ATOM: An Object-Based Formal Method for Real-time Systems

Hussein Zedan[1)], Antonio Cau, Zhiqiang Chen, Hongji Yang

Software Technology Research Laboratory,

SERCentre,

De Montfort University,

The Gateway, Leicester LE1 9BH, England,

http://www.cms.dmu.ac.uk/STRL/

June 25, 1999

[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

E-mail: zedan@dmu.ac.uk

Abstract

An object based formal method for the development of real-time systems, called **ATOM**, is presented. The method is an integration of the real-time formal technique TAM (Temporal Agent Model) with an industry-strength structured methodology known as HRT-HOOD. **ATOM** is a systematic formal approach based on the refinement calculus. Within **ATOM**, a formal specification (or abstract description statement) contains Interval Temporal Logic (ITL) description of the timing, functional, and communication behavior of the proposed real-time system. This formal specification can be analyzed and then refined into concrete statements through successive applications of sound refinement laws. Both *abstract* and *concrete* statements are allowed to freely intermix. The semantics of the concrete statements in **ATOM** are defined denotationally in *specification-oriented* style using ITL.

keywords: object-based, wide-spectrum language, refinement calculus, Temporal Agent Model, HRT-HOOD, Interval Temporal Logic

1 INTRODUCTION

The 'correctness' of real-time systems' design not only depends on satisfying *functional* requirements, as in most information processing systems, but also on *non-functional* requirements, such as timing, limited resources and dependability.

The development of a real-time system has been traditionally a somewhat ad-hoc affair. A system is designed from an informal requirements specification as a number of tasks with associated deadlines, execution periods, and resource requirements. The worst-case execution time is calculated for those tasks, and a resource allocation and schedule is computed which guarantees deadlines. Worst-case execution time, allocation, and scheduling are all complex procedures and research is still active in these areas; in the two latter cases the problems are known to be NP-complete. *Correctness* of systems developed in this way can only be performed by testing and detailed code inspection. However, when the consequence of system failure is catastrophic such as loss of life and/or damage to the environment, testing and code inspection can not alone be relied upon.

Therefore, there is clearly scope for *formalizing* some of the development process, particularly in the area of requirements specification and design [Fraser *et al.* 1991]. For this purpose, a large number of formalisms have been developed, for example RTL [F. Jahanian and A. Mok 1986], Timed CSP [Davies 1991], RTTL [Ostroff and Wonham 1985], MTL [Koymans 1990], XCTL [Harel *et al.* 1990], ITL [Moszkowski 1985], TAM [Scholefield *et al.* 1993; Scholefield *et al.* 1994b; Scholefield *et al.* 1994a; Lowe and Zedan 1995; He and Zedan 1996], TCSP [Schneider *et al.* 1992], TCCS [Yi 1991], TACP [Bergstra and Klop 1984] and time Petri Nets [Petri 1962; Merlin and Segall 1976; Ramchandani 1974].

However, we have shown [Chen 1997] that there are a significant number of limitations with existing real-time development formalisms. Most important of these is the lack of *method* or guidance on how to use a formalism for both specification writing and proving correctness. In addition, it is not clear how such formalisms can cope with the development of *large scale* real-time systems.

In real-time systems development we would benefit from a method which assists in the derivation of concrete designs from informal requirements specifications through a 'temporal' refinement notion.

A number of refinement calculi already exist for real-time systems, but they are either incomplete or use an unrealistic computational model. PL^{time} [He 1991] is a real-time design language which consists of a CSP-like syntax with extensions for real-time. However, the formalism is based on the maximal-parallelism hypothesis (i.e., the assumption that there are always sufficient resources available) which is too restrictive for most real time systems. In addition, since PL^{time} does not provide a separate specification statement as a syntactic entity, the refinement remains purely in the concrete domain. Similarly, RT-ASLAN [Auernheimer and Kemmerer 1986] is a refinement calculus which refines a specification into concrete code, but this again relies on the maximal parallelism model. The Duration Calculus [Zhou *et al.* 1991] (and to some extend timed Z [Hayes and Utting 1998] and B-method [Abrial *et al.* 1991]

in recent attempts), on the other hand, is a formalism based on ITL [Moszkowski 1985] and provides rules which are only applicable at the logical level of development.

Furthermore, with the advent of Object-Oriented (OO) paradigm, as a powerful approach in modeling and developing large-scale and complex software systems, research in object-oriented formalism has increased. This has ranged from extending process-oriented formalisms to cater for object structure (e.g., Z++ [Lano 1990] and VDM++ [Lano 1995]) to the development of new formal OO models (e.g., HOSA [Goguen and Diaconescu 1994; Malcom and Goguen 1994], Maude [Meseguer and Winkler 1992; Meseguer 1993], CLOWN [Battiston and Cindio 1993; Battiston *et al.* 1995; Battiston *et al.* 1996], CO [Bastide 1992; Bastide and Palanque 1993], COOPN/2 [Biberstein *et al.* 1996; Biberstein *et al.* 1997], TRIO+ [Morzenti and Pietro 1994] and OO-LTL [Canver and von Henke 1997]).

Although the use of formal methods in the development of real-time systems have their benefits, turning them into a sound engineering practice has proved to be extremely difficult. Some "pure" formal methods may keep practically-oriented software engineers from employing their benefits. This has led to investigating the integration of formal methods with well established structured techniques used by industry (e.g., System Analysis and Design Methodology (SSADM) [Meldrum and Lejk 1993], Yourdon [Yourdon 1989] and Jackson [Jackson 1983], for non-real-time systems, and ROOM [Celic *et al.* 1994] and HRT-HOOD [Burns and Wellings 1995] for real-time applications). As a result, in [Mander and Polack 1995; Semmens and Allen 1991], 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 [Fraser *et al.* 1991; Plat *et al.* 1991]. Recently, Liu, et al [Liu *et al.* 1998], provided a method that integrates both formal techniques, structured methodologies and the Object-Oriented paradigm. However, they still lack mechanisms for the systematic development of concrete design/code from formal specification. This has provisionally been addressed in [Chen *et al.* 1999].

