Integrating VDM^{++} and Real-time System Design

K. Lano, **S. Goldsack**, **J. Bicarregui**,
Dept. of Computing, Imperial College, 180 Queens Gate, London, SW7 2BZ.
S. Kent, Dept. of Computing, University of Brighton.

Abstract. This paper presents work performed in the EPSRC "Object-oriented Specification of Reactive and Real-time Systems" project. It aims to provide formal design methods for real-time systems, using a combination of the VDM^{++} formal method and the HRT-HOOD method.

We identify refinement steps for hard real-time systems in VDM^{++}, together with a case study of a mine-pump control system, involving a combination of VDM^{++} and HRT-HOOD.

We also consider the representation of hybrid systems in VDM^{++}.

1 Introduction

Formalisms for real-time object-oriented specification are still at an early stage of development. The TAM formalism [18] describes real-time systems without module structures, and hence, does not make use of principles of locality or encapsulation to support reasoning and transformation. TAM can be seen as a subset of the Real Time Action Logic (RAL) notation used here. Extensions of object-oriented methods to cover real-time aspects, such as HRT-HOOD [4] or Octopus [1] do not provide a formal semantics or refinement concept, although their design and analysis techniques are useful frameworks for development using formal notations such as VDM^{++}. VDM^{++} provides a number of constructs for specifying real-time constraints, such as the `whenever` statement [7]. However there does not yet exist a systematic method for the development of real-time systems using VDM^{++}.

We will show here that existing methods such as HRT-HOOD can be enhanced and combined with VDM^{++}/RAL, as follows:

- HRT-HOOD object descriptions can be given as VDM^{++} class definitions.
- Timing constraints in HRT-HOOD objects can be formally expressed using RAL formulae.
- Operation request types can be represented by VDM^{++} operation definitions, combined with RAL formulae.

The advantage of carrying out design in HRT-HOOD using VDM^{++} is that precise mathematical constraints can be expressed in the language, and that design can be carried out from an abstract functional and time-specification through to a level close to a final implementation language such as Ada95. VDM^{++} also allows a more abstract description of a system using a "model 0" flat specification, with timing and synchronisation constraints expressed via logical formulae rather than via pseudocode.

In the appendix we describe the RAL formalism, which combines aspects of RTL, LTL and modal action logic formalisms in a coherent framework in order to attempt to meet the criteria and capabilities for real-time formalisms given in [8], and to enable formal treatment of real-time constraints as found in methods such as HRT-HOOD. RAL can be used as an underlying semantics of VDM^{++} to support reasoning about internal consistency and refinement of classes.

2 VDM^{++}

A VDM^{++} specification consists of a set of *class* definitions, where these have the general form:

```
class C
types
  T = TDef
values
  const : T = val
functions
  f : A → B
  f(a) == Defn_f(a)
time variables
  input i_1 : T_1;
        ...
  input i_n : T_n;
        o_1 : S_1;
        ...
        o_m : S_m;
  assumption i_1, ..., i_n == Assumes;
  effect i_1, ..., i_n, o_1, ..., o_m == Effects
instance variables
  v_C : T_C;
inv objectstate == Inv_C;
init objectstate == Init_C
methods
  m(x : X_m,C) value y : Y_m,C
    pre Pre_m,C(x, v_C) == Defn_m,C;
   ...
sync ...
thread ...
aux reasoning ...
end C
```

The `types`, `values` and `functions` components define types, constants and functions as in conventional VDM (although class reference sets @**D** for class names **D** can be used as types in these items – such classes **D** are termed *suppliers* to **C**, as are instances of these classes. **C** is then a *client* of **D**). The `instance variables` component defines the attributes of the class, and the `inv` defines an invariant over a list of these variables: `objectstate` is used to include all the attributes. The `init` component defines a set of initial states in which an object of the class may be as a result of object creation. Object creation is achieved via an invocation of the operation **C!new**, which returns a reference to the new object of **C** as its result.

The `time variables` clause lists continuously varying attributes which are either inputs (the $\mathbf{i_j}$) with matching `assumption` clauses describing assumptions about how these inputs change over time, or outputs (the $\mathbf{o_k}$), with `effect` clauses defining how these are related to the inputs and to each other.

The methods of **C** are listed in the `methods` clause. The list of method names of **C**, including inherited methods, is referred to as $\underline{\mathbf{methods}}(\mathbf{C})$.

Methods can be defined in an abstract declarative way, using *specification statements*, or by using a hybrid of specification statements, method calls and procedural code. Input parameters are indicated within the brackets of the method header, and results after a `value` keyword. Preconditions of a method are given in the `pre` clause.

Other clauses of a class definition control how **C** inherits from other classes: the optional `is subclass of` clause in the class header lists classes which are being extended by the present class – that is, all their methods become exportable facilities of **C**.

Dynamic behaviour of objects of **C** is specified in the `sync` and `thread` clauses, which must not conflict. In the `sync` clause, which describes the behaviour of *passive* objects, either an explicit history of an object can be given, as a *trace* expression involving regular expressions in terms of method names, or as a set of *permission* statements of the form:

`per` **Method** $\Rightarrow$ **Cond**

restricting the conditions under which methods can initiate execution. The guard condition **Cond** can involve event counters $\#\mathbf{act(m)}$, $\#\mathbf{req(m)}$ and $\#\mathbf{fin(m)}$.

Threads describe the behaviour of active objects, and can involve general statements, including a `select` statement construct allowing execution paths to be chosen on the basis of which messages are received first by the object, similar to the `select` of Ada or `ALT` of OCCAM.

A set of internal consistency requirements are associated with a class, which assert that its state space is non-empty, and that the definition of a method maintains the invariant of the class, and that the initialisation predicate implies the invariant.

Refinement obligations, based on theory extension, can also be given.

In discussing the semantics of VDM^{++} and HRT-HOOD we will make use of the following RAL terms (Table 1).

2.1 Examples of Specification

An example of the type of properties that can be specified in an abstract declarative manner using RAL and VDM^{++} is a *periodic* timing constraint: "**m** initiates every **t** seconds, and in the order of its requests":

$$\forall \mathbf{i} : \mathbb{N}_1 \cdot \uparrow(\mathbf{m(x)}, \mathbf{i}+1) \;=\; \uparrow(\mathbf{m(x)}, \mathbf{i}) + \mathbf{t}$$

Symbol	*Meaning*
$@\mathbf{C}$	Set of possible object identifiers of objects of class $\mathbf{C}$
$\overline{\mathbf{C}}$	Set of identifiers of currently existing objects of class $\mathbf{C}$
$\uparrow(\mathbf{m}(\mathbf{x}),\mathbf{i})$	Activation time of $\mathbf{i}$-th invocation of action $\mathbf{m}(\mathbf{x})$
$\rightarrow(\mathbf{m}(\mathbf{x}),\mathbf{i})$	Request time of $\mathbf{i}$-th invocation of action $\mathbf{m}(\mathbf{x})$
$\downarrow(\mathbf{m}(\mathbf{x}),\mathbf{i})$	Termination time of $\mathbf{i}$-th invocation of action $\mathbf{m}(\mathbf{x})$
$\#\mathbf{act}(\mathbf{m})$	Number of activations of $\mathbf{m}$ to date
$\#\mathbf{req}(\mathbf{m})$	Number of received requests of $\mathbf{m}$ to date
$\#\mathbf{fin}(\mathbf{m})$	Number of terminated invocations of $\mathbf{m}$ to date
$\varphi \odot \mathbf{t}$	φ holds at time $\mathbf{t}$
$\mathbf{e} \circledast \mathbf{t}$	Value of $\mathbf{e}$ at time $\mathbf{t}$
$\clubsuit(\varphi := \mathbf{true}, \mathbf{i})$	Time at which φ becomes true for the $\mathbf{i}$-th time

Table 1. RAL Symbols and Meanings

More generally, in HRT-HOOD, periodic objects involve a periodic action such as $\mathbf{m}$, a period $\mathbf{P}$ for its execution period, and a deadline $\mathbf{D}$ for its completion within each period:

$$\begin{aligned} &\forall \mathbf{i} : \mathbb{N}_1 \cdot \\ &\qquad \mathbf{P} * (\mathbf{i} - 1) \ \leq\ \uparrow(\mathbf{m}, \mathbf{i}) \ \wedge \\ &\qquad \downarrow(\mathbf{m}, \mathbf{i}) \ \leq\ \mathbf{P} * (\mathbf{i} - 1) + \mathbf{D} \end{aligned}$$

where $\mathbf{D} < \mathbf{P}$.

This periodic behaviour can be expressed via a periodic thread and the `whenever` statement of VDM^{++}, provided that the methods of the class concerned are mutex:

```
effects ... ==
  whenever P  |  now
  also from D ==>
        #fin(m)  =  ←#fin(m)  +  1    ∧    #act(m)  =  ←#act(m)  +  1
 ...
thread
  periodic(P)(m)
```

In words "within $\mathbf{D}$ time units of any time $\mathbf{t}$ which is a multiple of $\mathbf{P}$, there will have been one more initiated and completed execution of $\mathbf{m}$ than at $\mathbf{t}$". Here $\overleftarrow{\mathbf{att}}$ denotes the value of $\mathbf{att}$ at the time of occurrence of the most recent trigger event.

Periodic objects may also support high-priority asynchronous actions. These methods, say **interrupt**, will have:

$$\forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{delay}(\mathbf{interrupt}, \mathbf{i}) \ < \ \delta$$

where $\mathbf{delay}(\mathbf{m}, \mathbf{i}) \ = \ \uparrow(\mathbf{m}, \mathbf{i}) - \rightarrow(\mathbf{m}, \mathbf{i})$, δ is some small time bound for the response time of the periodic object to this interrupt.

Sporadic constraints can also be directly expressed. In HRT-HOOD the sporadic action **m** may have a deadline **D**, which we can express as a constraint on its duration:

$$\forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{duration(m, i)} \ \leq \ \mathbf{D}$$

where $\mathbf{duration(m,i)} \ = \ \downarrow\!(\mathbf{m},\mathbf{i}) - \uparrow\!(\mathbf{m},\mathbf{i})$. There may also be interrupts specified as for periodic objects.

Timeouts can be specified for synchronous methods in HRT-HOOD. These require that an operation **exception** is executed if an operation **op** fails to respond to a request within **t** time units:

$$\begin{array}{l}
\mathbf{timeout\_on\_delay(op, exception, t)} \ \equiv \\
\quad (\forall \mathbf{i} : \mathbb{N}_1 \cdot \\
\qquad \mathbf{delay(op, i)} \ \geq \ \mathbf{t} \ \Rightarrow \\
\qquad\quad \exists! \mathbf{j} : \mathbb{N}_1 \ \cdot \uparrow\!(\mathbf{exception}, \mathbf{j}) \ = \ \rightarrow\!(\mathbf{op}, \mathbf{i}) + \mathbf{t}) \ \wedge \\
\quad (\forall \mathbf{j} : \mathbb{N}_1 \cdot \\
\qquad \exists! \mathbf{i} : \mathbb{N}_1 \ \cdot \uparrow\!(\mathbf{exception}, \mathbf{j}) \ = \ \rightarrow\!(\mathbf{op}, \mathbf{i}) + \mathbf{t} \ \wedge \\
\qquad\qquad \mathbf{delay(op, i)} \ \geq \ \mathbf{t})
\end{array}$$

Likewise, a timeout on the duration of **op** can be expressed.

