Abstract

In this paper we describe the application of a high-level programming language to the abstract modelling of transport systems. We show how METATEM, a language based upon the execution of temporal logics, can be used to model railway networks. The examples considered are abstractions of real rail systems, and we show how such abstractions can be modelled both by standard, and concurrent, METATEM. During this exercise, we highlight the features of executable temporal logics, and METATEM in particular, which aid the modelling of such dynamic, reactive systems.

1 Introduction

When developing computer applications which are implementations of dynamic or reactive systems, it is important to do so in such a way that (i) the specification can be shown to have the appropriate safety, liveness and fairness properties, and (ii) the implementation can be formally verified against the specification, and thus shown to have implemented those properties [Pnu86]. The emerging technology of executable temporal logic marries the operational model of reactive systems together with the formal methods theory of temporal logic to provide a framework within which reactive applications can be developed [Mos86, Gab87, BFG*89, AM89].

In this paper we model, abstractly, the signalling behaviour of several networks of stations of a railway system. We do not intend to describe all the details, but simply present executable temporal logic programs that characterise the general behaviour of the system. Consequently, the model we use will not correspond directly to a real-life transport system. It represents an abstraction of such a system exhibiting several features fundamental to the behaviour of transport systems. We will look at various configurations of lines and stations, and will use the following simplifying restrictions.

- Each station can be occupied by at most one train.
- Each train is assigned to a particular line, and can only ever travel on that line.
- Trains can only travel in one direction on a particular line, and each line has a direction associated with it.
- Networks consists only of stations, connected to each other.
- Signals have only two settings — red and green.

Given these restrictions, we will show how various configurations can be modelled, first in a single-threaded executable temporal logic, METATEM [BFG*89, FO92], then in an object based concurrent development of METATEM [FB91, Fis92b]. The former leads us to develop an implementation which takes a global view of each network, the latter exploits the object-oriented paradigm to develop a localised implementation. The benefits of the two approaches will be illustrated.

We begin by introducing several simple rail networks (§2), before giving a brief description of the METATEM system, including an introduction to temporal logic (§3). In §4, we show how the example networks can be modelled in METATEM, and discuss the benefits of this form of description. In §5, METATEM is extended to give Concurrent METATEM, which is based upon a novel model of concurrent object-based computation. An alternative description of the rail configurations described earlier will be given using Concurrent METATEM in §6. Finally, a summary and discussion of these different approaches is provided in §7.

2 Example Rail Networks

There are some elementary statements we can make about the behaviour of a simple railway system, as follows.
1. If a train is in a station, and its route is blocked by a red signal, then it remains in that station, and the station’s entry signal remains red.

2. If a train is in a station, and the entry signal of its next station is green, in the next state the train will be in the next station, with the entry signal of the next station set to red, and the entry signal of this station set to green.

3. If there is no train in a station, and there is no train about to enter the station, then the entry signal of the station will be green in the next state.

These rules are sufficient to drive the examples of train networks that we shall use. They may not, however, be good enough to ensure that the models possess all the properties that we may wish. We would like, for example, that the following be true.

1. No station ever has more than one train in it at any state.

2. A train is never stationary forever.

3. The system behaves ‘fairly’ at entry points to critical regions.

4. The network is ‘efficient’ — as busy as possible.

While we might hope that the above rules do indeed guarantee at least the first property, we should actually prove that they do by formal means. Once the rules (and the properties) have been stated in temporal logic, this is possible using a number of techniques devised to prove theorems in temporal logic [AM90, Fis91].

We will now describe several rail configurations that will be used as examples throughout the rest of the paper.

### 2.1 Ring Network

We begin with a simple abstraction of a circular track (see Figure 1), which has a fixed number of trains on it. Only a simple notion of a station with a single entry point, and a single exit point is required for this configuration. We then extend this to a double circle, in which a section of track is shared between two circles — a station which merges trains from two entry points onto a single exit point is then required. We describe the state of each network using the following predicates:

- \( \text{signal}(S, E, C) \) — the colour \( C \) of the entry signal \( E \) for each station \( S \).

- \( \text{station}(S) \) — the names of the stations.

- \( \text{line}(T, L) \) — the line \( L \) of each train \( T \).

- \( \text{has}(S, T) \) — a predicate storing the name of the train \( T \) currently at station \( S \).

