Steam Boiler Specification with TOP: An Executable Specification Language with Time, Objects and Productions

A. Mete Kabakçıoglu

Address (duringFeb. 95 - March 7, 1996): (Visiting Professor)
FH Albstadt-Sigmaringen
Fachbereich Informatik, Johannesstraße 3, 72458 Albstadt, Germany

tel: (49-7431) 579-127 **fax:** (49-7431)579-149
e-mail: mete@fh-albsig.de

Regular Address (not valid during Feb. 1, 95 - March 7, 1996) : (Professor)
Universidad de Los Andes, Depto. de Ing. Eléctrica
CRA. 1A. Este No. 18A-10, Apartado Aereo 4976
Bogotá, Colombia

tel: (57-1) 284-9911 Ext. 2839 **fax:** (57-1)284-1570
e-mail: mete@cdcnet.uniandes.edu.co

Abstract

The specification language TOP extends the traditional production system framework to support distributed object-oriented computing with explicit time support. TOP stands for time, objects and productions. I have chosen a rule based language, since it is easily understandable and there has been a wide variety of successful systems developed based on rules. In addition, since the language is based on logic it serves as an executable specification language. The interpretation process is similar to reasoning in a production system framework with data driven reasoning. TOP distinguishes itself from other systems in its execution mechanism and in the expressive power of the language that combines OOP with other language paradigms. TOP uses an event driven time management strategy on top of a logic-based inferencing mechanism.

TOP is briefly introduced and employed in this paper to specify the steam boiler problem defined in (Abrial 1994 a,b).

Keywords: Steam boiler problem, object oriented programming, production systems, distributed and real-time systems.

1 Introduction

Time is an important aspect of discrete systems including (real-time) programs (Pnueli 1977; Ishikawa et al. 1990; Nota and Pacini 1992; Stoyenko 1992; Degl'Innocenti et al. 1992), sequential hardware designs (Kabakçıoglu and Stabler 1990), discrete control systems (Ostroff, Wonham 1990), robots (Kabakçıoglu 1992a), communication protocols, biological systems, medicine and real-time patient monitoring (Kabakçıoglu 1992b). Real-time programming languages provide constructs and support to deal with time constraints such as time-outs, deadlines, process communication times, and constructs for distributed and concurrent processing such as interprocess communication links and use of shared resources.

A typical language for reasoning in time, also called temporal reasoning, is temporal logic (Allen 1984, Fisher and Owens 1995, Gabbay 1987; Shoham 1988; van Benthem 1988; Kabakçıoglu and Stabler 1990; Barringer et al. 1991). Classical temporal logic with its operators *always*, *eventually*, *next* or Allen's interval calculus with thirteen possible relations such as *before*, *after* do not allow explicit use of time. However, to model or control real-time systems we need first a specification language that supports time explicitly through time points, intervals, metric time in addition to classical temporal logic operators. Specification languages typically are used in a formal verification framework and do not necessarily have execution models. Second we want the language to be an executable temporal specification language that is machine implementable and to offer a temporal logic programming language, analog to what Prolog provides for first-order logic. An executable specification language allows rapid prototyping and early testing. There are only few languages that provide support in the two points mentioned (Degl'Innocenti et al. 1990, Fisher and Owens 1995, Kabakçıoglu 1995 a-b, 1992 a-b , Nota and Pacini 1992). In addition, we want to integrate distributed and object oriented programming (OOP) paradigm into the language. Work in distributed artificial intelligence (DAI) and agent theory has been investigating logic based frameworks that provide distributed computing (Agha 1986, Wooldridge and Jennings 1995). However, object-oriented programming has not received the attention it deserves, possibly due to difficulties in combining OOP paradigm with logic.

Object-oriented programming has emerged as a powerful programming methodology by providing support of reuse, inheritance and encapsulation and an organization that closely relates to the real-world problem being modeled.

To model or control real-time behavior we need to provide a process point of view and support of concurrent behavior. Among different approaches used for process-oriented programming the most important ones are process models as communicating sequential processes using channels (Hoare 1985), concurrent logic programming, functional programming and actors (Agha 1986). However these approaches do not express time explicitly, therefore we need to add time constructs on top of a process model. Another approach is from McCabe (1992) and provides an extension to logic programming to support object-oriented approach. This approach does not support explicit time. In addition, since logic programming does not support explicit state, there are limitations in expressing processes.

Objects can be seen as autonomous processes that function independently from each other. By viewing objects as processes we can provide an object-oriented view of process modeling. However a common criticism has been that traditional object oriented programming provides objects that are sequential in nature. Therefore large sized objects in execution would limit the concurrency.

I believe that a production system framework provides ample opportunities to model processes as objects with support of explicit time and concurrency. Based on this idea this paper extends the traditional production system framework to support real-time and object-oriented computing. I have chosen a rule based language, since it is easily understandable and there has been a wide variety of successful systems developed based on rules. In addition, since the language is based on logic it serves as an executable specification language. The interpretation process is similar to reasoning in a production system framework with data driven and goal driven reasoning. Therefore this knowledge representation and reasoning system is named a language with time, objects and productions (TOP). I have developed a system named temporal production system (TPS) that extends the traditional production system framework to support temporal knowledge representation and reasoning in time (Kabakçıoglu 1995 a-b, 1992 a-b). TOP extends TPS further with object oriented and distributed language paradigms.

TOP distinguishes itself from other systems in its execution mechanism and in the expressive power of the language system that combines OOP with other language paradigms. TOP uses an event driven time management strategy on top of a logic-based inferencing mechanism. Thus time management is not based on time points or intervals as main objects, like in most systems. As a result TOP is able to integrate time primitives typically found in different systems such as time points, intervals, classical temporal logic operators and partial (nonexact) time information through quantifiers and time variables in one framework.

2 Basic Language Constructs

The language constructs are introduced by the following definitions:

Definition 1 : A *term* is a variable, a constant or an n-ary function symbol
$f(t_1,..,t_n)$, where $t_1, .., t_n$ are terms.

Definition 2 : If p be an n-ary predicate symbol and $t_1, .., t_n$ are terms then
$p(t_1, .., t_n)$ is an *atomic formula* or simply an *atom*.

The names of predicates, functions and constants begin with a lower-case letter, variables begin with an upper-case letter.

Definition 3 : The symbol ¬ precedes the corresponding atom to express *negation*. The semantics of negation are based on the restricted version of closed world assumption (Reiter, 1978), named negation-as-failure. Note that the same principle is used in logic programming.

Definition 4 : A *literal* l_i is an atom or negation of an atom.

Definition 5: A *time expression* (**TE**) expresses the time when a formula is true. Typically the formula represents an event that takes place (i.e., is true) at the time given by TE. The so called interval endpoint indicators (IEI) are used in TE's. An IEI can be an integer expressing metric time information (relative to a reference time), ? (unknown future time), ∞ (infinity).
A **TE** is a) an integer constant representing a time point or,
b) a time interval expressed by a pair of IEI's limited by square bracket or parentheses. The square bracket is used for a closed and the parentheses is used for an open interval.

An *extended time expression* consists of two parts. The first part is named the *time information section* and consists of a TE that allows universally and existentially quantified variables as IEI's. These variables that are used to express time information about the literal are called *temporal variables*. We distinguish temporal variables from the logic variables that appear in the arguments of the predicates, since they are treated differently by the interpreter. Hence, we can say that TOP employs a two sorted language where the type of the variables may be logic variables or temporal variables. The second part of a TEX is named *the time constraint* section and consists of the constraints on the temporal variables in the time information section. The time constraint section is enclosed in curly brackets. Extended time expressions allow partially known and generalized time information about a literal.

