Combining UML and formal notations for modelling real-time systems

Luigi Lavazza
CEFRIEL - Politecnico di Milano
P.za Leonardo da Vinci, 32
20133 Milano, Italia
+39 02 23993553
lavazza@elet.polimi.it

Gabriele Quaroni
Technology Reply srl
Via Ripamonti, 89
20139 Milano, Italia
+39 02 53576613
g.quaroni@reply.it

Matteo Venturelli
TXT e-solutions
Via Frigia, 27
20126 Milano, Italia
+39 02 257711
Matteo.Venturelli@txt.it

ABSTRACT

This article explores a dual approach to real-time software development. Models are written in UML, as this is expected to be relatively easy and economic. Then models are automatically translated into a formal notation that supports the verification of properties such as safety, utility, liveness, etc. In this way, developers can exploit the advantages of formal notations while skipping the complex and expensive formal modelling phase.

The proposed approach is applied to the Generalised Railroad Crossing (GRC) problem, one of the best known benchmarks proposed in the literature. A UML model of the GRC is built, and then translated into TRIO (a first order temporal logic). The resulting specification properties are tested by a history checking tool which exploits the formality of TRIO.

The work described here highlights the shortcomings of UML as a real-time modelling language, proposes enhancements and workarounds to overcome UML limitations, and demonstrates the viability of using UML as a front-end for a formal real-time notation. By translating the GRC model into TRIO, we also give formal semantics to some of the UML constructs.

Keywords
Real-time software, formal methods, UML.

1. INTRODUCTION

Formal methods have been successfully employed in the development of several real-time software systems, where features like safety need to be formally proved. However, formal notations are generally considered too difficult or too expensive to be used in "ordinary" real-time software development, and it is a matter of fact that they are not widely employed. On the contrary, UML [5] is becoming extremely popular, essentially because it is a semi-formal notation relatively easy to use (being mostly graphical), and because it is provided with tools that support (although to a limited extent) code generation. However, UML was not conceived for modelling real-time software: its application in the real-time domain is limited by the lack of constructs to express time-related constraints and properties, as well as by the lack of formal semantics.

This article explores a dual approach to real-time software development. In a first phase models are written in UML. This is expected to be a relatively economic activity, since UML does not require a big learning effort, and it is well supported by tools. In a second phase UML models are automatically translated into one or more formal notations, which provide support to activities such as the verification of properties (like liveness, safety, utility, …), test case generation, etc. In this way, developers exploit the advantages of formal notations while skipping the complex and expensive formal modelling phase. The optimal situation is when the derived models can be used by fully automatic tools: the modeller gets the results without even being aware of the underlying formal methods.

As far as the formal notations are concerned, there are several possible candidates, such as the different extensions of the basic model of timed automata, e.g., linear hybrid systems, and timed automata with integer variables or multilabels. Actually, it can be useful to translate the same UML model into two or more notations, for instance into a declarative notation suited to support formal proofs, and into an operational one supporting simulation. The research activity from which this paper was derived is actually addressing the translation of UML models into TRIO, a first order theory augmented with a temporal domain that includes basic arithmetic [2] and Kronos timed automata, which come with a tool for model checking [8]. In this paper we cover only the usage of TRIO, because of space reasons, and because the integration with Kronos still needs some work.

The proposed approach is applied to the Generalised Railroad Crossing (GRC) problem [3], one of the best known benchmarks proposed in the literature. The GRC was used both to understand the limitations of UML as a real-time modelling language, and as a test-bed for verifying the viability of the proposed approach: a UML model of the GRC was built, and then translated into TRIO.

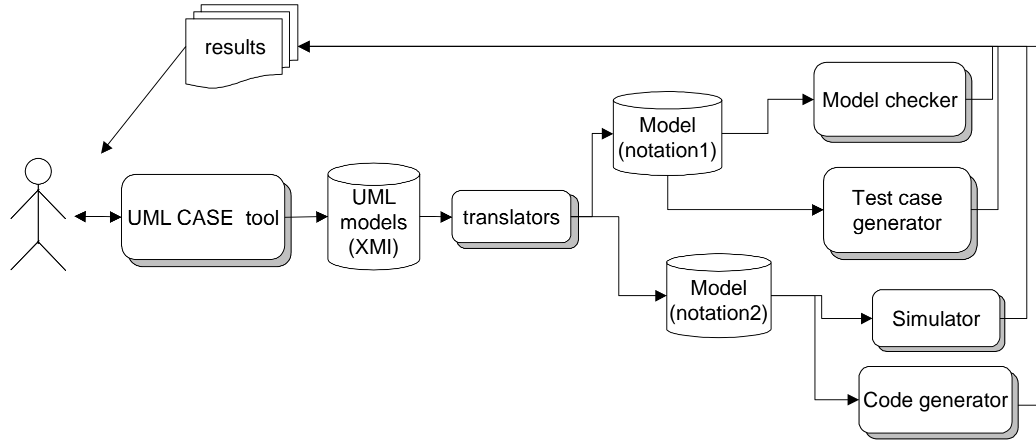

Figure 1. The envisaged environment.

More precisely, the translation does not generally need to take into consideration the whole UML models: here we consider only state diagrams and –to some extent– class diagrams. The resulting specification properties were tested by a history checking tool which exploits the formality of TRIO. Of course it would be possible to exploit several other TRIO tools [7][11][12], but this was out of the scope of the research. In fact, the main goal of the research underlying the work presented here is to demonstrate the viability of a development process based on UML modelling and the translation of the resulting models into several different notations. The correctness and effectiveness of the formal methods that can then be used is given for granted.

The development environment that will originate from the work presented here (and which is currently being developed at CEFRIEL) is described in Figure 1.

In order to validate the proposed approach, we applied it to the most critical part of a real-time system controlling a medical device. We do not report this experience here in detail for space reasons.

The paper is organised as follows: Section 2 introduces the GRC problem, Section 3 illustrates the corresponding UML model –together with the extensions of the language– and properties. Section 4 deals with the translation of the UML model into TRIO. Section 5 discusses the proof of the model properties. Section 6 briefly accounts for related work, while in Section 7 some conclusions are drawn, and future work is sketched.

2. THE GRC PROBLEM

2.1 The definition

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R (see Figure 2). Trains travel through R on K tracks in both directions. Trains may proceed at different speeds, and can even pass each other. Only one train per track is allowed to be in R at any moment. Sensors indicate when each train enters and exits regions R and I. A gate function g from real times to the interval [0,90] describes the state of the gate, g(t)=0 indicating that the bar is down (gate closed) and g(t)=90 indicating that the bar is up (gate open). A sequence of "occupancy intervals" is also defined, where each occupancy interval is a maximal time interval during which one or more trains are in I.