![Figure 1: A simple circular line](image)

These definitions of \( \text{station} \) and \( \text{next} \) for the simple ring shown in Figure 1 are:

- \( \text{station}(\text{North}) \)
- \( \text{next}(\text{North, circle, East}) \)
- \( \text{station}(\text{East}) \)
- \( \text{next}(\text{East, circle, South}) \)
- \( \text{station}(\text{South}) \)
- \( \text{next}(\text{South, circle, West}) \)
- \( \text{station}(\text{West}) \)
- \( \text{next}(\text{West, circle, North}) \)

These values of these predicates never change — \( \text{station} \) and \( \text{next} \) are said to be \text{rigid}. The other predicates are \text{flexible} — their values change over time.

The initial configuration for the railway system consisting of a simple ring is that train ‘1’ is at station ‘East’, and train ‘2’ is at station ‘West’. Also, the signals for these, initially occupied, stations must be set to red, and the other stations green:

- \( \text{signal}(\text{North, in, green}) \)
- \( \text{signal}(\text{East, in, red}) \)
- \( \text{signal}(\text{South, in, green}) \)
- \( \text{signal}(\text{West, in, red}) \)

While we might hope that the above rules do indeed guarantee at least the first property, we should actually prove that they do by formal means. Once the rules (and the properties) have been stated in temporal logic, this is possible using a number of techniques devised to prove theorems in temporal logic [AM90, Fis91].

We will now describe several rail configurations that will be used as examples throughout the rest of the paper.

### 2.2 Double Ring Network

We can merge two slightly larger versions of the above rings to form a figure-of-eight like system. Here, there are two lines, one moving clockwise and one moving anti-clockwise, i.e. as shown in Figure 2. Thus, the lines join at station ‘D’, split at station ‘F’, and use same track at station ‘E’. So, if ‘upper’ is the line running through A–B–C–D–E–F–A and ‘lower’ is the line running through G–H–I–D–E–F–G, then the definitions of \( \text{next} \) are as follows:

![Figure 2: A double ring network](image)
The initial configuration uses two trains on each line: trains ‘1’ and ‘2’ run only on the upper line, while trains ‘3’ and ‘4’ run only on the lower line. The initial placement of these four trains is train ‘1’ at station ‘C’, train ‘2’ at station ‘E’, train ‘3’ at station ‘F’, and train ‘4’ at station ‘H’.

Station ‘D’ differs from the other stations in this example, and from the stations in the simple ring network, in that it has two entry points — station ‘C’ on the upper line and station ‘I’ on the lower line. This ‘merging’ station requires more sophisticated control rules than the single-entry stations. In order to be able to write these rules (which will be presented in §4) we must distinguish between the two input signals. We do this by labelling them left and right in the signal predicate, rather than simply just in. Thus the initial configuration for the double ring is:

- signal(α, in, green) for α ∈ {A, B, G, I}
- signal(α, in, red) for α ∈ {C, E, F, H}
- signal(D, left, green) signal(D, right, red)
- has(train I, C) has(train 3, F)
- has(train 2, E) has(train 4, G)

Note that, station ‘F’ also represents a new type of station. This ‘splitting’ station has two exit points and directs trains to the appropriate next station according to their line.

Although the two configurations, shown in Figures 1 and 2 may seem trivial, a large range of complex rail networks can be constructed using the types of station exhibited in these two systems. The above networks are composed of ‘simple’ stations, ‘merging’ stations, and ‘splitting’ stations, which represent basic components from which more complex configurations can be constructed. For example, if we wish to represent a station in a rail network where three lines meet, we can either simulate such an element using two ‘merging’ stations, or can generalise the merging station’s rules to handle three, rather than two, incoming lines.

Before showing how the configurations described above can be modelled in METATEM, we first give an outline of the temporal logic used and the basic METATEM approach to execution.

### 3 Temporal Logic and Execution

METATEM is a framework for executing temporal logics [BFG’89]. The particular instance of this framework that we describe here is based upon the execution of discrete, linear temporal logics, and so we first give a brief outline of such a logic, First-order METATEM Logic (FML). This logic introduces two new connectives, ‘until’ (U) and ‘since’ (S), together with a number of other operators defnable in terms of U and S, to classical logic.