Priority levels for operations and objects can be stated in a number of ways. For example, we could say that a high priority action α being active suppresses any low-priority actions β:

$$\#\mathbf{active}(\alpha) > 0 \ \ \Rightarrow \ \ \#\mathbf{active}(\beta) = 0$$

where $\#\mathbf{active(m)} \ = \ \#\mathbf{act(m)} - \#\mathbf{fin(m)}$, the number of currently executing invocations of **m**. Alternatively, if both α and β are waiting to execute, it is always an α instance that is chosen:

$$\texttt{per}\ \beta \quad \Rightarrow \quad \#\mathbf{active}(\alpha) \ = \ 0$$

"there are no active invocations of α at initiation of execution of β".

Other forms of prioritisation constraints, permission constraints, timeouts and responsiveness constraints can all be specified in a direct manner using RAL [12]. All the forms of method invocation protocols for concurrent objects described in [4] can be precisely described in this logic in a similar way. The Ada rendez-vous interpretation is the default for VDM^{++}.

3 Models of Time

We wish to model both discrete and hybrid systems, with continuous behaviour specified using *time variables* which may be constrained by general differential and integral calculus formulae. In order that such class specifications are meaningful, we place the following constraints on instance and time variables and the set **TIME** of times.

We require that each (discrete) instance variable $\mathbf{att} : \mathbf{T}$ in a VDM^{++} specification can only take finitely many different values (including the undefined value $\mathbf{nil_T}$) over its lifetime:

$$\{\mathbf{att}\circledast\mathbf{t} \mid \mathbf{t} \in \mathbf{TIME}\} \ \in \ \mathbb{F}(\mathbf{T})$$

and the set of times over which $\mathbf{att}$ has a particular value is a finite union of intervals:

$$\begin{array}{l}\forall\, \mathbf{val} : \mathbf{T};\ \exists\, \mathbf{I} : \mathbb{N} \rightarrow \mathcal{I}(\mathbf{TIME});\ \mathbf{n} : \mathbb{N} \cdot \\ \qquad \{\mathbf{t} \mid \mathbf{t} : \mathbf{TIME} \wedge \mathbf{att}\circledast\mathbf{t} = \mathbf{val}\} \ = \ \bigcup_{\mathbf{i}\leq\mathbf{n}} \mathbf{I}(\mathbf{i})\end{array}$$

where $\mathcal{I}(\mathbf{TIME})$ is the set of interval subsets of $\mathbf{TIME}$, that is, subsets $\mathbf{J}$ such that

$$\begin{array}{l}\forall\, \mathbf{t}, \mathbf{t}' : \mathbf{J} \ \cdot \ \mathbf{t} < \mathbf{t}' \ \Rightarrow \\ \qquad\qquad \forall\, \mathbf{t}'' : \mathbf{TIME} \ \cdot \ \mathbf{t} \leq \mathbf{t}'' \leq \mathbf{t}' \ \Rightarrow \ \mathbf{t}'' \in \mathbf{J}\end{array}$$

This is a generalisation of the model of [20]. $\mathbf{e}\circledast\mathbf{t}$ is the value of $\mathbf{e}$ at time $\mathbf{t}$.

In addition, we usually expect that the lifetime of an object is contiguous:

$$\begin{array}{l}\forall\, \mathbf{obj} : @\mathbf{C} \cdot \\ \qquad \{\mathbf{t} \mid \mathbf{t} : \mathbf{TIME} \wedge (\mathbf{obj} \in \overline{\mathbf{C}})\circledcirc\mathbf{t}\} \ \in \ \mathcal{I}(\mathbf{TIME})\end{array}$$

where $@\mathbf{C}$ is the type of *possible* object identities of objects of class $\mathbf{C}$, whilst $\overline{\mathbf{C}} : \mathbb{F}(@\mathbf{C})$ is the set of identities of objects of $\mathbf{C}$ that currently exist.

The usual model of time in VDM^{++} is that $\mathbf{TIME}$ is the set of non-negative real numbers, with the usual ordering and operations. Because of the above properties of attributes representing instance variables however, they can be considered to use a simpler concept of time: a discrete non-dense set of points from the non-negative reals. Likewise for action symbols denoting methods.

In addition, in order for `whenever` φ ... statements to be well-defined, the times $\clubsuit(\mathbf{a}.\varphi := \mathbf{true}, \mathbf{i})$ at which $\mathbf{a}.\varphi$ becomes true for the $\mathbf{i}$-th time must be well-defined, ie, there must be clear cuttoff points for φ. This will be the case, if, for example, φ is of the form $\mathbf{time\_variable} \ \geq \ \mathbf{constant}$, where $\mathbf{time\_variable}$ is an input time variable with a piecewise continuous graph.

4 Formalising Real-time Refinement

The most interesting forms of refinement step are those which involve a transformation from a continuous model of the world to a discrete model. In VDM^{++} the continuous model is expressed by means of *time variables* of a class, which define attributes that change their value in a manner independent of the methods of the class, and, for real-valued time variables, possibly in a continuous manner.

Figure 1 shows the kinds of model and model transformation which are typically performed in VDM^{++}, starting from a highly abstract continuous model of both the combined controlled system/controller (an *essential* model in the

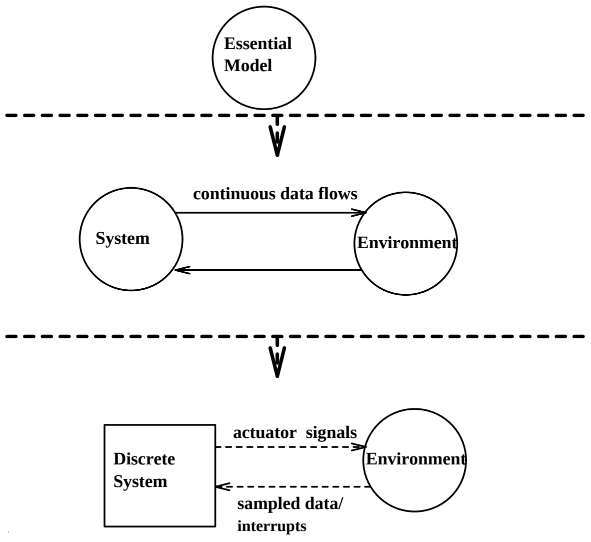

Fig. 1. Model Transformations: Continuous to Discrete

terms of Syntropy [6]), then moving to a model where the controller is separated from the controlled system, and then to a model where the controller is replaced by a discretisation. In the step to the discrete controller, output time variables received by the controlled system are replaced by calls to actuator devices to achieve the required effects. Inputs from the controlled system to the controller are replaced by sampling methods.

This is the approach taken in the paper [7]. Alternatively, the continuous controller model may be divided into a number of continuous classes before discretisation, as is done in the case study of Section 4.5 below.

A fundamental problem in each of these approaches is the transition from the continuous model to the discrete model. The transformation of data in this case does not fit into the usual VDM^{++} style of functional refinement: there is no simple retrieve function that converts the discretisation of a continuous quantity back to that quantity, because some information has been lost. Instead there are two alternatives:

1. accept that this is a case of *relational* data refinement. This complicates the proof theory of refinement and the difficulty of proof and provides no real "measure of closeness" between the continuous and discrete models;

2. use a concept of *approximate* refinement, whereby the retrieve function takes a concrete data item into an approximation of some ideal value (in this case, the retrieve function is some digital-analogue (DA) conversion which tries to recover a continuous function from a sequence of sampled points taken from it).

Other alternatives could include mapping the discrete value to the *set* of possible continuous abstractions of it, which should include the actual abstraction it derives from.

We will examine both 1 and 2 below.

4.1 Periodic Constraint Refinement

A common form of transformation from continuous to discrete views of a system starts from a class which abstractly describes a reaction to an event (the condition **C** becoming true):

```
class ContinuousController
time variables
  input it : X;
        ot : S;
  effect it, ot   ==
    whenever C(it)
    also from δ_0  ==>  ot  =  v
end ContinuousController
```

If we can assume that **C** remains true for at least $\mathbf{t_C} > 0$ time units from the points where it becomes true, then we can define a sampling approach:

```
class AbstractController
-- refines ContinuousController
time variables
  input it : X;
        ot : S;
  effect it, ot   ==
    whenever C(it)  ∧  (P  |  now)
    also from δ  ==>  ot  =  v
end AbstractController
```

where $\mathbf{P} \leq \mathbf{t_C}$ and $\delta + \mathbf{P} \leq \delta_0$. These constraints guarantee that no $\mathbf{C} := \mathbf{true}$ events are missed by the **AbstractController**, and that it responds within δ_0 to such events. $\mathbf{P} \mid \mathbf{now}$ denotes that **P** divides the current time.

This class is then implemented by a periodic action:

```
class ConcreteController
-- refines AbstractController
time variables
  input it : X;
instance variables
  id : X;
  ot_obj : @SClass
methods
  react()  ==
     (id  :=  it;
      if C(id)
      then
        ot_obj!set(v))
```

```
thread
  periodic($\mathbf{P}$)($\mathbf{react}$)
aux reasoning
  $\forall\ \mathbf{i} : \mathbb{N}_1 \cdot \downarrow(\mathbf{react},\mathbf{i}) \ \leq\ \mathbf{P} * (\mathbf{i} - 1) + \delta$
end $\mathbf{ConcreteController}$
```

The abstract specification requires that, at each periodic sampling time $\mathbf{P}$, if $\mathbf{C(it)}$ holds, then within a response deadline δ, $\mathbf{ot}$ has the value $\mathbf{v}$:

$$\begin{array}{l} \forall \mathbf{i} : \mathbb{N} \cdot \\ \quad \mathbf{C(it)} \circledcirc (\mathbf{P} * \mathbf{i}) \Rightarrow \\ \qquad \exists \mathbf{t} : \mathbf{TIME} \cdot \\ \qquad\quad \mathbf{P} * \mathbf{i} \ \leq\ \mathbf{t} \ \leq\ \mathbf{P} * \mathbf{i} + \delta \ \wedge\ (\mathbf{ot} = \mathbf{v}) \circledcirc \mathbf{t} \end{array}$$

The form of the test guarantees that the times $\clubsuit(\mathbf{C(it)} \wedge (\mathbf{P} \mid \mathbf{now}) := \mathbf{true}, \mathbf{i})$ are well-defined, regardless of $\mathbf{C}$. $\varphi \circledcirc \mathbf{t}$ expresses that φ holds at time $\mathbf{t}$.