Given two "positive tolerance" constants xi1 and xi2, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

- *Safety Property*: The gate is closed during all occupancy intervals.
- *Utility Property*: If t is not in any occupancy interval, nor within xi1 prior to an occupancy interval, nor within xi2 after an occupancy interval, then the gate is open.

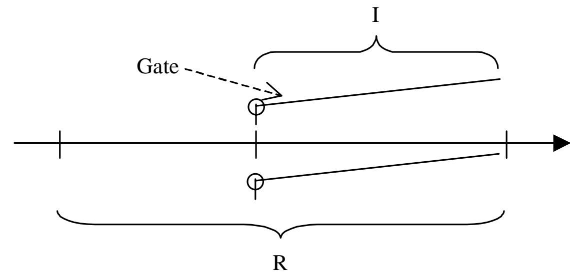

Figure 2. The railroad crossing regions.

Note that Figure 1 shows trains going in one direction. We adopt this simplification, since it has been demonstrated that the solution of the problem simplified in this way can be trivially extended to the general case.

2.2 Towards the solution of the GRC problem

We introduce relevant points in the interest region (see Figure 3):

- point RI indicates the position of the entrance sensor;
- RO indicates the position of the exit sensor;
- II indicates the position of the sensor which detects trains entering region I.

RI-RO defines zone R and II-RO defines zone I.

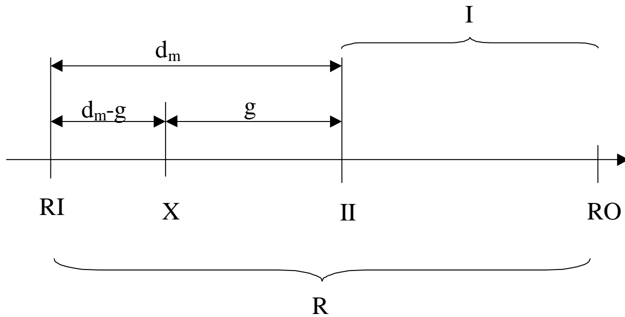

Figure 3. Annotated GRC.

Temporal constants describe maximum and minimum times for crossing the various zones, as well as the gate opening and closing times:

- dm: minimum time taken by a train to cross RI-II zone;
- dM: maximum time taken by a train to cross RI-II zone;
- hm: minimum time taken by a train to cross zone I (i.e., II-RO zone);
- hM: maximum time taken by a train to cross zone I (i.e., II-RO zone);
- g is the time taken by the bars of the gate for moving from the completely open to completely closed position (or vice versa).

Point X is thus defined as follows: when a train enters zone X-II it is time to start closing the gate, so that bars will be completely lowered when the train arrives at II (i.e., when the train enters the crossing zone I, or II-RO). We name RI-X and X-RO zones “Safe zone” and “Critical zone”, respectively. The exact position of X depends on the speed of each train, which is not known precisely. Thus the system cannot determine the right moment when a given train is at point X. However, it is clear that if we make the system safe for the fastest train, it will be safe also for the other trains. In order to have the gate closed when the fastest trains arrive at II, we must begin to close the gate dm-g seconds after the train entered region R. In this way when the fastest trains arrive at II the bars will be down and the crossing will be safe. Of course, the system will be safe for the slower trains as well.

In order to assure the safety property, the bars must be raised only when the critical zone is empty. Conversely, in order to assure the utility property, the system must start opening the gate *as soon as* the critical zone becomes empty.

3. MODELLING THE GRC WITH UML

3.1 The class diagram

We start modelling the GRC by defining the static structure of the model, by means of a class diagram (Figure 4). The following classes are defined:

- Gate and Train position, belonging to the problem domain, represent the gate and the position of the trains in each track within the interest region, respectively;
- Crossing represents both the crossing region, belonging to the problem domain, and the controller, belonging to the solution domain.

It can be noticed that the proposed model is oversimplified: for instance, dm, dM, etc. should be functions of train speed and region lengths. Nevertheless, we decided not to represent items like trains, tracks, sensors, etc., in order to keep the model as simple as possible, while focusing on the specification of the real-time behaviour of the system. It is easy to enhance the model in order to include all the omitted details, and in order to reflect more faithfully the structure of the real system.

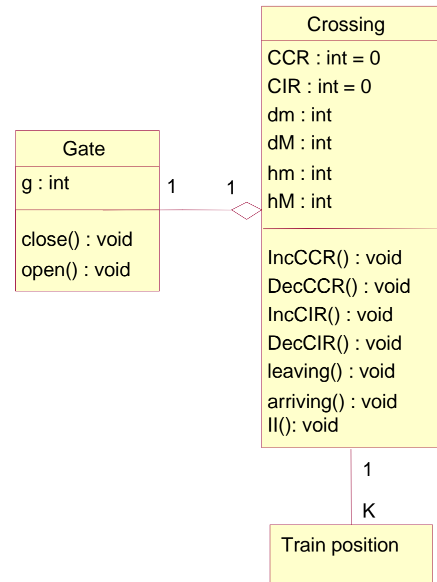

Figure 4. The class diagram of the GRC model.

The Gate class is characterised by the constant opening and closing time g. The Gate class provides two methods, open and close, for opening and closing the gate, respectively.

The Train position class represents the position of trains in each track of the interest region R. We are not interested in the exact position of trains, therefore the position is defined in an abstract way: out of region R, in RI-X, in X-II, in II-RO (see also Figure 5). The position of the train is determined according to the information provided by sensors, and taking into account the flowing of time. Whenever a train position changes (i.e., a train leaves a region and enters another one) this event is communicated to the controller, which reacts by sending appropriate commands to the gate. According to the specifications, there will be exactly K instances of this class, representing the trains travelling on the K tracks of R. Such a constraint is represented by specifying the number of instances involved in the relation linking the Crossing objects to the Train position objects.