Definition 6 : A *timed literal* (**TL**) is a literal with a time component that expresses the time when a Boolean formula of literals (F) is true. The time component is expressed by a TE and the syntax of the formula is of the following form: TE:F. TE's are given relative to current time (t=0) which will be explained later in detail.. A *timed atom* (**TA**) is an atomic formula with a TE. Negation of atoms is allowed in TL but not in TA's. Some examples are provided below:

5:F **time point**, F true at t=5 (relative to t=0).
[2,10]: F **interval**, F true in given interval (relative to t=0).
[5,?):F F true from t=5 **until** ? (an unknown future time). This is a nonpersistent formula that may be deleted later.
[4,∞):F F **always** true for t > 4.
(∃T,{T>10}):F F **eventually** true at t>10 (i.e. there exists a time T, T>10, when P is true).
(∃X∃Y[X,Y]{X>4 ∧ Y<15 ∧ X-Y=4}):F F is true in the given **interval** described by the constraints in curly brackets (Extended time expression).
As these examples demonstrate the time expressions support time points, intervals, conventional temporal operators, combination of intervals and temporal operators and provide a clear syntax.

Definition 7 : A special type of interval is an *unbreakable interval* (UBI). The TL's with an unbreakable interval cannot become false in face of new information. An unbreakable interval is a unit itself rather than a collection of time points. Certain events would make sense only as UBI's.

For example let *travel_to_europe* correspond to the activity of traveling to Europe. Such a process is not a collection of time points where a property holds, but an indivisible activity over time. This activity is expressed as a UBI as follows: ubi([t1,t2]:travel_to_europe)
The RTLF's in the database or RTLF's in the conclusion of a TPR can be made unbreakable. In this case when changes are made to the database by the TPR fired, the resulting new RTLF's are UBI.

Definition 8 : A TL with time component (t=0) that is in the form (t=0):F or 0:F is named TL_0 .

Definition 9 : A *relative timed logic formula* (**RTLF**) is a conjunction of TL's, where the TE or TEX of each TL expresses the time when the formula is true relative to time t_0=0 where t_0 is the reference time and will be called the *start time* (t_{start}). Hence RTLF represents literals and the temporal order in which they are true relative to TL_0. The following is an RTLF:

$$\forall X\ (0{:}p(X,a) \wedge 4{:}q \wedge [2,5]{:}r \wedge [5,\infty){:}s \wedge [0,?){:}p(d,e)\ )$$

Here p(X,a) and q are true at t_0 , and t_0+4 respectively; r is true during the interval [2,5], s is *always* true from t_0+5 until infinity, and p(d,e) is true from t_0 *until* an unknown time in the future. In order to avoid extra notation we adopt the same convention as in logic programming and allow dropping of universal quantification signs for logic variables (i.e.,$\forall$X in the above example).

Definition 10 : A *temporal production rule* (**TPR**) is in the following form:

IF conditions **THEN** actions

where the conditions and actions are RTLF's expressed relative to the same t_0 and are named $RTLF_{condition}$ and $RTLF_{action}$ respectively. The same logic variable in the conditions and actions denote the same object. In addition, the logic variables in the TPR are implicitly universally qualified if there are no explicit quantifiers. The scope of the implicit quantifier in a TPR covers both the conditions and actions. Extended TE's can be used in the conditions but actions can only use TE's. The reference time for the conditions is t_0 of the corresponding RTLF condition. A *positive* TPR is defined to be a TPR where the all conditions occur before the actions and there may be at most an overlap of one time point between conditions and actions. The interpreter requires *positive* TPR's. Definition 11 defines a rule to handle overlaps.

Sometmes we want to check conditions of a rule at every cycle (i.e., at constant intervals). In this case the TL *0:cycle(cycle_time)* is used in rule conditions to inform the interpreter, where cycle_time is the time between applications.

Definition 11: An *interval relation TPR* (**IRTPR**) is in the form

IF ubi:[t_1 , t_2]:p **THEN** [t_3 , t_4]:q, where t_i's are IEI's and p and q are atoms.

In an IRTPR the conditions should be an UBI (Definition 7) and unlike regular TPR's, overlap of actions and conditions in time is allowed. The reason why we can overlap intervals in IRTPR is because once the interpreter discovers an UBI in rule conditions, it knows that if *p* is true in the first point in the interval (i.e., 0:p), then it will be true in other points as well. Since $RTLF_{database}$ also provides an UBI definition for *p*, there is no need to check the database at different time points of the interval. As a result the actions of the rule can be asserted right away. It is at the time when 0:p occurs, we also know that rule condition is true, therefore an action cannot have a TE that refers to a time before *p* starts. If this rule did not have an ubi definition for *p*, we needed to prove that *p* is true at all points in the interval. Note that does not mean however checking all the time points in an interval. The interpreter employs an event driven timing mechanism, and checks rules against database, when it knows that database has changed (Section 4).

IRTPR can be used to express different interval relationships between *p* and *q*, such as after, during, equal, starts, finishes as suggested by Allen [1]. In TiEMPO these relationships can be more exact and metric as shown in the following example : **IF** ubi:[0 , 10]:p **THEN** [4,8]:q .

Definition 12 : The database for a given set of TPR's contains time dependent knowledge. This information is represented by RTLF formulas that give information about the past, present and future of the database. We will name this representation as an *RTLF fact* and the time dependent database as $RTLF_{database}$. For an RTLF fact the reference time (t_0=0) is the current time which corresponds to the current status of the database. In RTLF facts the negation of formulas is not allowed in accordance with the negation-as-failure model (Reiter 1978). The factual information has to provide complete and accurate time information to allow the projection of the future. Therefore only TE's are allowed in the RTLF facts.

Definition 13 : *Persistent properties* of the database consist of the knowledge that is true at all time instants. This include Horn clause (i.e., definite clause) rules and facts that are always true (e.g., [4,∞]:p). The latter is also named *persistent facts*. At a given instant of time the system is interpreted by methods used in logic programming. That is why Horn clause rules are allowed. Changes occur due to TPR's and system is brought from one world state to another by the TOP interpreter. The UBI's are also *persistent* during the interval defined. Other RTLF's can be overridden by TPR's that change them.

Next, language constructs that allow distributed knowledge representation and reasoning in time are introduced. Distributed representations and constructs to send and receive messages between objects is essential in object-oriented real-time programming.

Definition 14 : A Temporal Production System (TPS) consists of a set of TPR's, a corresponding $RTLF_{database}$ and a rule interpreter with time management mechanism (Section 4). A *distributed temporal production system* (**DTPS**) is a collection of *temporal production system objects*

(**TPSO**), where a TPSO is a temporal production system with its own interpreter and has its own database. A global database (**GD**) is available and can be accessed by all TPSO's. In addition TPSO's can access database of each other. The communication is achieved in asserting facts to each others database. Let $TL_{cond/i}$ and $TL_{act/k}$ be a timed literals in the condition and action part of a TPR respectively. The following mechanisms are available for the communication of TPSO's:

i) $TL_{cond/i}$ @ database_name means "match $TL_{cond/i}$ with the $RTLF_{database}$ given".
ii) $TL_{act/k}$ @ database_name means "insert $TL_{cond/i}$ to the $RTLF_{database}$ given".

The notation "@" provides a TPSO to access the database of another TPSO or the global database. Unless the expression "@ database_name" is provided the default database of the corresponding TPSO is employed. The TPSO's in a DTPS act as separate distributed objects, that reason based on their own knowledge about the world. The communication the TPSO's is through assertions of $TL_{act/k}$ to each others databases or through matching $TL_{cond/i}$ using each others databases. The extent of communication between these objects via this available communication mechanisms determines how loosely or tightly coupled the system is.

Definition 15 : *Parameterized class templates* allow object oriented representation of TPSO classes in the following form:
class_name(Parameter$_1$, ..,Parameter$_n$) **with** *database (List_of_databases_used)*
{ *TPR's, persistent facts (PF's), definitions,Horn clauses (HC's)***}**
The class body includes TPR's that express changes of the object state in time, PF's and HC's express the invariant properties of the object. Upper case is used for parameter names.

Definition 16: A *new instance of a TPSO* is created from a class template by
(i) Initializing parameters with constant values or variable names;
(ii) Definition of the initial states of databases used in the class definition.

The actual parameter names are written in lower case for parameters that are predicate names, constants or actual database names. The object instantiation is done in the following form:

object_instance **is_new** *class_name(actual_parameter$_1$, ..,actual_parameter$_n$)*
with *database(list_of_ actual_database_names)* .
*database$_1$***{** *RTLF for the initial database state***}**
...
*database$_n$***{** *RTLF for the initial database state***}**

Definition 17: The information in the following definition is not used in the steam boiler example. However, in many other specification tasks this may be an very useful tool. Therefore this definition has been included.

