 \,\mathsf{then}\, A_2 \,\Box \ldots \Box\, g_n \,\mathsf{then}\, A_n \,\mathsf{fi}$.
- $A \sqcap A'$ forms a nondeterministic choice between A and A'.
- $A \rhd_t^s A'$ monitors shunt s for t time units: if a write occurs within this time, then it executes A'; otherwise it times-out and executes A.
- $A \parallel A'$ executes the two agents concurrently, terminating when both agents ter-

minate.

- loop for n period t A executes A n times, giving each a duration of t.

We note here that no agent may share its local state space with concurrently executing agents, and only one concurrent agent may write to any given shunt: these restrictions allow the development of a compositional semantics and refinement calculus.

2.2.3 TAM Semantics

The semantics of TAM is given in terms of a predicate in the *TAM Logic Language* (TAMLL). This is a first-order logic with simple extensions to deal with times and the values held in variables and shunts. Formulae in the TAM Logic Language may include two distinguished variables: t_α representing the start time of the execution; and t_ω representing the termination time. It may also use terms of the form *x@t* and *s@t* to refer to the values held in variables and shunts at time *t*. We also use *s.ts@t* and *s.v@t* to refer to the time-stamp and value of a shunt.

TAMLL is powerful enough to express liveness and timeliness properties. Some safety properties may also be expressed. For example, consider a simple real-time control system in which an integer is read from a shunt *in* within 10 time units. After calculating the square of the integer, the system outputs the results to the shunt *out*. It is assumed that the shunt *in* is constrained by the environment, but that the shunt *out* is entirely under the control of the system we are specifying. The **liveness** and **timeliness** property of the system is thus captured by the following TAMLL formula:

$$\exists \sigma : N \cdot \sigma \in [t_\alpha, t_\omega] \wedge out.v@t_\omega = (in.v@\sigma)^2 \wedge t_\omega \leq t_\alpha + 10$$

Importantly, we must realise that no other value is written to the shunt *out* during the execution of the system, and so we provide a **safety** requirement that asserts that during the execution of the system, there is only one write to the shunt *out*. This can be done by counting the number of time-stamps which appear in *out* that are different to the time-stamps found at time t_ω:

$$\#\{n \mid \exists \sigma : N \cdot \sigma \in [t_\alpha, t_\omega] \wedge out.ts@\sigma = n \wedge n \neq out.ts@t_\omega\} = 1$$

The *specification-oriented* semantics for TAM is given by defining the semantic function:

$$F : TAM \rightarrow TAMLL,$$

such that $F\ [\![A]\!]$ gives the semantics of A. For example, the semantics of the specification statement is given by (the full semantics of TAM is given in [29]):

$$F\ [\![w : \Phi]\!] \triangleq \mathsf{stable}\,(W \setminus w, t_\alpha, t_\omega) \wedge \Phi,$$

where the predicate $\mathsf{stable}\,(x, t, t')$ specifies that the variable x remains constant between t and t'. Similar predicate is also defined for shunts.

3 Extended Computational Model

To integrate both HRT-HOOD and TAM, the underlying computational model of the TAM framework should be slightly modified to cater for the object structures found in HRT-HOOD.

A real-time system is thus viewed as a collection of concurrent *activities* which are initiated either periodically or sporadically, together with *services* that can be requested by the execution of the activities. The operations of the activities and services (in the form of *threads* and *methods*) are allocated to the corresponding *objects* according to their functional and temporal requirements and the relationships between them. An object is an encapsulated operation environment for the thread or methods.

Threads are activated and terminated with the corresponding objects and are concurrent with each other. *Methods* 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.

A *method* consists of a head and a body. The head specifies the method's name and its local environment (if necessary). The body specifies operations over either the object environment or the method environment, or both. The operations may be described at a high level of abstraction which can be refined to concrete implementation. A method and a thread are in fact TAM *agents*.

We define five types of objects, similar to those found in HRT-HOOD. These are:

(1) **sporadic object** — defines a unique thread which activates an operation sporadically by response to external events. The thread can not be requested and executed by other method's invocations. However, it can invoke methods provided by other objects. The thread may be concurrent with other activities in the system. A minimum interval can be specified to restrain responses to continuous event occurrences. Sporadic objects are used to model entities in a

system which are involved in random activities.

(2) **cyclic object** — is similar to a sporadic object except that its thread specifies an operation which is executed periodically. A cyclic object defines a period to specify how often the operation is executed and it is fixed. Every execution of the operation must be terminated within this period. Cyclic objects are used to model entities in a system which are involved in periodic activities.

(3) **protected object** — defines services which can be invoked. The services are implemented by *methods* which can be requested by others for execution. The methods can be requested arbitrarily, but their executions must be mutually exclusive. The execution order of invocations depends on their times of request. We use an invocation order based on "first come first serve". A method in a protected object can only request those methods which are (in)directly implemented by passive objects. Protected objects are used to model shared critical resources accessed by different objects or methods.

(4) **passive object** — is similar to a protected object except there are no constraints on invocations of its methods. A method in a passive object can be arbitrarily requested and immediately executed as a part of its client whenever being requested. A method in a passive object can only request the methods which are (in)directly implemented by other passive objects. Passive objects are used to define noninterfering operations on resources.

(5) **active object** — defines a framework for a number of *related* objects which are referred to as its *child objects*. An active object can be viewed as an independent system or subsystem. It encapsulates the methods of its child objects. Any object outside an active object can not request the methods defined in its child objects directly but through a method defined by it. The signature of a method defined in an active object must be consistent with that of its counterpart except its name. An active object can not include itself as a child object directly or indirectly and an object can not be a child object of different objects. An active object is used to model a composition process of systems or sub-systems.

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:

- **object type** — indicates if the object is either *active, sporadic, cyclic, protected* or *passive*.
- **provided methods** — provides signatures of the methods which can be invoked by other objects.
- **used methods** — declares the methods which will be invoked by the object and the objects which provide the methods.

Other attributes vary with the type of objects and may include:

- the activation interval of the thread for a cyclic object.

- the minimum activation interval of the thread for a sporadic object.
- the child object set for an active object.

The environment of an object is a set of data over which the methods of the object execute (i.e., set of variable and shunt names).

4 Objects and Methods in TAM

In this section we describe the necessary extension to TAM: *method*, *thread* and *object* structure.

4.1 Methods

A method *m* is defined in the form of

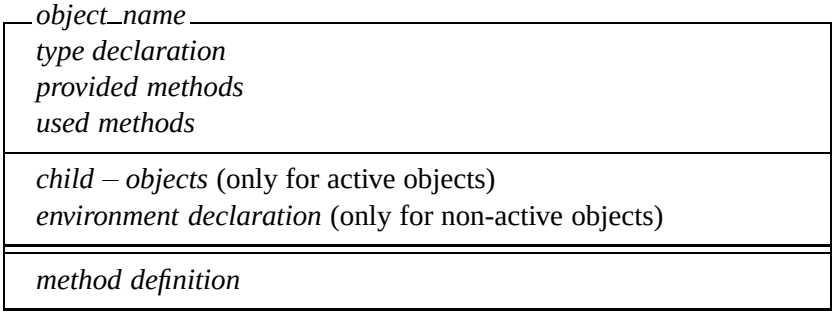
$$m([in, out]) \triangleq \mathsf{MthEnv}(m)\ A\ \mathsf{end}$$

where A (a TAM agent) is the body of the method *m*. *in* and *out* are its input and output parameter lists, respectively. We use A_m to denote the body of a method *m*.