The main objective of this paper is to provide a formal development technique whose underlying computational model is realistic and supports the development of large-scale systems. By realistic we take the view that it must reflect the basic developer's intuition about the target application area and that the resulting system can be analyzed for schedulability. In addition, to support large-scale system development, the computational model should adopt features advocated to in the OO paradigm.

The systematic derivation of a concrete design from an abstract specification requires that the formal development technique to be based on a wide-spectrum language in which concrete and abstract constructs can be freely intermixed. Further, a set of sound refinement laws must be provided enabling the software developers to transform a requirement specification into an executable program.

In this paper, we present a formal development technique **ATOM**. The formal language of **ATOM** contains both abstract and concrete statements. The development technique uses a refinement calculus to get from an abstract statement to a concrete statement. The concrete statements in the language include those of the Temporal Agent

Model (TAM) [Scholefield *et al.* 1993; Scholefield *et al.* 1994b; Scholefield *et al.* 1994a; Lowe and Zedan 1995]. The underlying computational structure of **ATOM** is an extension of TAM to cater for objects. The object structure in **ATOM** is based on that found in the industry strength structured technique known as HRT-HOOD [Burns and Wellings 1995]. HRT-HOOD is a real-time extension to HOOD [Robinson 1992]. The abstract statements in the language are Interval Temporal Logic [Moszkowski 1985] (ITL) formulae. ITL is also used to give a denotational semantics to the concrete statements so that abstract and concrete statements can be freely intermixed. The refinement calculus of **ATOM** is an extension of that of TAM to cater for the refinement into objects.

In Section 2, we introduce Interval Temporal logic. The computational object model of **ATOM** is detailed in Sect. 3. The syntax and informal semantics of the **ATOM** language are given in Sect. 4. The refinement calculus of **ATOM** is presented in Sect. 5. The systematic development technique is the outlined in Sect. 6 and illustrated with a small case-study in Sect. 7.

2 INTERVAL TEMPORAL LOGIC

We base our work on Interval Temporal Logic and its programming language subset Tempura [Moszkowski 1985]. ITL will be used both as our abstract specification language and to define the specification-oriented semantics of the concrete statements in **ATOM**.

Our selection of ITL is based on a number of points. It is a flexible notation for both propositional and first-order reasoning about periods of time. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL [Cau and Zedan 1997]. Tempura provides an executable framework for developing and experimenting with suitable ITL specifications.

2.1 Syntax

An interval is considered to be a (in)finite sequence of states, where a state is a mapping from variables to their values. The length of an interval is equal to one less than the number of states in the interval (i.e., a one state interval has length 0).

The syntax of ITL is defined in Table 1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol, p is a predicate symbol.

The informal semantics of the most interesting constructs are as follows:

- $\iota a{:}f$: the value of a such that f holds.

Table 1: Syntax of ITL

Expressions	
$e ::=$	$\mu \mid a \mid A \mid g(e_1,\ldots,e_n) \mid \imath a{:}f$
Formulae	
$f ::=$	$p(e_1,\ldots,e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathit{Skip} \mid f_1 ; f_2 \mid f^*$

- *Skip*: unit interval (length 1).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

These constructs enables us to define programming constructs like assignment, if then else, while loop etc. In table 2 some frequently used abbreviations are listed.

2.2 Data Representation

Introducing type system into specification languages has its advantages and disadvantages. An untyped set theory is simple and is more flexible than any simple typed formalism. Polymorphism, overloading and subtyping can make a type system more powerful but at the cost of increased complexity. While types serve little purpose in hand proofs, they do help with mechanized proofs.

There are two basic inbuilt types in ITL (which can be given pure set-theoretic definitions). These are integers N (together with standard relations of inequality and quality) and Boolean (*true* and *false*). In addition, the executable subset of ITL (Tempura) has basic types: integer, character, Boolean, list and arrays.

Further types can be built from these by means of $\times$ and the power set operator, P (in a similar fashion as adopted in the specification language Z).

For example, the following introduces a variable x of type T

$$(\exists x : T) \cdot f \quad \widehat{=} \quad \exists x \cdot \mathit{type}(T) \wedge f$$

Here $\mathit{type}(T)$ denotes a formula describing the desired type. For example, $\mathit{type}(T)$ could be $0 \leq x \leq 7$ and so on. Although this might seem to be rather inexpressive type system, richer type can be added following that of Spivey [Spivey 1996].