Resource Sharing - Mutually Exclusive Property of Facts : Selected TL_i or all TL_i's in a database can be declared *mutually match exclusive* (MUME). In this case, when more than one TPR matches the selected temporal fact, only one TPR is selected nondeterministically or according to a conflict resolution strategy to fire like a reactive system. Another type of conflict can occur, if at a given time instant two TPR's have contradictionary conclusions, positive and complement of the same predicate. The logic-based reasoning approach returns failure because of contradiction. However if the corresponding predicate is declared as *mutually action exclusive* (MANE), one of the rules is chosen according to a given strategy and all the effects of the other rules are deleted preemptively.

MUME and MANE provide solutions to resource sharing. MUME allows a TPS to have a variable level of reactiveness. One extreme level is that all available facts of the system are MUME, that is system is fully reactive and only one TPR is fired at a time. Note that a TPR is fired when its temporal conditions match the temporal information in the database (Section 4). The other extreme is that there is no fact with MUME. As a result all the rules that match the database at a given time instant are fired.

The strategy used for resource sharing in the case of MUME or MANE can be defined separately and this provides a meta-level reasoning capability. Typical choices can be priority based resource sharing, priority ceiling protocol, or a given order of TPR execution. When TPR's are organized as TPSO's each TPSO may have a given priority, and so we view each TPSO as a separate task.

Definition 18 : Once a TPSO instance is created from a class description, the TPSO starts being active, that is checking databases and firing rules. Explicit *TE:suspend*, and *TE:resume* commands can be send via assertion of the coressponding $TL_{act/k}$ to a TPSO database to suspend and resume the TPSO.

Definition 19: Like in (McCabe 1992) the symbols "<<" and "<=" are used to express class inheritance with and without override respectively.

Definition 20: The so called *definitions* allow RTLF's that appear in more than one TPR conditions or actions to be expressed in one definition. Let p be a predicate symbol and X_i arguments that are logic variables. The definition has the following form:

$p(X_i, X_{i,...,} X_i)$ ::= RTLF

This is a macro based approach and the semantics can be explained by substituting $p(X_i, X_{i,...,} X_i)$ in TPR by the body (RTLF) with the variables substituted.

Definition 21: Each TPSO can be associated to some facts in a database, so that when a fact becomes true at a time instant in the corresponding database of a TPSO, the corresponding TPS interpreter becomes active and begins to fire the rules whose conditions are satisfied. If such an

association is not given, the default is the TPSO interpreter is always active. The following notation is used for this purpose:

act_dis_set(*object_name*, *list_of _literals_for_activation*, *list_of _literals_for_disactivation*)

3 Operational (Procedural) Semantics: Interpreter Description

Each TPSO has its own independent interpreter, therefore the interpreter description for a TPS will be given here. The interpreter is defined for a TPS with positive TPR's. We are interested for a given TPS, to infer what is true over certain intervals of time, when and under which conditions (i.e. assignment to variables) an atomic formula is true in time. Typically temporal input to the database may be provided from external sources by external objects or sensors. The inference engine of the TPR system keeps track of the sequence of changes in the database and activates (fires) proper rules. Rule firing is like in production systems, however there are some differences. First, since conditions are in the form of an $RTLF_{cond}$, to fire a rule the corresponding $RTLF_{cond}$ has to be true. Second, once a TPR fires, the $RTLF_{action}$ is asserted to the database, which provides the information about the future of the database, i.e. future actions to be performed.

In TPS framework we are given a set of TPR's and the current temporal database in terms of RTLF formula which is named an RTLF fact (or $RTLF_{database}$ or *relative-time event set* (RTES)) since it provides factual knowledge with time information relative to current time t=0.

TA's with TE (t=0) (named TA_0) form the current database state (world), $M_{current}$. The changes to database are well defined, in order to avoid typical problems encountered in some STRIPS-like production systems (Lifschitz 1986). As a result Horn clause rules are allowed with the condition that they are valid at all times. Persistent facts are also true through the lifetime of the database, that is they are true at all current databases. The union of $M_{current}$ (i.e., TA_0's in RTES) and Horn clause rule set (HCRS) is called $MH_{current}$. A TL in the condition of a TPR *matches* a TA in $MH_{current}$ by interpreting the TL in the condition as a query against $MH_{current}$. Matching of literals is done by matching the predicate names first and, if the names match, terms are matched by finding the most general unifier as in logic programming. Negation-by-failure and resolution proof procedure is used to determine success or failure.

Definition 22: A *temporal query* (TQ), is a query with a time component. TPR's are used to represent a TQ's. The condition part of the TPR is used to expresses the query. In the data driven interpretation of a TPR, the query is successful once the database satisfies the conditions of the TPR representing the query. The action part of a TQ is different than a regular TPR. The actions include different actions such as interruption of the query process and information messages to the user. A TQ has the following form:

IF $RTLF_{query}$ **THEN** t_{rel}:terminate $\wedge$
t_{rel}: call(write('explanation, variable values: '))$\wedge$
t_{rel}: call(write(X)).

Once the $RTLF_{query}$ matches the database (more in Section 4), the query is satisfied and the action part of the TQ is processed at the desired relative time t_{rel}. The predicate *terminate* is a reserved predicate, and if it is in the actions part of a TQ, it is asserted to the database as a result of a successful query. The interpreter (or all active interpreters) checks at each cycle the fact *terminate* in the current database and terminates the interpretation once it encounters this fact. In addition, built-in input-output predicates such as *write* can be used to print information about variable assignments in $RTLF_{query}$ and other information. Actions of a regular TPR and TQ change the database. However the effect of an action with the reserved predicate *call* is unlike other actions where an assertion or deletion of timed atoms (TA's) in $RTLF_{database}$ take place. A *call* triggers only a database query and the predicates are called rather than asserted. Note that the use of *call* is not limited to TQ and can be used in TPR's as well. In this example calling a reserved input-output predicate causes the desired input-output function to be performed. Actions in queries allows the user to customize the desired response to queries.

We can also use TQ's to watch for certain temporal database constraints, such as liveliness and safety properties. A safety property involves system characteristics that are always true if the system makes progress. The liveliness properties are system characteristics that eventually become true, i.e., the system makes progress towards a specified behavior. In all these applications the TQ is not describing the system behavior, since its action part does not describe discrete system behavior in the future. TQ's are passive observers of the changes of the database, and provide the user information about successful goals and broken constraints etc.

Definition 23: A TPR is *activated* if all positive (formulas without negation) TL_0's in the $RTLF_{condition}$ matches a TA in $MH_{current}$. In addition, in case of negated TL's there has to be no matching TA in $MH_{current}$ (negation by failure). At each TPR activation the TL's in the condition part of the activated TPR excluding TL_0 are inserted to a set called *activated condition set* (ACS_i) of the corresponding TPR_i. Each ACS_i gives the TL's that have to be proven in the future to fire the corresponding TPR_i.

An important task in the data driven reasoning with rules and facts with time information is to match the $RTLF_{cond}$ against the temporal database. This is in general very difficult since the database is changing. In addition, a TPS database keeps on growing with new information as each new rule is fired. Therefore considering that there is a large number of distinct time points in the database, the process of checking the database fro all times whether a given $RTLF_{cond}$ holds, that is whether there is a match with the database becomes even more difficult. Quantifiers of time variables (i.e., in TE's) add to the complexity of the problem. Fortunately, this enormous complexity in reasoning in time can be overcome by a time management system based on the principles of event driven simulation.

Temporal nondeterminism is a typical situation that arises in temporal reasoning when there is more than one possible future. There are several causes of temporal nondeterminism such as

disjunctive formulas in $RTLF_{database}$ or when several TPSO's ask for the same resource. The former is not allowed in TPR's , the latter is managed through MUME and MANE and explicit resource access protocol as explained in Definition 17. By default, a situation with temporal nondeterminism that may arise will be resolved by a random choice if no explicit strategy is available. In a reasoning environment backtracking can be used together with nondeterministic choices in case of disjunctions to cover all possible paths. In such situations backtracking may be used when a choice does not lead to desired goal. However, in the environment TOP is designed to be used such as a real-time situation, or when a model of a process is being simulated and in case of nonterminating processes backtracking is not feasible.

