Reactive System Specification and Refinement

K. Lano

Dept. of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ.

Abstract. This paper describes formal approaches for reactive and real time system specification and development, using a process of systematic translation from statechart descriptions of a system into a specification language utilising real time logic (RTL), and refinement within this language. Alternative implementation strategies using synchronisation constraints and synchronisation code are also provided, together with examples of development using the approach.

The approach provides a unitary formalism which combines statecharts, RTL and temporal logic. Animation and proof tools are also briefly described.

1 Introduction

Software applications whose main purpose is to interact in real time with external systems or devices are termed 'reactive systems'. Examples include chemical process control systems, or patient monitoring systems, in which response to external events is a major part of the system requirements, and the timing of these responses is also critical. More generally, many safety critical systems are reactive or real time systems, for which formal specifications are becoming more frequently required by standards. Thus a formal language capable of treating issues of process scheduling, timing, concurrent execution and interrupts, and of supporting reasoning about such aspects at a suitable level of abstraction, is highly desirable.

In this context the use of an object-oriented paradigm is appropriate: objects may correspond to reactive components whose behaviour can be specified in isolation, or with real world concepts or entities, such as particular controlled devices, thus enabling a systematic tracing of requirements through specifications to code to be performed, supporting validation, assessment and maintenance.

The formal specification language Z^{++} [10, 6, 8] will be considered as the basis for a real time specification language. This paper extends the Z^{++} language by allowing the use of RTL formulae in the `HISTORY` clause of a class. In contrast to VDM^{++} [10], this formalism is more orientated towards *discrete event systems* rather than continuous variable dynamic systems [14]. However, the formalism presented here could be used as a unifying semantics for much of VDM^{++}, and was initially defined to serve this purpose.

Section 2 gives the syntax of Z^{++} specifications, together with examples. Section 3 summarises the temporal logic formalism used in the paper. Section 4 describes the development process intended to be used with the language, and summarises the steps in this process. Section 5 introduces a refinement technique using semaphores to implement synchronisation constraints, and Section 6 describes refinement using *synchronisation code* [13].

2 Z++/RTL Syntax

2.1 Classes

A Z++ specification consists of a collection of *class* definitions, including generic classes. A schematic Z++ class is of the form:

```
CLASS C[TypeParameters]
EXTENDS Ancestors
TYPES
  Tdefs
FUNCTIONS
  Axdefs
OWNS
  c
INVARIANT
  Inv_C
OPERATIONS
  [*] m : IN → OUT;
   ....
RETURNS
  r : IN → OUT;
   ....
ACTIONS
   Pre_m,C &
        [*] m x y  ==>  Def_m,C;
   ....
HISTORY H_C
END CLASS
```

Each of the clauses in the body of a class definition are optional, although an ACTIONS clause requires an OPERATIONS clause. The **TypeParameters** are a list (possibly empty) of *generic* type parameters used in the class definition. The EXTENDS list is the set of previously defined classes that are inherited by this class. Local types, functions and constants can be defined in the TYPES and FUNCTIONS clauses. The OWNS list gives attribute declarations. The INVARIANT specifies time-invariant properties of the internal state. The default invariant is **true**.

The ACTIONS list gives the definitions of the various operations that can be performed on instances of the object. The default action for a method, if no action for it is listed, is the completely non-deterministic operation on the state of the class and its parameter types. Input parameters are listed before the output parameters in the action definitions. Z predicates, method invocations and the B0 procedural code constructs of Abrial's B Notation [11] can be used to define methods. Operations are given explicit preconditions by the notations

$$\mathbf{Pre_{m,C}}\ \&\ \mathbf{m\ x\ y}\ \Longrightarrow\ \mathbf{Def_{m,C}} \quad \text{or} \quad \mathbf{m\ x\ y}\ \Longrightarrow\ \text{PRE}\ \mathbf{Pre_{m,C}}\ \text{THEN}\ \mathbf{Def_{m,C}}\ \text{END}$$

The default precondition is **true**. Methods with a preceeding * are *internal actions* and are discussed further below. The HISTORY of a class is an RTL predicate, the forms of which are given in Section 3.

Details of the reference semantics of Z^{++} are given in [6]. Essentially a class name **C**, when used as a type, denotes a countably infinite set @**C** of references to objects of **C**. A function

$$*_{\mathbf{C}} : @\mathbf{C} \nrightarrow \mathrm{State}_{\mathbf{C}}$$

then obtains actual object values (elements of the implicit state schema of **C**) from these references. $\overline{\mathbf{C}}$ denotes $\mathrm{dom} *_{\mathbf{C}}$, the set of existing objects of **C**.

An operation $\mathbf{New_C}$ creates a new instance of **C** and modifies the set of existing instances of **C** (and those of each identified supertype of **C**). A declaration **a** : **C** in the OWNS list of another class is interpreted as **a** : @**C**.