The classical component of FML is used to describe states, for example

\[ \neg(\text{track-blocked} \land \text{signal-green}). \]

The U and S connectives relate states together. αUA is true in a state s if the formula β is true in a state t in the future of state s, and the formula α is true in every state between states s and t. Thus, α will be true until β is true. The S connective mirrors the behaviour of U in the past, so that αSAβ is true in a state s if the formula β is true in a state t in the past of state s, and the formula α is true in every state between states t and s. Hence α was true since β was true.

Consider a simple specification of a signal on a track, which requires that should there be a train on a particular section of track, that track’s entry signal must have been red ever since the train passed it. We can express this by the following temporal logic rule:

\[ \text{train-in-section} \Rightarrow \text{signal-red} S \text{train-entered} \]

which is read as ‘if there is a train in the section, then there was a state in the past when the train entered the section of track, and the signal has been red in all the states since then.’

#### 3.1 Syntax of FML

Well-formed formulae of FML (WFF) are generated from the following symbols,

- The set, Lp, is a set of predicate symbols represented by strings of lowercase alphabetic characters. This set is partitioned into sets of environment, component and internal predicates.
- A set, Lv, of variable symbols, x, y, z, etc.
- A set, Lc, of constant symbols, a, b, c, etc.
- Quantifiers, ∀ and ∃.

We define set of terms, Ls, as the set of all strings that are either constant or variable symbols. The set of well-formed formulae of FML (WFF) is defined by:

1. If t₁, …, tₙ are in Ls, and p is a predicate symbol of arity n, then p(t₁, …, tₙ) is in WFF.
2. If A and B are in WFF, then so are true, false, ¬A, A V B, AUB, A S B, and (A).
3. If A is in WFF and v is in Lv, then ∃v: A and ∀v: A are both in WFF.
We define other classical connectives, such as ‘∧’, ‘⇒’, and ‘⇔’ in terms of the ones given above, and can also define several other useful temporal connectives in terms of \( \mathcal{U} \) and \( \mathcal{S} \). For example \( \diamond \alpha \), meaning that \( \alpha \) is true in some future state, can be defined as \( \text{true}_\mathcal{U} \alpha \) which can be read as ‘in some future state \( \alpha \) will be true, and \( \text{true} \) will be true in all the states until then.’ Because \( \text{true} \) is true in all states, \( \text{true}_\mathcal{U} \alpha \) simply requires \( \alpha \) to be true in some future state. All the connectives of FML are described in Figure 3, along with their definitions in terms of \( \mathcal{U} \) and \( \mathcal{S} \).

Temporal formulae can be classified as follows. A state-formula is either a literal (an atom or a negated atom) or a boolean combination of other state-formulae. Strict future-time formulae are defined as follows

- If \( A \) and \( B \) are either state or strict future-time, then \( A \cup B \) is strict future-time.
- If \( A \) and \( B \) are strict future-time, then \( \neg A, A \lor B \), and \( (A) \) are all strict future-time.

Strict past-time formulae are defined as the past-time duals of strict future-time formulae. Non-strict classes of formulae include state-formulae in their definition. Thus \( \bullet a \land e \diamond \delta d \) is a strict past-time formula, whereas \( p \lor \diamond q \) is a non-strict future-time formula.

3.2 Semantics of FML

The process of executing a METATEM program is essentially that of building a model for the FML formula which constitutes the program. The models for FML formulae are given by a structure, which consists of a sequence of states, together with an assignment of truth values to atomic sentences within states, a domain \( \mathcal{D} \), and mappings from elements of the language to denotations. More formally, a model is a tuple \( \mathcal{M} = (\sigma, \mathcal{D}, \pi, \pi_\sigma) \) where \( \sigma \) is the ordered set of states \( s_0, s_1, s_2, \ldots, \pi_\sigma \) is a map from the constants to \( \mathcal{D} \), and \( \pi_p \) is a map from \( \mathcal{N} \times \mathcal{L}_p \) to \( \mathcal{D} \rightarrow \{T,F\} \). Thus, for a particular state \( s \), and a particular predicate \( p \) of arity \( n \), \( \pi(s,p) \) gives truth values to atoms constructed from \( n \)-tuples of elements of \( \mathcal{D} \). Note that we use the constant domain assumption, i.e. that \( \mathcal{D} \) is constant for every state. A variable assignment \( V \) is a mapping from the variables to elements of \( \mathcal{D} \). Given a variable assignment and the valuation function \( \pi \), a term assignment \( \tau_{\pi,V} \) is a mapping from terms to \( \mathcal{D} \) defined in the usual way. We will usually deal with closed formulae, i.e. formulae containing no free variables. In this case, the empty mapping, [], is used as the initial variable assignment.