A. Interpreter

We will provide an interpreter description for a TPS. In TOP several TPSO's with different interpreters can exist at the same time. All TPR's to be interpreted have to be positive. A TPR is fired at the point we prove that all the TL's in the conditions ($RTLF_{conditions}$) are satisfied by the $RTLF_{database}$. This is the point when we can assert the actions described by $RTLF_{action}$. However, if the actions make changes in the past relative to current time after a TPR fires, the $RTLF_{database}$ we use to fire this TPR and other TPR's (now and rule firings in the past) may be changed. Hence we can never be sure about the conditions that match the $RTLF_{database}$ to fire a TPR, since later in time the current past may not hold. As a result we have restricted the ability of fired TPR's to change the past. The result is the positive TPR definition. As we will see with the description and the examples the interpreter has two important principles. First, the time points to be checked are determined by observing changes in database, second we avoid the high cost of checking $RTLF_{conditions}$ against all the past database at all times. A time point is picked up from TPR's and from data base and matching is done only for this time point (first-order logic for a given time point). A TPR condition is proven to be true after several interpreter cycles and checked parts of the database are deleted. The interpreter is summarized below:

0) Start with absolute time $t_{absolute} = 0$. The absolute time gives information about the time advance since the start of the reasoning process.

1) If the reserved predicate *terminate* is in the database, stop the interpretation process after completing the *call's* in the conclusion of the corresponding TQ. If there is no ACS_i available go to next step, else match TL_0's in the existing ACS_i's with $MH_{current}$. If TL_0 of an ACS_i does not match, delete that ACS_i.

2) Match TL_0's in the conditions of TPR's with $MH_{current}$. Activate the corresponding TPR if there is a match, i.e., form $ACSI_i$.

3) If match fails in 1 and 2, terminate the interpretation process.

4) Let t_{next} denote the next time point when the interpretation cycle is to be initialized relative to current time. Apply the time increment method to determine t_{next}.

5) All the TE's in RTES are decremented by t_{next} and the TA's with negative time are deleted. As a result elements of $M_{current}$ (i.e., with time t=0) are determined and all TE's of future TA's are

updated relative to the new world and new $t_{current}$. Also the start time of the interval TE's is advanced by t_{next}.

6) Perform the update on all ACS_i's decrementing all TE's by t_{next}. After the update, if there is a TL_i with a corresponding TE_i, where $TE_i < 0$, delete that TL_i. A TPR condition is satisfied if it has been activated before and its ACS_i becomes empty in this step.

7) For all TPR's whose conditions are satisfied (Step 6), the corresponding actions are taken according to the following procedure. Let t_{last} be the time of the last event in $RTLF_{conditions}$ relative to the first event in TPR conditions (i.e.,TL_0 in $RTLF_{conditions}$). For all TPR's whose conditions are satisfied, before performing TPR actions (i.e., $RTLF_{actions}$) we subtract t_{last} from the TE's of the TL´s in the actions. This adjustment is performed since TE's of TPR actions are given relative to the first event in TPR conditions, whereas t_{last} time units have passed since that time. Next the TL's in the TPR actions (i.e.TL_{act}'s) are added to RTES if TL has a positive atomic formula (i.e., TA), or deleted from RTES if TL has negation of an atomic formula. The conditions of an activated TPR are satisfied if the corresponding ACS_i becomes the empty set. At this point $M_{current}$ is defined in terms of TA_0's in RTES, and RTES contains all the currently available information about the future of the database relative to $t= 0 = t_{current}$. Increment time such that $t_{absolute} := t_{current} + t_{next}$ and continue with step 1.

We do not need a special mechanism for goals expressed as TQ's, since they are interpreted like TPR's. If $MH_{current}$ satisfies an activated TQ the reserved predicate *terminate* in the conclusion of TQ is asserted to the database (Step 7), and the interpreter stops interpretation once it detects terminate.

B. Time increment method

An event with an interval occurs at every time point in that interval. One can then view an interval of consisting of time points each separated by $\Delta\tau$ time units. Smaller $\Delta\tau$, larger is the number of time points in an interval. Therefore any temporal reasoning system that sees intervals as a collection of time points will face some difficulty, because of the large number of time. Some systems have used intervals as the primary objects in relations, thus avoiding this problem.

Time management is a key element of the data-driven reasoning process of TOP. In TOP each TPS interpreter employs an event driven time increment mechanism to determine t_{next}. Let t_d denote the time for the next database change relative to current time (i.e., t_d is the time difference). $RTLF_{database}$ (i.e., RTES) gives us this information about future relative to current time. Of course when new rules are fired $RTLF_{database}$ changes, but until t_{next} (relative to current time) no change will occur in the database. Let $t_{asc(i)}$ denote the time for the next event in ACS_i, i.e., the TL with a TE that has the smallest time point or the time for interval beginning point. The $t_{asc(i)}$ gives us the information about the next time (relative to current time) we need to check a condition of the active rule TPR_i against the database. Let t_{asc} = minimum ($\forall i$ $t_{asc(i)}$). The *time increment method* increments $t_{absolute}$ by t_{next}, where $t_{next} = \min(t_d, t_{asc})$. Briefly, we are checking the next projected change in database (i.e., t_d) and the next time we need to match a rule against the database (t_{asc}) to determine t_{next}. Clearly until one or the other occurs we do not need a new interpreter cycle.

4 Informal Specification and Semantic Mapping

Semantics is the study of the relation between a *language* and *objects* that the language speaks of. A formal specification language should have well defined semantics that is typically described through the mapping of symbols of the language to the symbols of the world being described, usually informally in natural language. If the semantic gap between the natural language specification and the formal specification language is too large, the specification process becomes too difficult and the specifications may end up to be very hard to understand. In addition, errors may easily be introduced during the phase of writing the formal specification, due to a large semantic gap. However, it is also expected that the specification language provides constructs to organize the ideas and the problem definition more clearly than the informal specification. In fact, ideally the formal specification should be easier to understand , clearly to a person that knows that language as well as the natural language. Otherwise this comparison would not be fair to a formal specification language. The development of the formal specification of the system also allows elimination of ambiguities that usually appears in the natural language definition.

As a result of the above observations I believe that the semantic mapping between formal specification and the informal specification should be clearly shown. This mapping will be divided into two different levels given below:

(i) *Raw semantic mapping* , that describes how a sentence (or a set of sentences) in the informal specification is related to the formal specification (i.e., which parts of the formal specification).

(ii) *Detailed semantic mapping* that describes how each symbol in the language is mapped to the informal natural language specification. The detailed semantic analysis is available in (Kabakçıoglu 1995b). Due to space limitations this issue will not be discussed any further.

The detailed semantic mapping can be quite long due to large number of symbols that have to be mapped. However, the raw semantic mapping can be done quite easily by putting after each sentence (or group of sentences) in the informal specification the information where and how in the formal specification the sentence is defined. The example problem consists of a program controlling a physical environment Next, I will give the principal part of the problem specification (that is the Program specification) used in this paper taken from (Abrial 1994a) together with the raw mapping. The raw mapping is written in italics and gives the corresponding rule name from the corresponding class and some useful comments. In addition, the object instance is given, since the same rule may be used by more than one object through inheritance mechanism. The reader is advised to read the following section without the raw semantic mapping first in order to understand the problem specification. Next, the raw semantic description can be read while going through the formal specification as an aid to understand the semantics of the formal specification. The section about the

physical environment preceding the Program specification (with the raw mapping) is also informative, since some of the TL's used in the specification and database names are introduced.

Physical Environment

This system comprises of the following units, the steam boiler (*TOP specification uses db_steam_boiler1 as the database*), a water level measurement device (*database: db_water_level _md1*), pumps (*database: db_pumps1*), a pump control device (*database: db_pump_cd1*), a steam measurement device (*database: db_steam_level_ md1*), an operator desk (*database:db_operator1*), a message transmission system (*TOP language provides a high level communication mechanism where TPR's communicate in asserting their conclusions into common databases and fire through a match of their conditions to a temporal database. Therefore no message transmission system is modeled. The implementation of database accesses is another level of abstraction*).