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 $\mathsf{MthEnv}(m)$.

4.2 Objects and Threads

Objects are represented in the graphic notation:



where

- *type declaration*—indicates the object type. We use A, S, C, P and Pr to represent, respectively, that the object is either active, sporadic, cyclic, passive, or protected.
- *provided methods*—presents signatures of methods defined in the object which can be requested by other objects. We use $\mathsf{ProvidedMethods}(o)$ to denote the provided method set of an object o where o is sometimes dropped if no confusion is caused. The signatures must be accordant with their definitions. 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. $\#in \geq 0$ and $\#out \geq 0$. We use $in(m)$ and $out(m)$ to denote them.
- *used methods*—presents a declaration of the set of object and method name pairs indicating methods to be invoked and objects which provide the 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.
- *child − objects*—presents a declaration of child objects for active objects. 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.
- *environment declaration*—defines data and their domain for non-active objects. 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.
- *method definition*—specifies operations accomplished by methods defined in the object. We use $\mathsf{Methods}(o)$ to denote the defined method set of an object o.

Obviously,

$$\mathsf{ProvidedMethods}(o) \subseteq \mathsf{Methods}(o)$$

The operations are described by means of agents which may be either abstract or concrete.

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 shown in Fig. 2.

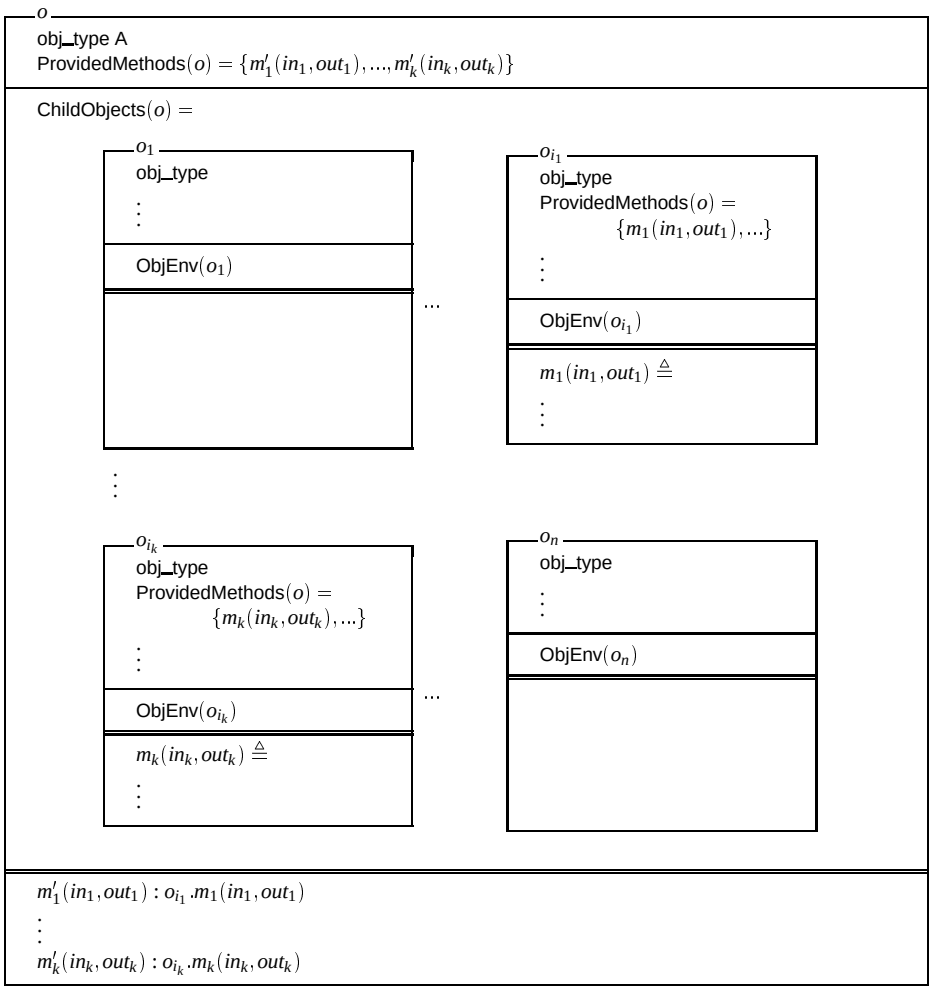


Fig. 2. Active Object

A *cyclic* or *sporadic* object defines a unique *thread* that operates periodically or sporadically. Let P and T be the activation period and the minimum activation interval, respectively, a cyclic and a sporadic object are given in the following forms.

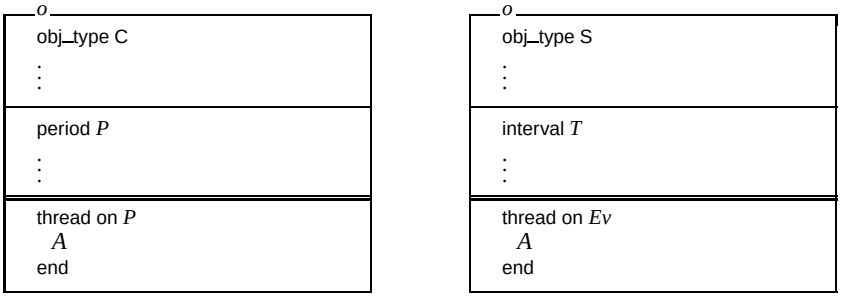


Fig. 3. Cyclic and Sporadic Object

Threads can not be requested by others, so no distinction between them is necessary. They are represented by a keyword thread. The event *Ev* in a sporadic object

is represented by a (set of) shunt(s), whose occurrence is caused by a write to these shunt(s).

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 only be executed exclusively while those defined in a passive object can be executed immediately when being requested:

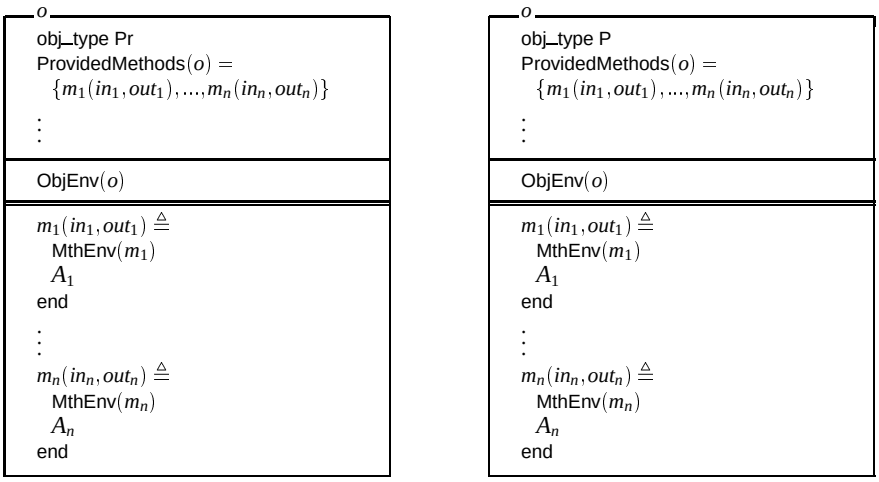


Fig. 4. Protected and Passive Object

5 Specification-Oriented Semantics

In this section we give a specification-oriented semantics for object and methods as defined in the last section.