We use the relation ‘\( \models \)’ to give the truth value of a formula in a model \( \mathcal{M} \), at a particular moment in time \( i \) and with respect to a variable assignment. This relation is defined for formulae of FML in Figure 4. Since an interpretation consists of a triple comprising model and time components, we say that a well-formed formula is satisfied in a particular model at a moment in time according to some variable assignment.

As FML formulae refer to discrete linear time-structures, which have a finite past, notice that the formula \( \bullet \text{false} \) can only be satisfied at the beginning of time and we can thus use this formula to mark that unique moment.

3.3 METATEM

METATEM programs are not just arbitrary FML formulae. They must be in a specific ‘past implies future’ form to be executed. Fortunately, however, any arbitrary FML formula can be transformed into this form \[\text{BFG}*89]. Further, arbitrary FML formulae can be transformed into a normal form, called First-Order Separated Normal Form (SNF\(_f\)) \[\text{Fis}92a\]. A formula in SNF\(_f\) is of the form

\[
\Box \bigwedge_{i=1}^n \forall \bar{x}_i. (\forall \bar{y}_i. P_i(\bar{x}_i, \bar{y}_i)) \Rightarrow \exists \bar{z}_i. F_i(\bar{x}_i, \bar{z}_i)
\]

where each of the conjoined implications is called a rule. The \( P_i \) and \( F_i \) are restricted in that each \( P_i \) is a (non-strict) past-time temporal formula, more precisely to be of the form

\[
\bullet \text{false} \land \bigwedge_{b=1}^h l_b(\bar{x}, \bar{y}) \lor (\bigotimes_{a=1}^g k_a(\bar{x}, \bar{y})) \land \bigwedge_{b=1}^h l_b(\bar{x}, \bar{y})
\]

and, each \( F_i \) is a (non-strict) future-time formula of either the form

\[
\exists \bar{z}_i. \bigwedge_{j=1}^r m_j(\bar{x}, \bar{z}_i) \lor \exists \bar{z}_i. \diamond l(\bar{x}, \bar{z})
\]

![Figure 3: Temporal operators of FML](image-url)
where each \( k_a, l_b, m_j \) or \( l \) is a literal. Here, \( \bar{x}_i \) represents a vector of variables, \( x_{i_1}, x_{i_2}, \ldots, x_{i_m} \), so \( \forall x_i \) represents \( \forall x_{i_1}, \forall x_{i_2}, \ldots, \forall x_{i_m} \) and \( P_i(\bar{x}_i, \bar{y}_i) \) represents
\[
P_i(x_{i_1}, x_{i_2}, \ldots, x_{i_m}, y_{i_1}, y_{i_2}, \ldots, y_{i_n}).
\]
Thus, a MetateM program, \( P \), is a set of SNF\(_f\) rules. Note that most versions of MetateM, including the ones used here, follow the restriction outlined in [BFG*89], whereby references to environment predicates are not allowed in any \( F_i \). This is because, intuitively, a component cannot ensure that its environment will behave as it requires.

The process of executing a MetateM program corresponds to constructing a model for the appropriate set of SNF\(_f\) rules, in the presence of input from the environment. The model is constructed iteratively, starting from state 0. At each step, the program rules are consulted, and the future-time components of those rules whose past-time components (\( P_i \)) are satisfied by the partial model constructed so far\(^1\), are collected together. These constraints on the present and future properties of the model, together with any outstanding constraints generated on previous steps, are then used to construct the current state. Any outstanding constraints are passed on to the next step.

Notice from Figure 4 that the \( U \) operator is non-deterministic — the model construction mechanism described above therefore has a choice of how to construct the current state. The basic algorithm followed in our version of MetateM is to attempt to satisfy eventualities (i.e., \( a \) in formulae such as \( \mathcal{O} a \) and \( \mathcal{L} a \)) as soon as possible and, if any such formulae are passed forward from previous states, to attempt to satisfy the oldest outstanding eventuality first [BFG*89, FO92].

