A Wide-Spectrum Language for Object-Based Development of Real-time Systems

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

A formal design notation is presented whose underlying computational model is object-based. The object structure of the model is based on the practical, industry-strength Object Oriented structured development technique HRT-HOOD. The computational model has been specifically chosen because it leads to designs which can be analyzed for their schedulability in a distributed hard real-time execution environment. It is a wide-spectrum language supporting abstract description statements in Interval Temporal Logic (ITL) for the description of the timing, functional, and communication behavior of the proposed real-time system, and concrete Temporal Agent Model (TAM) statements with objects which can be directly executed. The semantics of these concrete statements is defined denotationally in *specification-oriented* style using ITL. A system specified at a high level of abstraction can be systematically transformed into an executable program by the use of sound ITL refinement rules.

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

1 Introduction

Real-time systems are hard to model as their correctness depends on satisfying not only *functional* requirements, as in most information processing systems, but also

[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

on *non-functional* requirements, such as timing, limited resources and dependability.

Traditional real-time system development has been a somewhat ad-hoc affair. A system is designed from an informal requirement 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. For this purpose, a large number of formalisms have been developed, such as RTTL [35], MTL [22], XCTL [19], ITL [34], TAM [41,40,39,26], TCSP [38], TCCS [44], TACP [7], time Petri Nets [30,37].

However, we have shown [14] 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 in 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} [20] 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 [1] is a refinement calculus which refines a specification into concrete code, but this again relies on the maximal parallelism model. The Duration Calculus [46] (and to some extend timed Z in a recent attempt), on the other hand, is a formalism based on ITL [34] and provides rules which are only applicable at the logical level of development.

Furthermore, with the advent of the Object-Oriented (OO) method, as a powerful approach in modeling and developing large-scale and complex software systems,

the quest for sound framework within which such a paradigm can be used has increased. This has mainly been based on extending existing process-oriented formalisms:

- *model*-based: Z++ [23] and VDM++ [24];
- *algebraic*-based: HOSA [18,27] and Maude [32,31];
- *Petri net*: CLOWN [6,4,5], CO [2,3] and COOPN/2 [8,9] and
- *logic*-based: TRIO+ [33] and OO-LTL [11]

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) [29], Yourdon [45] and Jackson [21], for non-real-time systems, and ROOM [13] and HRT-HOOD [10] for real-time applications). As a result, in [28,42], 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 [17,36]. Recently, Liu, et al [25], 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 [15].

The main objective of this work is to provide a wide-spectrum language in which concrete and abstract constructs can be freely intermixed. This language forms the basis of a formal development technique for the systematic derivation of a concrete design from an abstract specification using sound refinement rules. The underlying computational model of this development technique must be realistic and should support 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 in the OO paradigm.

In Sect. 2 we present the syntax of Interval Temporal Logic (ITL) which is used for the description of the abstract constructs. In Sect. 3 we present the Temporal Agent Model (TAM) concrete constructs plus their semantics in ITL. In Sect. 4 we introduce the object model, based on that found in the industry strength structured technique known as HRT-HOOD, and give the syntax (an extension to TAM) and semantics (in ITL) of objects. In Sect. 5 we give a small case study to illustrate the use of this language. In Sect. 6 we end with a discussion on our development technique.

Table 1
Syntax of ITL

Expressions
$e ::= \quad \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \imath a{:}f$

Formulas
$f ::= \quad p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathit{Skip} \mid f_1 ; f_2 \mid f^*$

2 Interval temporal Logic

We base our work on Interval Temporal Logic (ITL) and its programming language subset Tempura[34]. ITL will be used both as our abstract specification language and to define the specification-oriented semantics of the concrete statements.

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 [12]. 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:

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

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
$\diamondsuit f$	$\widehat{=}$	*finite* ; f ; *true*	some subinterval
$\boxdot f$	$\widehat{=}$	$\neg(\diamondsuit \neg f)$	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	$\widehat{=}$	$\boxdot$(*Skip* $\supset$ f)	all unit subintervals
$\bigcirc e$	$\widehat{=}$	$\wr a$: $\bigcirc(e = a)$	next value
fin e	$\widehat{=}$	$\wr a$: *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 assignm.
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	$\widehat{=}$	$\wr a$: *intlen* (a)	interval length

2.2 Data Representation in ITL

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 $\mathcal{N}$ (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, $\mathcal{P}$ (in a similar fashion as adopted in the specification language Z).

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

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

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 [43].

3 Temporal Agent Model

The Temporal Agent Model (TAM) [41,40,39,26] 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.

3.1 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 model-led 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.