The concrete specification attempts to satisfy this specification by an explicit sampling and invocation. The periodic thread implies that:

$$\begin{array}{l} \forall \mathbf{i} : \mathbb{N}_1 \cdot \\ \qquad \mathbf{P} * (\mathbf{i} - 1) \ \leq\ \uparrow(\mathbf{react},\mathbf{i}) \ \wedge\ \downarrow(\mathbf{react},\mathbf{i}) \ \leq\ \mathbf{P} * \mathbf{i} \end{array}$$

and we also know from the deadline specification that in fact $\downarrow(\mathbf{react},\mathbf{i}) \ \leq\ \mathbf{P} * (\mathbf{i} - 1) + \delta$.

So assume $\mathbf{C(it)}$ holds at $\mathbf{P} * \mathbf{i}$ for some $\mathbf{i} \in \mathbb{N}$. Assume also that it remains true until the point $\uparrow(\mathbf{react},\mathbf{i}+1)$ where $\mathbf{it}$ is sampled by the concrete controller. Generally we will require that the period of sampling is fast enough with respect to the rate of change of $\mathbf{it}$ that it does not significantly change its value in the delay from $\mathbf{P} * \mathbf{i}$ to $\uparrow(\mathbf{react},\mathbf{i}+1)$.

Then **react** will send the message **set**(**v**) to the object that replaces the output time variable **ot**, ie, we could have the refinement that **ot** in the abstract class is interpreted by **ot_obj.ot** in the concrete, where **ot** is still a time variable in **ot_obj**.

We can then prove that the effect $\mathbf{ot} = \mathbf{v}$ is achieved by the time $\mathbf{P} * \mathbf{i} + \delta$, if **set** is synchronous, because then it terminates and achieves $\mathbf{ot} = \mathbf{v}$ before $\downarrow(\mathbf{react},\mathbf{i}+1) \ \leq\ \mathbf{P} * \mathbf{i} + \delta$ as required.

If **set** is asynchronous, with maximum delay plus duration ϵ, say, then we must use $\delta - \epsilon$ as the deadline for **react** in **ConcreteController**.

A consistency check that $\mathbf{duration(react)} \leq \gamma$ for whichever deadline γ is chosen must be made[1].

At the abstract specification level, a corresponding check must be made that multiple `whenever` statements do not require conflicting situations to occur at the same time.

Thus we can, in principle, prove such a refinement step correct.

[1] Under the assumption that the processor is adequate and not overloaded.

4.2 Discrete Sampling

The difficulty comes in the conversion from analogue to digital quantities – such as the relationship between **id** and **it** in the **ConcreteController** class. Consider the general case where we have an original abstract system which just contains a time variable (an example in the case study would be the operator console, containing **alarm**, or the module containing a sampled version of the CH4 level).

```
class Output
time variables
  ot : S
end Output
```

A refinement could instead contain a sampled version of the same data:

```
class Output_1
instance variables
  d : seq of S
methods
  set(v : S) == d := d ⌢ [v];

  access() value S
       pre len(d) > 0 == return d(len(d))
end Output_1
```

with the refinement relation being that **d** is some sampling of the abstract variable:

$$\mathbf{sampled}(\mathbf{d}, \mathbf{ot}, \mathbf{P}, \mathbf{D})$$

This is an abbreviation for:

$$\begin{aligned} &\forall \mathbf{k} : \mathbb{N} \cdot \exists \mathbf{t} : \mathbf{TIME} \cdot \\ &\qquad \mathbf{k} * \mathbf{P} < \mathbf{t} \leq \mathbf{k} * \mathbf{P} + \mathbf{D} \ \wedge \ \mathbf{k} + 1 \in \mathrm{dom}(\mathbf{d} \circledast (\mathbf{k} * \mathbf{P} + \mathbf{D})) \ \wedge \\ &\qquad \mathbf{d}(\mathbf{k} + 1) \circledast (\mathbf{k} * \mathbf{P} + \mathbf{D}) = \mathbf{ot} \circledast \mathbf{t} \end{aligned}$$

That is, **ot** is sampled in each interval of the form $(\mathbf{k} * \mathbf{P}, \mathbf{k} * \mathbf{P} + \mathbf{D}]$ and its value assigned to the $\mathbf{k} + 1$-th element of **d**.

More generally, we could sample an expression **e** involving various time variables and instance variables, including previous values of **d**. The predicate $\mathbf{sampled}(\mathbf{d}, \mathbf{ot}, \mathbf{P}, \mathbf{D})$ does describe a refinement *relation* between the abstract and concrete data, but not a retrieve *function* from the concrete state to the abstract state: for a given **d** there is not necessarily a unique **ot** such that $\mathbf{sampled}(\mathbf{d}, \mathbf{ot}, \mathbf{P}, \mathbf{D})$ is true. An alternative is to somehow characterise the error between a functional refinement mapping from **d** back to the continuous domain, and the original **ot** value (Figure 2 shows the general situation here). For example, let **ot** be a boolean quantity, modelled by a $\{0, 1\}$ value set. Assume that the minimum inter-arrival time of events that change the value of **ot** is **I**. Then the abstract model can be regarded as a sequence of intervals $\mathbf{I}_0$, $\mathbf{I}_1$, ...,

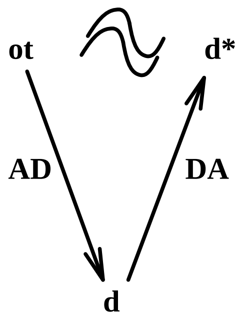

Fig. 2. Approximate Refinement

which are demarcated by changes in the value of $\mathbf{ot}$. Each of the $\mathbf{I_j}$ has length $\mathbf{len(I_j)} \geq \mathbf{I}$.

Define the DA conversion function from $\mathbf{d}$ to a time series $\mathbf{d}^*$ as follows:

$$\begin{aligned}\mathbf{d}^*(\mathbf{t}) &= \mathbf{d}(1) \quad \text{if } \mathbf{t} \leq \mathbf{t}_0 \\ &= \mathbf{d}(2) \quad \text{if } \mathbf{t}_0 < \mathbf{t} \leq \mathbf{t}_1 \\ &= \ldots \\ &= \mathbf{d}(\mathbf{i}+2) \quad \text{if } \mathbf{t_i} < \mathbf{t} \leq \mathbf{t_{i+1}}\end{aligned}$$

where $\mathbf{t_i}$ is the time at which the $\mathbf{i}$+1-th sample of $\mathbf{ot}$ is taken, so $\mathbf{d}^*(\mathbf{t_i}) = \mathbf{ot}(\mathbf{t_i})$.

Other DA functions are possible, but the error estimations are similar in each case. Here, we are concerned to estimate the integral

$$\int_0^{\mathbf{T}} \mid \mathbf{ot}(\mathbf{t}) - \mathbf{d}^*(\mathbf{t}) \mid \ \mathbf{dt}$$

for a given time $\mathbf{T}$. This is our measure of error in the approximate retrieve function that takes $\mathbf{d}$ to $\mathbf{d}^*$. We can break this integral down into a sum:

$$\Sigma_{\mathbf{i} \in \mathbb{N}_1, \mathbf{j} \in \mathbb{N}} \mid \mathbf{ot}(\uparrow \mathbf{I_j}) - \mathbf{d}^*(\mathbf{t_i}) \mid * \mathbf{len}(\mathbf{I_j} \cap (\mathbf{t_{i-1}}, \mathbf{t_i}])$$

over all the segments $\mathbf{I_j} \cap (\mathbf{t_{i-1}}, \mathbf{t_i}]$ where the values of both variables are constant, and $\mathbf{t_i} \leq \mathbf{T}$. $\uparrow \mathbf{I_j}$ refers to the starting time point of the $\mathbf{j}$-th interval.

If we then assume that $\mathbf{P} + \mathbf{D} < \mathbf{I}$, we can infer that:

$$\int_0^{\mathbf{k}*\mathbf{I}} \mid \mathbf{ot}(\mathbf{t}) - \mathbf{d}^*(\mathbf{t}) \mid \ \mathbf{dt} \leq 2\mathbf{k} * (\mathbf{P} + \mathbf{D})$$

This enables a crude upper bound to be placed on the error in the approximation. Clearly, the smaller we make the period and the deadline $\mathbf{P}$ and $\mathbf{D}$ involved in the sampling, the more accurate is the approximate refinement. $\mathbf{D} < \mathbf{P}$ is necessary, and $\mathbf{D}$ must be greater than the duration of the sampling method involved (in a system object such as **ConcreteController** above).

Similar reasoning can be applied to other discrete-valued time variables.

4.3 Approximate Refinement

Given some DA conversion function $\mathbf{R}$ as described in the previous section, we require the following conditions for an "approximate refinement" based on this function.

1. $\mathbf{R}$ should be the left inverse of an adequate (surjective) function from the continuous to the discrete space;
2. the error between the continuous approximated variable $\mathbf{c}$ and $\mathbf{R}(\mathbf{d})$ where $\mathbf{d}$ is the corresponding discrete variable, should be boundable in terms which can be engineered by the system designer;
3. each axiom φ of the continuous system should be provable in the interpreted form $\varphi[\mathbf{R}(\mathbf{d})/\mathbf{c}]$ in the discrete system.

These three conditions have the following justifications:

1. We should be able to see the transformation from a continuous to the discrete model as an abstraction step, removing domain detail that is irrelevant to the implemented system, similar to the transition from the *essential* models of Syntropy [6], which describe the "real world" domain, and the *specification* models, which describe the required software (Figure 3).
2. The developer should be able to control the quality of the discrete approximation, and trade off this quality against other aspects, such as efficiency.
3. Every property required by the abstract continuous class should be true in interpreted form in the new class – this is the usual meaning of refinement and subtyping as theory extension.

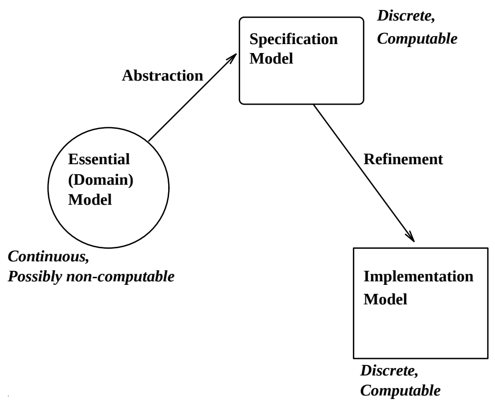

Fig. 3. Essential, Specification and Implementation Model Relationships

Examples of this process are given in the papers [16, 17].

4.4 Sporadic Constraint Refinement

The final major case of refinement from continuous to discrete views of a system involves the implementation of sporadic response requirements by sporadic, interrupt-driven classes. This can be treated by replacing a boolean input time variable by an interrupt [15].

4.5 Case Study

An example of combined analysis and design using VDM^{++}/RAL and HRT-HOOD is as follows, based on the mine pump example used in [4].