If a contradiction is generated (i.e., \textbf{false} is to be executed) then the system may backtrack to a previous choice-point and attempt to construct a model for the program in a different way. This backtracking undoes component actions and returns the execution to a previous choice point. If there are no more choice points left, the execution fails, signifying that the program is unsatisfiable.

Finally, though it is possible to execute sets of SNF\(_f\) rules\(^2\), as described above, in this paper we will use a particular subset of SNF\(_f\) which avoids some of the difficulties associated with the direct execution of quantifiers [FO92]. Thus, we remove the universal quantifier which appears \textit{inside} the past-time component of an SNF\(_f\) rule, giving the general form as
\[
\mathcal{O} \bigwedge_{i=1}^n \forall \bar{x}_i, P_i(\bar{x}_i) \Rightarrow \exists \bar{z}_j, F_i(\bar{x}_i, \bar{z}_j).
\]
Note that, as environment predicates are not allowed in the future-time components of such rules, then the component will have control over the value bound to the existential quantifier. The difference between component and \textit{internal} predicates is that when internal predicates are satisfied, this is recorded in the history of the computation, while when component predicates are satisfied, this is not only recorded, but their value is also output to the environment.

Now we consider the modelling of the configurations from §2 in MetateM.

## 4 Modelling Using MetateM

We assume that the definitions of the rigid predicates representing the network configuration and initial placement of trains are as described in §2. We now turn to the representation the transition rules of the system.

First, we introduce a new predicate \textit{can-move}(\(S, T, N\)) which is true when train \( T \) is able to move from station \( S \) to station \( N \). Thus, this predicate is constrained by the following rule.

\[
\forall S, \forall T, \forall N, \forall L, \quad \left[ \begin{array}{c} \text{has}(S, T) \land \text{next}(S, L, N) \land \\
\text{line}(T, L) \land \text{signal}(N, in, green) \end{array} \right] \Rightarrow \text{can-move}(S, T, N)
\]

\(^1\)It is in this process of ‘checking the past’ that input from the environment may be required as environment predicates can only appear in the \( P_i \) component.

\(^2\)By taking a suitable operational interpretation for quantifiers, for example.
∀S. ∀T. \left[ \text{station}(S) \land \text{has}(S, T) \land \text{line}(T, L) \land \text{next}(S, L, N) \land \text{signal}(N, \text{in}, \text{red}) \right] \Rightarrow \text{has}(S, T) \land \text{signal}(S, \text{in}, \text{red})

∀S. ∀T. ∀N. \left[ \text{station}(S) \land \text{can-move}(S, T, N) \right] \Rightarrow \text{has}(N, T) \land \text{signal}(N, \text{in}, \text{red}) \land \text{signal}(S, \text{in}, \text{green})

∀S. ∀T. ∀P. ∀U. \left[ \text{station}(S) \land \neg \text{has}(S, T) \land \neg \text{can-move}(P, U, S) \right] \Rightarrow \text{signal}(S, \text{in}, \text{green})

∀S. ∀T. ∀P. ∀U. ∀A. ∀B. \left[ \text{station}(S) \land \neg \text{has}(S, T) \land \text{signal}(S, A, \text{green}) \land \neg \text{can-move}(P, U, S) \land \text{signal}(S, B, \text{red}) \land \text{waiting}(S, B) \right] \Rightarrow \text{signal}(S, A, \text{red}) \land \text{signal}(S, B, \text{green})

Figure 5: Temporal description the global transition rules

The rules to move between one state of the network to another are presented in Figure 5. They are simply formalisations of the natural language rules presented at the start of §2 above. Thus, the statement

‘If a train is in a station, and its route is blocked by a red signal, then it remains in that station, and the station’s entry signal remains red’

can be represented by the rule 1 in Figure 5. The rule

‘If a train is in a station, and the entry signal of its next station is green, in the next state the train will be in the next station, with the entry signal of the next station set to red, and the entry signal of this station set to green’

can be formalised to rule 2 in Figure 5, while the constraint

‘if there is no train in a station, and there is no train about to enter the station, then the entry signal of the station will be green in the next state’

can be represented by rule 3 in Figure 5.

4.1 An Execution Step

To give the reader an impression of how the execution of such rules would proceed, we will present a single execution step for the system in a little more detail.