The steam boiler is characterized by several constants (*See the constant definitions, constants for the steam boiler in the specification. TOP uses a fact of the form: equal(constant_name, value,unit*). *Note that the fact has no TE, since it is assumed to be valid all the time and can be used by all TPSO's*):

The total capacity c.

The minimal quantity m1 of water(in liters) m1. Below m1 the steam boiler would be in danger after five seconds, if the steam continued to come out at its maximum quantity without supply of water from the pumps (*See ROC4, D_RISK in program_operating_cl*).

The maximal limit quantity m2 of waters (in liters). Above m2 the steam boiler would be in danger after five seconds, if the pumps continued to supply the steam boiler with water without possibility to evacuate the steam (*See ROC4, D_RISK in program_operating_cl*).

The minimal normal quantity n1 of water in liters to be maintained in the steam boiler insuring the regular operation (m1 < n1).

The maximal normal quantity n2 of water (in liters) to be maintained in the steam boiler during the regular operation (n2<m2).

The maximum quantity w of steam (liters/sec) at the exit of the steam boiler.

The maximum gradient u1 of increase of the quantity of steam (liters/sec/sec).

The maximum gradient u2 of decrease of the quantity of steam (liters/sec/sec).

A valve for the evacuation of water. It serves only to empty the steam boiler in its initial phase (*TL used for that purpose: TE:valve_on@db_steam_boiler1*).

The water level measurement device measures the level of water in the steam boiler provides the quantity of water in the steam boiler(*TL used for that purpose: TE:level(Q)@db_water_level_md1*).

The pump is characterized by :
Its capacity p (liters/sec). (*In this example there are 4 pumps of this capacity, however there is no specification about how to use the pumps together. Therefore I have used only one pump of capacity p4 = 4*p.*) .
Its functioning mode: on (*TL used for that purpose: TE:pump_state_open(all)@db_pumps1*), or off (*TL used for that purpose: TE: ¬pump_state_open(all)@db_pumps1. Since negation as failure is used the nonexistence of pump_state_open in the database at a given time means pumps are off. The pump is also started and stopped by pump_state_open*). After being switched on the pump needs five seconds to start pouring water into the boiler (*R_PD_C1*, *physical_devices_object*).

Each pump control device provides the following information:
The water circulates from the pump to the steam boiler (*TL used for that purpose: TE:pump_water_circulates@db_pump_cd1*), or , in the contrary, it does not circulate (*Same discussion of ¬pump_state_open applies*)

The steam measurement device measures the quantity of the steam which comes out of the steam boiler (*TL used for that purpose: TE: steam(V)@db_steam_level_md1*).

(*The behavior of the physical devices given in this section is described by the TPR's in the physical_devices_cl employing seven rules R_PD_C1 until R_PD_C7. See the comments in these rules. The physical_devices_object used is an instance of this class*).

Program

The program communicates with the physical units. (*In TOP the communication is through database accesses -for matching or assertion--No communication system has to be specified at this level of abstraction*). In the first approximation, the time for transmission can be neglected. The program follows a cycle and a priori does not terminate. This cycle takes place each 5 seconds (*tcycle = 5 sec is used in the TOP specification*) and consists of the following actions:
Reception of messages coming from physical units.
Analysis of the information which have been received.
Transmission of messages to the physical units.

To simplify the matters, and in first approximation, all messages coming from (or going to) the physical units are supposed to be received (emitted) simultaneously by the program at each cycle.
(*This assumption has also been used in TOP. However TOP provides good support of expressing events in time, so that more complicated requirements can be easily handled*).

The program operates in different modes namely: *initialization*, *normal*, *degraded*, *rescue*, *emergency stop*. (*The discussion of the program in the following part gives the raw semantic mapping in detail*).

The *initialization* is the mode to start with (*See R_PD_C4 that gets activated by operator start signal and changes the mode to initialization. Note also activation and disactivation sets (ADS) for initialization_mode_operating_object*). The program enters a state in which it waits for the message steam_boiler_waiting to come from the physical units (*ADS for initialization_mode_operating_object*). As soon as this message has been received the program checks whether the quantity of steam coming out of the steam boiler is really zero. If the unit for detection of the level of steam is defective - that is , when V is not equal to zero - the program enters the *emergency_stop_mode* (*R_IM_OC1, initialization_mode_operating_object*). If the quantity of water in the steam boiler is above n2 the program activates the valve in the steam boiler in order to empty it (*R_IM_OC2, initialization_mode_operating_object*). If the quantity of the water in the steam boiler is below n1 then the program activates a pump to fill the steam boiler (*R_IM_OC3, initialization_mode_operating_object*). If the program realizes a failure of water level detection unit it enters the *emergency_stop_mode* (*R_IM_OC4, initialization_mode_operating_object*). As soon as a level of water between n1 and n2 has been reached the program can send continuously the signal program_ready to the physical units until it receives the signal physical_units_ready which must necessarily be emitted by the physical units (*R_IM_OC5, initialization_mode_operating_object. (R_SL_C2, safety_liveliness_object) checks the livelinesss property expressed here*). As soon as the signal has been received, the program enters either the mode *normal* if all the physical units operate correctly or the mode *degraded* if any physical unit is defective (*R_IM_OC6 and R_IM_OC7, initialization_mode_operating_object. Also see the comments made in R_PD_C5*). A transmission failure puts the program into the mode *emergency_stop* (*R_GC1* (*inherited*), *initialization_mode_operating_object*).

The *normal* mode is the standard operating mode in which the program tries to maintain the water level in the steam boiler between n1 and n2 with all the physical units operating correctly. As soon as the water level is below n1 or above n2 the level can be adjusted by the program switching the pumps on or off (*R_OC1 and R_OC2* (*inherited*), *initialization_mode_operating_object*). The corresponding decision is taken on the basis of the information which has been received from the physical units. As soon as the program recognizes a failure of the water level measuring unit it goes into *rescue* mode (*R_OC3* (*inherited*), *normal_mode_operating_object*). Failure of any other physical unit puts the program into *degraded* mode (*R_NM_OC1* , *normal_mode_operating_object*). If the water level is risking to reach one of the limit values m1 or m2 the program enters the mode *emergency stop* mode (*R_OC4* (*inherited*), *normal_mode_operating_object*). This risk is evaluated on the basis of a maximal behavior of the physical units (*Definition D_RISK in program_operating_cl*). A transmission failure puts the program into *emergency stop* mode (*R_GC1* (*inherited*), *initialization_mode_operating_object*).

The *degraded* mode is the mode in which the program tries to maintain a satisfactory water level despite the presence of failure of some physical unit. It is assumed however that the water level measuring unit in the steam boiler is working correctly. (*See the comments made in R_PD_C5*). The functionality is the same as in the preceding case (*R_OC1 and R_OC2* (*inherited*), *degraded_mode_operating_object*). Once all the units which were defective have been repaired, the

program comes back to *normal* mode (*R_DM_OC1* , *degraded_mode_operating_object*). As soon as the program sees the water level measuring unit has a failure, the program goes into mode *rescue* (*R_OC3* (*inherited*), *degraded_mode_operating_object*). If the water level is risking to reach one of the limit values m1, m2 the program enters the mode *emergency stop* (*R_OC4* (*inherited*), *degraded_mode_operating_object*). A transmission failure puts the program into *emergency stop* mode (*R_GC1* (*inherited*), *degraded_mode_operating_object*).