The requirements are as follows:

1. the system should respond to the water low and water high conditions within 20 seconds – switching the pump on if the water goes high with the methane level below the danger level, and switching it off if the water level goes low. The minimum inter-arrival time for these events is 100 seconds;
2. the system should respond to high methane conditions within 1 second, switching off the pump and raising a "methane high" alarm. The methane is sampled at 5 second intervals;
3. the system should respond to critically high levels of CO within 1 second, raising a "CO high" alarm. The CO level is sampled at 60 second intervals;
4. the system should respond to a critically low air flow reading within 2 seconds, raising a "low air flow" alarm. The air flow is sampled at 60 second intervals;
5. the system should respond to a low water flow reading when the pump is on within 3 seconds, raising a "pump failed" alarm. The water flow is sampled at 60 second intervals.

Based on the analysis of the requirements, we could specify the system as a single class with input time variables representing the measured CH4 and CO levels, the water level and water and airflow levels, together with the outputs – the data logger, motor state and alarm. This class can be derived from a context diagram and the detailed requirements of functionality:

```
class Model_0
types
  Alarm_status = < high_methane > | < low_air_flow > | < high_water > |
                 < high_co > | < pump_failed > | < safe >
values
  ch4_high : ℕ = undefined;
  co_high : ℕ = undefined;
  jitter_range : ℕ = undefined;
  water_low : ℕ = undefined
time variables
  input co_reading : ℕ;
  input ch4_reading : ℕ;
  input low_sensor : bool;
  input high_sensor : bool;
  input water_flow : ℕ;
  input air_flow : ℕ;
        motor_on : bool;
        alarm : Alarm_status;

 /* Requirement 1: */

effect high_sensor, ch4_reading, motor_on ==
  whenever high_sensor ∧ (ch4_reading ≤ ch4_high − jitter_range)
  also from 20000 ==> motor_on;

effect low_sensor, motor_on ==
  whenever low_sensor
  also from 20000 ==> ¬ (motor_on);

 /* Requirement 2: */

effect ch4_reading, alarm ==
  whenever ch4_reading ≥ ch4_high ∧ (5000 | now)
  also from 1000 ==> alarm = < high_methane >;

effect ch4_reading, motor_on ==
  whenever ch4_reading ≥ ch4_high ∧ (5000 | now)
  also from 1000 ==> ¬ (motor_on);

 /* Requirement 3: */

effect co_reading, alarm ==
  whenever co_reading ≥ co_high ∧ (60000 | now)
  also from 1000 ==> alarm = < high_co >;

 /* Requirement 4: */

effect air_flow, alarm ==
  whenever air_flow ≤ flow_low ∧ (60000 | now)
  also from 2000 ==> alarm = < low_air_flow >;

 /* Requirement 5: */

effect water_flow, motor_on, alarm ==
  whenever (water_flow ≤ water_low) ∧
           motor_on ∧ (60000 | now)
  also from 3000 ==> alarm = < pump_failed >

end Model_0
```

We ignore data logging for simplicity in this initial model. The above class could also contain differential/integral calculus expressions relating the CH4 level to the air-flow rate. **jitter_range** provides some hysteresis for the system.

Notice that many of the above effect clauses give the same sample points $60000 * \mathbf{k}$ for the relevant tests on sensor values to be made. It may be infeasible for these tests to be made at exactly this time in the actual system, if the sampling tasks share a processor. However, any minor time-displacement of this kind should not make any difference to the truth or falsity of the test concerned. This is an auxilliary proof requirement which would require knowledge of the relevant rate of changes involved.

We can clearly factor this model into parts which contain smaller subsets of the data. The first subsystem is an environment monitor which contains the various gas monitors:

```
class EnvironmentMonitor
  is subclass of Basic_types
time variables
  input co_reading : ℕ;
  input ch4_reading : ℕ;
  input air_flow : ℕ;
        motor_on : bool;
        alarm : Alarm_status;

effect ch4_reading, alarm ==
  whenever ch4_reading ≥ ch4_high ∧ (5000 | now)
  also from 1000 ==>
               alarm = < high_methane >;

effect co_reading, alarm ==
  whenever co_reading ≥ co_high ∧ (60000 | now)
  also from 1000 ==> alarm = < high_co >;

effect ch4_reading, motor_on ==
  whenever ch4_reading ≥ ch4_high ∧ (5000 | now)
  also from 1000 ==> ¬ (motor_on);

effect air_flow, alarm ==
  whenever air_flow ≤ flow_low ∧ (60000 | now)
  also from 2000 ==> alarm = < low_air_flow >;

methods
  check_safe() value bool ==
    return (ch4_reading ≤ ch4_high − jitter_range)
end EnvironmentMonitor
```

We need the method **check_safe** to support the implementation of the first requirement.

This can be further decomposed into individual monitors plus a protected object to support external queries to the CH4 status:

```
class Ch4_Sensor
```

```
  is subclass of Basic_types
time variables
  input ch4_reading : ℕ;
        motor_on : bool;
        alarm : Alarm_status;
effect ch4_reading, alarm ==
  whenever ch4_reading ≥ ch4_high ∧ (5000 | now)
  also from 1000 ==>
              alarm = < high_methane >;

effect ch4_reading, motor_on ==
  whenever ch4_reading ≥ ch4_high ∧ (5000 | now)
  also from 1000 ==> ¬ (motor_on)
end Ch4_Sensor

class Ch4Status
  is subclass of Basic_types
instance variables
  environment_status : Ch4_status;
init objectstate ==
  environment_status := < motor_safe >
methods
  read() value Ch4_status ==
    return environment_status;

  write(v : Ch4_status) ==
    environment_status := v
end Ch4Status
```

Similarly for the **Air_flow_Sensor** and **CO_Sensor** classes.

Together these can be aggregated to refine the original subsystem specification:

```
class EnvironmentMonitor_1
-- refines EnvironmentMonitor
  is subclass of Basic_types, Ch4_Sensor,
     Air_flow_Sensor, CO_Sensor, Ch4Status
methods
  check_safe() value bool ==
    return (environment_status = < motor_safe >)
end EnvironmentMonitor_1
```

The refinement relation is that the abstract time variables in **EnvironmentMonitor** are implemented by the corresponding time variables in the individual classes, whilst the attribute **environment_status** is a discretisation of the test for motor safety, with a sampling time within 1 second of the times $5000 * \mathbf{k}$ for $\mathbf{k} \in \mathbb{N}$:

```
 sampled(environment_status,
   if ch4_reading ≤ ch4_high − jitter_range
   then < motor_safe >
```

```
$\mathbf{else}$
  $\mathbf{if\ ch4\_reading} \ \geq \ \mathbf{ch4\_high}$
  $\mathbf{then} \ \mathbf{< motor\_unsafe >}$
  $\mathbf{else} \ \overleftarrow{\mathbf{environment\_status}}, \ 5000, \ 1000)$
```

Here $\overleftarrow{\mathbf{environment\_status}}$ refers to the value at the previous sampling interval.

This approach corresponds to the architecture given in the paper [4]. Each of the objects with continuous variables will eventually be implemented as cyclic objects using a periodic thread. In the case of the CO sensor and air flow sensor objects the thread will have a periodicity of 60 seconds, whilst the CH4 sensor has a periodicity of 5 seconds.

Because we are using multiple inheritance to put together the components, which corresponds to aggregation in a object-oriented method such as Fusion [5], we must identify priorities for objects (generally, the objects with the shortest deadlines and periods will have the highest priority, etc). In a simple system such as this one, we can interpret the priorities as determining the precedence of execution of two operations: if class **C** has a higher priority than class **D**, then each object **a** of **C** can always commence execution of an operation **m** of **C** in preference to any object **b** of **D** waiting to execute an operation **n** of **D**:

$$\forall \mathbf{j} : \mathbb{N}_1 \cdot (\mathbf{a}.\#\mathbf{waiting}(\mathbf{m}) = 0) \odot \uparrow(\mathbf{b}!\mathbf{n}, \mathbf{j})$$

An internal consistency check must be made, that all the separate periodic and deadline constraints specified in the individual classes can still be satisfied, given these priorities, when they are aggregated into a mutex container class.

We take the priorities from [4]:

Class	*Priority*
CH4 status	7
motor	6
CH4 sensor	5
CO sensor	4
airflow sensor	3
water flow sensor	2
HLW handler	1

Formally we must have that

$$\forall \mathbf{i}, \mathbf{j} : \mathbb{N}_1 \ \cdot \ \mathbf{duration}(\mathbf{opcs\_periodic\_code}, \mathbf{i}) \ \leq \\ 1000 - \mathbf{duration}(\mathbf{co\_periodic\_code}, \mathbf{j})$$

where **co_periodic_code** is the periodic action of the CO sensor. Likewise:

$$\forall \mathbf{i}, \mathbf{j}, \mathbf{k} : \mathbb{N}_1 \cdot \\ \mathbf{duration}(\mathbf{opcs\_periodic\_code}, \mathbf{i}) + \\ \mathbf{duration}(\mathbf{co\_periodic\_code}, \mathbf{j}) \ \leq \\ 2000 - \mathbf{duration}(\mathbf{air\_flow\_code}, \mathbf{k})$$

in order that all three periodic codes can achieve their deadlines, assuming that they must be executed in a mutually exclusive and uninterrupted manner.

We can obtain slightly more refined requirements by examining the different periods concerned: **opcs_periodic_code** and **co_periodic_code** only possibly conflict, for example, in the time intervals of length 5000 beginning with a multiple of 60000. Thus we could require just:

$$\begin{array}{l} \forall \ell : \mathbb{N} \cdot \\ \quad \mathbf{duration}(\mathbf{opcs\_periodic\_code}, 1 + 12\ell) \leq \\ \qquad\qquad 1000 - \mathbf{duration}(\mathbf{co\_periodic\_code}, 1 + \ell) \end{array}$$

The **Basic_types** class encapsulates shared type and constant definitions:

```
class Basic_types
types
  Rate  =  ℕ;
  Ch4_reading  =  ℕ;
  Ch4_status  = < motor_safe > | < motor_unsafe >;
  Alarm_status  =
       < high_methane > | < low_air_flow > | < high_co > |
       < high_water > | < pump_failed > | < safe >
values
  water_low :  ℕ  =  undefined;
  ch4_high :  Ch4_reading  =  undefined;
  co_high :  ℕ  =  undefined;
  jitter_range :  Ch4_reading  =  undefined;
  ch4_sensor_period :  ℕ  =  5000
end Basic_types
```

The implementation of the CH4 sensor is a cyclic object:

```
class Ch4_Sensor_1
-- refines Ch4_Sensor
is subclass of Basic_types
time variables
  input ch4dbr :  Ch4_reading
instance variables
  ch4_present :  Ch4_reading;
  ch4_status :  Ch4_status;
  ch4status :  @Ch4Status;
  pump_controller :  @PumpController;
  operator_console :  @OperatorConsole;
  data_logger :  @DataLogger;
init objectstate ==
  (ch4_present  :=  0;
   ch4_status  :=  < motor_safe >)
methods
 opcs_periodic_code()  ==
   (ch4_present  :=  ch4dbr;
    ch4_status  :=  ch4status!read();
    if $\mathbf{ch4\_present} \geq \mathbf{ch4\_high}$
    then
      (if $\mathbf{ch4\_status} = < \mathbf{motor\_safe} >$
       then
         ($\mathbf{pump\_controller!not\_safe}()$;
          $\mathbf{operator\_console!set\_alarm}(< \mathbf{high\_methane} >)$;
          $\mathbf{ch4status!write}(< \mathbf{motor\_unsafe} >)$ ) )
    else
      if ($\mathbf{ch4\_present} \leq \mathbf{ch4\_high} - \mathbf{jitter\_range}$) $\wedge$
          $\mathbf{ch4\_status} = < \mathbf{motor\_unsafe} >$
      then
        ($\mathbf{pump\_controller!safe}()$;
         $\mathbf{ch4status!write}(< \mathbf{motor\_safe} >)$);
    $\mathbf{data\_logger!ch4\_status}(\mathbf{ch4\_status})$ )
thread
  $\mathbf{periodic}(\mathbf{ch4\_sensor\_period})(\mathbf{opcs\_periodic\_code})$
aux reasoning
  $\forall\ \mathbf{i} : \mathbb{N}_1 \cdot$
      $\downarrow(\mathbf{opcs\_periodic\_code}, \mathbf{i}) \leq (\mathbf{i} - 1) * \mathbf{ch4\_sensor\_period} + 1000$
end $\mathbf{Ch4\_Sensor\_1}$
```

The deadline (1000ms) of the **opcs_periodic_code** operation is expressed in the aux reasoning section of the class.