The Crossing class has only one instance, which represents the current situation and the criteria to be followed in sending

commands to the gate. The controller must always know how many trains are in the interest region, and how they are distributed into the different zones. To achieve this purpose we introduced the attributes CCR and CIR, which count how many trains are in the critical region and in zone I, respectively. Both counters are initially set to zero, and are modified by increment and decrement methods. Invariants concerning counters can be expressed by means of Object Constraint Language (OCL) [5] expressions:

```
context Crossing inv :
    self.CCR>=0
context Crossing::IncCCR() inv :
    CCR@pre<K and CCR=CCR@pre+1
context Crossing::DecCCR() inv :
    CCR@pre>0 and CCR = CCR@pre-1

context Crossing inv :
    self.CIR>=0 and self.CIR<=self.CCR
context Crossing::IncCIR() inv :
    CIR@pre<K and CIR=CIR@pre+1
context Crossing::DecCIR() inv :
    CIR@pre>0 and CIR = CIR@pre-1
```

Constraints concerning the minimum and maximum traversing times are specified by means of OCL:

```
context Crossing inv :
   Self.dM >= Self.dm
   Self.dm > Gate.g
   Self.hM >= Self.hm
   Self.hm > 0
```

3.2 State diagrams

The state diagram of Train position class is reported in Figure 5.

Initially the train is out of the interest region. When the sensor placed at RI sends its signal, the train is entering the safe zone. After dm-g seconds the fastest trains are at point X (i.e., entering the critical zone), thus we assume that the train is entering the Critical_zone, and the Crossing object is notified by means of an Arriving event. This transition is governed by a time-out, modelled by means of the After statement provided by UML [5].

Now we should specify that the train enters region I not earlier than g seconds nor later than dM-(dm-g) seconds after entering the critical zone. UML does not provide any means to specify that a transition cannot happen before a given time. Similarly, UML does not allow to say that a transition (or the corresponding event) must occur before a given time. In order to express such constraints we had to introduce the Close_to_Crossing state and the Error state. Note that these states were introduced only because of the limits of UML. If we modelled the GRC directly in TRIO [4] we would not need such “artificial” states, which make the model unnecessarily complex. Note also that the Error state here just indicates that the behaviour of the system deviated from the specifications. In designing a real system, we would wish to substitute the Error state with some exception handling state, for the sake of robustness.

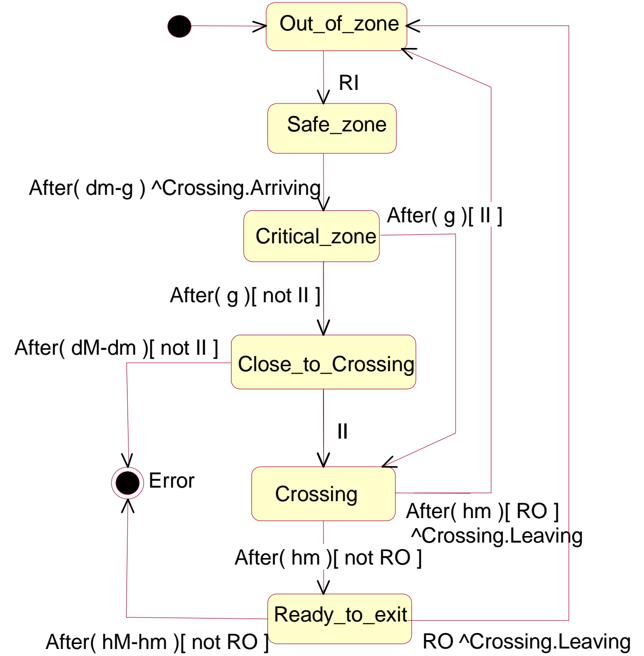

Figure 5. State diagram of class Train position.

Another problem is to specify what happens if a train reaches point II exactly after it has been in the Critical zone for g seconds (which actually happens, for the fastest trains). In such a case the train goes directly into the Crossing state, without passing for the Close_to_Crossing state. In order to express this constraint we borrowed UML conditions syntax, forcing their semantics to express the check of the occurrence of an event at a given moment (in plain UML conditions cannot contain references to events). The meaning we assign to this usage of the UML notation can be easily defined formally. For instance, the transition from Critical_zone to Close_to_Crossing in Figure 5 has the following meaning, expressed in temporal logic:

$$\forall t \ ((\exists \ \delta \ (\forall t_2 \ (t-\delta<t_2<t) \rightarrow \neg \ \text{Critical_zone}(t_2)) \land (\forall t_1 \ (t \leq t_1<t+g) \rightarrow \text{Critical_zone}(t_1)) \land \neg \ \text{II}(t+g)) \rightarrow \text{Close_to_Crossing}(t+g))$$

The meaning of the remaining transitions should be clear, thus they are not commented here.

The State diagram of class Gate is shown in Figure 6.

Most transitions are easy to understand, and presented no difficulty for modelling. Only the inversion of the movement deserves some comments. When the gate is closed (i.e., the bar is down), and an open command is received, the bars start to move upwards. If a new train enters the critical zone while the gate is still opening, the bars must start to move down immediately: this is modelled by a transition to state Inverted. According to the problem definition the bar will reach the close position again after a time equal to the time it has been opening. In order to represent this behaviour, we have to explicitly refer to the time when events occur: transition to state Down will occur after a time equal to the interval between the last open and close commands.

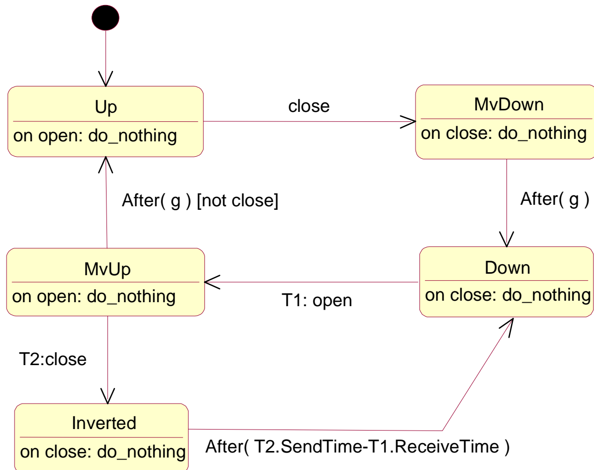

Figure 6. State diagram of class Gate

UML provides the keywords ReceiveTime and SendTime to indicate the time when an event was issued and received, respectively. However, we also need to indicate which of the open or close events we refer to (e.g., we are interested in the close event occurred while the gate was in state MvUp, not to the one occurred while in Up state). UML provides a labelling mechanism which can be used only in sequence diagrams and collaboration diagrams to identify individual transition *instances* [5]. State diagrams represent transition *types*, i.e., every transition may occur several times, that is, it may have several instances. We thus extended again UML syntax and semantics by applying labels to transitions in the state diagrams, intending that we always refer to the last occurrence. Thus, the transition from Inverted to Down occurs after a period equal to the interval between the last open command and the last close command.

The state diagram of class Crossing is reported in Figure 7.