2.2 Specification Examples

Examples of properties which may be expressed in an abstract declarative manner using Z^{++}/RTL formulae are: "**m** initiates every **t** seconds, and in the order of its requests":

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

$\uparrow(\mathbf{m}(\mathbf{x}), \mathbf{i})$ denotes the time that the **i**-th request for invocation of $\mathbf{m}(\mathbf{x})$ received by the current object begins to execute, whilst $\dagger(\uparrow\mathbf{m}(\mathbf{x}), \mathbf{i})$ denotes the **i**-th time of the form $\uparrow(\mathbf{m}(\mathbf{x}), \mathbf{j})$. These may be different if invocations are not initiated in the order that they are received by an object. The 'shortest job first' protocol is stated:

$$\forall \mathbf{i}, \mathbf{j} : \mathbb{N}_1 \mid \rightarrow(\mathbf{m}, \mathbf{i}) \leq \uparrow(\mathbf{m}, \mathbf{j}) \bullet \\ \mathbf{f}((\mathbf{m}, \mathbf{i}).\mathbf{x}) < \mathbf{f}((\mathbf{m}, \mathbf{j}).\mathbf{x}) \Rightarrow \uparrow(\mathbf{m}, \mathbf{i}) \leq \uparrow(\mathbf{m}, \mathbf{j})$$

That is, if $(\mathbf{m}, \mathbf{j})$ has not already started execution by the time the request for $(\mathbf{m}, \mathbf{i})$ arrives, then $(\mathbf{m}, \mathbf{i})$ will be started first if it has a smaller value of some priority assigning function **f** on parameter(s) **x**.

Liveness and fairness constraints can also be stated. An example of a complete (but highly abstract) Z^{++} class is a binary semaphore:

```
CLASS Semaphore
OPERATIONS
  signal : →;
  release : →
HISTORY
  #fin(signal) ≥ #act(release)  ∧
  #fin(release) + 1 ≥ #act(signal)
END CLASS
```

self_mutex({**signal**, **release**}) and **mutex**({**signal**, **release**}) follow from the remainder of the history constraint, which is implicitly quantified by a $\Box^{\tau}$ operator.

3 Z^{++}/RTL Logic

3.1 Logic

Events

- For each method $\mathbf{m}$ of $\mathbf{C}$: $\uparrow\mathbf{m(e)}$, $\downarrow\mathbf{m(e)}$, $\rightarrow\mathbf{m(e)}$ for $\mathbf{e} \in \mathbf{IN}$, denoting the initiation, termination and the arrival of a request at the object class, respectively, of an invocation instance of $\mathbf{m(e)}$;
- $\theta := \mathbf{true}$, $\theta := \mathbf{false}$ for a predicate θ without modal operators, which denote the events of this predicate becoming true or false, respectively.

These events, together with events of the form $\leftarrow(\mathbf{n(x1)}, \mathbf{a})$ for $\mathbf{a} : \mathbf{S}$ a supplier object to $\mathbf{C}$, and $\mathbf{n}$ a method of $\mathbf{S}$ (the sending of a request for $\mathbf{a}$ to execute $\mathbf{n(x1)}$), are collectively referred to as $\mathbf{BasicEvent_C}$. The complete set of events of $\mathbf{C}$ also include the following:

- $\uparrow(\mathbf{n(x1)}, \mathbf{a})$, $\downarrow(\mathbf{n(x1)}, \mathbf{a})$, $\rightarrow(\mathbf{n(x1)}, \mathbf{a})$ where $\mathbf{a}$ and $\mathbf{n}$ are as above;
- $\leftarrow(\mathbf{n(x1)}, \mathbf{a}, \mathbf{b})$ where both $\mathbf{a}$ and $\mathbf{b}$ are supplier objects to $\mathbf{C}$, $\mathbf{a} : \mathbf{S}$, and $\mathbf{n}$ is a method of $\mathbf{S}$.

The complete collection of events of $\mathbf{C}$ is denoted by $\mathbf{Event_C}$.

Terms For a given class $\mathbf{C}$, the following terms can occur in the formulae of its RTL language:

1. variables $\mathbf{v_i}$: $\mathbf{i} \in \mathbb{N}$ – only variables can be quantified over;
2. attributes of the class, its ancestors and supertypes;
3. $\mathbf{f}(\mathbf{e_1}, \ldots, \mathbf{e_n})$ for an $\mathbf{n}$-ary function symbol $\mathbf{f}$ and terms $\mathbf{e_1}$, ..., $\mathbf{e_n}$, and other Z expressions in terms and schema texts;
4. $\dagger\mathbf{e}$ where $\mathbf{e}$ is an *event occurrence* $(\mathbf{E}, \mathbf{i})$, where $\mathbf{E}$ is in $\mathbf{Event_C}$ – the time of the $\mathbf{i}$-th occurrence of $\mathbf{E}$, $\mathbf{i} : \mathbb{N}_1$;
5. $\mathbf{Op(m(e), i)}$ where $\mathbf{m} \in \underline{\mathbf{methods}}(\mathbf{C})$, $\mathbf{e}$ in the input type of $\mathbf{m}$, $\mathbf{i} : \mathbb{N}_1$ and $\mathbf{Op} \in \{\uparrow, \downarrow, \rightarrow\}$, and $\leftarrow((\mathbf{m(e)}, \mathbf{a}), \mathbf{i})$ for a supplier object $\mathbf{a} : \mathbf{D}$ and method $\mathbf{m}$ of $\mathbf{D}$;
6. $\mathbf{e} \circledast \mathbf{t}$ and $\bigcirc\mathbf{e}$ where $\mathbf{e}$ is a term, $\mathbf{t}$ a time-valued term – the value of $\mathbf{e}$ at $\mathbf{t}$ and at the next method initiation, respectively;
7. $\mathbf{\#act(m(e))}$, $\mathbf{\#fin(m(e))}$, $\mathbf{\#req(m(e))}$, $\mathbf{\#req(m)}$, $\mathbf{\#fin(m)}$, $\mathbf{\#act(m)}$ for $\mathbf{m} \in \underline{\mathbf{methods}}(\mathbf{C})$;
8. $\mathbf{self}$.

$\#\mathbf{req(m)}$ is the number of requests for $\mathbf{m}$ received by the current object up to the present time, $\#\mathbf{act(m)}$ the number of initiations of execution of $\mathbf{m}$ and $\#\mathbf{fin(m)}$ the number of terminations of $\mathbf{m}$.