Table 2: Frequently used abbreviations

true	$\widehat{=}$	$0 = 0$	true value
false	$\widehat{=}$	$\neg$*true*	false value
$f_1 \vee f_2$	$\widehat{=}$	$\neg(\neg f_1 \wedge \neg f_2)$	or
$f_1 \supset f_2$	$\widehat{=}$	$\neg f_1 \vee f_2$	implies
$f_1 \equiv f_2$	$\widehat{=}$	$(f_1 \supset f_2) \wedge (f_2 \supset f_1)$	equivalent
$\exists v \bullet f$	$\widehat{=}$	$\neg\forall v \bullet \neg f$	exists
$\bigcirc f$	$\widehat{=}$	*Skip* ; f	next
more	$\widehat{=}$	$\bigcirc$*true*	non-empty interval
empty	$\widehat{=}$	$\neg$*more*	empty interval
inf	$\widehat{=}$	*true* ; *false*	infinite interval
finite	$\widehat{=}$	$\neg$*inf*	finite interval
$\Diamond f$	$\widehat{=}$	*finite* ; f	sometimes
$\Box f$	$\widehat{=}$	$\neg\Diamond\neg f$	always
$\circledcirc f$	$\widehat{=}$	*finite* ; f ; *true*	some subinterval
$\boxed{a} f$	$\widehat{=}$	$\neg(\circledcirc\neg f)$	all subintervals
f^0	$\widehat{=}$	empty	0-chopstar
f^{n+1}	$\widehat{=}$	f ; f^n	$(n+1)$-chopstar
if f_0 then f_1 else f_2	$\widehat{=}$	$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
fin f	$\widehat{=}$	$\Box$(empty $\supset$ f)	final state
keep f	$\widehat{=}$	$\boxed{a}$(*Skip* $\supset$ f)	all unit subintervals
$\bigcirc e$	$\widehat{=}$	$\iota a{:}\bigcirc(e = a)$	next value
fin e	$\widehat{=}$	$\iota a{:}$*fin* $(e = a)$	end value
$A := e$	$\widehat{=}$	$\bigcirc A = e$	assignment
$e_1 \leftarrow e_2$	$\widehat{=}$	*finite* $\wedge$ (*fin* e_1) $= e_2$	temporal assignment
e_1 *gets* e_2	$\widehat{=}$	*keep* $(e_1 \leftarrow e_2)$	gets
stable e	$\widehat{=}$	e *gets* e	stability
intlen (e)	$\widehat{=}$	$\exists I \bullet (I = 0) \wedge (I$ *gets* $I + 1) \wedge I \leftarrow e)$	interval length e
len	$\widehat{=}$	$\iota a{:}$*intlen* (a)	interval length

3 OBJECT-BASED COMPUTATIONAL MODEL

In this section we present our object-based model which is a conservative extension to that adopted in the Temporal Agent Model (TAM) [Scholefield *et al.* 1993; Scholefield *et al.* 1994b; Scholefield *et al.* 1994a; Lowe and Zedan 1995]. TAM was developed to be a realistic formal software development method for real-time systems. Such an extension is based on an industry-strength structured methodology known as HRT-HOOD [Burns and Wellings 1995].

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.

3.1 Object Structure

In **ATOM** we can identify five types of objects. These are defined as follows.

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 methods' 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 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. A method in a protected object can only request the 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 non-interfering 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.

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.

Threads/methods are *agents* as defined in TAM. Threads activate and terminate 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.

3.2 Agent Structure

An agent is described by a set of computations, which may transform a local data space. Communication is asynchronous 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. 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 modeled 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 prioritized 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.

4 ATOM LANGUAGE

ATOM has the following syntactic form. 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 the object is either *active, sporadic, cyclic, protected* or *passive*.
- *provided methods* — signatures of the methods provided by the object for 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. $\mathsf{card}(in) \geq 0$ and $\mathsf{card}(out) \geq 0$ (where card denotes cardinality of the set). We use $in(m)$ and $out(m)$ to denote them.
- *used methods* — declare the methods invoked by the object and the objects which provide the methods.
  We use $\mathsf{UsedMethods}(o)$ to denote the used method set of an object o where o is sometimes dropped if no confusion is caused. 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:

- 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. 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.

The syntax of the ATOM language is defined in Table 3 where A is a TAM agent; $\mathsf{ProvidedMethods}$ is a set of provided methods; Ev is a shunt; P, t and T are time variables; S a shunt name; w a set of computation variables and shunts; f an ITL formula; x a variable; e an expression on variables; I some finite indexing set; g_i a boolean expression; and n a natural number.

Informally, the agents in Table 3 have the following meaning:

Table 3: Syntax of ATOM language

Method		
m	::=	$\langle$Method name$\rangle[in,out]$: A end
Object		
o	::=	cyclic $\langle$Object name$\rangle$ thread on P do A end |
		sporadic$_T$ $\langle$Object name$\rangle$ thread on Ev do A end |
		protected $\langle$Object name$\rangle$ ProvidedMethods $(m_1,\ldots,m_n)$ end |
		passive $\langle$Object name$\rangle$ ProvidedMethods $(m_1,\ldots,m_n)$ end |
		active $\langle$Object name$\rangle$ ProvidedMethods $(o_1,\ldots,o_2)$ end
Agent		
A	::=	$w:f$ | *Skip* | Δt | $x := e$ | $x \Leftarrow s$ | $e \Rightarrow s$ | $A\,;A'$ |
		var $x:T$ in A | shunt $s:T$ in A | $[t]\,A$ | if$_t$ $\Box_{i\in I}\, g_i$ then A_i fi |
		$A \sqcap A'$ | $A \rhd^s_t A'$ | $A \parallel A'$ | loop for n period t A

- $w:f$ is a specification statement. It specifies that only the variables in the *frame* w may be changed, and the execution must satisfy ITL formula f.
- 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'.
- var $x:T$ in A defines x to be a new local variable of type T within A; its initial value is chosen nondeterministically.
- shunt $s:T$ in A defines s to be a new local shunt of type $Time \times T$ 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.
- if$_t$ $\Box_{i\in I}\, g_i$ then A_i 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 $\mathsf{if}_t\ g_1$ then $A_1 \Box g_2$ then $A_2 \Box \ldots \Box g_n$ then A_n 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.
- $A \,||\, A'$ executes the two agents concurrently, terminating when both agents terminate.
- 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.

The formal semantics of the concrete part of the **ATOM** language is presented in Sect. A.

5 ATOM 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 X is *refined* by the component Y, denoted $X \sqsubseteq Y$, if and only if $Y \supset X$. A set of sound refinement laws are derived to transform an abstract specification into concrete objects. The following are some useful refinement laws. The soundness of these laws follows from the definition of the refinement relation.

The following law states that the operators in ITL are monotonic w.r.t. the refinement relation. Monotonicity means that the **ATOM** refinement calculus is compositional.