The refinement relation of this object compared to its specification is that **ch4dbr** implements **ch4_reading**, and that **ch4_present** is a sampled copy of this with period 5 seconds and deadline 1 second:

$$\mathbf{sampled}(\mathbf{ch4\_present}, \mathbf{ch4\_reading}, 5000, 1000)$$

We can recast this as an approximate function-based refinement as in Section 4.2.

In the above class the **motor_on** output time variable has been implemented by the internal state **pump_controller.motor_on**. Notice that we use the *abstract specification* of this class within the declarations of **Ch4_Sensor_1**, this is valid because any refinement of **PumpController** must be polymorphically compatible with its specification, and must provide some expression that implements **motor_on**. Likewise, **operator_console.alarm** is a sampled implementation of the abstract attribute **alarm**.

Ch4Status is already in a form that can be directly implemented as a protected (mutex) object.

The high/low water sensor object has the abstract specification:

```
class $\mathbf{HighLowWater\_Sensor}$
  is subclass of $\mathbf{Basic\_types}$
time variables
  input $\mathbf{low\_sensor} : \mathbf{bool}$;
  input $\mathbf{high\_sensor} : \mathbf{bool}$;
  input $\mathbf{ch4\_reading} : \mathbb{N}$;
        $\mathbf{motor\_on} : \mathbf{bool}$;
```

```
assumption low_sensor, high_sensor ==
                   ¬ (low_sensor ∧ high_sensor);

effect high_sensor, ch4_reading, motor_on ==
  whenever high_sensor ∧ (ch4_reading ≤ ch4_high − jitter_range)
  also from 20000 ==> motor_on;

effect low_sensor, motor_on ==
  whenever low_sensor
  also from 20000 ==> ¬ (motor_on);
end HighLowWater_Sensor
```

It is refined using the "sporadic constraints refinement" strategy:

```
class HighLowWater_Sensor_1
  is subclass of Basic_types
instance variables
  motor : @Motor
methods
  low_sensor_interrupt() == skip;

  high_sensor_interrupt() == skip
thread
  while true
  do
    sel
      answer high_sensor_interrupt -> motor!operate(),
      answer low_sensor_interrupt -> motor!turn_off()
end HighLowWater_Sensor_1
```

The **operate** method of **Motor** attempts to switch the motor on, checking first that the methane level is safe, using the **check_safe** method. Duration constraints are that the period of 100000ms that is the minimum inter-arrival time between these interrupts must be greater than the maximum duration of the select body, as described in Section 4.4 above. Likewise, this duration must be less than the deadline of 20000ms in the abstract requirements.

The structure of part of the development is shown in Figure 4. **Basic_types** is not shown, for simplicity. Notice that the inheritance of **EnvironmentMonitor** into **Model_1** must hide the method **check_safe** as this method does not appear in the external interface of the system. Likewise the operations of **Ch4Status** should be hidden in **EnvironmentMonitor_1**. The latter class is *hybrid*, as it contains both time variables and ordinary discrete variables. The refinement of **EnvironmentMonitor** by **EnvironmentMonitor_1** involves a *discrete sampling* refinement step on the **environment_status** variable with respect to the expression which tests if the motor is safe to operate.

The refinement from **Ch4_Sensor** to **Ch4_Sensor_1** is a *periodic constraint* refinement step, and also introduces discrete sampling.

The refinement of the high/low water sensor is a *sporadic constraint* refinement step.

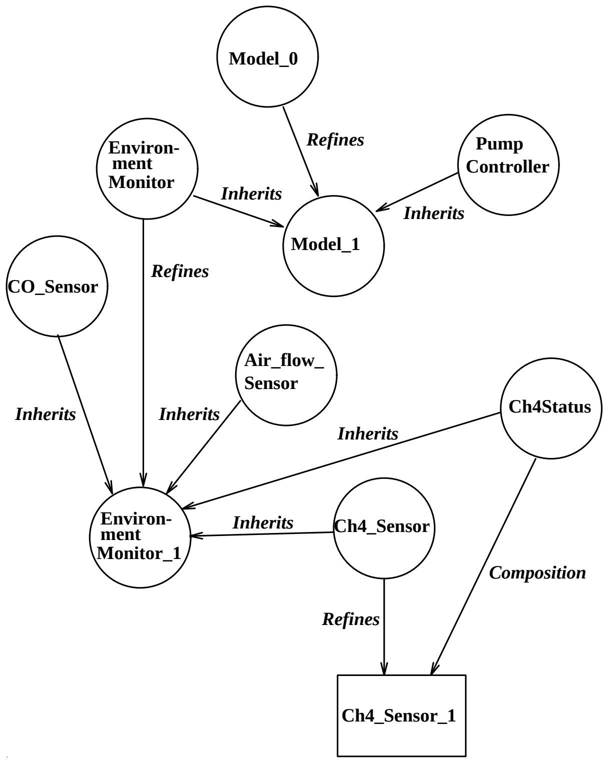

Fig. 4. Partial Structure of Mine System Development

In a complete development we would also refine the other sensors and classes to implementation-level descriptions.

5 Hybrid Systems

The analysis and verification of hybrid systems – systems containing both continuous and discrete aspects – is currently one of the most challenging areas in process control [3, 9]. We can extend the RAL formalism to treat some hybrid systems by adopting the model of *phase transition systems* from [19]. In this model, a hybrid system passes through a set of *phases*, which generalise the concept of a state in a statechart. Within each phase, time variables can modify their value in accordance with a set of differential/integral calculus equations. Transitions between phases occur as a result of some critical exit condition being reached.

We can relate this to RAL by defining a new form of actions termed *phases*:

$$
\begin{array}{l}
\alpha \; = \\
\quad \mathbf{wr}\ \ell \\
\quad \mathbf{pre\ G} \\
\quad \mathbf{do}\ \theta \\
\quad \mathbf{exit\ E} \\
\quad \mathbf{post\ Q}
\end{array}
$$

These have the intended meaning that $\mathbf{G}$ always holds at initiation of α:

$$\forall\, \mathbf{i} : \mathbb{N}_1 \;\cdot\; \mathbf{G} \circledcirc \uparrow(\alpha, \mathbf{i})$$

$\mathbf{Q}$ holds at termination of α: $\forall\, \mathbf{i} : \mathbb{N}_1 \;\cdot\; \mathbf{Q} \circledcirc \downarrow(\alpha, \mathbf{i})$ and that the phase equations θ hold during each execution of α:

$$
\begin{array}{l}
\forall\, \mathbf{i} : \mathbb{N}_1;\ \mathbf{t} : \mathbf{TIME} \cdot \\
\qquad \uparrow(\alpha, \mathbf{i}) \;\leq\; \mathbf{t} \;<\; \downarrow(\alpha, \mathbf{i}) \;\Rightarrow\; \theta \circledcirc \mathbf{t}
\end{array}
$$

Hooked attributes $\overleftarrow{\mathbf{var}}$ occurring in θ refer to the value of $\mathbf{var}$ at $\uparrow(\alpha, \mathbf{i})$.

We assume that **TIME** is the set of non-negative real numbers in the following.

The **exit** condition controls when α terminates – ie, at the first point in each execution interval where $\mathbf{E}$ becomes true:

$$
\begin{array}{l}
\forall\, \mathbf{i} : \mathbb{N}_1;\ \mathbf{t} : \mathbf{TIME} \cdot \\
\qquad \uparrow(\alpha, \mathbf{i}) \;\leq\; \mathbf{t} \;<\downarrow(\alpha, \mathbf{i}) \;\Rightarrow\; \neg\, \mathbf{E} \circledcirc \mathbf{t} \;\wedge \\
\qquad \mathbf{E} \circledcirc \downarrow(\alpha, \mathbf{i})
\end{array}
$$

Thus for example, in the "mouse and cat" example of [19], we can represent the activity of the mouse running, terminated by the **safe** state by:

$$
\begin{array}{l}
\mathbf{mrun_{norm}} \; = \\
\quad \mathbf{wr}\ \mathbf{x_m}, \mathbf{mstate} \\
\quad \mathbf{pre\ mstate} \; = < \mathbf{running} > \\
\quad \mathbf{do}\ \frac{\mathbf{dx_m}}{\mathbf{dt}} \; = \; -\mathbf{v_m} \\
\quad \mathbf{exit}\ \mathbf{x_m} \; = \; 0 \\
\quad \mathbf{post\ mstate} \; = < \mathbf{safe} >
\end{array}
$$

The mouse velocity is $\mathbf{v_m}$ and its distance to the hole is $\mathbf{x_m}$.

A similar action is used to represent the cat running and achieving the "mouse caught" state:

$$
\begin{array}{l}
\mathbf{crun_{norm}} \; = \\
\quad \mathbf{wr}\ \mathbf{x_c}, \mathbf{cstate} \\
\quad \mathbf{pre\ cstate} \; = < \mathbf{running} > \\
\quad \mathbf{do}\ \frac{\mathbf{dx_c}}{\mathbf{dt}} \; = \; -\mathbf{v_c} \\
\quad \mathbf{exit}\ \mathbf{x_c} \; = \; \mathbf{x_m} \;\wedge\; \mathbf{x_c} \; > \; 0 \\
\quad \mathbf{post\ cstate} \; = < \mathbf{mouse\_caught} >
\end{array}
$$

A "running" activity of the complete system is then a parallel combination of these two activities. It is not a simple $||$ combination however, because the cat achieving the "mouse caught" state must "pre-empt" the mouse running action. Likewise, the mouse achieving the "safe" state before the cat reaches it must pre-empt the cat running action.