Initially the Crossing is empty (this means that there are no trains in the critical zone). Whenever a train enters the critical zone the counter CCR is incremented. Whenever a train exits the critical zone the counter CCR is decremented. If a train arrives while the critical zone is empty the state is changed to non-empty, and a close command is sent to the gate. When the last train leaves the critical zone, the state is changed to empty, and an open command is sent to the gate.

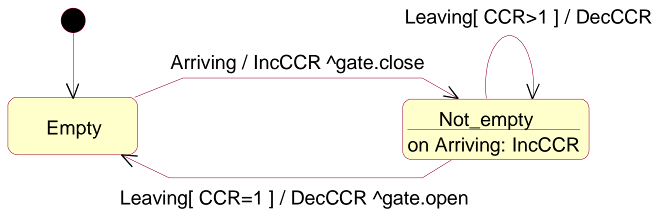

Figure 7. The state diagram of class Crossing.

3.3 Generalising the model

The model proposed above is suitable for a railroad crossing with just one track (i.e., only one train at a time is allowed to cross). It is easy to see that the model does not work for the generalised case, since all the instances of Train position would react to a single event: for instance, an event RI would cause all the objects of class Train position to go into the Safe_zone state.

It is relatively easy to modify the UML model in order to manage the general case:

- Sensors indicate the track where a relevant event occurred, i.e., events are parameterised with the track identifier: for instance II(2) means that a train travelling on track 2 entered zone I.
- Train position contains an attribute (named Track) indicating the corresponding track. Every instance of Train position will have a different value of the attribute Track.

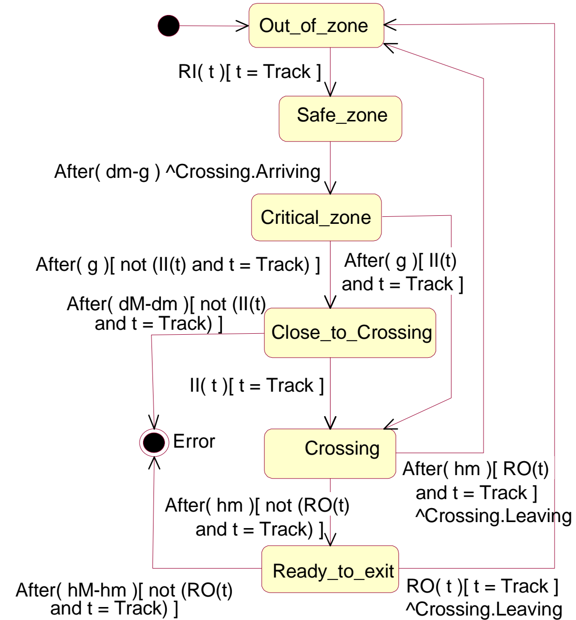

Figure 8. The state diagram of class Train position with parameterised events.

Transitions in the state diagram of class Train position are guarded by a condition that checks whether the event parameter matches the value of the Track attribute (see Figure 8). It is easy to see that the proposed solution works for any number of tracks. Of course, it requires that every track is equipped with its own sensors. It is possible to build several different models dealing with the generalised case, depending on the way sensors work. In fact we built a couple of alternative models, which are not reported here since they do not require additional extensions of UML.

3.4 Expressing model properties

Although the model discussed so far is sufficient to guarantee that the gate controller works properly, it is not detailed enough to express the safety property. In fact, we need to say that "when there is at least one train in zone I, the gate is closed", but we have no state corresponding to "zone I occupied". If we had defined such a state, the safety condition could be written in OCL as follows:

```
Context Crossing inv:
   Self.oclInstate(Zone_I_occupied) implies
Gate.oclInstate(Down)
```

Expressing the utility property involves some problems. Exploiting the state diagrams, it is only possible to state the following property:

```
Context Crossing inv:
    Self.oclInstate(Empty) implies
Gate.oclInstate(Mvup)or Gate.oclInstate(Up)
```

However, it is easy to see that the property above is too weak, as it does not require the gate to eventually reach the open position. The utility property should require that the gate is open whenever the crossing has been empty for at least g seconds (i.e., for the minimum time required to open the gate):

```
Context Crossing inv:
   Self.oclInstate(Empty) for at least g
implies Gate.oclInstate(Up)
```

Unfortunately, the sentence above is not a legal OCL statement, as OCL does not provide means to deal with time. In order to cope with this situation, we have to introduce a new state. Figure 9 shows the new state diagram for the Crossing class, extended with the states needed in order to express safety and utility properties.

Now, we can express the utility property in OCL as follows:

```
Context Crossing inv:
    Self.oclInstate(EmptyOpen) implies
Gate.oclInstate(Up)
```

It is interesting to note that on one hand the limits of OCL forced us to complicate the state diagram of class Crossing, on the other the new diagram represents more faithfully the real world, where the crossing can be in several states: closed with trains crossing (Zone_I_occupied), open with cars and pedestrians crossing (Empty_Open), closing (Zone_I_free) or opening (Empty_Opening).

Note also that the OCL expressions reported above apply to both the single track and multiple tracks models described above.

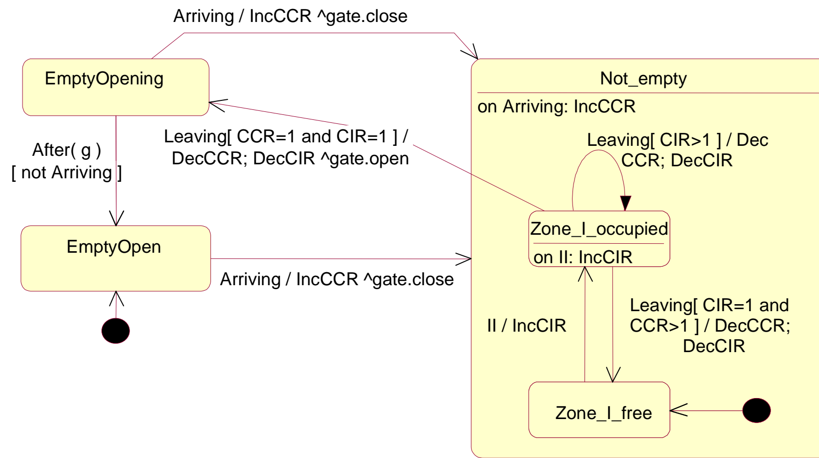

Figure 9. The modified state diagram of class Crossing

4. Translating the UML model into TRIO

4.1 Translation procedure

The translation was defined by identifying fragments of state diagrams, and defining the corresponding TRIO axioms. For space reasons, we do not report here all the fragment types mentioned above, instead we introduce just the more representative, especially those dealing with real-time features and constraints. The definition of TRIO is reported in [2]. In order to make the paper as self-contained as possible, a brief introduction to the most common constructs of the language is reported in the appendix.