An invocation is viewed as a special shunt whose value and time-stamp 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, which corresponds

with the invocation status of *Request*, *Activation* and *Termination* respectively. 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 free variables t_α and t_ω, in TAM, are 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 function $\copyright$, similar to the @ timing function for variables and shunts, is defined over pairs (x,t), where x is a an invocation, and t is a time, resulting in the invocation status and its occurrence time at the given time:

$$\copyright : \bigcup_{\text{all } m} \mathsf{Inv}_m \times \mathsf{Time} \rightarrow \mathsf{InvStatus} \times \mathsf{Time}$$

The corresponding projection functions for $\copyright$ are defined as $.v\copyright$ and $.ts\copyright$, and will result in an invocation status and its occurrence time respectively.

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

$$1\ \forall i \in [1, \mid \mathsf{Inv}_m \mid], t : \mathsf{Time} \cdot \mathsf{Inv}_m(i) \copyright t = (\mathsf{REQ}, t) \Rightarrow \mathsf{req\_time}(\mathsf{Inv}_m(i)) = t$$

$$2\ \forall i \in [1, \mid \mathsf{Inv}_m \mid], t : \mathsf{Time} \cdot \mathsf{Inv}_m(i) \copyright t = (\mathsf{ACT}, t) \Rightarrow \mathsf{act\_time}(\mathsf{Inv}_m(i)) = t$$

$$3\ \forall i \in [1, \mid \mathsf{Inv}_m \mid], t : \mathsf{Time} \cdot \mathsf{Inv}_m(i) \copyright t = (\mathsf{TER}, t) \Rightarrow \mathsf{ter\_time}(\mathsf{Inv}_m(i)) = t$$

It is clear that a normal invocation begins with request, and then activation and finally 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.

Following TAM, the specification-oriented semantics of the extension to include HRT-HOOD is given in terms of a predicate in the *TAM Logic Language* (TAMLL). To begin with, a few useful predicates are defined. A shunt *s* being written to at the time *t* can be specified by:

$$\mathsf{written}(s,t) \triangleq s.ts@t = t$$

A shunt *s* being written with a value *x* exactly once during time interval $[t_1, t_2]$ can

be specified as:

$$\mathsf{write\_sh}(x,s,t_1,t_2) \triangleq$$
$$\exists t \in [t_1,t_2] \cdot s@t = (x,t) \wedge \mathsf{stable}(\{s\},t_1,t-1) \wedge \mathsf{stable}(\{s\},t,t_2)$$

A method m is requested at a time t:

$$\mathsf{requested}(m,t) \triangleq \exists I \in \mathsf{Inv}_m \cdot \mathsf{req\_time}(I) = t$$

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

$$\mathsf{exclusive}(M) \triangleq$$
$$\forall m,n \in M, i_1 \in [1,\mid \mathsf{Inv}_m \mid], i_2 \in [1,\mid \mathsf{Inv}_n \mid] \cdot$$
$$\mathsf{act\_time}(\mathsf{Inv}_m(i_1)) \leq \mathsf{act\_time}(\mathsf{Inv}_n(i_2)) \leq \mathsf{ter\_time}(\mathsf{Inv}_m(i_1))$$
$$\Rightarrow m = n \wedge i_1 = i_2$$

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

The specification-oriented semantics of object, method and agent are given in the following subsections.

5.1 Agents

If a method invocation $o'.m'(\overline{in},\overline{out})$ is used in a method m in an object o, and

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

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

$$m'(in,out) \triangleq A \ \mathsf{end}$$

then the semantics of Invocation is defined as:

$$F\ [\![o'.m'(\overline{in},\overline{out})]\!] \triangleq$$
$$\mathsf{stable}(\mathsf{MthEnv}(m) \backslash \overline{out}, t_\alpha, t_\omega) \wedge$$
$$\exists t_1 \in [t_\alpha,t_\omega], i \in [1,\mid \mathsf{Inv}_{m'} \mid] \cdot \mathsf{req\_time}(\mathsf{Inv}_{m'}(i)) = t_1 \wedge$$
$$\exists t_2,t_3 \cdot t_1 \leq t_2 \leq t_3 \wedge \quad F\ [\![A[t_2/t_\alpha, t_3/t_\omega, \overline{in}/in, \overline{out}/out]]\!] \wedge$$
$$((t_\omega = t_1 \Leftrightarrow out(\mathsf{Inv}_{m'}(i)) = \emptyset) \vee (t_\omega = t_3 \Leftrightarrow out(\mathsf{Inv}_{m'}(i)) \neq \emptyset))$$

The definition indicates that

(1) *out* may be changed by calling a method.
(2) the method is executed at or after the time at which the request is received.
(3) If the calling is a control flow ($out = \emptyset$), then it terminates when the request is received. If the calling is a data flow ($out \neq \emptyset$), then it terminates with the execution of the method.

Suppose o is an active object which provides a method m implemented by a method m' defined in one of its children objects, o'. The semantics of Encapsulation is as follows:

$$F\ [\![\, m([in,out]) : o'.m'([in',out'])]\!] \triangleq$$
$$o' \in \mathsf{ChildObjects}(o) \wedge m' \in \mathsf{ProvidedMethods}(o') \wedge F\ [\![\, o'.m'([in',out'])]\!]$$

5.2 Objects

Suppose o is a cyclic, sporadic, protected, passive, or active object with a corresponding notation given in Section 4.

(1) Cyclic object
A cyclic object defines a periodic operation which is time-driven and can not be called by other objects. The period is specified by the field period. For a cyclic object o with period P and thread body A, we define

$$\begin{aligned} F\ [\![o]\!] \triangleq\ & \mathsf{ChildObjects}(o) = \emptyset \wedge \mathsf{ProvidedMethods}(o) = \emptyset \wedge \\ & \forall n \in N\ , \exists t_1 \in [t_\alpha, t_\omega] \cdot t_1 = P \times n \Rightarrow \\ & \exists t_2 \in [t_\alpha, t_\omega] \cdot t_1 \leq t_2 \leq t_1 + P \wedge F\ [\![A[t_1/t_\alpha, t_2/t_\omega]\ ]\!] \end{aligned}$$

(2) Sporadic object
A sporadic object defines a event-driven operation where an event is identified by a (set of) shunt(s) and only one response to the event is valid in the given interval which is specified by the field interval. For a sporadic object o with interval T and thread body A which responds to the event represented by a shunt set *Ev*, we define

$$\begin{aligned} F\ [\![o]\!] \triangleq\ & \mathsf{ChildObjects}(o) = \emptyset \wedge \mathsf{ProvidedMethods}(o) = \emptyset \wedge \\ & \exists t \in [t_\alpha, t_\omega] \cdot \bigwedge_{s \in Ev} \mathsf{written}(s,t) \wedge \forall t' \in [t_\alpha, t) \cdot t - s\ .ts@t' \geq T \\ & \Rightarrow \exists t'' \cdot t'' \leq t + T \wedge F\ [\![A[t/t_\alpha, t''/t_\omega]\ ]\!] \end{aligned}$$