Law 1 (Monotonicity) *Let f_i be an ITL formula then*

$(\sqsubseteq -1)$ *If* $f_0 \sqsubseteq f_1$ *and* $f_1 \sqsubseteq f_2$ *then* $f_0 \sqsubseteq f_2$

$(\sqsubseteq -2)$ *If* $f_0 \sqsubseteq f_1$ *and* $f_2 \sqsubseteq f_3$ *then* $(f_0 \wedge f_2) \sqsubseteq (f_1 \wedge f_3)$

$(\sqsubseteq -3)$ *If* $f_0 \sqsubseteq f_1$ *and* $f_2 \sqsubseteq f_3$ *then* $(f_0 \vee f_2) \sqsubseteq (f_1 \vee f_3)$

$(\sqsubseteq -4)$ *If* $f_1 \sqsubseteq f_2$ *then* $f_0 ; f_1 \sqsubseteq f_0 ; f_2$

$(\sqsubseteq -5)$ *If* $f_1 \sqsubseteq f_2$ *then* $f_1 ; f_0 \sqsubseteq f_2 ; f_0$

$(\sqsubseteq -6)$ *If* $f_0 \sqsubseteq f_1$ *then* $f_0^* \sqsubseteq f_1^*$

$(\sqsubseteq -7)$ *If* $f_0 \sqsubseteq f_1$ *then* $\forall v \bullet f_0 \sqsubseteq \forall v \bullet f_1$

The following law states that any interval of length 1 can be refined into the *Skip* statement.

Law 2 (Skip)

$$len = 1 \quad \sqsubseteq \quad Skip$$

The following law states that any interval of length t can be refined into the delay statement Δt.

Law 3 (Delay)

$$len = t \quad \sqsubseteq \quad \Delta t$$

The following law states that when variable x gets the value of exp in the next state of an interval this can be refined into the assignment $x := exp$.

Law 4 (Assignment) *If $x \in w$ then*

$$w : \bigcirc x = exp \quad \sqsubseteq \quad x := exp$$

The following law states that shunt reading corresponds to reading of the stamp and the value of the shunt.

Law 5 (Shunt read) *If $x \in w$ then*

$$w : x_1 = \surd s \wedge x_2 = \mathsf{read}\,(s) \quad \sqsubseteq \quad x \Leftarrow s$$

The following law states that shunt writing is like assignment but that also the stamp is increased by 1.

Law 6 (Shunt read) *If $x \in w$ then*

$$w : \bigcirc s = (\surd s + 1, x) \quad \sqsubseteq \quad x \Rightarrow s$$

The following law states that sequential composition is associative and distributes over the $\vee$.

Law 7 (Sequential composition)

$$\begin{array}{lcl} f_0 \,;\, (f_1 \,;\, f_2) & \sqsubseteq & (f_0 \,;\, f_1) \,;\, f_2 \\ (f_0 \,;\, f_1) \,;\, f_2 & \sqsubseteq & f_0 \,;\, (f_1 \,;\, f_2) \\ f_0 \,;\, (f_1 \vee f_2) & \sqsubseteq & (f_0 \,;\, f_1) \vee (f_0 \,;\, f_2) \\ (f_0 \,;\, f_1) \vee (f_0 \,;\, f_2) & \sqsubseteq & f_0 \,;\, (f_1 \vee f_2) \end{array}$$

The following law is for the introduction of a variable v.

Law 8 (New variable)

$$w : f \quad \sqsubseteq \quad \mathsf{var}\ v\ \mathsf{in}\ w \cup \{v\} : f$$

The following law is for the introduction of a new shunt.

Law 9 (New shunt)

$$w : f \quad \sqsubseteq \quad \text{shunt } s \text{ in } w \cup \{s\} : \surd(s) = 0 \wedge f$$

The following law is for the introduction of the deadline.

Law 10 (Deadline)

$$\Delta t \wedge f \,;\, true \wedge (f \supset len <= t) \quad \sqsubseteq \quad [t]\, f$$

The alternation is introduced with the following law.

Law 11 (Alternation)

$$(f_0 \wedge f_1) \vee (f_2 \wedge f_3) \quad \sqsubseteq \quad \text{if } f_0 \textit{ then } f_1 \Box f_2 \textit{ then } f_3 \text{ fi}$$

The following law states that the nondeterministic choice corresponds to the $\vee$ of ITL.

Law 12 (Nondeterministic choice)

$$f_0 \vee f_1 \quad \sqsubseteq \quad f_0 \sqcap f_1$$

The timeout is introduced with the following law.

Law 13 (Timeout)

$$(\Delta t \wedge \text{stable}\,(s)) \,;\, f_0 \vee (\Delta t \wedge \neg\, \text{stable}\,(s)) \,;\, f_1 \quad \sqsubseteq \quad f_0 \rhd_t^s f_1$$

The following law introduces the parallel composition.

Law 14 (Parallel composition) *If* $w_0 \cap w_1 = \emptyset$, *then*

$$w_0 \cup w_1 : f_0 \wedge f_1 \quad \sqsubseteq \quad (w_0 : f_0) \parallel (w_1 : f_1).$$

This law will introduce the loop.

Law 15 (Loop)

$$([t]\, f)^n \quad \sqsubseteq \quad \text{loop for } n \text{ period } t \;\; f$$

The following 5 laws is for the introduction of the **ATOM** objects.