The *rescue* mode is the mode in which the program tries to maintain a satisfactory water level despite of the failure of the water level measuring unit (*Always when the water level measuring unit has a failure the working program has to change into this rescue mode. The rule (R_SL_C1, safety_liveliness_object) checks this safety property)*. The water level is then estimated by a computation which is done taking into account the maximum dynamics of the quantity of steam coming out of the steam boiler. (*The useful formula for that purpose are given in (Abrial 1994b). The ten consecutive rules R_RM_OC3 until R_RM_OC12 in rescue_mode_operating_cl (used by rescue_mode_operating_object) implement what is described in (Abrial 1994b). In order not to make the document unacceptably long I am not giving the raw semantic mapping for that part and leave as an exercise to the reader. However, I see a problem with one of the formulas that I have corrected in rule R_RM_OC10 (see comments) defined in rescue_operating_mode_cl*). As soon as the water measuring unit is repaired, the program returns into mode *degraded* or into mode *normal* (*R_RM_OC1 and R_RM_OC2, initialization_mode_operating_object. Here the specification does not say openly how to choose between degraded or normal. However, one can easily deduce from the rest of the specification that the next mode is degraded if there are still failing physical units, else to normal*). The program goes into *emergency stop* mode if it realizes that one of the following cases holds: the unit which measures the outcome of steam has failure, or the units which control the pumps have a failure, or the water level risks to reach one of the two limit values (*R_RM_OC3, initialization_mode_operating_object*). A transmission failure puts the program into *emergency stop* mode (*R_GC1* (*inherited*), *rescue_mode_operating_object*).

The *emergency stop* mode is the mode into which the program has to go, as we have seen already, when either the vital units have a failure or when the water level risks to reach one of its two limit values. This mode can also be reached after detection of an erroneous transmission between the program and the physical units (*Since TOP language provides a high level communication mechanism, no extra communication mechanism has been modeled. Therefore we assume that rule conclusions are asserted or deleted properly to corresponding databases. The mechanism for database accesses is another level of abstraction*). This mode can also be set directly from outside (*In TOP model we or the operator can assert mode(emergency_stop) to corresponding database-the name db_mode1 used in next section- to set this mode*). Once the program has reached the *emergency stop* mode, the physical environment is then responsible to take appropriate actions, and the program stops (*See ADS for the objects . It is clear that when at a given time the mode is mode(emergency_stop)@db_mode1, then the disactivation set of all the objects (except the physical_devices_object) are true (i.e.,satisfied). For example normal_mode_operating_object is not active since ¬mode(initialization)@db_mode1 is satisfied when*

mode(emergency_stop)@db_mode1 is true. Note that change_mode predicate is defined such that the modes are mutually exclusive (i.e., no two modes at the same time in db_mode1)) .

When the stop message has been received three times in a row by the program the program must go into emergency stop mode.(*R_GC2, shared by all objects of the program*).

5 TOP Specification

This Section provides the TOP specification of the steam boiler problem. The raw semantic mapping introduced in the previous section can be used to understand the specification and further comments have been added whenever necessary. Most of the time the specification language is self explanatory due to the small semantic gap with the informal specification in the natural language.

The object-oriented approach supported by TOP allows an easy comprehensible organization that shows how the rules and different databases are shared between different objects. The inheritance structure between different classes is shown below, followed later with the class definitions.

```
/* Class organization: ------------------------------------------------------------------------------------*/
    program_operating_cl  <=  global_cl
    rescue_mode_operating_cl  <=  global_cl
    initialization_mode_operating_cl <= global_cl
    failure_cl <= global_cl
    normal_mode_operating_cl <= program_operating_cl
    degraded_mode_operating_cl <= program_operating_cl
/* ------------------------------------------------------------------------------------------------------*/
```

There are several constants used in the specification (Abrial 1994a, pgs 2-3). In TOP the constants are defined as a fact in the following form:

equal(constant,value,unit of measure).

Following is the list of the constants used in the specification. Note that the constants are represented in lower case in TOP. Associated assigned values are arbitrarily chosen, since they have not been specified in the specification (Abrial 1994a,b). A "?" can be used for unassigned values.

```
/* Constant definitions: ----------------------------------------------------------------------------*/
/* Constants for steam boiler  */
equal(c, 100,liters).     equal(m1,10 ,liters).          equal(m2, 80,liters).
equal(n1, 20    ,liters).      equal(n2, 90,liters).          equal(w, 2 ,liters/sec).
equal(u1, 0.2   ,liters/sec/sec).      equal(u2,   0.2  ,liters/sec/sec).
```

```
/* Constants for the pumps, not used in TOP specification (p4 is used instead) */
equal(p, ? ,liters/sec).
/* p4 is the capacity of all the pumps together . p4 is used since the pumps are all on or all off */
equal(p4,5 ,liters/sec).

/* Other constants */
equal(tcycle, 5 , sec). /* time of cycle is 5 seconds */
equal(tsmall, 0.1    ,sec).  /*  a small time increment */
/* ----------------------------------------------------------------------------------------------------*/
```

Next the specification of the classes is given. Global class contains two shared rules, one definition, one input Parameter and several database parameters that are inherited by other classes:

```
/* Global_cl: ----------------------------------------------------------------------------------------*/
global_cl(CURRENT_MODE)
    with database([default = DB_PROGRAM, DB_MODE, DB_PUMPS,
                        DB_WATER_LEVEL_MD,DB_STEAM_LEVEL_MD]) {
/*   References without any explicit database name / parameter are made to given
     default database */
/* Rule name: R_GC1  */
IF  0:failure(transmission)      THEN  0:change_mode(CURRENT_MODE,emergency_stop).

/* Definition name: D_CHANGE   */
change_mode(X,Y) ::= ¬[0,?)mode(X)@DB_MODE  ∧ [0,?):mode(Y)@DB_MODE .

/* Rule name: R_GC2  */
/* Evaluation of arithmetic expressions is done like in Prolog through is relation */
IF  0:stop ∧ tcycle:(stop ∧ T is (2*tcycle))  ∧ T:stop
THEN  0:change_mode(CURRENT_MODE,emergency_stop).
}
/* ----------------------------------------------------------------------------------------------------*/
```

The class named program_operating_cl contains rules that are shared by the two classes normal_mode_operating_cl and degraded_mode_operating_cl. As the names suggest, these classes describe normal and degraded operation modes.

```
/* Program_operating_cl: ------------------------------------------------------------------------------*/
/* Program_operating_cl inherits from global_cl and there are no extra input and database
parameters except the ones inherited. */
program_operating_cl( )
    with database([ ]) {
```

/* Rule name: R_OC1. The response (close the pumps) is given one cycle time (tcycle= 5 sec) later after the input (level(Q)) is received. Note that the effect of closing the pumps is through deleting pump_state_open (i.e.,rule conclusion has: ¬pump_state_open(all)). */
IF 0:cycle(5) ∧ 0:(level(Q)@DB_WATER_LEVEL_MD ∧ Q > n2)
THEN tcycle: ¬pump_state_open(all)@DB_PUMPS .

/* Rule name: R_OC2. Note that the effect of opening the pumps is through asserting pump_state_open (i.e.,rule conclusion has: pump_state_open(all)).*/
IF 0:cycle(5) ∧ 0:(level(Q)@DB_WATER_LEVEL_MD ∧ Q < n1)
THEN tcycle: pump_state_open(all)@DB_PUMPS .

/* Rule name: R_OC3.*/
IF 0:(failure(water_level_measuring_unit) ∧ mode(X)@DB_MODE ∧(X = normal∨ X=degraded))
THEN 0: change_mode(X,rescue) .

/* Rule name: R_OC4. According to informal specification there are 5 seconds before danger after a risk situation and the information level(Q) is accessed by the program every cycle
(tcycle=5 seconds). Therefore mode change action is done immediately and the risk definition uses a level slightly larger than m1 or smaller than m2. */

IF 0:(cycle(5) ∧ level(Q)@DB_WATER_LEVEL_MD ∧ risk(Q))
THEN 0: change_mode(CURRENT_MODE,emergency_stop) .

/* Definition name: D_RISK. <= means less than or equal */
risk(Q) ::= (Q <= m1 + w/1000) ∨ (Q >= m2 - w/1000) .
}
/* ----------*/

/* Normal_mode_operating_cl: ----------*/
/* Inherits from program_operating_cl and global_cl and there are no extra input and database parameters except the ones inherited. */
normal_mode _operating_cl( )
with database([ ]) **{**

/* Rule name: R_NM_OC1. */
IF 0:(failure(X) ∧ X != water_level_measuring_unit)
THEN 0: change_mode(normal,degraded) .
}
/* ----------*/

/* Degraded_mode_operating_cl: ----------*/
/* Inherits from program_operating_cl and global_cl and there are no extra input and database parameters except the ones inherited. */
degraded_mode _operating_cl( )
with database([ ]) **{**
/* Rule name: R_DM_OC1. The predicate *setof* is defined like in Prolog, *setof(X,P,L)* produces a list L of objects X that satisfy P. */
IF 0:setof(X,failure(X),List_of_failures) ∧ tcycle:repaired_all(List_of_failures)
THEN tcycle: change_mode(degraded,normal) .