4.1.1 Initial and final states and state persistence

The initial state of a class is characterised by the following TRIO axiom:

$Som(Initial_state \wedge AlwP(Initial_state))$

which states that at some time the object is in state Initial_state, and previously the object was always in that state. Note that in the GRC system all the objects have the same lifetime as the whole system. This fact allows us to say that an object was in a given state *always* before some time. In other systems, where objects are created and destroyed during the system lifetime, we should introduce a "Not_existent" state, indicating that the corresponding object does not exist (i.e., it has not yet been instantiated).

Similarly, final states are characterised by an axiom that indicates that when the final state is reached, it will be never left:

$Becomes(Final_state) \rightarrow AlwF(Final_state)$

States are persistent, i.e., an object remains in the same state until some relevant event occurs. For instance, the permanence of an instance of Train position (Figure 5) in state Out_of_zone is represented by the following axiom:

$Becomes(Out_of_zone) \rightarrow Until_{ie}(Out_of_zone, RI)$

which states that once the object enters state Out_of_zone (at time some t) it will remain in that state until event RI occurs (at some time t'>t). The meaning of the "ie" subscript notation is explained in the appendix.

4.1.2 Time-out transitions and event generation

An example of time-dependent transition is given by the transition from Safe_zone to Critical_zone in the state diagram of class Train position (Figure 5). It is represented by the following axioms:

$Becomes(Safe_Zone) \rightarrow Lasts_{ie}(Safe_Zone, dm\text{-}g)$

$Lasted_{ie}(Safe_Zone, dm\text{-}g) \rightarrow Becomes(Critical_zone) \wedge Arriving$

The first axiom says that the train will remain in the Safe_Zone state for dm-g seconds, while the second axiom states that after that period the train will enter the Critical_zone state, and at the same time the event Arriving is generated.

4.1.3 *Time-out transitions guarded by events*

The behaviour of an object may depend not only on time-outs, but also on the concurrent occurrence of events. An example of this kind of transition is given by the transitions from Critical_zone to Close_to_Crossing and Crossing in state diagram of class Train position (Figure 5). These transitions are represented by the following axioms:

```
Becomes(Critical_zone) → Lasts_ie(Critical_zone,g)
Lasted_ie(Critical_zone,g) ∧ ¬ II →
   Becomes(Close_to_Crossing)
Lasted_ie(Critical_zone,g) ∧ II →
   Becomes(Crossing)
```

These axioms state that a train remains in the critical zone for g seconds. After exactly g seconds, if event II occurs the train goes into the Crossing state, otherwise it goes into the Close_to_Crossing state.

4.1.4 *Transitions dependent on other transitions' times*

In some cases a transition occurs at a time which depends on other transitions' occurrence time. An example is given by the transition from Inverted to Down in the state diagram of class Gate (Figure 6). This transition is represented by the following axioms, where LastTime(E, t) indicates that the most recent occurrence of E was at time t:

```
Becomes(Inverted) ∧ LastTime(open,t1) ∧
   LastTime(close,t2) → Lasts_ie(Inverted,t2-t1)
Lasted_ie(Inverted, t2-t1) ∧ LastTime(open,t1) ∧
   LastTime(close,t2) → Becomes(Down)
```

The former axiom states that the permanence of the gate in state "Inverted" is equal to the time passed between the last pair of "close" and "open" events, i.e., t2-t1. The latter axiom states that after such period the gate is closed (bar down).

Note that we could have simplified these axioms, considering that the last occurrence of event close caused the state to become Inverted, i.e., t2 is always the current time in the axioms above.

Note also that here we consider ReceiveTime and SendTime to be equal (i.e., the transmission time of events is negligible). Had the transmission time been relevant, we would have modelled it explicitly, e.g., introducing a connecting element between the sensor and the gate.

4.2 Automatic translation

We developed a tool that automates the translation of UML models into TRIO axioms. The translator is written in Java, it takes in input the XMI [10] file generated by a UML CASE tool (we employed Rational Rose [13] and ARGO/UML [14]), parses it (by means of standard libraries provided by W3C) and applies the rules described above in order to generate TRIO axioms.

Currently the user needs to edit the XMI file in order to make it compliant with the extensions of UML we defined, since these extensions are not supported by available tools like Rose or Argo. This problem is dealt with in Section 7.

4.3 TRIO specifications

Here we report the TRIO specification of the first model introduced in Section 3, i.e., the one concerning the single track case. Then we also discuss how to deal with the generalised case.

The specifications reported here were obtained by means of the translator mentioned above.

4.3.1 *Class Train position*

```
{Possible states and mutual exclusion}
(Out_of_zone ∨ Safe_zone ∨ Critical_zone ∨
   Close_to_Crossing ∨ Crossing ∨ Ready_to_Exit ∨
   Error)
Out_of_zone → ¬ (Safe_zone ∨ Critical_zone ∨
   Close_to_Crossing ∨ Crossing ∨ Ready_to_Exit ∨
   Error)
Safe_zone → ¬ (Out_of_zone ∨ Critical_zone ∨
   Close_to_Crossing ∨ Crossing ∨ Ready_to_Exit ∨
   Error)
Critical_zone → ¬ (Out_of_zone ∨ Safe_zone ∨
   Close_to_Crossing ∨ Crossing ∨ Ready_to_Exit ∨
   Error)
{other formulae imposing mutual exclusion of
   states are omitted}
{Transitions}
Som(Out_of_zone ∧ AlwP(Out_of_zone))
UpToNow(Out_of_zone) ∧ RI → Becomes(Safe_Zone)
Becomes(Safe_Zone)→ Lasts_ie(Safe_Zone,dm-g)
Lasted_ie(Safe_Zone,dm-g) → Becomes(Critical_zone)
   ∧ Arriving
Becomes(Critical_zone) → Lasts_ie(Critical_zone,g)
Lasted_ie(Critical_zone,g) ∧ ¬ II →
   Becomes(Close_to_Crossing)
Becomes(Close_to_Crossing) →
   Until_ie(Close_to_Crossing, II ∨
   Lasted_ie(Close_to_Crossing,dM-dm))
Lasted_ie(Critical_zone,g) ∧ II →
   Becomes(Crossing)
Lasted_ie(Close_to_Crossing,dM-dm) ∧ ¬ II →
   Becomes(Error)
Becomes(Error) → AlwF(Error)
UpToNow(Close_to_Crossing) ∧ II →
   Becomes(Crossing)
Becomes(Crossing) → Lasts_ie(Crossing,hm)
Lasted_ie(Crossing,hm) ∧ RO → Becomes(Out_of_zone)
   ∧ Leaving
Lasted_ie(Crossing,hm) ∧ ¬ RO →
   Becomes(Ready_To_Exit)
Becomes(Ready_To_Exit)→ Until_ie(Ready_To_Exit, RO
   ∨ Lasted_ie(Ready_To_Exit,hM-hm))
Lasted_ie(Ready_To_Exit,hM-hm) ∧ ¬ RO →
   Becomes(Error)
UpToNow(Ready_To_Exit) ∧ RO →
   Becomes(Out_of_zone) ∧ Leaving
Becomes(Out_of_zone) → Until_ie(Out_of_zone,RI)
```