3.2 TAM Syntax

Agents in TAM are described as follows.

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

where w is a set of computation variables and shunts; Φ is a predicate in ITL, 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 ITL.
- The agent *Skip* is a delay of 1.
- 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.
- $\mathcal{A}; \mathcal{A}'$ performs a sequential composition of $\mathcal{A}$ and $\mathcal{A}'$.
- var $x : T$ in $\mathcal{A}$ defines x to be a new local variable of type T within $\mathcal{A}$; its initial value is chosen nondeterministically.
- shunt $s : T$ in $\mathcal{A}$ defines s to be a new local shunt of type $Time \times T$ within $\mathcal{A}$; its initial value is chosen nondeterministically, but it is time-stamped with the time of its declaration.
- $[t]\ \mathcal{A}$ gives agent $\mathcal{A}$ 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 $\mathcal{A}_i$ fi evaluates all the boolean guards g_i, and executes an $\mathcal{A}_i$ 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 if$_t$ g_1 then $\mathcal{A}_1 \Box g_2$ then $\mathcal{A}_2 \Box \ldots \Box g_n$ then $\mathcal{A}_n$ fi.
- $\mathcal{A} \sqcap \mathcal{A}'$ forms a nondeterministic choice between $\mathcal{A}$ and $\mathcal{A}'$.
- $\mathcal{A} \rhd_t^s \mathcal{A}'$ monitors shunt s for t time units: if a write occurs within this time, then it executes $\mathcal{A}'$; otherwise it times-out and executes $\mathcal{A}$.
- $\mathcal{A} \parallel \mathcal{A}'$ executes the two agents concurrently, terminating when both agents terminate.
- loop for n period t $\mathcal{A}$ executes $\mathcal{A}$ 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.

3.3 TAM Semantics

The semantics of TAM is given denotationally in terms of an ITL formula. We begin by first introducing some extensions to ITL in order to describe the formal semantics of TAM.

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 domain of the variable *time* is a linear order $(TIME, <, +, 0)$, where 0 is the least element, and + is an addition operator.

The ITL semantics of TAM is given as follows

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

3.4 Procedures

We can trivially introduce various structures within TAM such as *function* and *procedure*. We show this by introducing procedures.

Let A be a TAM agent and $x_1, x_2, ..., x_n$ be a set of state variables which are free in A. A procedure P is denoted by

$$P(x_1, x_2, ..., x_n) \ \widehat{=}\ \mathcal{A}$$

The semantics of which is given by

$$P(y_1, y_2, ..., y_n) \ \widehat{=}\ \mathcal{A}(y_1/x_1, y_2/x_2, ..., y_n/x_n)$$

4 Object model

In this section we introduce the object structure which we use in our framework. Such structure is being used in an industry-strength structured methodology known as HRT-HOOD [10].

4.1 Computation

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

(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.

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.

4.2 Syntactic Structure

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 environment of a non-active object is a set of data over which the methods of the object execute for computations and communications. The data include constants, variables and shunts. For cyclic and sporadic objects, an activation period and a minimum activation interval are specified in the environment declaration respectively. We use $\mathsf{ObjEnv}(o)$ to denote the environment set of an object *o*.

A **method** consists of a head and a body. The head specifies a method name and a local environment (if necessary) of the method. The body specifies operations over either the object environment or the method environment, or both. We use $\mathsf{Methods}(o)$ to denote the set of method defined by the object *o*. [2] The operations are described by means of agents which may be either abstract or concrete. A method can define its local execution environment. We use $\mathsf{MthEnv}(m)$ to denote the local environment of the method *m*. A method *m* is defined in the form of

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

where $\mathcal{A}$ (a TAM agent) is the body of method *m*, *in* and *out* are its input and output parameter list. We use $\mathcal{A}_m$ to denote the body of a method *m*.

The syntax of an object is as follows.

Method

m ::= $\langle$Method name$\rangle[in, out]$: $\mathcal{A}$ end

Object

o ::= cyclic $\langle$Object name$\rangle$ thread on P do $\mathcal{A}$ end |
sporadic$_T$ $\langle$Object name$\rangle$ thread on *Ev* do $\mathcal{A}$ 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

where $\mathcal{A}$ is a TAM agent, ProvidedMethods is a set of provided methods, *P* and *T* are times, and *Ev* is a shunt.

[2] Obviously, $\mathsf{ProvidedMethods}(o) \subseteq \mathsf{Methods}(o)$

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.

4.3 Invocation and Encapsulation

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

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

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

4.4 Specification-oriented semantics of objects

In this section we give an ITL semantics for the objects presented above.