In order to represent the abortion or pre-emption of an action, we use the choice combinator $\sqcap$:

$$\begin{array}{l} \forall \mathbf{i} : \mathbb{N}_1 \cdot \\ \qquad \exists \mathbf{j} : \mathbb{N}_1 \cdot \\ \qquad\qquad (\uparrow(\alpha,\mathbf{i}) \ = \ \uparrow(\beta,\mathbf{j}) \ \wedge \ \downarrow(\alpha,\mathbf{i}) \ = \ \downarrow(\beta,\mathbf{j})) \ \vee \\ \qquad\qquad (\uparrow(\alpha,\mathbf{i}) \ = \ \uparrow(\gamma,\mathbf{j}) \ \wedge \ \downarrow(\alpha,\mathbf{i}) \ = \ \downarrow(\gamma,\mathbf{j})) \end{array}$$

where $\alpha \ = \ \beta \ \sqcap \ \gamma$.

If we want α to be capable of being aborted, ie, to terminate without its normal postcondition holding, then we define a normal behaviour action $\alpha_{\mathbf{norm}}$ and an action for abnormal behaviour cases $\alpha_{\mathbf{abort}}$.

α itself is defined as the $\sqcap$ combination of these:

$$\alpha \ = \ \alpha_{\mathbf{norm}} \ \sqcap \ \alpha_{\mathbf{abort}}$$

For example, if we consider the cat and mouse problem, the activity of the mouse running has the abort termination:

$$\begin{array}{l} \mathbf{mrun_{abort}} \ = \\ \qquad \mathbf{wr} \ \mathbf{x_m}, \mathbf{mstate} \\ \qquad \mathbf{pre} \ \mathbf{mstate} \ = < \mathbf{running} > \\ \qquad \mathbf{do} \ \frac{\mathbf{dx_m}}{\mathbf{dt}} \ = \ -\mathbf{v_m} \\ \qquad \mathbf{exit} \ \mathbf{x_m} = \mathbf{x_c} \wedge \mathbf{x_m} > 0 \\ \qquad \mathbf{post} \ \mathbf{mstate} \ = < \mathbf{caught} > \end{array}$$

Similarly the cat running has an abnormal termination:

$$\begin{array}{l} \mathbf{crun_{abort}} \ = \\ \qquad \mathbf{wr} \ \mathbf{x_c}, \mathbf{cstate} \\ \qquad \mathbf{pre} \ \mathbf{cstate} \ = < \mathbf{running} > \\ \qquad \mathbf{do} \ \frac{\mathbf{dx_c}}{\mathbf{dt}} \ = \ -\mathbf{v_c} \\ \qquad \mathbf{exit} \ \mathbf{x_c} = 0 \\ \qquad \mathbf{post} \ \mathbf{cstate} \ = < \mathbf{failed} > \end{array}$$

This means that the composed action can be combined with the cat running action using the pre-emption operator $\lightning$:

$$\begin{array}{l} \forall \mathbf{i} : \mathbb{N}_1 \cdot \\ \qquad \exists \mathbf{j}, \mathbf{k} : \mathbb{N}_1 \cdot \\ \qquad\qquad \uparrow(\alpha \lightning \beta,\mathbf{i}) \ = \ \uparrow(\alpha,\mathbf{j}) \ = \ \uparrow(\beta,\mathbf{k}) \ \wedge \\ \qquad\qquad \downarrow(\alpha \lightning \beta,\mathbf{i}) \ = \ \downarrow(\alpha,\mathbf{j}) \ = \ \downarrow(\beta,\mathbf{k}) \end{array}$$

The overall system action is then:

$$\mathbf{running} \ = \ (\mathbf{mrun} \lightning (\mathbf{wait}(\delta);\ \mathbf{crun}))$$

At termination of an instance $(\mathbf{running}, \mathbf{i})$ with duration greater than δ we know that both the **mrun** and **crun** actions have terminated. If **mrun** has terminated normally, ie, the instance of **mrun** is an instance of $\mathbf{mrun_{norm}}$, then we know that $\mathbf{mstate} \ = < \mathbf{safe} >$ at $\downarrow(\mathbf{running}, \mathbf{i})$, and that $\mathbf{x_m} \ = \ 0$, and hence, that the **crun** action has terminated abnormally: the instance of **crun** involved is an instance of $\mathbf{crun_{abort}}$.

Conversely, if the instance of **crun** is an instance of $\mathbf{crun_{norm}}$ then $\mathbf{cstate} \ = \ < \mathbf{mouse\_caught} >$ at $\downarrow(\mathbf{running}, \mathbf{i})$, and that $\mathbf{x_m} \ = \ \mathbf{x_c} \neq 0$, so that the instance of **mrun** involved must be an instance of $\mathbf{mrun_{abort}}$.

It is not possible for both actions to fail – in that case $\mathbf{x_m} \ = \ \mathbf{x_c} > 0$ and $\mathbf{x_c} = 0$, a contradiction.

Discretisation of this problem will require sampling of the $\mathbf{v_c}$ and $\mathbf{v_m}$ variables, and calculation of the $\mathbf{x_c}$ and $\mathbf{x_m}$ variables. We must choose a fine enough granularity of sampling so that the point where the cat catches the mouse is not missed. This requires some tolerance $\mid \mathbf{x_m} - \mathbf{x_c} \mid < \ \epsilon$ in the distances as the criterion for "catches", instead of equality. The sampling period τ must then satisfy

$$\tau \ < \ \frac{2 * \epsilon}{\mathbf{v_c} - \mathbf{v_m}}$$

6 Conclusions

We have described some techniques for combining VDM^{++} with HRT-HOOD, and how real-time refinement can be formalised in RAL. All the forms of constraint described in [8] can be expressed in RAL, except required internal non-determinism. External required non-determinism (the capability to respond to several different messages) is expressed via the **enabled** predicate.

The formalism possesses a sound semantics, and it is therefore consistent relative to ZF set theory. The advantage of the formalism over other real-time and concurrency formalisms is the conciseness of the core syntax and axiomatisation, and its ability to express the full range of reactive and real-time system behaviour via derived constructs. The TAM formalism of [18] can be regarded as a subset of RAL, and could be used to transform specification and code fragments that are purely local to one class and that are within its language. For practical development, we also need higher-level design transformations such as design patterns, and a systematic, tool-supported combination of formal and diagrammatic notations.

We have argued that the concept of approximate (functional) refinement is preferable to the use of relational refinement in carrying out the step from a continuous or hybrid specification of a system to a discrete specification. This is because it provides a simpler formulation of refinement in terms of theory extension, and enables us to measure the degree to which information about the continuous world can be recovered from the discrete refinement.

Animation of VDM^{++} specifications can be performed at the abstract continuous description level, using tools such as gPROMS [2], in order to validate the formal model of the real-world situation expressed in terms of predicates and time variables. This is in contrast to implementation-level simulation as described in [4], which is in terms of threads and processes and may be unconnected to the real-world model. Tool support for proof obligation generation for internal consistency, refinement and subtyping obligations, and for animation of event sequences against VDM^{++} classes is being developed in the "Object-oriented Specification of Reactive and Real-time Systems" project.

Examples of using the logic to express properties of distributed and concurrent systems can be found in the papers [13, 14]. Similar techniques could be applied using the formal language Z^{++}, although VDM^{++} is more suited to the later design and implementation stages.

References

1. M Awad, J Kuusela, and Jurgen Ziegler. *Object-oriented Technology for Real-time Systems.* Prentice Hall, 1996.
2. P I Barton, E Smith and C C Pantelides. Combined Discrete/Continuous Process Modelling Using gPROMS, 1991 AIChE Annual Meeting: Recent Advances in Process Control, Los Angeles, 1991.
3. P Barton and T Park. *Analysis and Control of Combined Discrete/Continuous Systems: Progress and Challenges in the Chemical Processing Industries*, in proceedings of *Chemical Process Control - V: Assessment and New Directions for Research*, January, 1996.
4. A Burns and A Wellings. HRT-HOOD: A structured design method for hard real-time systems. *Real-Time Systems*, 6(1):73–114, January 1994.
5. D Coleman, P Arnold, S Bodoff, C Dollin, H Gilchrist, F Hayes, and P Jeremaes. *Object-oriented Development: The FUSION Method.* Prentice Hall Object-oriented Series, 1994.
6. S Cook and J Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, Sept 1994.
7. E Durr, S Goldsack, and J van Katjwick. Specification of a cruise controller in VDM^{++}. In *Proceedings of Real Time OO Workshop, ECOOP 96*, 1996.
8. S M Celiktin. Interval-Based Techniques for the Specification and Analysis of Real-Time Requirements, PhD thesis, Catholic University of Louvain, September 1994.
9. S Engell and S Kowalewski. *Discrete Events and Hybrid Systems in Process Control*, Proceedings of *Chemical Process Control - V: Assessment and New Directions for Research*, January, 1996.
10. J Fiadeiro and T Maibaum. *Describing, Structuring and Implementing Objects*, in de Bakker *et al.*, *Foundations of Object Oriented languages*, LNCS 489, Springer-Verlag, 1991.
11. F Jahanian and A K Mok. Safety Analysis of Timing Properties in Real-time Systems, *IEEE Transactions on Software Engineering*, SE-12, pp. 890–904, September 1986.

12. S Kent and K Lano. *Axiomatic Semantics for Concurrent Object Systems*, AFRODITE Technical Report AFRO/IC/SKKL/SEM/V1, Dept. of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ.
13. K Lano. *Distributed System Specification in VDM^{++}*, FORTE '95 Proceedings, Chapman and Hall, 1995.
14. K Lano, J Bicarregui and S Kent. *A Real-time Action Logic of Objects*, ECOOP 96 Workshop on Proof Theory of Object-oriented Systems, Linz, Austria, 1996.
15. K Lano. *Semantics of Real-Time Action Logic*, Technical Report GR/K68783-3, Dept. of Computing, Imperial College, 1996.
16. K Lano, S Goldsack and A Sanchez. *Transforming Continuous into Discrete Specifications with VDM^{++}*, IEE C8 Colloquium Digest on Hybrid Control for real-time Systems, 1996.
17. K Lano. *Refinement and Simulation of Real-time and Hybrid Systems using VDM^{++} and gPROMS*, ROOS project report GR/K68783-13, November 1996, Dept. of Computing, Imperial College.
18. G Lowe and H Zedan. Refinement of complex systems: A case study. *The Computer Journal*, 38(10):785–800, 1995.
19. Z Manna and A Pnueli. Time for concurrency. Technical report, Dept. of Computer Science, Stanford University, 1992.
20. B Mahony and I J Hayes. Using continuous real functions to model timed histories. In P A Bailes, editor, *Proceedings of 6th Australian Software Engineering Conference*. Australian Computer Society, July 1991.
21. J S Ostroff. *Temporal Logic for Real-Time Systems.* John Wiley, 1989.
22. A Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J de Bakker, W P de Roever, and G Rozenberg, editors, *Current Trends in Concurrency*, LNCS vol. 224, Springer-Verlag, 1986.