4.3.2 *Class Gate*

```
{Possible states and mutual exclusion}
(Up ∨ MvDown ∨ MvUp ∨ Down ∨ Inverted )
Up → ¬ (MvDown ∨ MvUp ∨ Down ∨ Inverted )
MvDown → ¬ (Up ∨ MvUp ∨ Down ∨ Inverted )
MvUp → ¬ (Up ∨ MvDown ∨ Down ∨ Inverted )
Down → ¬ (MvDown ∨ MvUp ∨ Up ∨ Inverted )
Inverted → ¬ (MvDown ∨ MvUp ∨ Down ∨ Up )
```

```
{Transitions}
Som(Up ∧ AlwP(Up))
UpToNow(Up) ∧ Close → Becomes(MvDown)
Becomes(MvDown) → Lasts_ie(MvDown,g)
Lasted_ie(MvDown,g) → Becomes(Down)
Becomes(Down) → Until_ie(Down,Open)
UpToNow(Down) ∧ Open → Becomes(MvUp)
Becomes(MvUp) → Until_ie(MvUp, Close ∨
   Lasted_ie(MvUp,g))
UpToNow(MvUp) ∧ Close → Becomes(Inverted)
Becomes(Inverted) → Lasts_ie(Inverted,t2-t1) ∧
   LastTime(open,t1) ∧ LastTime(close,t2)
Lasted_ie(MvUp,g) ∧ ¬ Close → Becomes(Up)
Becomes(Up) → Until_ie(Up,Close)
Lasted_ie(Inverted, t2-t1) ∧ LastTime(open,t1) ∧
   LastTime(close,t2) → Becomes(Down)
```

4.3.3 Class Crossing

```
{Possible states and mutual exclusion}
Empty_Opening ∨ Empty_Open ∨ Not_Empty
Empty_Opening → ¬ Empty_Open ∧ ¬ Not_Empty
Empty_Open → ¬ Empty_Opening ∧ ¬ Not_Empty
Not_Empty → ¬ Empty_Open ∧ ¬ Empty_Opening
{Substates of Not_Empty}
Not_Empty ↔ Zone_I_occupied ∨ Zone_I_free
Zone_I_occupied → ¬ Zone_I_free
Zone_I_free → ¬ Zone_I_occupied
{Operations}
IncCCR ∧ CCR(X) → Becomes(CCR(X+1))
DecCCR ∧ CCR(X) → Becomes(CCR(X-1))
Becomes(CCR(X)) → Until_ie(CCR(X),IncCCR ∨ DecCCR)
CCR(X) → ¬ CCR(Y) ∧ X ≠ Y
{axioms for CIR, IncCIR and DecCIR are omitted,
   being similar to those concerning CCR, IncCCR
   and DecCCR}
{Transitions}
Som(Empty_Open ∧ AlwP(Empty_Open))
Empty_Open ∧ AlwP(Empty_Open) → CCR(0) ∧ CIR(0)
UpToNow(Empty_Open) ∧ Arriving →
   Becomes(Zone_I_free) ∧ Close ∧ IncCCR
UpToNow(Empty_Opening) ∧ Arriving →
   Becomes(Zone_I_free) ∧ Close ∧ IncCCR
Not_Empty ∧ Arriving → IncCCR
Becomes(Zone_I_free) → Until_ie(Zone_I_free, II)
UpToNow(Zone_I_free) ∧ II →
   Becomes(Zone_I_occupied) ∧ IncCIR
Zone_I_occupied ∧ II → IncCIR
Zone_I_occupied ∧ Leaving ∧ CCR(X) ∧ X>1 → DecCCR
   ∧ DecCIR
Becomes(Zone_I_occupied) →
   Until_ie(Zone_I_occupied, Leaving ∧ CIR(1))
UpToNow(Zone_I_occupied) ∧ Leaving ∧ CCR(1) ∧
   CCR(1) → Becomes(Empty_Opening) ∧ DecCCR ∧
   DecCIR ∧ Open
UpToNow(Zone_I_occupied) ∧ Leaving ∧ CCR(X) ∧ X>1
   → Becomes(Zone_I_free) ∧ DecCCR ∧ DecCIR
Becomes(Empty_Opening) → Lasts_ie(Empty_Opening,
   g)
Lasted_ie(Empty_Opening, g) → Becomes(Empty_Open)
Becomes(Empty_Open) → Until_ie(Empty_Open,
   Arriving)
```

4.3.4 Dealing with the generalised case

The TRIO specifications for the generalized case are derived from the UML model of the generalized railroad crossing system (i.e., the model containing the state diagram given in Figure 8). With respect to the single track case, the translation has to take into account a few more issues, which are illustrated below.

When there is just one track, at most one increment (or decrement) of CCR at a time is required. This situation is easily managed by means of sentences like the following:

```
UpToNow(Empty_Open) ∧ Arriving →
   Becomes(Zone_I_free) ∧ Close ∧ IncCCR
```

When multiple tracks are considered, it is necessary to take into account what happens if multiple trains enter the critical zone at the same time. It is clear that the formula above would not work, since CCR would be incremented only once.