- A cyclic object:

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

- A sporadic object:
  an object in which the agent $\mathcal{A}$ is executed whenever the shunt *Ev* is written to. The interval between two successive executions can not be less than T:

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

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

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

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

(1) A protected object:
is an object in which the method body $\mathcal{A}_i$ is executed when method m_i has been requested, but the execution must be 'mutually exclusive' within the object.

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

where

$$m_i \mathrel{\widehat{=}} (\textsf{Status}_i = Req \wedge \textsf{stable}(\textsf{Status}_i)) \,;\, \textit{Skip};$$
$$(\textsf{Status}_i = Act \wedge \textsf{stable}(\textsf{Status}_i) \wedge \mathcal{A}_{m_i}) \,;\, \textit{Skip};$$
$$(\textsf{Status}_i = Idle \wedge \textsf{stable}(\textsf{Status}_i))$$

and

$$\textit{Mut} \mathrel{\widehat{=}} \Box(\Sigma_i(\textsf{Status}_i = Act) \leqslant 1)$$

(2) A passive object:
Is similar to the protected object except that it responds to all method invocations at anytime:

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

- An active object:
If $\textsf{ProvidedMethods}(o) \neq \emptyset$, then every method $m \in \textsf{ProvidedMethods}(o)$ is implemented by one of its child object.

$$\textsf{active } \langle \text{Object name} \rangle \textsf{ ProvidedMethods } (o_1, \ldots, o_n) \textsf{ end} \mathrel{\widehat{=}}$$
$$\bigwedge_i (o_i)$$

5 A small case study

The case study used here is a simplified version of "The Mine Control System" [10], by keeping activities on motor and gas, and adding a sporadic activity initiated by the operator, as depicted in Fig. 1.

The requirement of the system is given as follows.

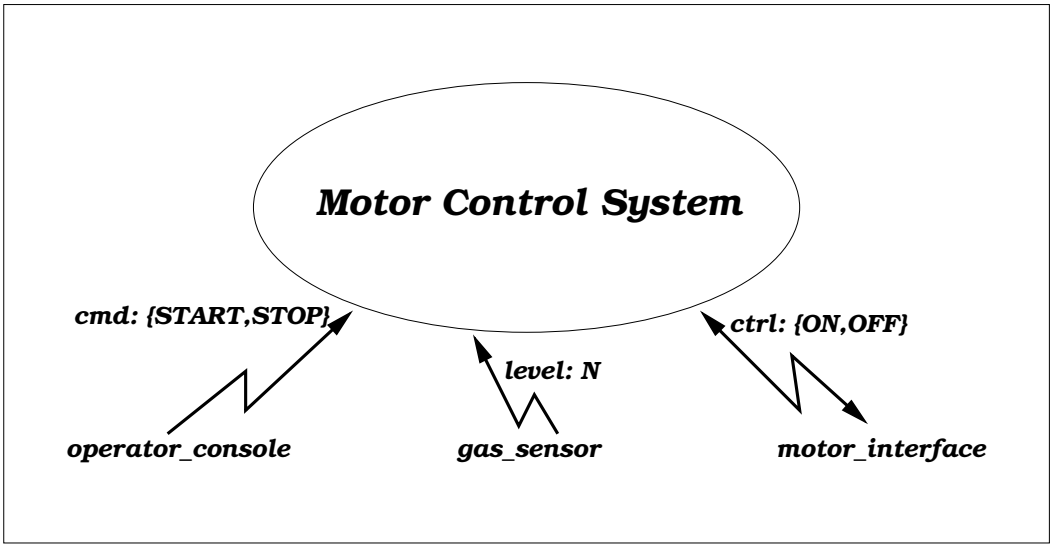

Fig. 1. Motor Control System

(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[3] 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] This decomposition may be done using various techniques provided by the various structured methodologies

(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

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

and

$$
\begin{array}{l}
f_{on} \mathrel{\widehat{=}} Skip \wedge \bigcirc Motor = (\surd Motor + 1, on) \\
f_{off} \mathrel{\widehat{=}} Skip \wedge \bigcirc Motor = (\surd Motor + 1, off)
\end{array}
$$

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

$$
\begin{array}{l}
f_{React} \sqsubseteq \\
\mathsf{sporadic}_{10}\ \langle \mathrm{React} \rangle\ \mathsf{thread\ on}\ Cmd\ \mathsf{do}\ f_{cmd}\ \mathsf{end}
\end{array}
$$

and f_{cmd} can be refined into

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

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 that will get the respectively the values of shunts *Cmd* and *Motor*, i.e,

$$
\begin{array}{l}
\sqsubseteq \\
\mathsf{var}\, X, Y \,\mathsf{in} \\
(X \Leftarrow Cmd \parallel Y \Leftarrow Motor) \parallel \\
\text{if } (X = start \wedge Y = off \wedge Gas\_level \leq 40) \text{ then } [5]\ (f_{on}) \\
\Box\ (X = stop \wedge Y = on) \text{ then } [5]\ (f_{off}) \\
\text{fi}
\end{array}
$$