Law 16 (Cyclic object)

$$\textit{finite} \wedge (\textit{len} = P \wedge (f \,;\, \textit{true}))^* \quad \sqsubseteq \quad \text{cyclic } \langle \textit{Object name} \rangle \text{ thread on } P \text{ do } f \text{ end}$$

Law 17 (Sporadic object)

$$\begin{aligned} &\mathit{finite} \wedge (\mathsf{stable}\,(\mathit{Ev})\,;\ (\mathit{Skip} \wedge \surd \mathit{Ev} \neq \bigcirc \surd \mathit{Ev})\,;\ (f\,;\ \mathit{true} \wedge \mathit{len} = T))^* \quad \sqsubseteq \\ &\mathsf{sporadic}_T\ \langle \mathit{Object\ name} \rangle\ \mathsf{thread\ on}\ \mathit{Ev}\ \mathsf{do}\ f\ \mathsf{end} \end{aligned}$$

Law 18 (Protected object) *Let*

$$\mathit{Mut} \quad \widehat{=} \quad \Box(\Sigma_i(\mathsf{Status}_i = \mathit{Act}) \leqslant 1).$$

If

$$\begin{aligned} m_i \quad \widehat{=} \quad & (\mathsf{Status}_i = \mathit{Req} \wedge \mathsf{stable}\,(\mathsf{Status}_i))\,;\ \mathit{Skip};\\ & (\mathsf{Status}_i = \mathit{Act} \wedge \mathsf{stable}\,(\mathsf{Status}_i) \wedge f_{m_i})\,;\ \mathit{Skip};\\ & (\mathsf{Status}_i = \mathit{Idle} \wedge \mathsf{stable}\,(\mathsf{Status}_i)) \end{aligned}$$

then

$$\mathit{finite} \wedge \textstyle\bigwedge_i m_i \wedge \mathit{Mut} \quad \sqsubseteq \quad \mathsf{protected}\ \langle \mathit{Object\ name} \rangle\ \mathsf{ProvidedMethods}\ (m_1, \ldots, m_n)\ \mathsf{end}$$

Law 19 (Passive object) *If*

$$\begin{aligned} m_i \quad \widehat{=} \quad & (\mathsf{Status}_i = \mathit{Req} \wedge \mathsf{stable}\,(\mathsf{Status}_i))\,;\ \mathit{Skip};\\ & (\mathsf{Status}_i = \mathit{Act} \wedge \mathsf{stable}\,(\mathsf{Status}_i) \wedge f_{m_i})\,;\ \mathit{Skip};\\ & (\mathsf{Status}_i = \mathit{Idle} \wedge \mathsf{stable}\,(\mathsf{Status}_i)) \end{aligned}$$

then

$$\mathit{finite} \wedge \textstyle\bigwedge_i m_i \quad \sqsubseteq \quad \mathsf{passive}\ \langle \mathit{Object\ name} \rangle\ \mathsf{ProvidedMethods}\ (m_1, \ldots, m_n)\ \mathsf{end}$$

Law 20 (Active object) *Let* $\mathsf{ProvidedMethods}(o) \neq \emptyset$.

$$\textstyle\bigwedge_i (o_i) \quad \sqsubseteq \quad \mathsf{active}\ \langle \mathit{Object\ name} \rangle\ \mathsf{ProvidedMethods}\ (o_1, \ldots, o_n)\ \mathsf{end}$$

6 ATOM DEVELOPMENT TECHNIQUE

In order to derive a concrete design from an abstract specification, a refinement calculus was developed. In the first stage, the designer builds a system model and states the system's requirements (or 'expectation') along with assumptions/constraints of the environment. Using HRT-HOOD such system's requirement may be decomposed into sub-requirement. Each sub-requirement is formalized, using the specification statement which is subsequently refined into objects using the refinement laws.

A development method is therefore suggested:

Given an informal requirement *REQ* of a system.

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

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

$$SPEC \mathrel{\widehat{=}} \bigwedge_{i \in [1,n]} spec_i$$

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 [Lowe and Zedan 1995].

(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.

7 A SMALL CASE STUDY

The case study used here is a simplified version of "The Mine Control System" [Burns and Wellings 1995], by keeping activities on motor and gas, and adding a sporadic activity initiated by the operator.

The requirement of the system is given as follows.

1. *Every 20 time units, the gas level is checked.* **If** *the gas level is higher than 40 and the motor is on,* **then** *the motor is switched off within 5 time units.*

2. *An operator can issue one of two commands: 'Start' or 'Stop'. The System reacts upon receiving the operator's command whenever it is received at least 10 time units has elapsed since the last command. The reaction is as follows:*

- **if** *the command is 'Start', the motor is switched 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' and the motor is switched on,* **then** *the motor is switched off within 5 time units.*

We can decompose[1)] this requirement into the following three sub-requirements (or components).

1. *React***:** *The reaction of the system depends on the command received from the operator.*
   - *The reaction is performed at least* 10 *time units since the last command was received.*
   - **if** *the command is*
     (a) *'Start', the motor is switched off, and the gas level is not higher than* 40, **then** *the motor is switched on within* 5 *time units.*
     (b) *'Stop' and the motor is switched on,* **then** *the motor is switched off within* 5 *time units.*
2. *Gas_Check***:**
   - *Check the gas level every* 20 *time units.*
   - **If** *the level is higher than* 40 *and the motor is in operation,* **then** *switch the motor off within* 5 *time units.*
3. *Switch***:** *Switch the motor on or off if requested. Only one operation can be done at the same time.*

We now give the formal specification of *React*.

$$
\begin{array}{l}
f_{React} \mathrel{\widehat{=}} \\
\quad (stable(Cmd); \\
\quad\ (Skip \wedge \surd Cmd \neq \bigcirc \surd Cmd)\,;\ (len = 10 \wedge f_{cmd}\,;\ true) \\
\quad)^*
\end{array}
$$

where

$$
\begin{array}{l}
f_{cmd} \mathrel{\widehat{=}} \\
\quad (\mathsf{read}\,(Cmd) = start \wedge \mathsf{read}\,(Motor) = off \wedge Gas\_level \leq 40 \wedge \\
\quad\ len = 5 \wedge \mathsf{stable}\,(Motor)\,;\ f_{on}\,;\ \mathsf{stable}\,(Motor) \\
\quad) \vee \\
\quad (\mathsf{read}\,(Cmd) = stop \wedge \mathsf{read}\,(Motor) = on \wedge \\
\quad\ len = 5 \wedge \mathsf{stable}\,(Motor)\,;\ f_{off}\,;\ \mathsf{stable}\,(Motor) \\
\quad)
\end{array}
$$

[1)] This decomposition may be done using various techniques provided by the various structured methodologies

and

$$f_{on} \;\widehat{=}\; Skip \wedge \bigcirc Motor = (\surd Motor + 1, on)$$
$$f_{off} \;\widehat{=}\; Skip \wedge \bigcirc Motor = (\surd Motor + 1, off)$$

f_{React} can be refined into the following object using law 17.