In the general case (i.e., multiple tracks, each equipped with one sensor), several trains, travelling on different tracks, may activate at the same time the corresponding sensors. This case can be managed easily if we add the Track identifier as a parameter of event Arriving. This requires a trivial modification of the state diagram of class Train position reported in Figure 8: e.g., the transition from Safe_zone to Critical_zone should be labelled "After(dm-g)^Crossing.Arriving(Track)". In fact, the parallel activation of several sensors can be easily recognised and dealt with by expressions like the following, which indicate that at any moment CCR must be incremented by the number of trains which are crossing point X and decremented of the number of trains which are crossing point RO at that moment:

```
S(0,0) ∧ ∀j,v S(j,v) ↔ (S(j-1,v-1)∧Arriving(j)
   ∨ S(j-1,v+1)∧Leaving(j) ∨ S(j-1,v)∧
   (¬Arriving(j) ∧ ¬Leaving(j)))
CCR(X) ∧ S(K,Y) → Becomes(CCR(X+Y))
```

The first axioms assures that at any moment t S(K,Y) is true, where K is the number of tracks and Y is the number of Arriving events occurring at time t minus the number of Leaving events occurring at t. The second axiom states that the number of trains in the critical region (CCR(X)) is modified as indicated by S(K,Y).

The same specification can be written in a non recursive way:

```
Arriving(N) → DeltaC(N,1)
Leaving(N) → DeltaC(N,-1)
¬Arriving(N) ∧ ¬Leaving(N) → DeltaC(N,0)
CCR(X) ∧ DeltaC(1,X1) ∧ DeltaC(2,X2) ∧ … ∧
   DeltaC(n,Xn)→ Becomes(CCR(X+X1+X2+…+Xn))
```

This is the only type of substantial modifications required to deal with the generalised case.

4.4 Dealing with non-determinism

It is generally recognized that some amount of non-determinism can be useful in specifications. Our approach does not take into account non-determinism in the sense that for each of our state diagrams there is always only one transition which can occur, and thus only one active state. Nevertheless, we considered the possibility of modelling non-determinism in time, for instance to express that a given transition may occur at any moment within a given time interval.

For instance, when modelling a sensor, we should say that signal II can follow signal RI after a time comprised between dm and

dM seconds. We represent this kind of constraints as illustrated by Figure 10.

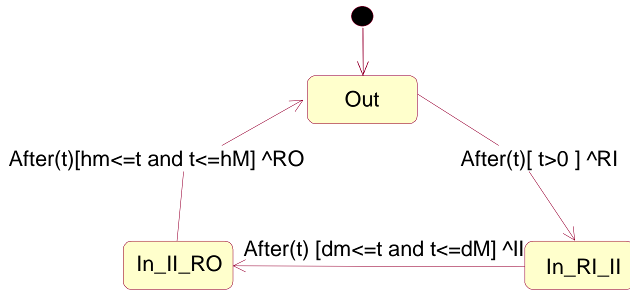

Figure 10. The state diagram of a sensor

5. Validation

As already mentioned in the introduction, the goal of the proposed approach is to allow UML models to be used as inputs for formal methods. In the case of TRIO there are several tools that can be used, for different purposes [7][11][12]. In order to validate our work with relatively little effort, we decided to test the properties of the model by means of TRIO-Matic. This is a tool, still under development at Politecnico di Milano, which is able to check whether a given history (i.e., an evolution of the system in a time interval) is consistent with a TRIO specification.

5.1 Testing the GRC model

By means of TRIO-Matic we proved that histories which verify the safety and utility properties are consistent with the specification derived from the UML model. On the contrary, histories which violate the properties were proved to be inconsistent with the specification. The safety and utility properties were obtained translating the OCL expressions into the following TRIO statements:

```
Zone_I_occupied → Down          {safety}
Empty_Open → Up                 {utility}
```

Of course, in this way we did not prove that the safety and utility properties always hold. However this is not a big issue, since our goal was to show that it is possible to derive from a UML model a formal specification, about which it is possible to reason with the help of automated tools.

5.2 An industrial case study

In order to further prove the viability of the approach we applied it to a piece of real-time software being developed by TXT e-solutions (http://www.txt.it). In particular, we modelled the program controlling the step motor of ACL8000, an automated blood analysis machine [9]. The program has to drive the motor and program timers controlling the acquisition of data while respecting the features of the motor, the required speed (determined by the time needed to analyse blood samples), the constraints posed by the nature of the samples, etc.

We proceeded as for the GRC: the model of the system was built in terms of class diagrams and state diagrams. Constraints and features, as well as the properties to be satisfied, were expressed by means of OCL. The model was then translated into TRIO, and by means of TRIO-Matic we were able to test the properties of the model against several histories. For additional details see [9].

The application of the proposed approach to the case study showed that the process is relatively straightforward and economic. The extended UML model was easily created by experienced UML users. The automatic translation made it possible to apply TRIO-Matic right away. The resulting confidence that the model was correct with respect to the requirements largely rewarded developers for the little additional effort required for writing histories and running TRIO-Matic.

6. Related work

UML is a widely adopted standard notation. Although not originally conceived for modelling real-time software, several efforts have been done to employ UML in the development of real-time software.

Douglass has shown how several features of UML can be exploited in the specification and design of real-time software [1]. However, he has not solved the problems connected with the lack of formal semantics of UML. Moreover, he has not tackled difficult problems which –like the GRC– cannot be modelled by means of plain UML.

Selic has proposed the Real-Time Object-Oriented Modeling (ROOM) language [6], which has been merged with standard UML to form a proposal of 'UML for real-time'. This language is being used in the Rose for Real-Time tool. ROOM favours the design and coding activities, at the expenses of the specification phase. The language provides rich information on the structure of the system thanks to the facilities provided to describe active objects and their communications, and tools are available to translate the models into executable code which behaves like the models. The limits of ROOM are in the lack of constructs to describe non-trivial time constraints like those occurring in the GRC, as well as in the lack of formal semantics (semantics is hardwired in the translators). As a consequence, a ROOM model cannot be analysed in order to prove whether the model has desirable properties such as safety.

The limits of UML are well known by the Object Management Group (OMG) which has issued two requests for proposals concerning a "UML Profile for Scheduling, Performance, and Time" (in other words, UML for real-time) and "Action Semantics for UML".

7. Conclusions and future work

The work presented here is a first step towards the definition of a method for real-time software development. The goal is to allow developers to specify and design real-time systems by means of UML models, which are then automatically converted into equivalent formal descriptions of the system. Being formal, such descriptions are amenable of being understood by several kinds of tools, such as model checkers, simulators, test case generators, etc. In particular, the models can be analysed in order to guarantee that they have some desired properties. The analysis activities are carried out at an early stage of development (i.e., as soon as models are available), are reliable (being based on formal descriptions), and are cheap (being largely automated).

In order to make such a method actually usable in practice, UML was extended with constructs for real-time systems modelling. Then, the translation of UML into a formal notation (TRIO) was defined, and a tool performing the translation of XMI into TRIO was built. The translation achieves a double goal: it provides UML (at least state diagrams and –partly– class diagrams) with

formal semantics, and allows the developer to exploit the available tools supporting the target formal notation.