Real-time Action Logic

Logic

RAL is an extension of the Object Calculus of Fiadeiro and Maibaum [10] to cover durative actions and real-time constraints. The syntactic elements of an RAL theory are: *action symbols*, *attribute symbols*, plus the usual type, function and predicate symbols of typed predicate calculus, including the operators $\in$, set comprehension, $\cup$, $\mathbb{F}$, etc, of ZF set theory. These aspects are as for the standard object calculus.

For each action α, there are function symbols $\rightarrow(\alpha, \mathbf{i})$ the time of request of the $\mathbf{i}$-th invocation of action α, $\uparrow(\alpha, \mathbf{i})$ the time of activation of this invocation, and $\downarrow(\alpha, \mathbf{i})$ the time of termination of this invocation. $\mathbf{i}$ ranges over $\mathbb{N}_1$.

Modal operators are $\odot$ "holds at a time" and $\circledast$ "value at a time".

The type **TIME** is assumed to be totally ordered by a relation $<$, with a least element 0, and with $\mathbb{N} \subseteq \mathbf{TIME}$. It satisfies the axioms of the set of non-negative elements of a totally ordered ring, with addition operation $+$ and unit 0, and multiplication operation $*$ with unit 1.

The following operators can be defined in terms of the above symbols: (i) the modal action formulae $[\alpha]\mathbf{P}$ "α establishes $\mathbf{P}$". $\mathbf{P}$ may contain references

$\overleftarrow{\mathbf{e}}$ to the value of $\mathbf{e}$ at commencement of the invocation of α being considered; (ii) the operator $\supset$ representing the calling relation between two actions; (iii) the RTL [11] event-time operators $\clubsuit(\varphi := \mathbf{true}, \mathbf{i})$ and $\clubsuit(\varphi := \mathbf{false}, \mathbf{i})$ giving the times of the $\mathbf{i}$-th occurrences of the events of a predicate φ becoming true or false, respectively; (iv) counters $\#\mathbf{req}(\alpha)$, $\#\mathbf{act}(\alpha)$ and $\#\mathbf{fin}(\alpha)$ for request, activation and termination events; (v) the temporal logic operators $\Box$, $\diamond$, $\bigcirc$; (vi) action combinators ; , || (parallel non-interfering execution), assignment, etc.

Specific to the object-oriented view are types $@\mathbf{Any}$ of all possible object identifiers, and subsorts $@\mathbf{C}$ of this type which represent the possible object identifiers of objects of class $\mathbf{C}$.

A predicate added for concurrent object-oriented systems is a test for *enabling* of an action α (whether a request for execution of α will be serviced or not). This is expressed by $\mathbf{enabled}(\alpha)$.

Attributes and Actions For a specification $\mathbf{S}$ consisting of a set of classes, the attribute symbols are as follows: $\mathbf{x.att}$ for $\mathbf{x} : @\mathbf{C}$ and $\mathbf{att}$ an instance or time variable of a class $\mathbf{C}$ of $\mathbf{S}$. The attribute $\overline{\mathbf{C}}$ for each class $\mathbf{C}$ represents the set of *existing* objects of $\mathbf{C}$. This is of type $\mathbb{F}(@\mathbf{C})$.

Derived attributes of a class will include *event counters* $\#\mathbf{act}(\mathbf{m})$, $\#\mathbf{fin}(\mathbf{m})$ as defined below.

The action symbols are: $\mathbf{new}_{\mathbf{C}}(\mathbf{c})$ for $\mathbf{C}$ a class of $\mathbf{S}$ and $\mathbf{c} : @\mathbf{C}$; $\mathbf{x}!\mathbf{m}(\mathbf{e})$ for $\mathbf{x} : @\mathbf{C}$ and $\mathbf{m}$ a method of $\mathbf{C}$, with $\mathbf{e} : \mathbf{X}_{\mathbf{m},\mathbf{C}}$ a term in the type of the input parameters of $\mathbf{m}$ in $\mathbf{C}$.

pre Guard post Post where **Guard** is an expression over a set of attributes, and **Post** can additionally contain expressions of the form $\overleftarrow{\mathbf{e}}$ referring to the value of the expression $\mathbf{e}$ at commencement of execution of the action.

We write $\mathbf{x}.{\uparrow}(\mathbf{m}(\mathbf{e}), \mathbf{i})$ for ${\uparrow}(\mathbf{x}!\mathbf{m}(\mathbf{e}), \mathbf{i})$ etc to make the notation used for objects more uniform.

Derived Actions and Attributes For an object $\mathbf{x} : @\mathbf{C}$ event occurrence times $\clubsuit(\varphi := \mathbf{true}, \mathbf{i})$ and $\clubsuit(\varphi := \mathbf{false}, \mathbf{i})$ can be defined from the above language.

Event counters are also derived operators:

$$\begin{aligned}&\mathbf{x}.\#\mathbf{act}(\mathbf{m}(\mathbf{e})) = \\ &\qquad \mathbf{card}(\{\mathbf{j} : \mathbb{N}_1 \mid \mathbf{x}.{\uparrow}(\mathbf{m}(\mathbf{e}), \mathbf{j}) < \mathbf{now}\})\end{aligned}$$

This definition involves $<$ because we consider $\#\mathbf{act}(\mathbf{m})$ to be incremented indivisibly *just after* the moment at which $\mathbf{m}$ initiates execution. Similarly we can define $\mathbf{x}.\#\mathbf{req}(\mathbf{m}(\mathbf{e}))$ and $\mathbf{x}.\#\mathbf{fin}(\mathbf{m}(\mathbf{e}))$.

The actions **pre G post P** name actions α with the following properties:

$$\forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{now} = {\uparrow}(\alpha, \mathbf{i}) \;\Rightarrow\; \mathbf{G} \circledcirc {\uparrow}(\alpha, \mathbf{i})$$
$$\forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{now} = {\uparrow}(\alpha, \mathbf{i}) \;\Rightarrow\; \mathbf{P}[\mathbf{att} \circledast {\uparrow}(\alpha, \mathbf{i}) / \overleftarrow{\mathbf{att}}] \circledcirc {\downarrow}(\alpha, \mathbf{i})$$

Formulae $\Box_{\mathbf{a},\mathbf{C}}\phi$ denotes that ϕ holds at each future initiation time of a method invocation $\mathbf{a!m}$ on an object $\mathbf{a} : @\mathbf{C}$, where $\mathbf{m}$ is a method of the class $\mathbf{C}$. In other words it abbreviates

$$\begin{array}{l}\forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{a}.\uparrow(\mathbf{m}_1, \mathbf{i}) \geq \mathbf{now} \Rightarrow \phi \circledcirc \mathbf{a}.\uparrow(\mathbf{m}_1, \mathbf{i}) \\ \wedge \ldots \ \wedge \\ \forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{a}.\uparrow(\mathbf{m_n}, \mathbf{i}) \geq \mathbf{now} \Rightarrow \phi \circledcirc \mathbf{a}.\uparrow(\mathbf{m_n}, \mathbf{i})\end{array}$$

where $\underline{\mathbf{methods}}(\mathbf{C}) = \{\mathbf{m}_1, \ldots, \mathbf{m_n}\}$.

The calling operator $\supset$ is defined by:

$$\begin{array}{l}\alpha \supset \beta \ \equiv \\ \quad \forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{now} = \uparrow(\alpha, \mathbf{i}) \ \Rightarrow \\ \qquad \exists \mathbf{j} : \mathbb{N}_1 \cdot \uparrow(\beta, \mathbf{j}) = \uparrow(\alpha, \mathbf{i}) \ \wedge \ \downarrow(\beta, \mathbf{j}) = \downarrow(\alpha, \mathbf{i})\end{array}$$

In other words: every invocation interval of α is also one of β.

The MAL operator $[\alpha]\mathbf{P}$ is defined as:

$$\begin{array}{l}[\alpha]\mathbf{P} \ \equiv \\ \quad \forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{now} = \uparrow(\alpha, \mathbf{i}) \ \Rightarrow \ \mathbf{P}[\mathbf{att} \circledast \uparrow(\alpha, \mathbf{i}) / \overleftarrow{\mathbf{att}}] \circledcirc \downarrow(\alpha, \mathbf{i})\end{array}$$

where the same substitution is used as for the definition of **pre G post P** above.

We can then show $[\mathbf{pre\,G \ \ post\,P}](\overleftarrow{\mathbf{G}} \wedge \mathbf{P})$ and that

$$(\alpha \supset \beta) \ \Rightarrow \ ([\beta]\mathbf{P} \Rightarrow [\alpha]\mathbf{P})$$

for any $\mathbf{P}$ in the language concerned.

Conditionals have the expected properties:

$$\begin{array}{l}\mathbf{E} \ \Rightarrow \ (\mathbf{if\ E\ then\ S_1\ else\ S_2} \quad \supset \quad \mathbf{S}_1) \\ \neg\,\mathbf{E} \ \Rightarrow \ (\mathbf{if\ E\ then\ S_1\ else\ S_2} \quad \supset \quad \mathbf{S}_2)\end{array}$$

Similarly, **while** loops can be defined.

A synchronous method invocation $\mathbf{a!m(e)}$ is interpreted as an **invoke** statement:

$$\mathbf{invoke \ \ a!m(e)}$$

An instance $(\mathbf{S}, \mathbf{i})$ of this statement has the properties:

$$\begin{array}{l}\forall \mathbf{i} : \mathbb{N}_1 \cdot \exists \mathbf{j} : \mathbb{N}_1 \cdot \\ \qquad \uparrow(\mathbf{S}, \mathbf{i}) \ = \ \mathbf{a}.\rightarrow(\mathbf{m(e)}, \mathbf{j}) \ \wedge \\ \qquad \downarrow(\mathbf{S}, \mathbf{i}) \ = \ \mathbf{a}.\downarrow(\mathbf{m(e)}, \mathbf{j})\end{array}$$

Axioms The axioms of predicate calculus and ZF set theory are adopted, with some modifications.

The core logical axioms include:

$$(\mathbf{C}1): \ \forall \mathbf{i} : \mathbb{N}_1 \cdot \rightarrow(\mathbf{m(e)}, \mathbf{i}) \ \leq \ \rightarrow(\mathbf{m(e)}, \mathbf{i} + 1)$$

"the $\rightharpoondown(\mathbf{m(e), i})$ times are enumerated in order of their occurrence."

$$(\mathbf{C}2) : \forall \mathbf{i} : \mathbb{N}_1 \cdot \rightharpoondown(\mathbf{m(e),i}) \leq \uparrow(\mathbf{m(e),i}) < \downarrow(\mathbf{m(e),i})$$

"every invocation must be requested before it can initiate, and initiates before it terminates."

The *compactness* condition is that for every $\mathbf{p} \in \mathbb{N}_1$ there are only finitely many values $\uparrow(\alpha, \mathbf{i}) < \mathbf{p}$, for each action α. Similar conditions are required for the $\rightharpoondown$ and $\downarrow$ times.

