A Brief Introduction to TRIO

Abstract

This document introduces the main characteristics of the TRIO specification language. First, the basic logic features of the language are described; then, its object-oriented extensions are presented.

1 TRIO in-the-small

TRIO [2] is a first order temporal logic language that supports a linear notion of time. Besides the usual propositional operators and quantifiers, one may compose formulas by using a single basic modal operator, called $Dist$, that relates the *current time*, which is left implicit in the formula, to another time instant: the formula $Dist(F, t)$, where F is a formula and t a term indicating a time distance, specifies that F holds at a time instant at t time units from the current instant.

A number of *derived temporal operators* can be defined from the basic $Dist$ operator through propositional composition and first order quantification on variables representing a time distance. Table 1 reports the formal definition of some TRIO derived operators [1]. Most of the operators are symmetrically defined with reference to the past and the future of the current instant.

TRIO is well suited to deal with both continuous and discrete time. What changes in the two cases is the domain over which time variables range (real or integer numbers)[2]. In the following the time domain is assumed to be continuous.

The natural tendency to describe systems in an operational way is supported by TRIO through the so-called *ontological constructs*, such as events and states. An *event* is a particular predicate that models instantaneous conditions such as a change of state or the occurrence of an external stimulus. A *state* is a predicate representing a property of a system. A state may have duration over a time interval.

The semantics of such constructs is defined by means of (built-in) TRIO axioms. In fact, a distinguishing feature of TRIO is that every high level concept is defined in terms of lower level ones down to the $Dist$ operator.

For example, an event E must satisfy the following (non-Zeno) behavior [1]:

$$UpToNow(\neg E) \wedge NowOn(\neg E).$$

[1] Notice that there exist versions of the temporal operators with explicitly included/excluded bounds - indicated with subscripts i/e, respectively. For instance, the definition of $Lasts_{ie}(F, t)$ is $\forall d\ (0 \leq d < t \rightarrow Dist(F, d))$.

[2] The definitions of some operators also change, for example $UpToNow$ and $NowOn$

Operator	TRIO Definition
$Past(A, d)$	$d > 0 \land Dist(A, -d)$
$Futr(A, d)$	$d > 0 \land Dist(A, d)$
$Som(F)$	$\exists d \; Dist(F, d)$
$Alw(F)$	$\neg Som(\neg F)$
$SomF(F)$	$\exists d \; (d > 0 \land Dist(F, d))$
$SomP(F)$	$\exists d \; (d > 0 \land Dist(F, -d))$
$AlwF(F)$	$\neg SomF(\neg F)$
$AlwP(F)$	$\neg SomP(\neg F)$
$Lasts(F, t)$	$\forall d \; (0 < d < t \rightarrow Dist(F, d))$
$Lasted(F, t)$	$\forall d \; (0 < d < t \rightarrow Dist(F, -d))$
$WithinF(F, t)$	$\neg Lasts(\neg F, t)$
$WithinP(F, t)$	$\neg Lasted(\neg F, t)$
$Until(F, G)$	$\exists d \; (d > 0 \land Lasts(F, d) \land Dist(G, d))$
$Since(F, G)$	$\exists d \; (d > 0 \land Lasted(F, d) \land Dist(G, -d))$
$UpToNow(F)$	$\exists d \; (d > 0 \land Lasted(F, d))$; $Dist(F, -1)$ if T is not dense
$NowOn(F)$	$\exists d \; (d > 0 \land Lasts(F, d))$; $Dist(F, 1)$ if T is not dense
$LastTime(A, d)$	$d \geq 0 \land Dist(A, -d) \land Lasted(\neg A, d)$
$NextTime(A, d)$	$d \geq 0 \land Dist(A, d) \land Lasts(\neg A, d)$
$Becomes(F)$	$F \land UpToNow(\neg F)$

Table 1: Derived Temporal Operators

A state S, on the other hand, obeys the following axiom:

$$
\begin{array}{ll}
& (UpToNow(S) \land NowOn(S) \land S) \\
\lor & (UpToNow(\neg S) \land NowOn(\neg S) \land \neg S) \\
\lor & (UpToNow(\neg S) \land NowOn(S)) \\
\lor & (UpToNow(S) \land NowOn(\neg S)).
\end{array}
$$

TRIO items (values, predicates, functions, events, states, etc.) are divided into time-independent (TI), whose value does not change during system evolution, and time-dependent (TD), whose value may change during system evolution.