(3) Protected object

A protected object provides a number of exclusive operations on shared data. For a protected object o, with A_m the body of a method m in o, we define

$$\begin{aligned} F\ [\![o]\!] \triangleq\ & \mathsf{ChildObjects}(o) = \emptyset \wedge \mathsf{ProvidedMethods}(o) \neq \emptyset \wedge \\ & (\forall m \in \mathsf{ProvidedMethods}(o), t \in [t_\alpha, t_\omega] \cdot \mathsf{requested}(m,t) \Rightarrow \\ & (\exists t' \geq t \cdot \mathsf{exclusive}(\mathsf{ProvidedMethods}(o)) \wedge F\ [\![A_m[t'/t_\alpha]]\!])) \end{aligned}$$

(4) Passive object
A passive object defines methods of which any is viewed as a part of the caller's when it is invoked. For a passive object o, with A_m the body of a method m in o, we define

$$\begin{aligned} F\ [\![o]\!] \triangleq\ & \mathsf{ChildObjects}(o) = \emptyset \wedge \mathsf{ProvidedMethods}(o) \neq \emptyset \wedge \\ & \forall m \in \mathsf{ProvidedMethods}(o), t \in [t_\alpha, t_\omega] \cdot \\ & \mathsf{requested}(m,t) \Rightarrow F\ [\![A_m[t/t_\alpha]]\!] \end{aligned}$$

(5) Active object
An active object defines a decomposition of a system/subsystem in which a number of objects are identified, namely its child objects. Methods in an active object are implemented by either its own agents or its child objects. For an active object o, we define

$$\begin{aligned} F\ [\![o]\!] \triangleq\ & \mathsf{ChildObjects}(o) \neq \emptyset \wedge \bigwedge_{o' \in \mathsf{ChildObjects}(o)} F\ [\![o']\!] \wedge \\ & \forall m \in \mathsf{ProvidedMethods}(o), t \in [t_\alpha, t_\omega] \cdot \\ & \exists o' \in \mathsf{ChildObjects}(o), m' \in \mathsf{ProvidedMethods}(o') \cdot \\ & \mathsf{requested}(m,t) \Rightarrow \mathsf{requested}(m',t) \end{aligned}$$

6 Refinement Calculus

As in TAM, we define the refinement as follows:

$$A \sqsubseteq A' \triangleq A \Leftarrow A'$$

If A' implies A, then A' refines A. It is easily shown that refinement is a pre-congruence: that is, if $A \sqsubseteq A'$ then $C(A) \sqsubseteq C(A')$ for any context $C(\_)$ written in terms of the TAM language. This means that if we refine one part of a program, then we refine the whole program.

A set of refinement laws are specified to transform an abstract specification into concrete objects. In the Appendix A, we give some useful refinement laws. Here

we give two example laws that refine a specification into, respectively a sporadic and a protected object.

Let

$$F\ [\![o]\!] \Rightarrow F\ [\![w : \Phi]\!]$$

where $F\ [\![o]\!]$ is the semantics of sporadic object o as defined in the previous section, and T is a constant of time, then

$$w : \Phi \sqsubseteq o$$

where o is in the form of the Sporadic Object of Fig. 3.

Let

$$F\ [\![o]\!] \Rightarrow F\ [\![w : \Phi]\!]$$

where $F\ [\![o]\!]$ is the semantics of protected object o as defined in the previous section, then

$$w : \Phi \sqsubseteq o$$

where o is in the form of the Protected Object of Fig. 4.

7 The Development Technique

Given an informal requirement of a system, *REQ*, the development process is described as follows.

(1) Use HRT-HOOD to decompose *REQ* to produce a set of sub-requirements: req_1, req_2, ..., req_n.

(2) Formalise each sub-requirement req_i using the specification statement to produce $spec_1$, $spec_2$, ..., $spec_n$. Note that the formal specification, *SPEC*, which corresponds to *REQ*, is given by

$$SPEC \triangleq \bigwedge_{i \in [1,n]} spec_i$$

(3) Using the refinement calculus, each specification $spec_i$ may be refined into an object obj_i:

$$spec_i \sqsubseteq obj_i$$

(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.

We note the following:

(a) In Step 1, the decomposition of *REQ* is left to the designer of the system, and various visual techniques are offered by HRT-HOOD. In this step, a *logical architecture* of the system is developed in which appropriate classes of objects, together with their timing properties are identified. We note here that in the logical architecture we do not address those requirements which are dependent on the physical constraints imposed by the execution environment. Such constraints as scheduling analysis are dealt with in a similar fashion as in [18].
(b) Due to compositionality, the final concrete system in the Step 4 is a refinement of *SPEC* as defined in the Step 2.
(c) Various properties may be proved at the specification level in the Step 2.

8 A Case Study

The case study used here is a simplified version of "The Mine Control System"[3], by keeping activities on motor and gas, and adding a sporadic activity initiated by the operator, as depicted in Fig. 5.

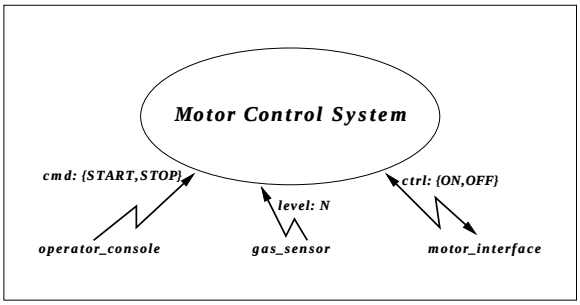


Fig. 5. Motor Control System

8.1 Requirements of the Motor Control System

We can express the system requirements *REQ* as:

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, as shown in Fig. 6.

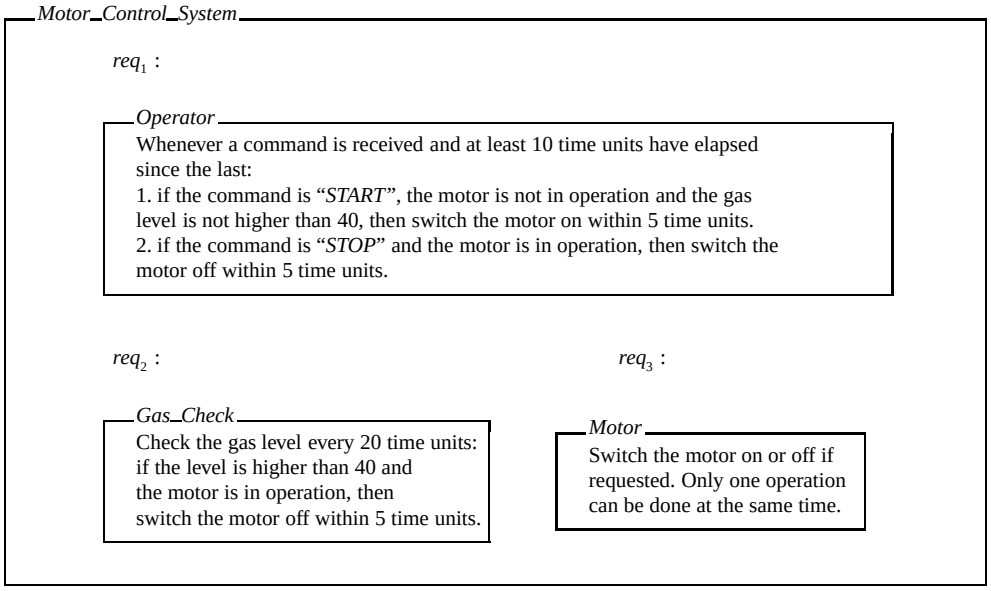


Fig. 6. Decomposition of the System

8.2 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:

$cmd : \{START, STOP\}$

$level : N$

$ctrl : \{ON, OFF\}$

Thus the system can be represented as shown in Fig. 7.