A potential problem with our method is that current UML tools do not support our extensions. Therefore we have to manipulate "by hand" the XMI files produced by tools, in order to add our extensions. This may easily lead to inconsistency and double maintenance problems. In order to avoid these problems we are modifying an existing tool (namely Argo/UML [14]) in order to make it support our extended version of UML.

Another activity that we are currently carrying out is the study of the translation of UML models into the timed automata supported by Kronos. The goal is to perform effective automated model checking, thus providing early indications of a model properties at a very low cost (Kronos is also equipped with on-the-fly model checking capabilities, which allow to process large models efficiently).

8. Acknowledgements

The work described here was carried out as part of the ITEA project DESS (Software Development Process for Real-Time Embedded Software Systems). Project DESS is partly supported by MURST.

The authors wish to thank Vincenzo Martena for his help to use Trio-Matic.

The work described here was carried out while Gabriele Quaroni was with TXT e-solutions.

9. References

[1] Douglass B. P. Real-Time UML, Addison Wesley, 1998.

[2] Ghezzi C., Mandrioli D., Morzenti A., TRIO, a logic language for executable specifications of real-time systems. The Journal of Systems and Software, 12, 2 (May 1990).

[3] Heitmeyer C.L., Jeffords R.D., Labaw B.G., Comparing different approaches for Specifying and verifying Real-Time Systems, in Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software (New York May 1993), 122-129.

[4] Mandrioli D., Morzenti A., Pezzè M., San Pietro P., Silva S., A Petri Net and Logic Approach to the Specification and Verification of Real-Time Systems, Trends in Software n.5, Heitmeyer C. and Mandrioli D. Eds., Wiley, 1996

[5] OMG, Unified Modeling Language Specification Version 1.3, First Edition, March 2000.

[6] Selic B., Gullekson G., Ward P.T., Real-Time Object-Oriented Modeling, Wiley, 1999.

[7] Mandrioli D., Morasca S., Morzenti A., Generating test cases for real-time systems from logic specifications, ACM-TOCS - Transactions on Computer Systems, 13,4 (November 1995)

[8] Yovine S., Kronos: A Verification Tool for Real-Time Systems. In Springer International Journal of Software Tools for Technology Transfer, 1, ½ (October 1997).

[9] Quaroni G. and Venturelli M., Utilizzo integrato di UML e TRIO per la specifica di sistemi Real Time, Politecnico di Milano, February 2001, Thesis (in Italian).

[10] OMG, XML Metadata Interchange (XMI) Specification, Version 1.0, June2000, available at http://www.omg.org/technology/documents/formal/xml_metadata_interchange.htm

[11] Coen-Porisini A., Pradella M., San Pietro P., A finite domain semantics for testing temporal logic specifications, in Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems 5th International Symposium (Eds. A.P. Ravn and H. Rischel), LNCS Vol. 1486, Springer, September 1998.

[12] Felder M., Morzenti A., Validating real-time systems by history-checking TRIO specifications, ACM TOSEM-Transactions On Software Engineering and Methodologies, 3,4 (October 1994).

[13] http://www.rational.com/

[14] http://argouml.tigris.org/

Appendix TRIO: a shortest overview

TRIO is a first order theory augmented with a temporal domain that includes basic arithmetic (total order, metrics, and operations). The temporal operator Dist is provided with an intuitive semantics that states that for a given formula W, Dist(W, t) means that W is true at a time instant whose distance is exactly t time units from the current instant, i.e., the instant when the sentence is claimed. The current instant is implicit in TRIO formulas.

On the basis of the fundamental operator Dist many other temporal operators can be defined as derived operators:

Futr(F,d) $\stackrel{def}{=}$ d≥0 ∧ Dist(F,d) {future}

Past(F,d) $\stackrel{def}{=}$ d≥0 ∧ Dist(F,-d) {past}

Lasts(F,d) $\stackrel{def}{=}$ ∀d'(0<d'<d → Dist(F,d')) {F holds over a period of length d}

Lasted(F,d) $\stackrel{def}{=}$ ∀d'(0<d'<d → Dist(F,-d')) {F held over a period of length d in the past}

Until (A_1,A_2) $\stackrel{def}{=}$ ∃t (t>0 ∧ Futr (A_2,t) ∧ Lasts(A_1,t)) {A_1 holds until A_2 becomes true}

Since (A_1, A_2) $\stackrel{def}{=}$ ∃t (t>0 ∧ Past (A_2,t) ∧ Lasted(A_1,t)) {A_1 held since A_2 became true}

Alw(F) $\stackrel{def}{=}$ ∀d Dist(F, d) {F always holds}

AlwF(F) $\stackrel{def}{=}$ ∀d (d>0 → Dist(F, d)) {F will always hold in the future}

SomF (A) $\stackrel{def}{=}$ ∃d(d>0 ∧ Dist(F,d)) {Sometimes in the future F will hold}

Som (A) $\stackrel{def}{=}$ ∃d Dist(F,d) {Sometimes F held or will hold}

UpToNow (F) $\stackrel{def}{=}$ ∃δ (δ>0 ∧ Past(F,δ) ∧ Lasted (F,δ)) {F held for a nonnull time interval that ended at the current instant}

Becomes (F) $\stackrel{def}{=}$ F ∧ UpToNow(¬F) {F holds at the current instant but it did not hold for a nonnull interval that preceded the current instant}

LastTime (F,t) $\stackrel{def}{=}$ Past(F,t) ∧ ($Lasted_{ei}$ (¬F,t)) {F occurred for the last time t units ago}

Notice that the operators expressing a duration over a time interval (for example Lasts) are given definitions where the extremes of the specified time interval are excluded, i.e. the interval is open.

Operators including either one or both of the extremes can be easily derived from the basic ones we listed above. For notational convenience, in order to indicate inclusion or exclusion of the lower or upper bound of the interval, we append to the operator name two subscripts, 'i' or 'e', respectively. For example, $Lasts_{ie}$ and $Lasts_{ei}$ are defined as follows.

$$Lasts_{ie}(A,t) \stackrel{def}{=} A \wedge Lasts(A,t)$$

$$Lasts_{ei}(A,t) \stackrel{def}{=} Lasts(A,t) \wedge Dist(A,t)$$

The former axiom states that A is true at the current instant and for an open interval of duration t.

The latter statement states that A is true for an open interval of duration t and at a time instant whose distance is exactly t time units from the current instant.