2 Modular TRIO

For specifying large and complex systems, TRIO has the usual object-oriented concepts and constructs such as classes, inheritance and genericity. Classes denote collections of objects (class instances) that satisfy a set of axioms. Notice that TRIO, being a logic language, does not support object creation/destruction. Therefore, if one wants to model an entity having a limited lifetime he/she must simulate creation/destruction using other TRIO mechanisms, such as a time-dependent predicate that is true when the object exists, and false otherwise. In addition, TRIO objects do not have *a priori* a unique identifier to distinguish one object from the others. However, object identity

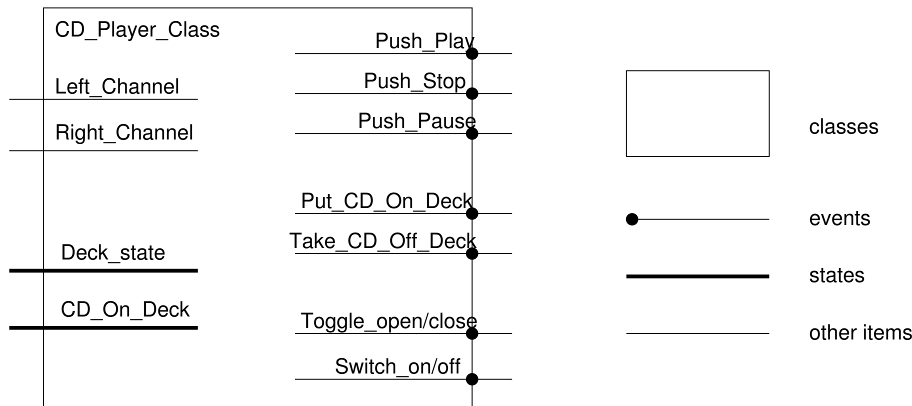

Figure 1: A TRIO simple class

can be modeled by introducing an item that represents the identity of the object and some axioms assuring that different objects have different identities.

Classes can be either *simple* or *structured* - the latter term denoting classes obtained by composing simpler ones. A simple class is defined through a set of axioms premised by a declaration of all items that are referred therein. Some of such items are *visible*, that is, they may be referenced from outside, in the context of a complex class whose instances include an instance of that class.

Suppose, for example, that we want to specify the behavior of basic CD players that allow only three operations: *play*, *stop* and *pause*, plus the obvious possibility to switch the player on/off and to open/close the CD deck. The graphical description of the class representing the CD players is given in Figure 1. The class includes seven events (recognizable by the dots on the lines, for example Push_Play, Put_CD_On_Deck, and Switch_on/off), two states (Deck_state and CD_On_Deck, represented by a bold line) and two generic predicates (Left_channel and Right_channel, drawn as simple lines).

All of the items of CD_Player_Class are in its interface, that is, they may be referenced from outside it in the context of a complex class that includes an instance of CD_Player_Class.

A class is completely defined through a set of axioms premised by a declaration of all items that are referred therein. These axioms describe the dynamic behavior of that class, while the graphical representation gives only a static idea of what items are included in the class. A possible definition for class CD_Player_Class might be, for example:

```
Class CD_Player_Class

signature:
visible: Left_channel, Right_channel,
         Push_Play, Push_Stop, Push_Pause,
```

```
          Put_CD_On_Deck, Take_CD_Off_Deck,
          Toggle_open/close, Switch_on/off,
          Deck_state, CD_On_Deck;
temporal domain: real;
domains:
  DeckState : {open, close, opening, closing};

items:
  TD total Left_Channel : real;
  TD total Right_Channel : real;

  state Deck_state (DeckState);
  state CD_On_Deck;

  event Push_Play;
  event Push_Stop;
  event Push_Pause;
  event Put_CD_On_Deck;
  event Take_CD_Off_Deck;
  event Toggle_open/close;
  event Switch_on/off;

axioms:
  /* axioms that define the behavior of the class */

end
```

Notice that state Deck_state has an argument of type enumerative, which represents not only when the deck is stopped in an open/close position, but also when it is moving; on the other hand, state CD_On_Deck has no arguments, it is simply true when a CD is on the deck, false otherwise. In addition, none of the events has an argument (which is no surprise since they simply represent a button pushed). Values (i.e. functions with arity zero in TRIO terms) Left_channel and Right_channel represent the sound wave that is transmitted by the CD player.