$f_{React} \sqsubseteq$
sporadic$_{10}$ ⟨React⟩ thread on Cmd do f_{cmd} end

and f_{cmd} can be refined, using law 11 and 10, into

$f_{cmd} \sqsubseteq$
if $(\mathsf{read}\,(Cmd) = start \wedge \mathsf{read}\,(Motor) = off \wedge Gas\_level \leq 40)$ then $[5]$ (f_{on})
□ $(\mathsf{read}\,(cmd) = stop \wedge \mathsf{read}\,(Motor) = on)$ then $[5]$ (f_{off})
fi

Since $\mathsf{read}\,(Cmd)$ and $\mathsf{read}\,(Motor)$ are not concrete constructs these should be further refined. This is done with the introduction of variables X, Y with law 8 and 14 that will get, respectively the values of shunts *Cmd* and *Motor*, i.e,

$\sqsubseteq$
var X, Y in
$(X \Leftarrow Cmd \;||\; Y \Leftarrow Motor)$ ||
if $(X = start \wedge Y = off \wedge Gas\_level \leq 40)$ then $[5]$ (f_{on})
□ $(X = stop \wedge Y = on)$ then $[5]$ (f_{off})
fi

The final step consists of refining f_{on} and f_{off}, using law 6, into respectively

$$f_{on} \;\sqsubseteq\; (on \Rightarrow Motor)$$
$$f_{off} \;\sqsubseteq\; (off \Rightarrow Motor)$$

Using law 1 we get the final concrete code.

var X, Y in
$(X \Leftarrow Cmd \;||\; Y \Leftarrow Motor)$ ||
if $(X = start \wedge Y = off \wedge Gas\_level \leq 40)$ then $[5]$ $(on \Rightarrow Motor)$
□ $(X = stop \wedge Y = on)$ then $[5]$ $(off \Rightarrow Motor)$
fi

8 DISCUSSION

In this paper we have introduced an object-based development technique called **ATOM**. IT is a wide-spectrum formal design language that extends the Temporal Agent Model (TAM) with the capability of describing behaviors of objects and method invocations. It also supports mixing of abstract statements, known as 'specification' statements, and 'concrete' executable statements.

The novelty of our treatment lies in the underlying computational model. The model was particularly constructed so that the resulting concrete system can be easily analyzed for their schedulability in a distributed hard real-time execution environment. The computational model prescribes the use of object structure which facilitates the development of large scale systems. The object structure was based on an industry-strength object methodology known as HRT-HOOD. Within an object, agents are statically allocated which may communicate asynchronously using (single writer - multiple reader) shunts.

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 constrained by the obligations themselves.

In addition, a graphical notation was provided for the presented object-based structure. For example, an sporadic object *o* with can be represented as Fig. 1.

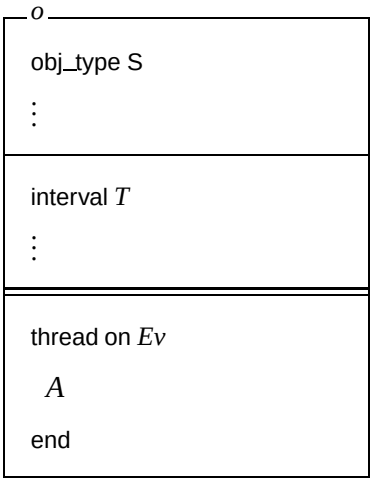

Figure 1: Sporadic Object

REFERENCES

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

Auernheimer, B. and R. Kemmerer (1986), "RT-ASLAN: a Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering 12*, 9, 879–889.

Bastide, R. (1992), "Objets Coopératifs: un formalisme pour la modélisation des systèmes concurrents," Ph.D. thesis, Université Paul Sabatier de Toulouse.

Bastide, R. and P. Palanque (1993), "Cooperative Objects : a Concurrent Petri Net Based Object-Oriented Language," In *IEEE / System Man and Cybernetics 93*, Elseiver Science Publisher, Le Touquet (France).

Battiston, E., A. Chizzoni, and F. D. Cindio (1995), "Inheritance and concurrency in CLOWN," In *Proceedings of the Application and Theory of Petri Nets 1995 workshop on Object-Oriented Programming and Models of Concurrency*, Italy.

Battiston, E., A. Chizzoni, and F. D. Cindio (1996), "Modeling a cooperative environment with clown," In *Proceedings of the second international workshop on Object-Oriented Programming and Models of Concurrency within the 16th International Conference on Application and Theory of Petri Nets*, G. Agha, F. D. Cindio, and A. Yonezawa, Eds., Osaka, Japan, pp. 12–24.

Battiston, E. and F. D. Cindio (1993), "Class orientation and inheritance in modular algebraic nets," In *Proceedings International Conference on Systems, Man and Cybernetics*, volume 2, Palais de L'Europe Hôtel Westminster, Le Touquet, France, pp. 717–723.

Bergstra, J. A. and J. W. Klop (1984), "Process Algebra for Synchronous Communication," *Information and Control 60*, 109–137.

Biberstein, O., D. Buchs, and N. Guelfi (1996), "COOPN/2 : A Specification Language for Distributed Systems Engineering," Technical Report 96/167, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland.

Biberstein, O., D. Buchs, and N. Guelfi (1997), "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 formalism," In *Advances in Petri Nets on Object-Orientation*, G. Agha and F. D. Cindio, Eds., LNCS, Springer-Verlag, To appear.

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

Canver, E. and F. von Henke (1997), "Formal specification and verification of objectbased systems in a temporal logic setting," Technical report, University of Newcastle Upon Tyne, England, Department of Computing Science, Technical Report Second Year Report of the Esprit Long Term Research Project 20072 Design For Validation.