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

$$
\begin{array}{l}
f_{on} \sqsubseteq (on \Rightarrow Motor) \\
f_{off} \sqsubseteq (off \Rightarrow Motor)
\end{array}
$$

6 Discussion

In this paper we have introduced a wide-spectrum formal design language for the development of real-time systems. The language is an extension to 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 are formulae in Interval Temporal Logic) and 'concrete' statements (which could be executed).

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. Agents are implemented as preemptive priority dispatched tasks; shunts are implemented as protected resources.

In order to derive a concrete design from an abstract specification a refinement calculus has been developed. The refinement relation $\sqsubseteq$ is defined on a component (agent, method and object) in a similar fashion to that of TAM. A component $\mathcal{X}$ is

refined by the component $\mathcal{Y}$, denoted $\mathcal{X} \sqsubseteq \mathcal{Y}$, if and only if

$$\mathcal{F}[\![\mathcal{Y}]\!] \Rightarrow \mathcal{F}[\![\mathcal{X}]\!].$$

In [16] we presented a comprehensive set of refine laws for the development of object-based real-time systems. However, instead of ITL we used TAMLL (Temporal Agent Model Logic Language) as our underlying wide-spectrum language. TAMLL is first order predicate logic with simple extensions to deal with times and the values held in variables and shunts. A disadvantage of TAMLL is that formulae tend to grow rather rapidly in size and has an excessive use of the time variable *t*. ITL formulae are short, simple and there is no need of time variable *t* because of the temporal operators in ITL. The set of refinement rules of [16] can be easily transformed into ITL based one.

Furthermore we presented in [16] a formal development method for object-based real-time systems. This method is as follows: 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. So one proceeds as follows:

(1) Use HRT-HOOD to decompose the system requirement, namely *REQ*, to produce sub-requirements: req_1, req_2, ..., req_n.
(2) Formalize each sub-requirement req_i using the specification statement of TAM to produce $spec_1$, $spec_2$, ..., $spec_n$. Note that the formal specification, *SPEC*, which corresponds to *REQ*, is given by

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

(3) Construct corresponding object o_i based on spe_i, such that (following laws from 1 to 5),

$$spec_i \sqsubseteq obj_i$$

(4) The collection of resulting objects are then composed to produce the final concrete system.
(5) Use HRT-HOOD to map the resulting concrete code to an equivalent Ada code.

A characteristic of our approach is that during the refinement stages, all necessary timing information may be gathered in the form of 'proof-obligations'. These obligations are obviously proved correct (as a result of the soundness of the refinement laws) and are vital to scheduling theorists. Once these obligations are available, various scheduling tests and analysis may be applied. In fact these tests could also

be applied after each refinement step; if the test is not valid then the step is repeated until the obligation is satisfied.

It is clear that some of the timing characteristics may be left as ‘variables’ to be determined at a later stage of development. These variables are constraints by the obligations themselves.

In addition, a graphical notation was provided for the presented object-based structure. For example, 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 Fig. 2.