One possible axiom might state that it is possible to put/take away a CD on/from the deck only when this is open:

$$Put_CD_On_Deck \lor Take_CD_From_Deck \rightarrow Deck_state(open)$$

Similarly, we must define the behavior of state CD_On_Deck; this can be done using the following two axioms (we assume that at the beginning the CD player is empty):

$$\begin{aligned} CD_On_Deck \quad \leftrightarrow \quad & \neg Take_CD_From_Deck \land \\ & SomP_i(Put_CD_On_Deck) \land \\ & Since_{ei}(\neg Take_CD_From_Deck, Put_CD_On_Deck) \end{aligned}$$

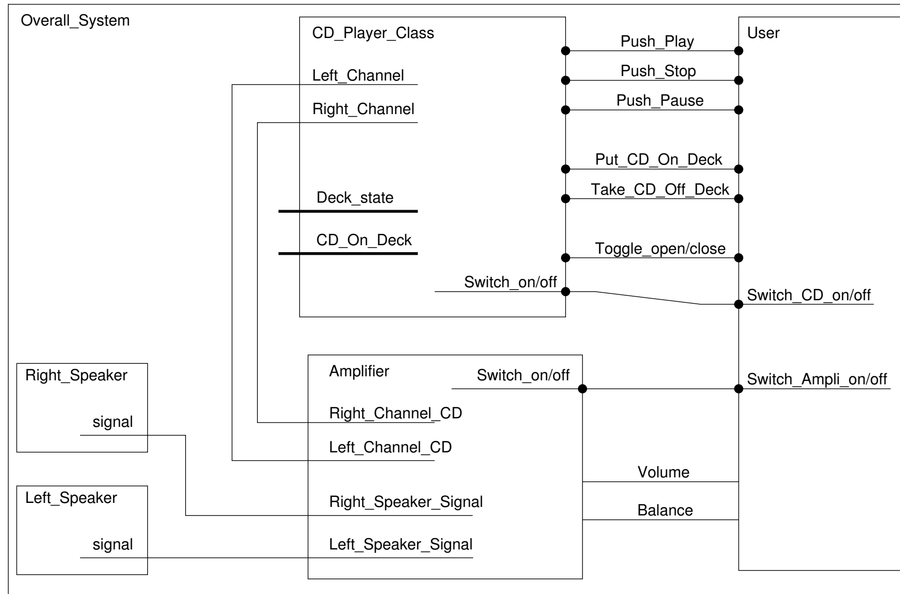

Figure 2: A TRIO structured class

Figure 2 depicts a situation in which a user operates a hi-fi composed of a CD player, an amplifier and two speakers. The signature definition of the corresponding class Overall_System is the following:

```
Class Overall_System
import: CD_Player_Class, Amplifier_Class,
        Speaker_Class, Hi-Fi_User_Class;

signature:
temporal domain: real;
modules:
  CD_Player : CD_Player_Class;
  Amplifier : Amplifier_Class;
  Left_Speaker : Speaker_Class;
  Right_Speaker : Speaker_Class;
  User : Hi-Fi_User_Class
connections:
  (direct CD_Player.Push_Play, User.Push_Play)
  (direct CD_Player.Switch_on/off, User.Switch_CD_on/off)
  (direct Amplifier.Switch_on/off, User.Switch_Ampli_CD_on/off)
  (direct Amplifier.Volume, User.Volume)
  (direct Amplifier.Right_Speaker_signal, Right_Speaker.signal)
  (direct Amplifier.Right_channel_CD, CD_Player.Right_channel)
  /* rest of connections not shown */
axioms:
```

```
  /* axioms that define the behavior of the class. */
end
```

Structured class Overall_System contains one instance each of classes CD_Player_Class, Amplifier_Class and Hi-Fi_User_Class and two instances of class Speaker_Class. A connection joining two items states that they are the same thing. For example, event Switch_on/off of module CD_Player is the same as Switch_CD_on/off of module User, they are only called differently by the two modules. On the other hand, items Volume in Amplifier and Volume in User are not only the same, they are also called in the same way in the two modules (hence the unique naming of the corresponding line in Figure 2).

The specification of any non-trivial system is usually made up of one structured class; this models the overall system along with its environment, and its instances include instances of other classes representing the different components of the system. The global semantics of a structured class is defined by the logical conjunction of all axioms of the class and of its modules.

References

[1] A. Gargantini and A. Morzenti. Automated deductive requirements analysis of critical systems. *ACM TOSEM - Transactions On Software Engineering and Methodologies*, 3(3):225–307, 2001.

[2] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. TRIO: A logic language for executable specifications of real-time systems. *The Journal of Systems and Software*, 12(2):107–123, May 1990.