Time-valued terms are arithmetic combinations of terms of the form 4 or 5 and elements of $\mathbb{N}$. The time domain satisfies the axioms of the set of non-negative elements of a totally ordered topological ring, with addition operation $+$ and unit 0, and multiplication operation $*$ with unit 1. Thus $\mathbb{N} \subseteq \mathbf{TIME}$ can be assumed. Quantification over **TIME** is not allowed.

Relativised versions $\#\mathbf{act(m(e), a)}$, etc of event counters for suppliers $\mathbf{a} : \mathbf{D}$ to $\mathbf{C}$ are also included, as are attributes $\mathbf{a.att}$ of such suppliers.

In addition, method names $\mathbf{m}$ from supertypes $\mathbf{D}$ of $\mathbf{C}$ can be used in its events. They will be interpreted (if unambiguous) as $\phi(\mathbf{m})$ in the semantics of the language, where $\mathbf{D} \sqsubseteq_{\phi,\mathbf{R}} \mathbf{C}$ is asserted in the specification.

$\overline{\mathbf{A}}$ and $*_{\mathbf{A}}$ can be referred to, for any class $\mathbf{A}$ in the specification. This enables control over object sharing and aliasing.

Formulae For any class $\mathbf{C}$ the following are the formulae in its RTL language.

1. $\mathbf{P(e_1, \ldots, e_n)}$ for an $\mathbf{n}$-ary predicate symbol $\mathbf{P}$ and terms $\mathbf{e_1}, \ldots, \mathbf{e_n}$;
2. $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, $\neg \phi$ for formulae ϕ and ψ;
3. $\phi \circledcirc \mathbf{t}$ for formulae ϕ and time-valued terms $\mathbf{t}$ – "ϕ holds at time $\mathbf{t}$";
4. $\forall \mathbf{SD} \bullet \phi$, $\exists \mathbf{SD} \bullet \phi$ for declarations $\mathbf{SD}$ and formulae ϕ;
5. $\Box^{\tau}\theta$, $\Box\theta$ and $\bigcirc\theta$ for formulae θ;
6. $\Diamond^{\tau}\theta$, $\Diamond\theta$ for formulae θ;
7. $\mathbf{enabled(m)}$ and $\mathbf{enabled(m(e))}$ for methods $\mathbf{m}$, $\mathbf{e}$ in the input type of $\mathbf{m}$, and $\mathbf{enabled(m, a)}$ for suppliers $\mathbf{a}$ of $\mathbf{C}$.

$\Box\psi$ denotes that ψ holds at all times of the form $\uparrow(\mathbf{m}, \mathbf{i})$ for $\mathbf{m}$ a method of $\mathbf{C}$ which are at or greater than the present time. $\bigcirc\psi$ denotes that ψ holds at the next time of the form $\uparrow(\mathbf{m}, \mathbf{i})$ (if there is any). In contrast $\Box^{\tau}$ and $\Diamond^{\tau}$ refer to all present and future times.

Axioms Only selected axioms will be presented here. A full list, together with details of the semantics and a proof of soundness, is contained in [7]. The axioms include those of classical predicate logic in this language, and the axioms of the Z mathematical toolkit. Of particular importance is a *frame* axiom which asserts that attributes of $\mathbf{C}$ can only change in value if a method of $\mathbf{C}$ is currently executing – it is a form of *locality* property in the sense of [4].

The `ACTIONS` specification of a method $\mathbf{m}$ is interpreted as:

$$(\mathbf{xi}) : \forall \mathbf{e} : \mathbf{IN};\ \mathbf{i} : \mathbb{N}_1 \bullet \mathbf{Pre_{m,C}} \circledcirc \uparrow(\mathbf{m(e)}, \mathbf{i}) \Rightarrow \mathbf{Def_{m,C}}[\mathbf{v} \circledast \downarrow(\mathbf{m(e)}, \mathbf{i})/\mathbf{v}'] \circledcirc \uparrow(\mathbf{m(e)}, \mathbf{i})$$

where the appropriate versions $(\mathbf{m(e)}, \mathbf{i}).\mathbf{x_j}$ of formal input or output parameters are used in $\mathbf{Pre_{m,C}}$ or $\mathbf{Def_{m,C}}$.

Axioms of linear temporal logic (LTL) [15] hold in this formalism, including the induction scheme:

$$\phi \circledcirc \mathbf{min}(\{\dagger(\uparrow \mathbf{m_1}, 1), \ldots, \dagger(\uparrow \mathbf{m_n}, 1)\}) \ \wedge \ \Box(\phi \Rightarrow \bigcirc\phi) \ \Rightarrow \ \Box\phi$$

which holds for each LTL formula ϕ, where $\underline{\mathbf{methods}}(\mathbf{C}) = \{\mathbf{m_1}, \ldots, \mathbf{m_n}\}$.

In [7] it is shown that a version of Manna-Pnueli logic [14] is provable from Z^{++}/RTL, and that Z^{++}/RTL is conservative over Manna-Pnueli logic.

Abbreviations $\underline{\mathbf{methods}}(\mathbf{C})$ abbreviates the set of methods of a class $\mathbf{C}$, including inherited methods.

$\#\mathbf{active}(\mathbf{m})$ abbreviates $\#\mathbf{act}(\mathbf{m}) - \#\mathbf{fin}(\mathbf{m})$, the number of currently active instances of $\mathbf{m}$. $\#\mathbf{waiting}(\mathbf{m})$ abbreviates $\#\mathbf{req}(\mathbf{m}) - \#\mathbf{act}(\mathbf{m})$, the number of instances of $\mathbf{m}$ awaiting execution.