Starting from the initial configuration given in §2.1:

- The condition of rule (2) applies to the initial state with two variable assignments,
  \([S \mapsto \text{East}, T \mapsto \text{train 1}, N \mapsto \text{South}]\)
  and
  \([S \mapsto \text{West}, T \mapsto \text{train 2}, N \mapsto \text{North}]\).
- Both of these variable assignments are applied to the conclusion of rule (2), yielding the actions

  \(\text{has}(\text{South}, \text{train 1}) \land \text{signal}(\text{East}, \text{in}, \text{green}) \land \text{signal}(\text{South}, \text{in}, \text{red})\)

from the first variable assignment, and

\(\text{has}(\text{North}, \text{train 2}) \land \text{signal}(\text{West}, \text{in}, \text{green}) \land \text{signal}(\text{North}, \text{in}, \text{red})\)

from the second.

- Neither rule (1) nor rule (3) apply, so the actions from the previous step are the only ones to be taken.
- A new state is built, in which the flexible predicates receive the values assigned to them by the actions, and the rigid predicates retain the values assigned to them in the initial configuration. Hence the flexible component of the second state of the system of §2.1 is given by:

\(\text{signal}(\text{North}, \text{in}, \text{red}) \land \text{has}(\text{North}, \text{train 2}) \land \text{signal}(\text{East}, \text{in}, \text{green}) \land \text{signal}(\text{South}, \text{in}, \text{red}) \land \text{has}(\text{South}, \text{train 1}) \land \text{signal}(\text{West}, \text{in}, \text{green})\)

4.2 Rules for Merging

These rules are all that we need to describe the simple ring of §2.1. To handle the double ring of §2.2, we need to add some rules. The rules for the merging stations are essentially the same as those for the ordinary single-entry stations, except that

- care must be taken to ensure that when there is a train in the station, both entry signals are set to red.
- at most one entry signal is set to green at any time, to prevent trains entering the merging station from multiple lines simultaneously
- should one entry signal be red, and the other green, then a train should not be delayed behind the red signal if there is no train behind the green signal.

The existing rules are extended to cope with the multiple signals, and the following new rules are added. First we
add a rule to define a predicate \( \text{waiting}(S, Z) \) which is true when there is a train waiting for signal \( Z \) of station \( S \):

\[
\forall S, \forall T, \forall P, \forall U. \quad \left( \text{station}(P) \land \text{has}(P, T) \land \text{next}(P, L, S) \land \text{line}(T, L) \right) \Rightarrow \text{waiting}(S, Z)
\]

This new predicate is used in the new rule for merging stations (rule 4 in Figure 5). This rule states that if \( A \) is an entry signal set to green, and \( B \) is an entry signal set to red, and there is a train waiting behind \( B \) but not behind \( A \) then the order of the signals should be reversed.

Similar rules apply to the other cases, when trains are waiting behind both signals, and no trains are waiting.

### 4.3 Implementing the Rules

The rules that have been presented in the preceding sections have been implemented in an early version of the single-threaded executable temporal logic system. Owing to limitations of this interpreter, the conditions defining the \textit{can-move} and \textit{waiting} predicates were directly inserted into the state-changing rules, but in all other aspects, the implemented rules are those presented here. The ability to directly interpret the rules has enabled the behaviour of the various networks to be ‘tuned’ — a specification may be correctly implemented in a number of ways, but not all of them need possess all the non-functional properties one might desire. By being able to manipulate the rules and have immediate feedback via the actions taken by the interpreter, one can better understand not only what the specification is describing, but also the various options available to implement the description.

We now turn to an alternative, extended version of METATEM, in which sets of separate METATEM processes are executed using a particular concurrent operational model. We will show that the network configurations described above can be modelled by viewing each station as an individual METATEM process, and that this approach provides a different perspective to the one presented above.

### 5 Concurrent METATEM

Concurrent METATEM provides an operational model for communicating METATEM processes, based on the CMP Model [Fis92b]. This, in turn, provides a new way to view concurrent systems, where computation is carried out within groups of objects broadcasting, listening and executing asynchronously.

Communication between processes in Concurrent METATEM occurs when, during execution, the value of a predicate which is controlled by some other process is required. Thus, although the computation mechanism for a single object in the Concurrent METATEM system is provided by a METATEM-like computational engine, the three types of predicate used in §3 now take on a specific operational interpretation, with several categories of predicate corresponding to messages to and from the object.