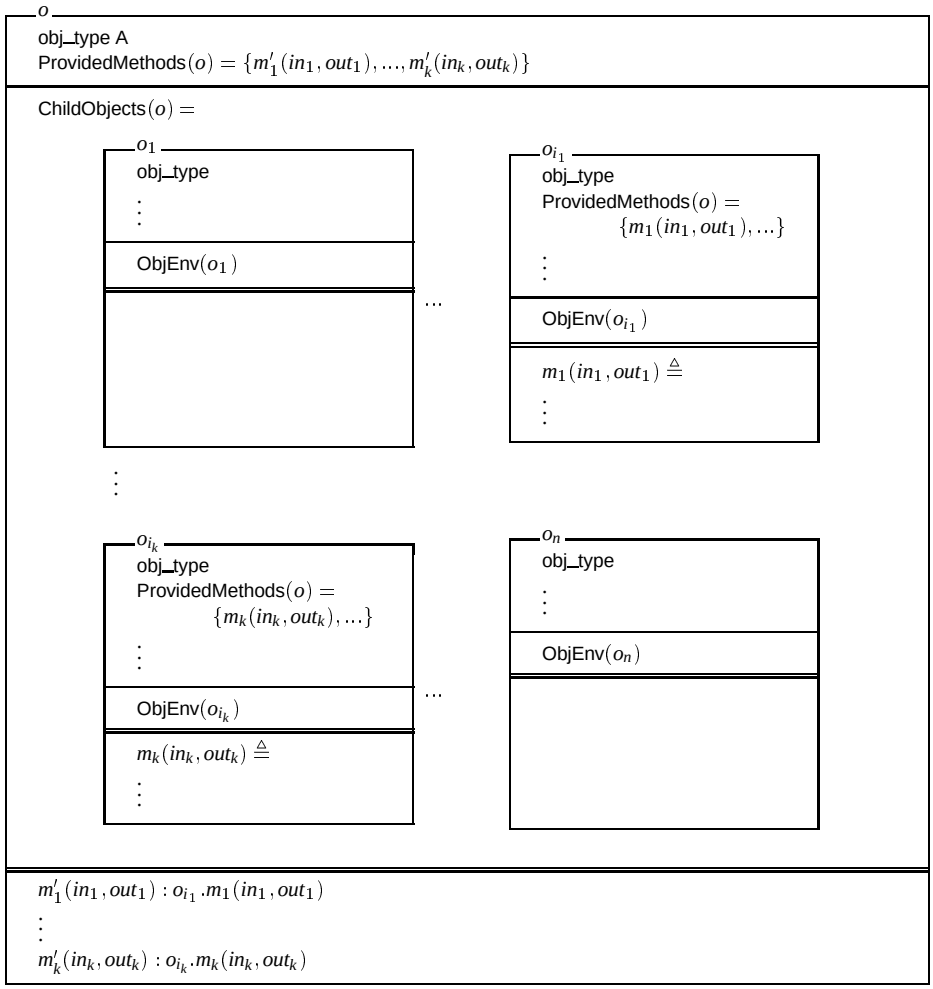

Fig. 2. Active Object

References

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

[2] R. Bastide. *Objets Coopératifs: un formalisme pour la modélisation des systèmes concurrents*. PhD thesis, Université Paul Sabatier de Toulouse, 1992.

[3] R. Bastide and P. Palanque. Cooperative objects : a concurrent petri net based object-oriented language. In *IEEE / System Man and Cybernetics 93*, Le Touquet (France), October 1993. Elseiver Science Publisher.

[4] E. Battiston, A. Chizzoni, and F. De Cindio. 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, 1995.

[5] E. Battiston, A. Chizzoni, and F. De Cindio. Modeling a cooperative environment with clown. In G. Agha, F. De Cindio, and A. Yonezawa, editors, *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*, pages 12–24, Osaka, Japan, 1996.

[6] E. Battiston and F. De Cindio. Class orientation and inheritance in modular algebraic nets. In *Proceedings International Conference on Systems, Man and Cybernetics*, volume 2, pages 717–723, Palais de L'Europe Hôtel Westminster, Le Touquet, France, October 1993.

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

[8] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. COOPN/2 : A specification language for distributed systems engineering. Technical Report 96/167, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1996.

[9] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha and F. De Cindio, editors, *Advances in Petri Nets on Object-Orientation*, LNCS. Springer-Verlag, 1997. To appear.

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

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

[12] 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.

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

[14] Z. Chen. Formal methods for object-oriented paradigm applied to the engineering of real-time systems: A review. Technical report, De Montfort University, 1997.

[15] Z. Chen, A. Cau, H. Zedan, and H. Yang. Integrating structured oo approaches with formal techniques for the development of real-time systems. To appear in International Journal of Information and Software Technology, 1999.

[16] 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.

[17] 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.

[18] J. Goguen and R. Diaconescu. 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, 1994.

[19] 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.

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

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

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

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

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

[25] 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.

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

[27] G. Malcom and J. Goguen. Proving correctness of refinement and implementation. Technical Report Prg-114, Oxford University, Oxford Technical Monograph, 1994.

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

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

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

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

[32] J. Meseguer and T. Winkler. *Parallel Programming in Maude*, volume 574 of *LNCS*, pages 253–293. Springer-Verlag, New York, N.Y., 1992.

[33] A. Morzenti and P. San Pietro. Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology*, 3(1):56–98, January 1994.

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

[35] 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.

[36] 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.

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

[38] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 640–675, Berlin, Germany, June 1992. Springer.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] J. M. Spivey. Richer types for Z. *Formal Aspects of Computing*, 8:565–584, 1996.

[44] Wang Yi. CCS + time = an interleaving model for real time systems. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *LNCS*, pages 217–228, Madrid, Spain, 1991. Springer-Verlag.

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

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