/* These are Horn clauses, written in Prolog syntax. Used to define the relations used in rules. Horn clauses allow computations to be defined, recursion and list data structure is used frequently */
repaired_all([ ]).
repaired_all(H|T) :- repaired(H), repaired(T).
}
/* ----------*/

/*Rescue_mode_operating_cl: ----------*/
/* Inherits from global_cl and there are no extra input and database parameters except the ones inherited. */
rescue_mode _operating_cl( )
with database([ ]) **{**

/* Rule name: R_RM_OC1. != means not equal */
IF 0:(repaired(water_level_measuring_unit) ∧ failure(X) ∧ X != water_level_measuring_unit)
THEN [0,?):(¬failure(water_level_measuring_unit)) ∧ 0:change_mode(rescue,degraded) .

/* R_RM_OC2 */
IF 0:(repaired(water_level_measuring_unit) ∧ :¬(failure(X) ∧ X != water_level_measuring_unit))
THEN [0,?):(¬failure(water_level_measuring_unit) ∧ 0:change_mode(rescue,normal) .

/* R_RM_OC3 */
IF 0:failure(X) ∧ 0:(X = pump_controller ∨ X = steam_level_measuring_unit ∨
(estimate(Qmin,Qmax) ∧ risk(Qmin,Qmax))
THEN 0:change_mode(rescue,emergency_stop) .

/* Definition name: D_ESTIMATE */
estimate(Qmin,Qmax) ::= qa1(Qmin) ∧ qa2(Qmax).

/* Definition name: D_RISK2*/
risk(Qmin,Qmax) ::= (Qmin <= m1 + c/1000) ∨ (Qmax >= m2 - c/1000).

/* R_RM_OC4 */
IF 0:condition_ci **THEN** (0,tcycle]:pa1(0) .

/* R_RM_OC5 */
IF 0: ¬condition_ci **THEN** (0,tcycle]:pa1(p4) .

/* R_RM_OC6 */
IF 0:condition_di **THEN** (0,tcycle]:pa2(p4) .

/* R_RM_OC7 */
IF 0: ¬condition_di **THEN** (0,tcycle]:pa2(0) .

/* Definition D_Ci */
condition_ci ::= 0:failure(pump_monitor) ∨ 0:¬pump_state_open(all) .

/* Definition D_Di */
condition_di ::= 0:failure(pump_monitor) ∨ 0:pump_state_open(all) .

/* The following rules implement water level estimation in the rescue mode using the information in (Abrial 1994b) */

/*R_RM_OC8 */
IF 0:cycle(tcycle) ∧ 0: ¬failure(steam_level_measuring_unit) ∧
0:steam(V)@ DB_STEAM_LEVEL_MD
THEN [0, tcycle):va1(V) ∧ [0, tcycle):va2(V).

/*R_RM_OC9 */
IF 0:cycle(tcycle) ∧ 0:va1(VA1) ∧ 0:va2(VA2) ∧
tcycle: failure(steam_level_measuring_unit)
THEN tcycle:(VC1 is VA1- u2 * tcycle) ∧
tcycle:(VC2 is VA2 + u2 * tcycle) ∧ T is (2 * tcycle) ∧
[tcycle, T):va1(VC1) ∧[tcycle,T):va2(VC2).

/*R_RM_OC10 .The formula used in the specification (Abrial 1994b) to estimate the water level does not include (pa1* Δt) (i.e., multiplication with Δt is missing). Here we have corrected this (note that Δt is equal to tcycle in our definitions). */

IF 0:cycle(tcycle) ∧ 0: (qa1(QA1) ∧ va1(VA1) ∧ pa1(PA1)
qa2(QA2) ∧ va2(VA2) ∧ pa2(PA2))
THEN tcycle:(QC1 is (QA1 - VA2 * tcycle - 0.5 * u1 * tcycle * tcycle + PA1*tcycle)) ∧
tcycle:(QC2 is (QA2 - VA1 * tcycle + 0.5 * u2 * tcycle * tcycle + PA2*tcycle)) ∧
T is (2 * tcycle) ∧ [tcycle, T):qa1(QC1) ∧ [tcycle, T):qa2(QC2).

/*R_RM_OC11 .Case 1 and 2 from (Abriel 1994b, pg4) that define when to open the pumps*/
IF 0:cycle(tcycle) ∧ 0: (qa1(QA1) ∧ qa2(QA2) ∧ QA1 < n1 ∧ QA2 < n2)
THEN [tcycle,?):pump_state_open(all)@DB_PUMPS

/*R_RM_OC12 */
/* Case 5 and 6 from (Abriel 1994b, pg4) that define when to close the pumps*/
IF 0:cycle(tcycle) ∧ 0: (qa1(QA1) ∧ qa2(QA2) ∧ QA1 > n1 ∧ QA2 > n2)
THEN [tcycle,?): ¬ pump_state_open(all)@DB_PUMPS
}
/* ----------*/

/* Initialization_mode_operating_cl: ----------*/
/* Inherits from global_cl and there is an extra database parameters in addition to the ones inherited.
*/
initialization_mode _operating_cl( )
with database([DB_PHYSICAL_UNITS]) **{**
/* Rule name: R_IM_OC1 */
IF /* This part is not needed since ADS uses steam_boiler_waiting in the activation set:
0:steam_boiler_waiting ∧ */
0:(steam(V)@DB_STEAM_LEVEL_MD ∧ V != 0)
THEN [0,?):failure(steam_level_measuring_unit) ∧ change_mode(initialization,emergency_stop) .

/* Rule name: R_IM_OC2*/
IF 0:(level(Q)@DB_WATER_LEVEL_MD ∧ Q > n2) ∧
THEN tcycle:(T is 2 * tcycle) ∧ [tcycle,T):valve_on@DB_STEAM_BOILER .

/* Rule name: R_IM_OC3*/
IF 0:(level(Q)@DB_STEAM_LEVEL_MD ∧ Q < n1) ∧
THEN [tcycle,?):pump_state_open(all)@DB_PUMPS .

/* Rule name: R_IM_OC4*/
IF 0:failure(water_level_measuring_unit)
THEN 0:change_mode(initialization,emergency_stop).

/* Rule name: R_IM_OC5. Here the informal specification uses the term " as soon as", therefore the usual response time to an input information of tcycle = 5 seconds has not been used here. Clearly this part of the specification has to be clarified with the customer. */
IF 0:level(Q)@DB_WATER_LEVEL_MD ∧ Q>n1 ∧ Q < n2)
THEN [0,?):program_ready .

/* Rule name: R_IM_OC6 */
IF 0:physical_units_ready ∧ 0:all_physical_units_operate@DB_PHYSICAL_UNITS
THEN [0,?): ¬program_ready ∧ 0:change_mode(initialization,normal).

/* Rule name: R_IM_OC7*/
IF 0:physical_units_ready ∧ 0: ¬all_physical_units_operate@DB_PHYSICAL_UNITS
THEN [0,?): ¬program_ready ∧ 0:change_mode(initialization,degraded) .
/* ----------*/

/* Physical_devices_cl: ----------*/
/* The following rules specify behavior of physical system components */

physical_devices_cl()
with database([**default =** DB_PROGRAM, DB_MODE, DB_PUMPS,
DB_WATER_LEVEL_MD, DB_STEAM_LEVEL_MD, DB_OPERATOR,
DB_PHYSICAL_DEVICES]) **{**
/* Rule name: R_PD_C1 */
IF 0: ¬pump_state_open(all)@DB_PUMPS ∧
∃T,T: pump_state_open(all)@DB_PUMPS ∧ T:level(Q)@DB_WATER_LEVEL_MD
THEN T: (T1 is T + tcycle) ∧ [T , T1):level(Q)@DB_WATER_LEVEL_MD ∧
[T,?]:pump_water_circulates@DB_PUMP_CD .