- \textbf{Environment} predicates represent incoming messages.
  
  An environment predicate can be made true if, and only if, the corresponding message has just been received. Thus, a formula containing an environment predicate, such as \textit{request}('Y'), is only satisfied if a message of the form \textit{request}('b') has just been received (for some argument 'b').

- \textbf{Component} predicates represent messages broadcast from the object.
  
  When a component predicate is satisfied, it has the (side-)effect of broadcasting the corresponding message to the environment. For example, if the formula \textit{permit}('e') is satisfied, where \textit{permit} is a component predicate, then the message \textit{permit}('e') is broadcast.

- \textbf{Internal} predicates have no external effect.
  
  These predicates are used as part of formulæ participating in the internal computation of the object and, as such, do not correspond either to message-sending or message reception. This category of predicates may include various \textit{primitive} operations.

Once the object has commenced execution, it continually follows a cycle of reading incoming messages, collecting together the rules that ‘fired’ (i.e., whose left-hand sides are satisfied by the current history), and executing one of the disjuncts represented by the conjunction of right-hand sides of ‘fired’ rules, as described in §3.3. However, each METATEM object is considered as an asynchronously executing process where

1. The basic method of communication between objects is \textit{broadcast} message-passing.

2. Objects are not message driven — they begin executing from the moment they are created and continue even while no messages are received.

3. An object can change its interface (i.e., the messages that it recognises and how it manipulates incoming messages) dynamically.

We will not utilise (3) in this paper, but will depend on the fact that messages are \textit{broadcast}.

Individual objects have two components: an abstract \textit{interface definition}, and, an \textit{internal definition}. Networks of Concurrent METATEM objects communicate via broadcasted messages and individual objects only act upon certain identified messages. Thus, an object must be able to filter out messages that it wishes to recognise, ignoring all others. The definition of which messages an object recognises, together with a definition of the messages that an object may itself produce, is provided by the \textit{interface definition} for that particular object.
The interface definition for an object, for example ‘stack’, is defined in the following way

\[ \text{stack}(\text{pop}, \text{push})[\text{popped}, \text{stack-full}] \]

Here, \{ \text{pop}, \text{push} \} is the set of messages the object recognises, while \{ \text{popped}, \text{stack-full} \} is the set of messages the object might produce itself. Note that these sets of messages need not be disjoint – an object may broadcast messages that the object itself recognises. In this case, messages sent by an object to itself are recognised immediately. Note also that many distinct objects may broadcast and recognise the same messages. The internal definition of a Concurrent METATEM object is simply the set of METATEM rules associated with the object.

Objects may backtrack, with the proviso that an object may not backtrack past the broadcasting of a message. Consequently, in broadcasting a message to its environment, an object effectively commits the execution to that particular path. Thus, the basic operation of an object can be thought of as a period of internal execution, possibly involving backtracking, followed by appropriate broadcasts to its environment. (For more details, see [Fis92b].)

6 Modelling Using Concurrent METATEM

We will now show how the rail configurations described in §2 can be modelled in Concurrent METATEM. Rather than directly following the style of specification presented in §4, which represents a global view of the system, we will here present an object-based model. Thus, each station will be represented as a separate Concurrent METATEM object and, rather than having signals between stations, we will introduce a communications protocol which will enable stations to negotiate the passing of trains between stations. Thus, rather than having a global model of the system, the state of each station will be represented locally, with the station’s has and next constraints being private to that station. Thus, for example, the Concurrent METATEM object representing station ‘F’ in the double ring network described in §2.2 would contain the following initial constraints

\[ \text{station}(F) \quad \text{has}(\text{train} \ 3) \]

together with the following network details

\[ \text{next}(\text{upper}, A) \quad \text{next}(\text{lower}, G) \]

Each separate station will use the same set of METATEM rules. The only differences between stations will be their local next and has constraints. Since the constraints on these two predicates will be local to each station, we do not require a ‘station’ argument to each of these predicates, as the context will be obvious. The formulae representing initial placement and network configuration will thus be partitioned amongst the appropriate objects.