$\mathbf{delay}(\mathbf{m}, \mathbf{i})$ abbreviates $\uparrow(\mathbf{m}, \mathbf{i}) - \rightarrow(\mathbf{m}, \mathbf{i})$. $\mathbf{duration}(\mathbf{m}, \mathbf{i})$ abbreviates $\downarrow(\mathbf{m}, \mathbf{i}) - \uparrow(\mathbf{m}, \mathbf{i})$. $\mathbf{mutex}(\{\mathbf{m_1}, \ldots, \mathbf{m_n}\})$ abbreviates the assertion

$$\#\mathbf{active}(\mathbf{m_1}) = \sum_{\mathbf{i}=1}^{\mathbf{n}} \#\mathbf{active}(\mathbf{m_i}) \ \vee \ \ldots \ \vee \ \#\mathbf{active}(\mathbf{m_n}) = \sum_{\mathbf{i}=1}^{\mathbf{n}} \#\mathbf{active}(\mathbf{m_i})$$

$\mathbf{self\_mutex}(\{\mathbf{m_1}, \ldots, \mathbf{m_n}\})$ abbreviates $\#\mathbf{active}(\mathbf{m_1}) \leq 1 \ \wedge \ \ldots \ \wedge \ \#\mathbf{active}(\mathbf{m_n}) \leq 1$.

$\underline{\mathbf{m}}$ abbreviates $\#\mathbf{active}(\mathbf{m}) > 0$.

$\mathbf{fires}(\mathbf{t}, \mathbf{i})$ denotes $\neg\ (\mathbf{false} \circledcirc \uparrow(\mathbf{t}, \mathbf{i}))$, that is, the object exists at this time point.

A *durative method* $\mathbf{m}$ is a method which satisfies $\mathbf{durative(m)}$: $\forall \mathbf{i} : \mathbb{N}_1 \bullet \downarrow(\mathbf{m}, \mathbf{i}) > \uparrow(\mathbf{m}, \mathbf{i})$. A *durative class* is a class all of whose methods are durative. For such a class, the property $\forall \mathbf{i} : \mathbb{N}_1 \bullet \underline{\mathbf{m}} \circledcirc \uparrow(\mathbf{m}, \mathbf{i})$ holds for each method $\mathbf{m}$.

3.2 Semantics

A model Ω of a Z^{++}/RTL specification $\mathbf{S}$ consists of a family $(@\mathbf{C})_{\mathbf{C} \in \underline{\mathbf{classes\_of}}(\mathbf{S})}$ of countably infinite sets of object references for each class $\mathbf{C}$ of $\mathbf{S}$, and a family $(\Omega_\mathbf{C})_{\mathbf{C} \in \underline{\mathbf{classes\_of}}(\mathbf{S})}$ with typing $\Omega_\mathbf{C} : @\mathbf{C} \rightarrow \mathbf{Object_C}$ where $\mathbf{Object_C}$ is the set of pairs $\beta = (\alpha, \sigma)$ with the type

$$(\alpha, \sigma) : (\mathbf{BasicEvent_C} * \mathbb{N}_1 \rightarrow \mathbf{TIME}) \times (\mathbf{TIME} \nrightarrow \mathbf{State_C^\beta})$$

which satisfy a set of conditions corresponding to basic properties of events. $\mathbf{State_C^\beta}$ is $\mathbf{State_C}$ with each declaration $\mathbf{att} : \mathbf{T}$ of $\mathbf{C}$ replaced by $\mathbf{att} : \mathbf{T}^\beta$.

α assigns a time (not necessarily in the lifetime of the object) to each occurrence of an event of $\mathbf{C}$, and σ gives the state of the object at each time point in its history. From these the value of terms and formulae of $\mathcal{L}_\mathbf{C}$ can be computed at each time point.

4 Development Process

The development process envisaged for the use of the language for real time systems is as follows:

1. *requirements capture and analysis* of the problem, using a structured method suitable for real time problems, such as OMT [16];
2. *formalisation* of structured method notations in Z^{++}/RTL, using systematic processes for the translation of object classes in OMT into Z^{++} classes (similar to the processes for B AMN described in [11]) and statecharts into history constraints. The abstract declarative nature of RTL allows fairness, liveness and safety constraints to be stated in an implementation-independent manner;
3. *refinement* of Z^{++}/RTL specifications into implementation-oriented classes, making use of reusable specified components and the code of these components;
4. *implementation* of classes using classes which contain procedural code, with timing information derived from a particular execution environment being used to prove the final refinement step.

In the following sections stages 2, 3 and 4 will be illustrated using a small example. Larger applications are given in [7, 8].

4.1 Formalisation of OMT Analysis Models

Integrating formal and structured methods has a number of advantages as a development approach: it can make use of the complementary strengths of these two techniques, and it can make use of existing software engineering expertise, rather than attempting to replace it. A recent survey on the use of formal methods in industry reported that 31% of those companies using formal methods were using them in conjunction with structured methods [2].

Object Models There are two main models which form the input to the formalisation process. The first is the *Object Model* which describes the entities involved in the system, and their attributes, operations and the relationships between them (including inheritance or subtyping). An object model describing a railway station consisting of a set of track sections is shown in Fig. 1.

These models are used to build an initial outline specification, which will later be enhanced by consideration of the dynamic model, and by the addition of semantic detail which could not be expressed in the structured models.