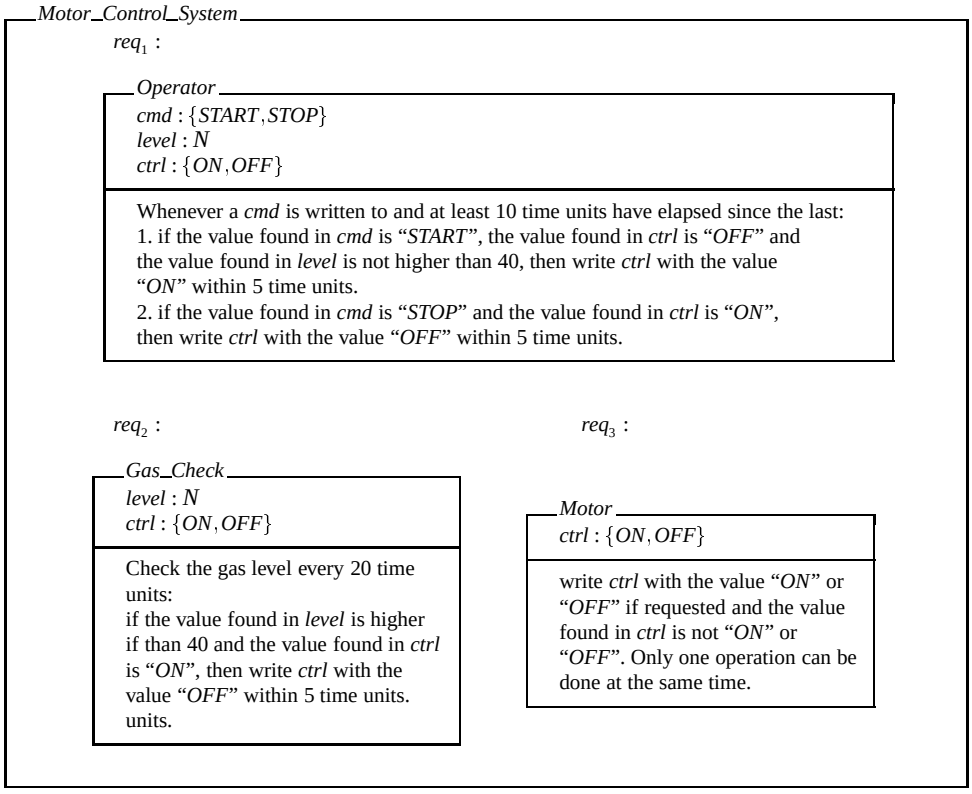


Fig. 7. Basic Objects of the System

Sub-specifications $spec_1$, $spec_2$ and $spec_3$ corresponding to req_1, req_2 and req_3 can be specified as:

Operator:

$$
\begin{aligned}
&spec_1 \triangleq \{ctrl\} : \\
&\forall t \in [t_\alpha, t_\omega], t' < t \cdot t - cmd.ts@t' \geq 10 \wedge \\
&\quad (cmd@t = (START, t) \wedge level.v@t \leq 40 \wedge ctrl.v@t = OFF \Rightarrow \\
&\qquad \exists d : \mathsf{Time} \cdot \mathsf{write\_sh}(ON, ctrl, t, t+d) \wedge d \leq 5) \vee \\
&\quad (cmd@t = (STOP, t) \wedge ctrl.v@t = ON \Rightarrow \\
&\qquad \exists d : \mathsf{Time} \cdot \mathsf{write\_sh}(OFF, ctrl, t, t+d) \wedge d \leq 5)
\end{aligned}
$$

Gas_Check:

$$
\begin{aligned}
&spec_2 \triangleq \{ctrl\} : \\
&\forall n \in N \cdot \exists t \in [t_\alpha, t_\omega] \cdot (t = 20 * n \wedge level.v@t > 40) \Rightarrow \\
&\quad (ctrl.v@t = ON \Rightarrow \exists d : \mathsf{Time} \cdot \mathsf{write\_sh}(OFF, ctrl, t, t+d) \wedge d \leq 5)
\end{aligned}
$$

Motor:

$$
\begin{aligned}
&spec_3 \triangleq \{ctrl\} : \\
&\forall t \in [t_\alpha, t_\omega] \cdot \\
&\quad \neg((ctrl.v@t_1 = OFF \Rightarrow \mathsf{write\_sh}(ON, ctrl, t, t+d) \wedge d \leq 5) \wedge \\
&\qquad (ctrl.v@t_1 = ON \Rightarrow \mathsf{write\_sh}(OFF, ctrl, t, t+d) \wedge d \leq 5)) \vee \\
&\quad \exists d : \mathsf{Time} \cdot \mathsf{stable}(ctrl, t, t+d)
\end{aligned}
$$

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 *gas_st* with values $\{SAFE, UNSAFE\}$ is introduced to represent the gas status. A corresponding specification $spec_4$ is defined as:

Gas_Status:

$$
\begin{aligned}
&spec_4 \triangleq \{gas\_st, s\} : \\
&\forall t \in [t_\alpha, t_\omega] \cdot \\
&\quad \neg((\exists t' \in [t, t+d] \cdot s@t' = gas\_st@t) \wedge \\
&\qquad (\exists t' \in [t, t+d] \cdot gas\_st@t' = s@t)) \vee \\
&\quad \exists d : \mathsf{Time} \cdot \mathsf{stable}(\{gas\_st, s\}, t, t+d)
\end{aligned}
$$

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

Gas_Check:

$$
\begin{aligned}
&spec'_2 \triangleq \{ctrl, gas\_st\} : \\
&\forall n \in N \cdot \exists t \in [t_\alpha, t_\omega] \cdot t = 20 * n \wedge \\
&\quad (level.v@t_\alpha > 40 \Rightarrow \\
&\qquad (ctrl.v@t_\alpha = ON \Rightarrow \exists d : \mathsf{Time} \cdot \mathsf{write\_sh}(OFF, ctrl, t_\alpha, t_\omega) \wedge d \leq 5) \wedge \\
&\qquad (gas\_st@t_\alpha = SAFE \Rightarrow \exists d' : \mathsf{Time} \cdot gas\_st@(t_\alpha + d') = UNSAFE)) \vee \\
&\quad (level.v@t_\alpha \leq 40 \wedge gas\_st@t_\alpha = UNSAFE \Rightarrow \\
&\qquad \exists d' : \mathsf{Time} \cdot gas\_st@(t_\alpha + d') = SAFE)
\end{aligned}
$$

The specification is then $SPEC \triangleq spec_1 \wedge spec'_2 \wedge spec_3 \wedge spec_4$.