The METATEM rules will use all the predicates defined in §4, except for signal(). As we have replaced signal by a communications protocol, we use the extra predicates request(), permit(), and moved, whose effect can be described as follows.

request(S, T) — a request message which is broadcast asking for permission for train T to move to station S.

permit(S, T) — a message which is broadcast permitting train T to move to station S.

moved(S, T) — a message which is broadcast as train T moves to station S.

Thus, in order for a train to move, the station it occupies must request the ‘next’ station for permission to move the train on. A station can only give permission for a train to move to it if the station is currently empty and no permission has been given to another train.

6.1 Common Rules Amongst Stations

The interface definition for each station object is simply

\[ \text{station}(\text{request, permit, moved})[\text{request, permit, moved}] \]

showing that each station object both recognises and broadcasts instances of request, permit and moved messages.

The internal definition of each station is represented by the specification given in Figure 6\(^3\). These rules can be described as follows

1. If a station had a train, and the train has not moved, then the station still has that train.
2. If a station had a train, and a request to move the train on has not been made while it has had the train, then the station will make such a request to the train’s next station.
3. If a station had a train, and it has just received permission to move the train on to its next station, then the train can be moved to its next station.
4. If a train has just moved to a station, then that station has it.
5. If a train has just moved to a station, then that station cannot give permission to any other trains to move to that station.
6. If a station receives a request for a train to be moved to the station, then eventually it will give permission for this move.
7. A station can only ever give permission to one move at a time.

\(^3\)The variables ‘\(T\)’, ‘\(T1\)’ and ‘\(T2\)’ represent trains, while ‘\(M\)’ and ‘\(N\)’ represent stations.
7 Conclusions

We have described two approaches to the modelling of reactive systems based on the use of executable temporal logics. One approach, based upon a global system view, represents the standard specification approach; the other, based on a more modular view, provides an object-based specification for interacting reactive elements. Thus, the style of specification required, whether it be global or object-based, can be provided by one of the dialects of METATEM described above.

The ability to prototype specifications, and hence to change them and experiment with them, is essential, especially for complex reactive systems. As mentioned earlier, the ability to directly interpret the rules has enabled the behaviour of the various networks to be 'tuned' — a specification may be correctly implemented in a number of ways, but not all of them need possess all the non-functional properties one might desire. By being able to manipulate the rules and have immediate feedback via the actions taken by the interpreter, one can better understand not only what the specification is describing, but also the various options available to implement the description.

Although the examples given have been simple abstractions, they represent abstractions of real systems. Further, the components of these abstractions, i.e., the different types of station, can be combined to form larger, and perhaps more realistic, railway networks.

In conclusion, we have found this form of temporal logic to be a useful tool for specifying reactive systems and the ability to execute such specifications under more than one operational model has been invaluable in removing any ‘mistakes’ in the specifications, and in clarifying the core elements of the systems.

7.1 Future Work

The future work that we intend to undertake, based upon this study, can be split into two areas. The first involves proving that the specifications, both using METATEM and Concurrent METATEM, satisfy the criteria required of the railway networks which they model. The second area involves developing larger and more complex examples of railway systems, becoming more realistic and, necessarily, more difficult to specify without tool support, such as the prototyping provided by METATEM.

One particular area that we intend to investigate is the modelling of complex railway systems, where the elements (stations, trains, etc) may exhibit more ‘intelligent’ behaviour. In the network configurations given earlier, there was little room for choice in the routing of trains. We can think of more complex topologies, where a train might come to a branch in the line and either of the branches taken will eventually lead to the train’s final destination. However, as normal trains must call at many intermediate stations, such choices are usually determined by the train’s next station.

Consider a new type of train – the Express Train. Such a train is not constrained to run only on a particular line and only calls at specific stations. Crucially, alternative routes can now be taken and, when an express train reaches a branch in the line, it has a true choice of routes. In such a situation, the train\(^4\) can use information broadcast by sta-

\[^4\text{Note that, in our case it would be the station that directs the train up the appropriate route.}\]
tions further up the line to decide upon the best route. For example, the train might record the average delay between ‘request’ and ‘permit’ messages on each possible route, or it might just calculate the ratio of ‘request’ messages to stations on each route (in an attempt to gauge which line is the busiest). Locally, the train might check whether either of the possible next stations are responsive to requests or not.

In this way, the train (or station) is able to build up a model of the behaviour of stations that are not necessarily local. This type of system is readily modelled in Concurrent METATEM, where the ability to broadcast messages is crucial.

References