The formalisation process is as follows:

1. For each entity C in the object model, create a Z^{++} object class **C**;
2. Each attribute of C becomes an attribute of **C**, with corresponding type;

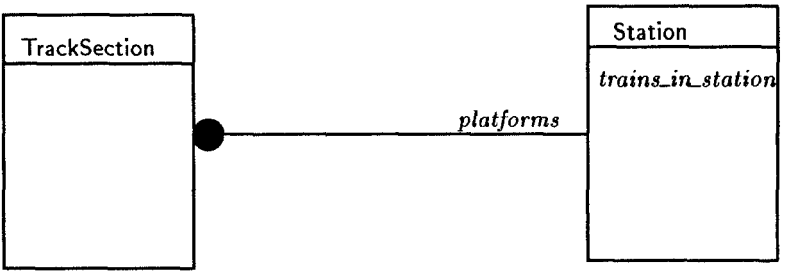

Fig. 1. Object Model of Station

3. Each association r between entities C and D is examined to determine if both directions of the association are required in the final system. The required directions are then formalised as attributes $\mathbf{r}_1$ of **C**, of type **D** (in the case of a many-one or one-one association from C to D), or of type $\mathbb{F}(\mathbf{D})$ or seq(**D**) (in the case of an unordered or ordered many-many or one-many association from C to D, respectively). Similarly for the inverse map $\mathbf{r}_2$ from **D** to **C**, if this is required;

4. If D inherits from C, then the clause `EXTENDS C` is placed in the header of **D**. Conformant subtypes are expressed via suitable $\sqsubseteq$ assertions;

5. Operations are translated into outline specifications of operations, with however all signatures completed;

6. Formalisation of all assertions on an object model can be attempted, using the `INVARIANT` and `HISTORY` components of a class.

Dynamic Model For reactive systems the dynamic model, based upon Harel statecharts, is the most significant analysis model. Here we will consider extensions of statecharts to include time bounds on transitions. The formalisation process for dynamic models is:

1. Condition triggered transitions and anonymous (automatic) transitions are formalised as operations which are internally invoked, whilst event-triggered transitions are formalised as operations which are invokable from other objects;

2. states are formalised as elements of an enumerated set, and an attribute of this type is defined to record the current state. Methods formalising a transition modify this variable appropriately;

3. time bounds $[\mathbf{l}, \mathbf{u}]$ on transition t are expressed via the formula $\forall \mathbf{i} : \mathbb{N}_1 \bullet \mathbf{fires}(\mathbf{t}, \mathbf{i}) \Rightarrow \mathbf{l} \leq \mathbf{delay}(\mathbf{t}, \mathbf{i}) \leq \mathbf{u}$;

4. if **t** has source state S1 and destination state S2 (assumed distinct) and guard condition **cond**, then:

$$\begin{array}{l}(\mathbf{enabled(t)} \equiv (\mathbf{state} = \mathbf{S1}) \wedge \mathbf{cond}) \quad \wedge \\ \forall \mathbf{i} : \mathbb{N}_1 \bullet \exists \mathbf{j}, \mathbf{k1}, \mathbf{k2} : \mathbb{N}_1 \bullet \\ \qquad \dagger((\mathbf{state} = \mathbf{S1}) \wedge \mathbf{cond} := \mathbf{true}, \mathbf{j}) = \rightarrow(\mathbf{t}, \mathbf{i}) \ \wedge \\ \qquad ((\mathbf{state} = \mathbf{S1}) \wedge \mathbf{cond}) \circledcirc \uparrow(\mathbf{t}, \mathbf{i}) \ \wedge \\ \qquad \downarrow(\mathbf{t}, \mathbf{i}) = \dagger((\mathbf{state} = \mathbf{S1}) := \mathbf{false}, \mathbf{k1}) \ \wedge \\ \qquad \downarrow(\mathbf{t}, \mathbf{i}) = \dagger((\mathbf{state} = \mathbf{S2}) := \mathbf{true}, \mathbf{k2})\end{array}$$

If **t** is durative (in particular if $\mathbf{S1} \neq \mathbf{S2}$) then the LTL properties

$$\Box(\underline{\mathbf{t}} \Rightarrow \bigcirc(\mathbf{state} = \mathbf{S2})) \qquad \Box(\underline{\mathbf{t}} \Rightarrow \mathbf{state} = \mathbf{S1})$$

can be derived (if **t** has no other source or destination);

5. if **t** has an associated action **act** on supplier object **a**, then:

$$\forall \mathbf{i} : \mathbb{N}_1 \bullet \exists \mathbf{k3} : \mathbb{N}_1 \bullet \uparrow(\mathbf{t}, \mathbf{i}) = \leftarrow((\mathbf{act}, \mathbf{a}), \mathbf{k3})$$

6. event triggered transitions are formalised in the same way, however clause 4 is replaced by:

$$\begin{array}{l}\mathbf{enabled(t(p))} \equiv (\mathbf{state} = \mathbf{S1}) \ \wedge \\ \forall \mathbf{i} : \mathbb{N}_1 \bullet \exists \mathbf{j} : \mathbb{N}_1 \bullet \\ \qquad (\mathbf{state} = \mathbf{S1}) \circledcirc \dagger(\mathbf{event(p)}, \mathbf{j}) \ \wedge \\ \qquad \dagger(\mathbf{event(p)}, \mathbf{j}) = \rightarrow(\mathbf{t(p)}, \mathbf{i}) \ \wedge \\ \qquad (\mathbf{state} = \mathbf{S1}) \circledcirc \uparrow(\mathbf{t(p)}, \mathbf{i})\end{array}$$

Each class corresponding to a statechart is mutex and self-mutex. In addition there are liveness constraints asserting that any non-terminal state must eventually be exited, and constraints asserting that states can only become true as a result of a transition into them. If there are no self-transitions on S1 then the assertion $\Box(\mathbf{state} = \mathbf{S1} \Rightarrow \underline{\mathbf{t_1}} \vee \ldots \vee \underline{\mathbf{t_n}})$ states that S1 can only be exited via transitions $\mathbf{t_1}, \ldots, \mathbf{t_n}$.