Cau, A. and H. Zedan (1997), "Refining Interval Temporal Logic Specifications," In *Transformation-Based Reactive*

Systems Development, M. Bertran and T. Rus, Eds., number 1231 In LNCS, AMAST, Springer-Verlag, pp. 79–94.

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

Chen, Z. (1997), "Formal Methods for Object-Oriented Paradigm Applied to the Engineering of Real-Time Systems: A Review," Technical report, De Montfort University.

Chen, Z., A. Cau, H. Zedan, and H. Yang (1999), "Integrating Structured OO Approaches with Formal Techniques for the Development of Real-time Systems," To appear in International Journal of Information and Software Technology.

Davies, J. (1991), "Specification and Proof in Real-Time Systems," Ph.D. thesis, Oxford University, Cambridge University Press.

F. Jahanian and A. Mok (1986), "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering 12*, 9, 890–904.

Fraser, M. D., K. Kumar, and V. K. Vaishnavi (1991), "Informal and Foraml Requirements Specification Languages: Bridging the Gap," *IEEE Transactions on Software Engineering 17*, 5, 454–466.

Goguen, J. and R. Diaconescu (1994), "Towards an algebraic semantics for the object paradigm," In *In RECENT trends in data type specification: workshop on specification of abstract data types: COMPASS: selected papers*, number 785 In LNCS, Springer Verlag.

Harel, E., O. Lichtenstein, and A. Pnueli (1990), "Explicit Clock Temporal Logic," In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Philadelphia, Pennsylvania, pp. 402–413.

Hayes, I. J. and M. Utting (1998), "Deadlines are termination," Technical Report Technical Report 98-01, Software Verification Reseach Centre (SVRC).

He, H. and H. Zedan (1996), "A Fast Prototype Tool for Parallel Reactive Systems," *Euromicro Journal of Systems Architecture* .

He, J. (1991), "Specification oriented semantics for ProCoS programming language PL^{time}," Technical Report PRG-OU-HJF-71, Oxford University.

Jackson, M. A. (1983), *System Development*, Prentice Hall, New Jersey.

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

Lano, K. (1990), "Z++," In *Proceedings of Z User Workshop Oxford*, J. E. Nicholls, Ed., Springer-Verlag.

Lano, K. (1995), "Distributed System Specification in VDM++," In *Proceedings of FORTE'95*, Chapman and Hall.

Liu, S., A. J. Offutt, Y. Sun, and M. Ohba (1998), "SOFL: A Formal Engineering Methodology for Industrial Applications," *IEEE Transactions on Software Engineering 24*, 1.

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

Malcom, G. and J. Goguen (1994), "Proving correctness of refinement and implementation," Technical Report Prg-

114, Oxford University, Oxford Technical Monograph.

Mander, K. C. and F. Polack (1995), "Rigorous Specification using Structured Systems Analysis and Z," *Information and Software Technology 37*, 5–6, 285–291.

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

Merlin, P. M. and A. Segall (1976), "Recoverability of communication protocols - implications of a theoretical study," *IEEE Transactions on Communications* , 1036–1043.

Meseguer, J. (1993), *Research Directions in Concurrent Object-Oriented Programming*, chapter A logical theory of concurrent objects and its realization in the maude language, The MIT Press, Cambridge, Mass., pp. 314–390.

Meseguer, J. and T. Winkler (1992), *Parallel Programming in Maude*, volume 574 of *LNCS*, Springer-Verlag, New York, N.Y., pp. 253–293.

Morzenti, A. and P. S. Pietro (1994), "Object-Oriented Logical Specification of Time-Critical Systems," *ACM Transactions on Software Engineering and Methodology 3*, 1, 56–98.

Moszkowski, B. (1985), "A temporal logic for multilevel reasoning about hardware," *Computer 18*, 2, 10–19.

Ostroff, J. S. and W. M. Wonham (1985), "A Temporal Logic Approach to Real Time Control," In *Proc. of 24th Conf. Decision and Control*, Fort Lauderdale, FL, USA, pp. 6565–6567.

Petri, C. A. (1962), "Communication with Automata," Ph.D. thesis, Univ. Bonn.

Plat, N., J. Katwijk, and K. Pronk (1991), "A Case for Structured Analysis/Formal Design," In *Proceedings of VDM'91*, number 551 In LNCS, Springer-Verlag.

Ramchandani, C. (1974), "Analysis of asynchronous concurrent systems by timed Petri nets," Technical Report MAC TR 120, MIT.

Robinson, P. J. (1992), *HOOD: Hierarchical Object-Oriented Design*, Prentice Hall.

Schneider, S., J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe (1992), "Timed CSP: Theory and Practice," In *Proceedings of Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds., volume 600 of *LNCS*, Springer, Berlin, Germany, pp. 640–675.

Scholefield, D., H. Zedan, and J. He (1993), "Real-Time Refinement: Semantics and Application," In *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, A. M. Borzyszkowski and S. Sokolowski, Eds., volume 711 of *lncs*, Springer, Gdansk, Poland, pp. 693–702.

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

Scholefield, D. J., H. Zedan, and J. He (1994b), "A Predicative Semantics for the Refinement of Real-Time Systems," In *LNCS*, number 802, Springer-Verlag, pp. 230–249.

Semmens, L. T. and P. M. Allen (1991), "Using Yourdon and Z: An Approach to Formal Specification," In *Z User Workshop, Oxford 1990*, J. E. Nicholls, Ed., Workshops in Computing, Springer-Verlag, pp. 228–253.

Spivey, J. M. (1996), "Richer Types for Z," *Formal Aspects of Computing 8*, 565–584.

Yi, W. (1991), "CCS + Time = An Interleaving Model for Real Time Systems," In *Automata, Languages and Programming, 18th International Colloquium*, J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, Eds., volume 510 of *LNCS*, Springer-Verlag, Madrid, Spain, pp. 217–228.