The sub-specification $spec_1$ specifies a random operation based on the occurrence of the issuing command by the operator, which can be identified by a sporadic object, shown in Fig. 8

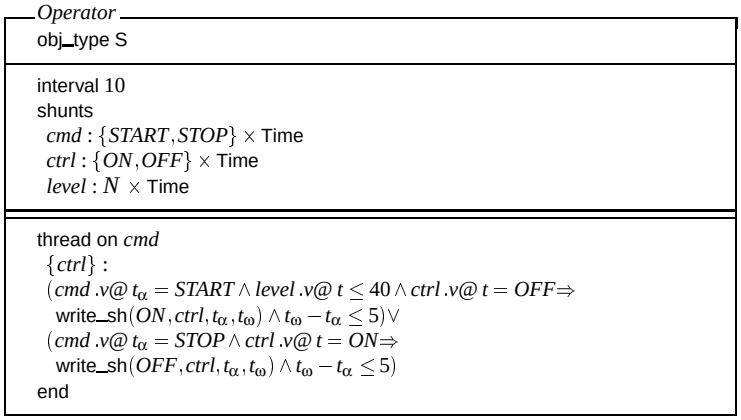


Fig. 8. Object *Operator*

$spec'_2$ describes an operation which starts every 20 time units. This can be defined by a cyclic object, shown in Fig. 9.

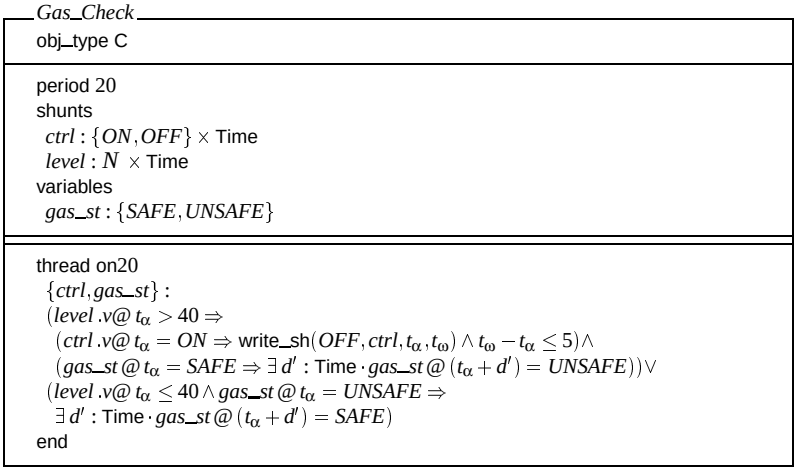


Fig. 9. Object *Gas_Check*

Both $spec_3$ and $spec_4$ define mutually exclusive operations on shared data, *ctrl*, *gas_st*, respectively. We introduce methods *set_on*, *set_off*, *read* and *write* to correspond to relative predicates. Thus they can be defined by protected objects *Motor* and *Gas_Status*, shown both in Fig. 10.

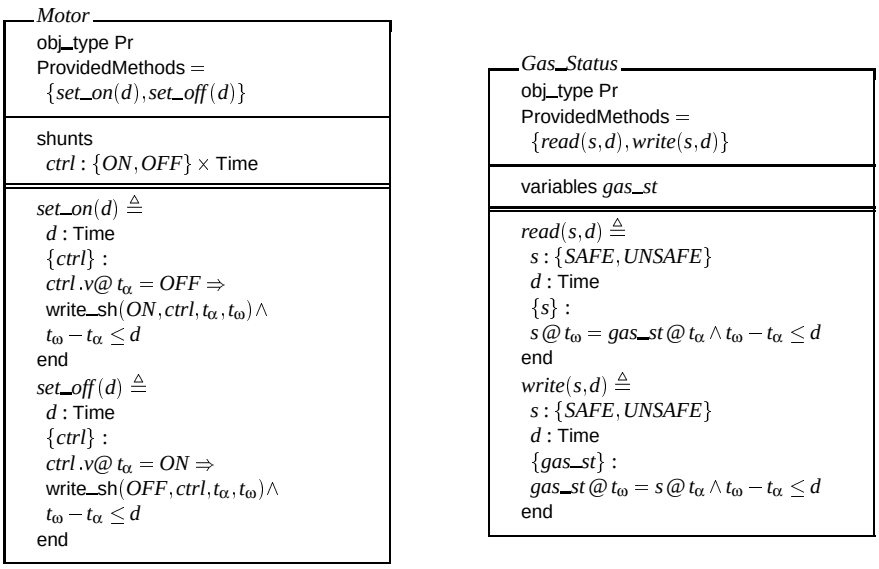


Fig. 10. Object *Motor* and object *Gas_Status*

8.4 Refinement

In this section, we use refinement laws listed in the Appendix A to refine objects constructed in the last subsection. They are shown in Figure 11- 14. Some time parameters are removed at the concrete level and time obligations.