For example, consider the statechart of track sections shown in Fig. 2.

A *track section* is a defined contiguous segment of track which can be occupied by at most one train at any time. In addition, it may be closed (eg, for engineering work) so that no trains may enter the section. A train will take a minimum of 60 seconds to clear a track section, and a closed track section will be closed for a minimum of 120 seconds. The corresponding outline class, with a state variable **tstate** : **TState** is:

```
CLASS TrackSection
TYPES
  TState ::= closed | free | blocked
OWNS
  tstate : TState
```

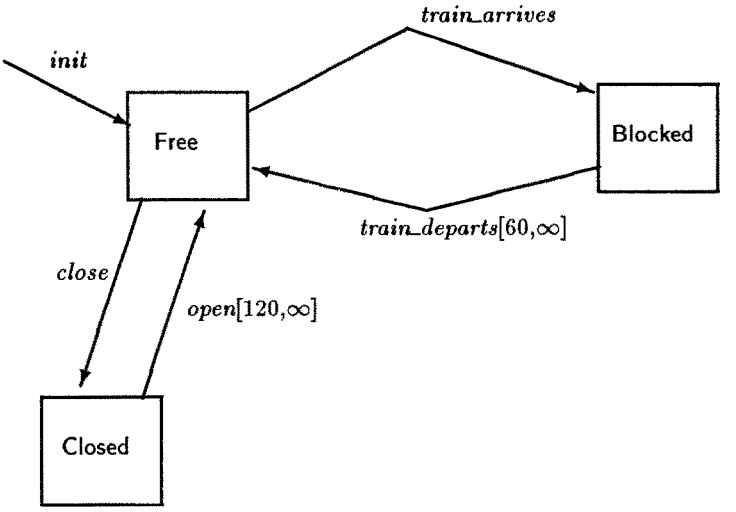

Fig. 2. Statechart of **TrackSection**

```
OPERATIONS
  init : →;
  train_arrives : →;
  train_departs : →;
  open : →;
  close : →
ACTIONS
  init ==>
          tstate' = free;

  train_arrives ==>
          tstate' = blocked;

  train_departs ==>
          tstate' = free;

  open ==>
          tstate' = free;

  close ==>
          tstate' = closed
HISTORY
  mutex({ init, train_arrives, train_departs, open, close }) ∧
  self_mutex({ init, train_arrives, train_departs, open, close }) ∧
  □(train_arrives ⇒ tstate = free) ∧
  □(train_departs ⇒ tstate = blocked) ∧
  □(open ⇒ tstate = closed) ∧
  □(close ⇒ tstate = free) ∧
  ∀ i : ℕ1 • fires(train_departs, i) ⇒ 60 ≤ delay(train_departs, i) ∧
```

$\forall$ i : $\mathbb{N}_1$ • **fires(open,i)** $\Rightarrow$ 120 $\leq$ **delay(open,i)**
END CLASS

There are additional logical properties which can be derived from the statechart. More detail on the formalisation process, including the treatment of entry and exit actions, activities within a state, nested statecharts and AND composition of statecharts is given in [8]. The precision of the translation is such that statecharts could be explicitly presented in the HISTORY clause of a class in place of (some) temporal assertions.

4.2 Refinement

Refinement in Z^{++}/RTL corresponds to *theory extension* [6, 7]. That is, all specified dynamic properties of a class **C** should be provable in any refinement **D** of **C**. The notation for refinement of **C** by **D**, via a renaming ϕ of the methods of **C**, and a data refinement relation **R** on the combined states is $\mathbf{C} \sqsubseteq_{\phi,\mathbf{R}} \mathbf{D}$.

Subtyping of classes is equated with class refinement, in contrast to [12], although the roles of subtyping and refinement within the development process are clearly distinguished. Alternative concepts of refinement, related to *operational compatibility* [3], and strictly stronger than subtyping, are discussed in [7].

4.3 Implementation

In the final refinement of a subsystem, a restricted language is used, in which methods are defined using constructs corresponding to procedural code structures, and in which types are restricted to be arrays, strings, object reference types or scalars.

This language *preserves refinement* in the sense that if a class **D** is a client of a class **C**, and $\mathbf{C}_1$ refines **C**, then substituting $\mathbf{C}_1$ for **C** in **D** to produce a class $\mathbf{D}_1$ implies that $\mathbf{D}_1$ refines **D**. This supports compositional (separate) development of subsystems. It is also direct to implement such classes in an object-oriented programming language such as C++. Timing specifications for such implementations can be derived from the timing specifications of the hardware which is the ultimate destination of the compilation process.

The RTL language is extended by times $\uparrow(\mathbf{s},\mathbf{i})$, $\downarrow(\mathbf{s},\mathbf{i})$ and $\rightarrow(\mathbf{s},\mathbf{i})$ for each statement occurrence **s** of the implementation language in **C**. These denote respectively start, end, and request times for the **i**-th invocation of **s** within an instance of **C**.