Yourdon, E. (1989), *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey.

Zhou, C., C. Hoare, and A. Ravn (1991), "A Calculus of Durations," *Information Processing Letters 40*, 5, 269–276.

A ATOM FORMAL SEMANTICS

The semantics of the concrete statements in the language of **ATOM** is given denotationally in terms of a formula in ITL. We begin by first introducing some extensions to ITL in order to describe the formal semantics of ATOM. Let W be a set of state variables then $frame(W)$ denotes that only the variables in W can possible change, i.e., the variables outside the frame don't change. Here, we adopt a combined state-communication model for the system behavior where the observables correspond to the following variables:

- The normal state variables of ITL.
- variables s representing shunts whose values are tuples (t,v) where t is a stamp and v the value written. The stamp value of s will be denoted by $\surd s$ and the value stored in s will be denoted by $\mathsf{read}(s)$.

The ITL semantics of ATOM is given as follows

1. **Agent Structure**: The semantics is given in Table 4.

Table 4: Semantics of ATOM agents

$w : f$	$\widehat{=}$	$\mathsf{frame}(w) \wedge f$
$Skip$	$\widehat{=}$	$Skip$
Δt	$\widehat{=}$	$len = t$
$x := e$	$\widehat{=}$	$\bigcirc x = e$
$x \Leftarrow s$	$\widehat{=}$	$x_1 = \surd s \wedge x_2 = \mathsf{read}(s)$
$x \Rightarrow s$	$\widehat{=}$	$\bigcirc s = (\surd s + 1, x)$
$A\,;\,A'$	$\widehat{=}$	$A\,;\,A'$
$\mathsf{var}\, x \,\mathsf{in}\, A$	$\widehat{=}$	$\exists x \bullet A$
$\mathsf{shunt}\ s \,\mathsf{in}\, A$	$\widehat{=}$	$\exists s \bullet \surd s = 0 \wedge A$
$[t]\,A$	$\widehat{=}$	$\Delta t \wedge A\,;\, true \wedge (A \supset len <= t)$
$\mathsf{if}_t\ \Box_{i\in I} g_i \,\mathsf{then}\, A_i\ \mathsf{fi}$	$\widehat{=}$	$\bigvee_{i\in I}([t]\,(g_i \wedge A_i)) \vee [t]\,(\bigwedge_{i\in I} \neg g_i)$
$A \sqcap A'$	$\widehat{=}$	$A \vee A'$
$A \rhd^s_t A'$	$\widehat{=}$	$(\Delta t \wedge \mathsf{stable}(s))\,;\, A \vee (\Delta t \wedge \neg\,\mathsf{stable}(s))\,;\, A'$
$A \parallel A'$	$\widehat{=}$	$A \wedge A'$
$\mathsf{loop\,for}\ n\ \mathsf{period}\ t\ \ A$	$\widehat{=}$	$([t]\,A)^n$

2. **Object Structure**

- A cyclic object

  $$\text{cyclic } \langle \text{Object name} \rangle \text{ thread on } P \text{ do } A \text{ end} \mathrel{\widehat{=}}$$
  $$\mathit{finite} \wedge (\mathit{len} = P \wedge (A\,;\, \mathit{true}))^*$$

- A sporadic object:

  an object in which the agent A is executed whenever the shunt Ev is written to. The interval between two successive executions can not be less than T:

  $$\text{sporadic}_T \ \langle \text{Object name} \rangle \text{ thread on } Ev \text{ do } A \text{ end} \mathrel{\widehat{=}}$$
  $$\mathit{finite} \wedge (\text{stable}(Ev)\,;\ (\mathit{Skip} \wedge \surd Ev \neq \bigcirc \surd Ev)\,;\ (A\,;\ \mathit{true} \wedge \mathit{len} = T))^*$$

- For protected and passive objects, we need to identify all possible states for method invocation. Let

  $$\text{Status}_i \in \{Idle, Req, Act\}$$

  denote the status of a method, idle (or terminated), requested or active respectively.

  (a) A protected object:

  is an object in which the method body A_i is executed when method m_i has been requested, but the execution must be 'mutually exclusive' within the object.

  $$\text{protected } \langle \text{Object name} \rangle \text{ ProvidedMethods } (m_1, \ldots, m_n) \text{ end} \mathrel{\widehat{=}}$$
  $$\mathit{finite} \wedge \bigwedge_i m_i \wedge \mathit{Mut}$$

  where

  $$\begin{aligned} m_i \quad \mathrel{\widehat{=}} \quad & (\text{Status}_i = \mathit{Req} \wedge \text{stable}(\text{Status}_i))\,;\ \mathit{Skip}; \\ & (\text{Status}_i = \mathit{Act} \wedge \text{stable}(\text{Status}_i) \wedge A_{m_i})\,;\ \mathit{Skip}; \\ & (\text{Status}_i = \mathit{Idle} \wedge \text{stable}(\text{Status}_i)) \end{aligned}$$

  and

  $$\mathit{Mut} \quad \mathrel{\widehat{=}} \quad \Box(\Sigma_i(\text{Status}_i = \mathit{Act}) \leqslant 1)$$

  (b) A passive object:

  Is similar to the protected object except that it responds to all method invocations at anytime:

  $$\text{passive } \langle \text{Object name} \rangle \text{ ProvidedMethods } (m_1, \ldots, m_n) \text{ end} \mathrel{\widehat{=}}$$
  $$\mathit{finite} \wedge \bigwedge_i (m_i)$$

- An active object:

  If ProvidedMethods$(o) \neq \emptyset$, then every method $m \in$ ProvidedMethods(o) is implemented by one of its child object.

  active ⟨Object name⟩ ProvidedMethods $(o_1, \ldots, o_n)$ end $\widehat{=}$

  $\bigwedge_i (o_i)$