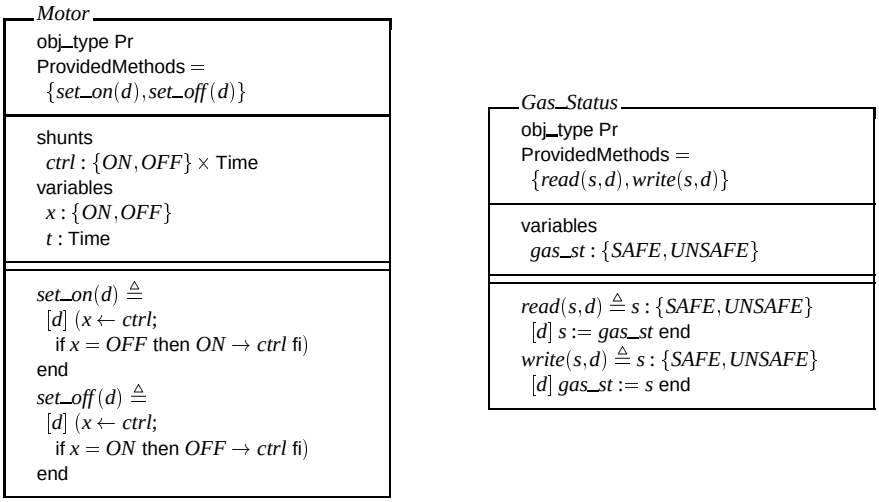


Fig. 11. Refined object *Motor* and refined object *Gas_Status*

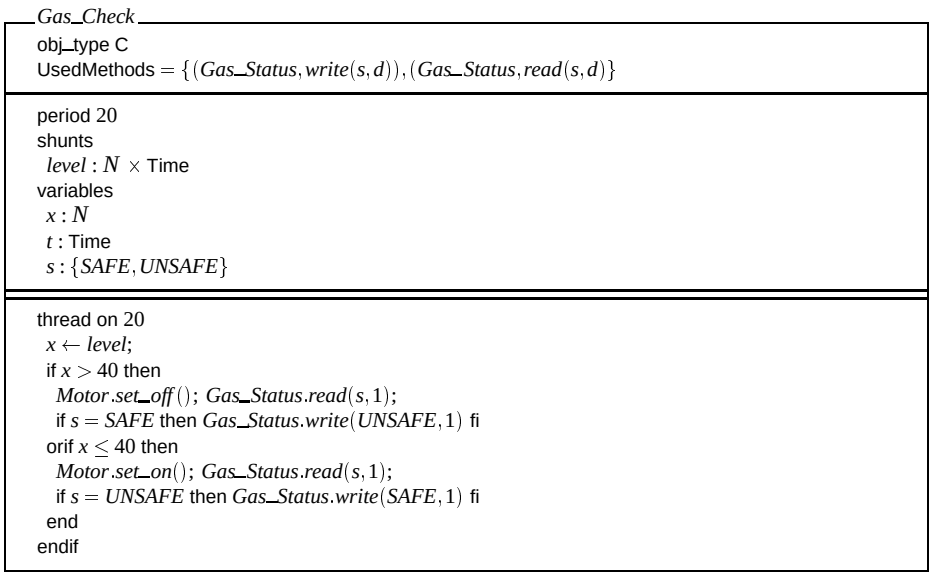


Fig. 12. Refined object *Gas_Check*

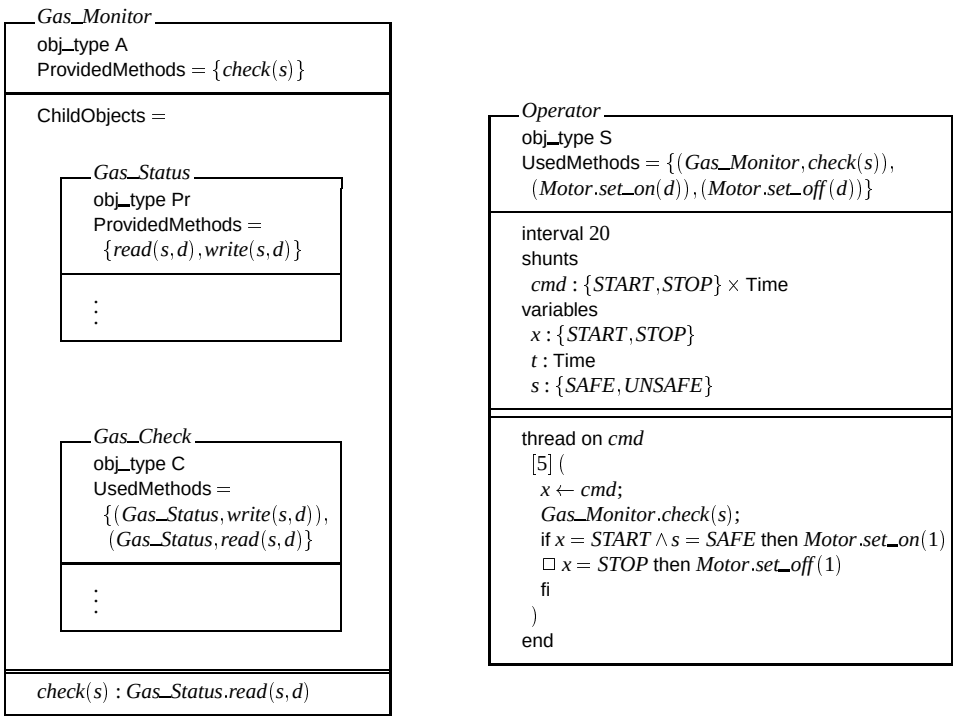


Fig. 13. Refined object *Gas_Monitor* and refined object *Operator*

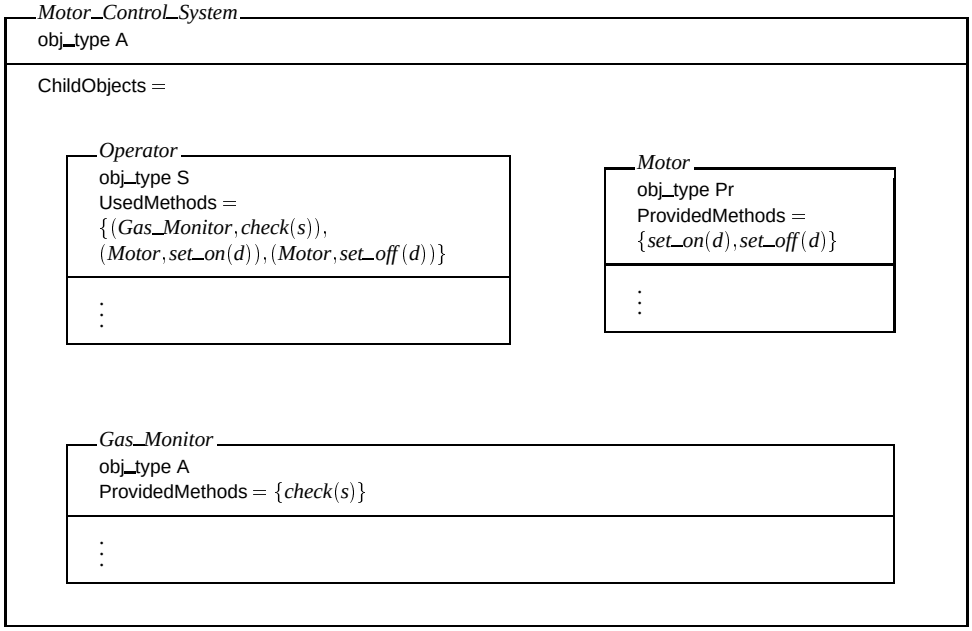


Fig. 14. Design of the System

9 Conclusions

The debate about the use and relevance of formal methods in the development of computing systems has always attracted a considerable attention and is continually doing so. One school of thought (the protagonists) hold the view that formal techniques offer a complete solution to the problems of system development. Another school (the detractors) claims that formal methods have little, or no, use in the development process (at least due to the cost involved).

For safety critical applications (in particular those which are *time-critical*), a high degree of confidence in their correct operations must be attained. This is due to the fact that a failure may lead to catastrophic consequences including loss of life and/or damage to the environment. For these sort of applications the use of formal methods have been put forward as techniques to ensure the correctness of the systems, as they have sound basis in mathematics.

However, the use of formal methods has not been as popular in industry as *structured* methodologies which are well defined and supported by the necessary software tools but do not have a sound basis in mathematics.

The main objective of the paper was to explore the issue of *linkage* between formal and structured methods for the development of time-critical applications. For this purpose, we have selected HRT-HOOD, as industry- strength object-oriented struc-

tured technique, and TAM, as a systematic formal development technique. TAM offers a sound calculus that allows the developer to freely intermix specifications at a high level of abstraction and concrete code. Through its associated calculus, a specification can be gradually and systematically be refined into concrete code [2]. This is achieved in a highly *compositional* manner allowing the development of large-scale complex systems. In addition, using the *delayed specification* technique (which we have developed in [18]), makes formalising many requirements much easier and less error prone. It allows the chaining of two such delayed specifications together: this is useful when we introduce new intermediate shunts which themselves satisfy a delayed specification. Using the *limited resource* model of TAM [18] allows us to calculate appropriate **scheduler** for the resulted concrete system.

We believe that the technique developed in this paper provides a solution to increase the accessability of formal methods. We need to try the technique on other case studies and investigate the design of an associated integrated tool support. We also would like to investigate how this technique could be integrated to standard **safety** techniques (such as *fault tree analysis*[16]). In addition, we are currently investigating the use of Interval Temporal Logic (**ITL**) [23,5,4] to replace the TAMLL. This will give us a more powerful logic to express a larger class of **liveness** properties. The result of this investigation will be reported elsewhere [7].

References

[1] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-method. In S. Prehn and W. J. Toetenel, editors, *VDM'91: Formal Software Development Methods, Volume 2*, volume 552 of *LNCS*, pages 398–405. Springer-Verlag, 1991.