These times are assumed in general to be non-negative real numbers. The approach of [5] is used: For each primitive operation **op**, which is one of: **gaddr** (get an address), **stor** (store a value in an address), **eval**(e) (evaluate expression **e**), **decl**(**T**) (define storage for a variable of type **T**), **brt**, **brtloop** (branch on true), **brf**, **brfloop** (branch on false), there are corresponding *sets* **T**(**op**) of possible durations for this operation.

$\mathbf{T(op)}$ should always be a small finite set, thus allowing reasoning by case analysis. The definition of $\mathbf{T}$ will vary between destination processor architectures. An operation $\dot{+}$ performs addition of such sets, ie: $\mathbf{S \dot{+} T} = \{\mathbf{s + t} \mid \mathbf{s} \in \mathbf{S} \wedge \mathbf{t} \in \mathbf{T}\}$. An example of the timing rules is:

Assignment The behaviour of assignment is given by:

$$\rightarrow(\mathbf{x := e, i}) = \uparrow(\mathbf{x := e, i})$$
$$\downarrow(\mathbf{x := e, i}) - \uparrow(\mathbf{x := e, i}) \in \mathbf{T(gaddr)} \dot{+} \mathbf{T(stor)} \dot{+} \mathbf{T(eval(e))}$$

That is, there is no delay in executing the assignment after its request, and its duration is a sum of possible durations for its constituent steps.

5 Development Using Semaphores

A general approach for the refinement of classes with non-trivial synchronisation constraints is to utilise classes, such as semaphores, which provide specific synchronisation facilities. That is, a class **C** with synchronisation requirements expressed in its `HISTORY` clause will be refined by a class **C_1** which has a supplier class **S** whose properties can be used to prove the requirements of **C**. **C_1** itself may not need to contain any synchronisation mechanisms.

An example is the case where it is required that a particular method **m** is self-mutex (Fig. 3). Synchronisation constraints of **C** are implemented using a

```
CLASS C
OWNS
  ...
OPERATIONS
  m: X → Y;
  ...
ACTIONS
  m x y  ==>  Def_m;
  ...
HISTORY
  self_mutex({ m })
END CLASS
```

```
CLASS C_1
OWNS
  s: Semaphore;
  ...
OPERATIONS
  m: X → Y;
  ...
ACTIONS
  m x y  ==>
          BEGIN
            s.signal;
            Code;
            s.release
          END;
  ...
END CLASS
```

Fig. 3. Refinement Using Semaphores

Semaphore instance, where **Code** implements the state transitions defined in $\mathbf{Def_m}$. The refinement is formally provable because:

$$\begin{aligned}\#\mathbf{active(m)} &= \#\mathbf{act(m)} - \#\mathbf{fin(m)} \\ &= \#\mathbf{act(signal, s)} - \#\mathbf{fin(release, s)} \leq 1\end{aligned}$$

from the history constraint of **Semaphore**.

Other examples are given in [8], including a development of the dining philosophers problem using this approach.

6 Refinement Using Synchronisation Code

An alternative refinement route, which allows synchronisation and fairness requirements expressed in a Z^{++} specification to be discharged, makes use of the concept of *synchronisation code* [13]. This involves the definition of state transformations on *synchronisation variables* which are performed at event occurrences. An event is either the creation of an object, a reception of a request for a method execution, the start of an execution of a method, or the termination of execution of a method. Thus the approach is consistent with the Z^{++}/RTL formalism, and with statecharts.

Synchronisation variables and functions over these are then used to control the permission to execute methods via permission guards.

The refinement of a class (at the end of a data and procedural refinement process) into a C++ class with synchronisation code can be formally checked by an induction over events. This induction will be that the abstract declarative history constraint of the Z^{++} class is always true at each event time, with respect to the translation between Z^{++} and C++ variables.

As an example, consider the specification that requests of **m** are served in a first-come, first-served manner (FCFS):

```
CLASS C
OPERATIONS
  m : →
HISTORY
  ∀ i, j : ℕ1 | →(m,i) < →(m,j) • ↑(m,i) ≤ ↑(m,j)
END CLASS
```

An appropriate extended C++ class is (following [13]):

```
class C{
  m() {}
synchronisation
  int clk;
  int arr_time local to m;

  start(C) --> clk = 0;
  arrival(m) --> this_inv.arr_time = clk++;

  m: there_is_no(p in waiting(m): p.arr_time < this_inv.arr_time);
}
```

The final clause here asserts that the **this_inv** invocation instance of **m** cannot initiate execution unless the priority condition holds for this instance.

We can relate these two versions of **C** as follows. The concrete expression `this_inv.arr_time` refers to a local (synchronisation) variable of a specific instance $(\mathbf{m}, \mathbf{i})$ of an invocation of $\mathbf{m}$. This can be formalised as $(\mathbf{m}, \mathbf{i}).\mathbf{arr\_time}$. From the code it follows that: $(\mathbf{m}, \mathbf{i}).\mathbf{arr\_time} = \mathbf{clk} \circledast \rightarrow(\mathbf{m}, \mathbf{i})$.

By induction on events it can be shown that this value is also

$$\mathbf{previous\_requests}(\mathbf{m}, \rightarrow(\mathbf{m}, \mathbf{i})) \;=\; \#\{\mathbf{j} : \mathbb{N}_1 \mid \rightarrow(\mathbf{m}, \mathbf{j}) < \rightarrow(\mathbf{m}, \mathbf{i})\}$$

(The only relevant events are those listed in the concrete class, and it is assumed that object creation occurs before any $(\mathbf{m}, \mathbf{i})$ invocation.)