Of key importance for reasoning about objects is a *framing* or *locality* constraint [10], which asserts that over any interval in which no action executes, no attribute representing an instance variable changes in value.

This locality principle reduces to that of the object calculus in the case that all actions have duration 1 and $\mathbf{TIME} = \mathbb{N}$.

The usual inference rules of predicate logic are taken. In addition the following rule is adopted:

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall \mathbf{t} : \mathbf{TIME} \cdot \varphi \circledcirc \mathbf{t}}$$

Interpretations of Class Features

The theory $\Gamma_{\mathbf{S}}$ of a system is the union of the theories $\Gamma_{\mathbf{C}}$ of the separate classes within it, which are defined as follows.

If we have a method definition in class $\mathbf{C}$ of the form:

```
m(x : X_m,C) value y : Y_m,C
    pre Pre_m,C == Code_m,C ;
```

then the action $\mathbf{a!m(e)}$ has the properties:

$$\mathbf{a.Pre_{m,C}[e/x]} \wedge \mathbf{a} \in \overline{\mathbf{C}} \Rightarrow$$
$$\mathbf{a!m(e)} \supset \mathbf{a.Code_{m,C}[e/x]}$$

where each attribute $\mathbf{att}$ of $\mathbf{C}$ occurring in $\mathbf{Pre_{m,C}}$ is renamed to $\mathbf{a.att}$ in $\mathbf{a.Pre_{m,C}}$ and similarly for $\mathbf{Code_{m,C}}$. Additionally, invocations of actions $\mathbf{b!n(f)}$ within $\mathbf{Code}$ are explicitly written as $\mathbf{invoke\ b!n(f)}$ statements.

The initialisation of a class $\mathbf{C}$ can be regarded as a method $\mathbf{init_C}$ which is called automatically when an object $\mathbf{c}$ is created by the action $\mathbf{new_C}$: $\mathbf{new_C(c)} \supset \mathbf{c!init_C}$.

$\mathbf{new_C}$ itself has the property: $\mathbf{c} \notin \overline{\mathbf{C}} \Rightarrow [\mathbf{new_C(c)}](\overline{\mathbf{C}} = \overleftarrow{\overline{\mathbf{C}}} \cup \{\mathbf{c}\})$.

A method must be enabled when it initiates execution:

$$\forall \mathbf{x} : @\mathbf{C};\ \mathbf{i} : \mathbb{N}_1;\ \mathbf{e} : \mathbf{X_{m,C}} \cdot$$
$$\mathbf{enabled(x!m(e))} \circledcirc \mathbf{x}.\uparrow(\mathbf{m(e),i})$$

for all methods $\mathbf{m}$ of $\mathbf{C}$.

The invariant of a class is true at every method initiation and termination time: $\Box_{\mathbf{a},\mathbf{C}}\mathbf{Inv_C} \wedge \forall \mathbf{i} : \mathbb{N}_1 \cdot \mathbf{Inv_C} \circledcirc \mathbf{a}.\downarrow(\mathbf{m_j}, \mathbf{i})$ for each method $\mathbf{m_j}$ of $\mathbf{C}$ and $\mathbf{a}$: $@\mathbf{C}$. However, the typing constraints for attributes are *always* true: $\Box^{\tau}(\mathbf{a.att} \in \mathbf{T})$ for each attribute declaration $\mathbf{att} : \mathbf{T}$ of $\mathbf{C}$.

Permission guards for a method $\mathbf{m}$ give conditions which must be implied by $\mathbf{enabled(m)}$:

$$\texttt{per}\ \mathbf{m} \quad \Rightarrow \quad \mathbf{G}$$

yields the axiom $\mathbf{enabled(m)} \Rightarrow \mathbf{G}$.

The `whenever` construct of VDM^{++} is interpreted as follows. A statement

`whenever` χ `also from` δ `==>` φ

asserts that φ must be true at some point in each interval of the form $[\mathbf{t}, \mathbf{t} + \delta]$ where $\mathbf{t}$ is a time at which χ becomes true.

Thus it can be expressed directly as:

$$\forall \mathbf{i} : \mathbb{N}_1;\ \exists \mathbf{t} : \mathbf{TIME} \cdot \varphi \circledcirc \mathbf{t} \ \wedge \\ \clubsuit(\chi := \mathbf{true}, \mathbf{i}) \leq \mathbf{t} \leq \clubsuit(\chi := \mathbf{true}, \mathbf{i}) + \delta$$

This definition yields a transitivity principle.

We can extend this interpretation to classes involving time variables, provided that we restrict $\mathbf{TIME}$ to be the set of non-negative real numbers. The representation of `assumption` and `effect` clauses then uses *phase actions*, which have ongoing activities terminated by critical conditions. These activities only change a certain subset of the time variables (ie, for each assumption clause, the variables listed in the header, and for effect clauses, those output variables listed in the clause header).

In detail, for each `assumption` clause

$$\texttt{assumption}\ \mathbf{it}_1,\ \ldots,\ \mathbf{it_p} \quad == \quad \mathbf{A}(\mathbf{it}_1,\ \ldots,\ \mathbf{it_p})$$

we have a phase action

$$\begin{array}{l} \mathbf{wr}\ \mathbf{it}_1, \ldots, \underline{\mathbf{it_p}} \\ \mathbf{pre\ self} \in \overline{\mathbf{C}} \\ \mathbf{do}\ \mathbf{A}(\mathbf{it}_1, \ldots, \mathbf{it_p}) \\ \mathbf{exit\ self} \notin \overline{\mathbf{C}} \\ \mathbf{post\ true} \end{array}$$

which continues for the lifetime of the current object, allows only $\mathbf{it}_1, \ldots, \mathbf{it_p}$ to change, and requires that their changes obey the formula $\mathbf{A}$ at all times in this lifetime.

If an input time variable $\mathbf{it}$ does not appear in an `assumption` clause, then there is a default action for $\mathbf{it}$ with $\mathbf{A}$ being $\mathbf{true}$. Similarly for output time variables without effect clauses. The $\mathbf{now}$ attribute is treated in this way, except that its activity clause is $\frac{\mathbf{d\ now}}{\mathbf{dt}} = 1$.

Likewise, an `effect` clause

`effect` $\mathbf{it}_1, \ldots, \mathbf{it_p}, \mathbf{ot}_1, \ldots, \mathbf{ot_q} \quad == \mathbf{E}(\mathbf{it}_1, \ldots, \mathbf{it_p}, \mathbf{ot}_1, \ldots, \mathbf{ot_q})$

has an interpretation as an action:

$$\begin{array}{l}
\mathbf{wr\ ot}_1, \ldots, \mathbf{ot_q} \\
\mathbf{pre\ self} \in \overline{\mathbf{C}} \\
\mathbf{do\ E}(\mathbf{it}_1, \ldots, \mathbf{it_p}, \mathbf{ot}_1, \ldots, \mathbf{ot_q}) \\
\mathbf{exit\ self} \notin \overline{\mathbf{C}} \\
\mathbf{post\ true}
\end{array}$$

All of these actions are lifted to be actions at the class level by substitution of particular object references $\mathbf{a} : @\mathbf{C}$ for **self**, and $\mathbf{a.it_i}$ for $\mathbf{it_i}$, etc.

Notice that since these actions execute over the entire lifetime of an object of the class, a more refined concept of locality, involving write frames for actions, is necessary in order to reason about changes to attributes over intervals. More precisely, if an attribute **att** of $\mathbf{a} : @\mathbf{C}$ changes in value between times $\mathbf{t}_1 < \mathbf{t}_2$, then there is some $\mathbf{t} : \mathbf{TIME}$ with $\mathbf{t}_1 \leq \mathbf{t} \leq \mathbf{t}_2$ such that some action **m** of **C** is executing on **a** at **t**, and has **att** in its write frame.

Finally, the formulae listed in the `aux reasoning` part of a class are conjoined together, and lifted to refer to particular objects, in order to obtain their meaning in the class theory.

If class **C** inherits class **D**, the theory of **D** is included in that of **C**, except that methods **m** of **D** defined in both classes are renamed to **D‘m** in the theory of **C**.

Subtyping and Refinement Concepts

Theory Morphisms

The concept of a theory morphism for RAL is similar to that for the object calculus. A morphism $\sigma : \mathbf{Th}1 \rightarrow \mathbf{Th}2$ maps each type symbol **T** of **Th**1 to a type symbol $\sigma(\mathbf{T})$ of **Th**2, each function symbol of **Th**1 to a function symbol of **Th**2, and each attribute of **Th**1 to an attribute of **Th**2. Actions of **Th**1 are mapped to actions of **Th**2.

The type **TIME** is always mapped to itself.

We can construct a category of theories with theory morphisms as categorical arrows as usual. Theory morphisms can be used to decompose the description of a class or system theory into theories for individual objects, and theories of the individual classes.

Refinement

The concepts of subtyping and refinement in VDM^{++} correspond to a particular form of theory morphism. Class **C** is a supertype of class **D** if there is a *retrieve function* $\mathbf{R} : \mathbf{T_D} \rightarrow \mathbf{T_C}$ between the respective states, and a renaming ϕ of methods of $\mathbf{C}$ to those of $\mathbf{D}$, such that for every $\varphi \in \mathcal{L}_{\mathbf{C}}$, $\mathit{\Gamma}_{\mathbf{C}} \vdash \varphi$ implies that $\mathit{\Gamma}_{\mathbf{D}} \vdash \phi(\varphi[\mathbf{R}(\mathbf{v})/\mathbf{u}])$ where $\mathbf{v}$ is the tuple of attributes of $\mathbf{D}$, $\mathbf{u}$ of $\mathbf{C}$.

ϕ must map internal methods of $\mathbf{C}$ to internal methods of $\mathbf{D}$, and external methods to external methods. The notation $\mathbf{C} \sqsubseteq_{\phi,\mathbf{R}} \mathbf{D}$ is used to denote this relation.

$\mathbf{D}$ is a refinement of $\mathbf{C}$ if it is a subtype of $\mathbf{C}$ and the retrieve function $\mathbf{R}$ satisfies the condition of *adequacy*:

$$\begin{aligned} &\forall \mathbf{u} \in \mathbf{T}_{\mathbf{C}} \cdot \mathbf{Inv}_{\mathbf{C}}(\mathbf{u}) \;\Rightarrow \\ &\qquad\qquad \exists \mathbf{v} \in \mathbf{T}_{\mathbf{D}} \cdot \mathbf{Inv}_{\mathbf{D}}(\mathbf{v}) \wedge \mathbf{R}(\mathbf{v}) = \mathbf{u} \end{aligned}$$

That is, $\mathbf{R}$ is onto. In addition, no new external methods can be introduced in $\mathbf{D}$.

Refinement proofs can be decomposed into modular proofs of stronger but more local obligations, such as that preconditions can be weakened and postconditions strengthened, etc.

This article was processed using the LaTeX macro package with LLNCS style