[2] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

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

[4] A. Cau and H. Zedan. Refining interval temporal logic specifications. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development*, number 1231 in LNCS, pages 79–94. AMAST, Springer-Verlag, 1997.

[5] A. Cau, H. Zedan, N. Coleman, and B. Moszkowski. Using ITL and Tempura for Large Scale Specification and Simulation. In *Proceedings of Fourth Euromicro Workshop On Parallel And Distributed Processing*, pages 493–500, Braga, Portugal, 1996. IEEE.

[2] The existence of the refinement calculus makes TAM more powerful than others such as Z++ [14], VDM++ [15] and Duration Calculus [36] in which calculi only exist at a logical level

[6] B. Celic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[7] Z. Chen, A. Cau, H. Zedan, and H. Yang. A wide-spectrum language for object-based development of real-time systems. To appear in International Journal of Information Sciences, 1999.

[8] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and foraml requirements specification languages: Bridging the gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, May 1991.

[9] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 402–413, Philadelphia, Pennsylvania, 1990. IEEE Computer Society Press.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[11] M. A. Jackson. *System Development*. Prentice Hall, New Jersey, 1983.

[12] C. B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice-Hall, 1986.

[13] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[14] K. Lano. Z++. In J. E. Nicholls, editor, *Proceedings of Z User Workshop Oxford*. Springer-Verlag, 1990.

[15] K. Lano. Distributed system specification in vdm++. In *Proceedings of FORTE'95*. Chapman and Hall, 1995.

[16] N. G. Levenson and P. R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(9), 1983.

[17] S. Liu, A. J. Offutt, Y. Sun, and M. Ohba. Sofl: A formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1), 1998.

[18] G. Lowe and H. Zedan. Refinement of complex systems: a case study. *The Computer Journal*, 38(10), 1995.

[19] K. C. Mander and F. Polack. Rigorous specification using structured systems analysis and Z. *Information and Software Technology*, 37(5–6):285–291, 1995.

[20] M. Meldrum and P. Lejk. *SSADM techniques: an introduction to Version 4*. Chartwell-Bratt, 1993.

[21] P. M. Merlin and A. Segall. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, pages 1036–1043, September 1976.

[22] R. Milner. *A Calculus of Communicating Systems*. Number 92 in LNCS. Springer-Verlag, 1980.

[23] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, 18(2):10–19, February 1985.

[24] J. S. Ostroff and W. M. Wonham. A temporal logic approach to real time control. In *Proc. of 24th Conf. Decision and Control*, pages 6565–6567, Fort Lauderdale, FL, USA, December 1985.

[25] C. A. Petri. *Communication with Automata*. PhD thesis, Univ. Bonn, 1962.

[26] N. Plat, J. Katwijk, and K. Pronk. A case for structured analysis/formal design. In *Proceedings of VDM'91*, number 551 in LNCS. Springer-Verlag, 1991.

[27] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical Report MAC TR 120, MIT, February 1974.

[28] P. J. Robinson. *HOOD: Hierarchical Object-Oriented Design*. Prentice Hall, 1992.

[29] D. Scholefield, H. Zedan, and He Jifeng. A specification-oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 131(1):219–241, August 1994.

[30] D. J. Scholefield, H. Zedan, and J. He. A predicative semantics for the refinement of real-time systems. In *LNCS*, number 802, pages 230–249. Springer-Verlag, 1994.

[31] David Scholefield, Hussein Zedan, and Jifeng He. Real-time refinement: Semantics and application. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, volume 711 of *lncs*, pages 693–702, Gdansk, Poland, 1993. Springer.

[32] L. T. Semmens and P. M. Allen. Using Yourdon and Z: An approach to formal specification. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 228–253. Springer-Verlag, 1991.

[33] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cillfs, N.J., 1989.

[34] P. H. J. van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project*. NORTH-HOLLAND, 1989.

[35] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[36] C. Zhou, C. A. R. Hoare, and A. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

A Refinement Laws

Law. 1 (Strengthen specification)
If $\Phi' \Rightarrow \Phi$, then $w : \Phi \sqsubseteq w : \Phi'$

Law. 2 (Assignment)
If $x \in w$ then $w : (\exists t \in [t_\alpha, t_\omega] \cdot x@t_\omega = e@t) \sqsubseteq x := e$

Law. 3 (Input)
If $x, t \in w$, then $w : (\exists t' \in [t_\alpha, t_\omega] \cdot x@t_\omega = s.v@t') \sqsubseteq x \leftarrow s$

Law. 4 (Output)
If $x \in w$, then $w : \mathsf{write\_sh}(e, s, t_\alpha, t_\omega) \sqsubseteq e \rightarrow s$

Law. 5 (Invocation)
If a method

$$m'(in, out) \triangleq \mathsf{MthEnv}(m')\ A\ \ \mathsf{end}$$

is defined in an object o', B is an agent in an object o, and

$$B \sqsubseteq A[\overline{in}/in, \overline{out}/out]$$

then

$$B \sqsubseteq o'.m'(\overline{in}, \overline{out})$$

and if $w_B \subseteq w_{o'}$, then $w_o = w_o \backslash w_B$, where w_B, w_o and $w_{o'}$ are frames of B, o and o' respectively, and

$$\mathsf{UsedMethods}(o) = \mathsf{UsedMethods}(o) \cup \{(o', m'(in, out)\}$$

Law. 6 (Condition)
If $(\bigvee_{i \in [1,n]} (g_i \wedge \Phi_i) \vee (\bigwedge_{i \in [1,n]} \neg g_i \wedge \mathsf{stable}(w, t_\alpha, t_\omega))) \Rightarrow \Phi$, then

$$w : \Phi \sqsubseteq \text{ if } g_1 \text{ then } w : \Phi_1 \ldots \Box\ g_n \text{ then } w : \Phi_n \text{ fi}$$

Law. 7 (Duration)

$$w : \Phi \wedge t_\omega - t_\alpha = d \sqsubseteq [d]\ \ w : \Phi$$

Law. 8 (Method)
Suppose o_1, o_2 are two objects with the same type. If

$$\forall m' \in \mathsf{ProvidedMethods}(o_1), \exists m'' \in \mathsf{ProvidedMethods}(o_2) \cdot A_{m'} \sqsubseteq A_{m''}$$

then

$$o_1 \sqsubseteq o_2$$

Law. 9 (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

$$o_1 \sqsubseteq o_2$$

Law. 10 (Cyclic object)
Let

$$F\ [\![o]\!] \Rightarrow F\ [\![w : \Phi]\!]$$

where $F\ [\![o]\!]$ *is the semantics of cyclic object o as defined in Sect. 5, then*

$$w : \Phi \sqsubseteq o$$

where o is in the form of the Cyclic Object of Fig. 3

Law. 11 (Passive object)
Let

$$F\ [\![o]\!] \Rightarrow F\ [\![w : \Phi]\!]$$

where $F\ [\![o]\!]$ *is the semantics of passive object o as defined in Sect. 5, then*

$$w : \Phi \sqsubseteq o$$

where o is in the form of the Passive Object of Fig. 4

Law. 12 (Active object)
Let

$$F\ [\![o]\!] \Rightarrow F\ [\![w : \Phi]\!]$$

where $F\ [\![o]\!]$ *is the semantics of active object o as defined in Sect. 5, then*

$$w : \Phi \sqsubseteq o$$

where o is in the form of Fig. 2