The set $\mathbf{waiting}(\mathbf{m})$ of outstanding invocation instances of $\mathbf{m}$ has the formal counterpart $\mathbf{waiting\_instances}(\mathbf{m}, \mathbf{t}) = \{\mathbf{i} : \mathbb{N}_1 \mid \rightarrow(\mathbf{m}, \mathbf{i}) \leq \mathbf{t} \wedge \neg\, (\uparrow(\mathbf{m}, \mathbf{i}) \leq \mathbf{t})\}$ for each time-valued term $\mathbf{t}$.

Therefore the concrete guard has corresponding formal permission requirement

$$\begin{array}{l} \forall\, \mathbf{i} : \mathbb{N}_1 \bullet \\ \quad (\neg\ \exists\, \mathbf{pind} : \mathbb{N}_1 \mid \mathbf{pind} \in \mathbf{waiting\_instances}(\mathbf{m}, \uparrow(\mathbf{m}, \mathbf{i})) \bullet \\ \qquad \mathbf{previous\_requests}(\mathbf{m}, \rightarrow(\mathbf{m}, \mathbf{pind})) \;< \\ \qquad\qquad \mathbf{previous\_requests}(\mathbf{m}, \rightarrow(\mathbf{m}, \mathbf{i}))) \circledast \uparrow(\mathbf{m}, \mathbf{i}) \end{array}$$

Given this, it is impossible for there to be $\mathbf{pind} < \mathbf{i}$ with $\rightarrow(\mathbf{m}, \mathbf{pind}) < \rightarrow(\mathbf{m}, \mathbf{i})$ but $\uparrow(\mathbf{m}, \mathbf{i}) < \uparrow(\mathbf{m}, \mathbf{pind})$.

Assume otherwise. Then $\mathbf{pind} \in \mathbf{waiting\_instances}(\mathbf{m}, \uparrow(\mathbf{m}, \mathbf{i}))$. But

$$\mathbf{previous\_requests}(\mathbf{m}, \rightarrow(\mathbf{m}, \mathbf{pind})) < \mathbf{previous\_requests}(\mathbf{m}, \rightarrow(\mathbf{m}, \mathbf{i}))$$

since **pind** is a member of the second set and not the first (and $\mathbf{previous\_requests}(\mathbf{m}, \mathbf{t})$ is monotonically increasing with $\mathbf{t}$), contradicting the guard for $\mathbf{m}$.

Tools

Animation and proof tools have been developed for this formalism. Animation is based upon the checking of proposed scenarios expressed as sequences of events. The tool allows a sequence of events and corresponding times to be incrementally constructed, checking permission guards, duration and delay constraints, synchronisation constraints involving event counters, and mutual exclusion properties. Post-states are generated from pre-states, with user interaction being required if there is non-determinism in the method specification [9].

Conclusions

This paper has detailed development processes for real time system development which integrate structured and formal methods in an object-oriented framework, and which combine the benefits of these methods. The proposed formalism provides many of the facilities of real time formalisms such as interval logic [1] or RTTL [14] without excessive notational overhead.

References

1. J. F. Allen: *Maintaining knowledge about temporal intervals*, *CACM*, 26(11):832–843, November 1983.
2. S. Austin, G. I. Parkin: *Formal Methods: A Survey*, National Physical Laboratory, Queens Road, Teddington, Middlesex, TW11 0LW, March 1993.
3. R. Duke, P. King P, G. Smith: *Formalising Behavioural Compatibility for Reactive Object-oriented Systems*, in **Proc 14th Australian Compt. Sci. Conf. (ACSC-14)**, 1991.
4. J. Fiadeiro, T. Maibaum: *Sometimes "Tomorrow" is "Sometime"*, in **Temporal Logic**, D. M. Gabbay and H. J. Ohlbach (editors), LNAI 827, Springer-Verlag 1994, 48–66.
5. C. Fidge: *Proof Obligations for Real-Time Refinement*, **Proceedings of 6th Refinement Workshop**, Springer-Verlag Workshops in Computing, 1994.
6. K. Lano: *Refinement in Object-oriented Specification Languages*, **Proceedings of 6th Refinement Workshop**, Springer-Verlag Workshops in Computing, 1994.
7. K. Lano: *Formal Object-oriented Specification of Real Time Systems*, Dept. of Computing, Imperial College, 1994.
8. K. Lano: *Software Specification and Development in Z^{++}*, to appear in **The Z Handbook**, J. Bowen and M. Hinchey (eds.), McGraw-Hill 1995.
9. K. Lano: *Reasoning Techniques in VDM^{++}*, AFRODITE project report AFRO/IC/KL/RT/V1, Dept. of Computing, Imperial College, 1994.
10. K. Lano, H. Haughton: **Object-oriented Specification Case Studies**, Prentice Hall, 1993.
11. K. Lano, H. Haughton: *Improving the Process of System Specification and Refinement in B*, **Proceedings of 6th Refinement Workshop**, Springer-Verlag Workshops in Computing, 1994.
12. B. Liskov, J. Wing: *Family Values: A Behavioral Notion of Subtyping*, School of Computer Science, Carnegie Mellon University, report CMU-CS-93-187, 1993.
13. C. McHale, S. Baker, B. Walsh, A. Donnelly: *Synchronisation Variables*, Amadeus Project report TCD-CS-94-01, University of Dublin, 1994.
14. J. S. Ostroff: **Temporal Logic for Real-Time Systems**, John Wiley, 1989.
15. A. Pnueli: *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*, **Current Trends in Concurrency**, de Bakker J., de Roever W.-P., Rozenberg G. (Eds), Springer-Verlag Lecture Notes in Computer Science, Vol. 224, 1986.
16. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, **Object-Oriented Modelling and Design**, Prentice-Hall International, 1991.