/* Rule name: R_PD_C2 */
IF 0: ¬pump_state_open(all)@DB_PUMPS ∧ 0:(level(Q)@DB_WATER_LEVEL_MD)
THEN [0,tcycle]:level(Q)@DB_WATER_LEVEL_MD .

/* Rule name: R_PD_C3 */
/* This rules updates water level quantity at every small time interval tsmall. The time tsmall is assumed to be small enough so that the change (increase or decrease) in steam exiting the broiler is negligible */
IF 0: cycle(tsmall) ∧ 0: pump_water_circulates@DB_PUMP_CD ∧
0:(level(Q)@DB_WATER_LEVEL_MD ∧ steam(V)@DB_STEAM_LEVEL_MD)
THEN 0: Q1 is (Q-V*tsmall +p4*tsmall) ∧ (0,tsmall]:level(Q1)@DB_WATER_LEVEL_MD.

/* Rule name: R_PD_C4 */
/* The start signal by the operator is assumed to turn the system up */
IF 0:start@DB_OPERATOR
THEN [0,tcycle]:steam_boiler_waiting ∧ 0:change_mode(not_active,initialization) .

/* Rule name: R_PD_C5 */
/* This rule covers the case when the program is waiting but the water level measurement unit is defective. According to the specification the program goes from initialization to the degraded mode when a unit is defective. However, in degraded mode water level measurement unit is assumed to be not defective. As a result I have concluded that the physical system is responsible to send to the program the physical_units_ready signal only when the water measurement unit is in working condition. The operator is informed about the repair job if that is not the case in R_PD_C7 */
IF 0:program_ready ∧ 0:¬failure(water_level_measurement_unit)
THEN [0,?]:physical_units_ready .

/* R_PD_C7 */
IF 0:program_ready ∧ 0:failure(water_level_measurement_unit)
THEN [0,?]:repair_water_level_measurement_unit@DB_OPERATOR .
}
/* ----------*/

/* Safety_liveliness_cl: ----------*/
The following class definition shows TQ's (Definition21) that check one safety and one liveliness property of the steam boiler example. Further properties can be added as desired.

safety_liveliness_cl( )
with database([**default =** DB_PROGRAM, DB_MODE, DB_PUMPS,
DB_WATER_LEVEL_MD,DB_STEAM_LEVEL_MD]) **{**
/* Rule name: R_SL_C1. This safety rule checks the mode a small time units (tsmall)later after the failure of the water level measuring unit in order to insure that the program had the time to make the mode change . */
IF (∃T,T:failure(water_level_measuring_unit)@DB_WATER_LEVEL_MD ∧
T: (T1 is T+tsmall) ∧ T1: mode(X)@DB_MODE ∧
(X != rescue ∨ X != emergency_stop)
THEN T1:(call(print("safety property1 not satisfied!")) ∧ terminate).

/* Rule name: R_SL_C2. This liveliness rule checks that after the program begins to send program_ready signal eventually physical_units_ready signal arrives. Since one has to think in the simulation of the system, an upper limit of 1000 seconds is given. */

IF (∃T1,T1:(mode(initialization)@DB_MODE ∧ program_ready@DB_PROGRAM) ∧
(∀T2,{T2>T1 ∧ T2<1000}: ¬physical_units_ready@DB_PROGRAM)
THEN T1:call(print("liveliness property in initialization mode not satisfied!"))
∧ terminate).

/* The following rules demonstrate failure detection */
/* Rule name: R_SL_C3. This rule is an example of a failure detection rule */
IF 0:(steam(V)@STEAM_LEVEL_MD ∧ (V > w ∨ V < 0))
THEN [0,?):failure(steam_level_measuring_unit) .

/* Rule name: R_SL_C4. This rule is an example of a failure detection rule */
IF 0:(level(Q)@WATER_LEVEL_MD ∧ (Q > c ∨ Q < 0))
THEN [0,?):failure(water_level_measuring_unit) .
}
/* ----------*/

/* Objects: ----------*/
initialization_mode_operation_object **is_new** initialization _mode_operating _cl(initialization)
with database([db_program1,db_mode1,db_pumps1,db_water_level_md1, db_steam_level_md1]).

normal_mode_operating_object **is_new** normal_mode_operating _cl(normal)
with database([db_program1,db_mode1,db_pumps1,db_water_level_md1, db_steam_level_md1]).

degraded_mode_operating_object **is_new** degraded_mode_operating _cl(degraded)
with database([db_program1,db_mode1,db_pumps1,db_water_level_md1, db_steam_level_md1]).

rescue_mode_operating_object **is_new** rescue_mode_operating _cl(rescue)
with database([db_program1,db_mode1,db_pumps1,db_water_level_md1, db_steam_level_md1]).

safety_liveliness_object **is_new** safety_liveliness__cl( )
with database([db_program1,db_mode1,db_pumps1,db_water_level_md1, db_steam_level_md1]).

physical_mode_operating_object **is_new** physical_mode_operating _cl( )
with database([db_program1, db_mode1, db_pumps1, db_water_level_md1,
db_steam_level_md1,db_operator1, db_physical_devices1]).
/* ----------*/

/* Activation and disactivation sets (ADS) for objects ----------*/
act_dis_set(initialization_mode_operating_object, [steam_boiler_waiting@db_program1],
[¬mode(initialization)@db_mode1]).

act_dis_set(normal_mode_operating_object, [mode(normal)@db_mode1],
[¬mode(normal)@db_mode1]).
act_dis_set(degraded_mode_operating_object,[mode(degraded)@db_mode1],
[¬mode(degraded)@db_mode1]).
act_dis_set(rescue_mode_operating_object, [mode(rescue)@db_mode1],
[¬mode(rescue)@db_mode1]).
act_dis_set(safety_liveliness_object, [¬mode(not_active)@db_program1],
[mode(emergency_stop)@db_mode1]).
act_dis_set(physical_devices_object, [start@db_operator1], [mode(stop)@db_operator1]).
/* ----------*/

Execution of a small Scenario

The execution mechanism of TOP is based on an even driven simulation method. The changes in the temporal database used by a TPSO are used to advance time to next change as described in section 3. However the interpreter is not implemented. Therefore here we will informally illustrate through an example how the steam boiler specification described in TOP executes for a given scenario. An initial situation in databases will be used to show the sequence of TPR firings and events in terms of database changes.

Initially :
db_operator1: { 0:start. }
db_water_level_md1: { [0,?):level(5) }
db_steam_level_md1: { [0,?):steam(0) }
db_mode1: { [0,?):mode(not_active) }
db_physical_units: { [0,?):all_physical_units_operate }
db_program1: { 90:stop ∧ 95:stop ∧ 100:stop }
The TL 0:start@db_operator1 activates physical_devices_object. Next R_PD_C4 (used by physical_devices_object, defined in physical_devices_cl) fires that results in the following changes in the databases:
db_mode1: { [0,?):mode(initialization) }
db_program1: {[0,tcycle):steam_boiler_waiting ∧ 90:stop ∧ 95:stop ∧ 100:stop }

Due to [0,tcycle):steam_boiler_waiting@db_program1 initialization_mode_operating_object is activated. Among the rules defined in initialization_mode_operating_class, R_IM_OC1 for example does not fire since steam level V = 0. The rule that fires is R_IM_OC3, resulting the pumps to be opened: db_pumps1: { [0,?):pump_state_open }, this in turn causes R_PD_C1 (used by physical_devices_object) to fire. Note that R_PD_C1 has been active since one of its conditions 0: (¬pump_state_open(all)@db_pumps1) has been satisfied and hence was following the events in time to see whether the other actions in the condition RTLF follow (See section 3 and (Kabakcioglu 1995

a,b)). The resulting changes are tcycle= 5 seconds after the R_PD_C1 conditions are satisfied (note that 5 seconds pass until pump_water_circulates becomes true since pump needs 5 seconds to start pouring water into boiler): db_water_level_md1: { [0,tcycle):level(5) } and db_pump_cd1: { [0,?):pump_water_circulates }. Note that by all this interpretation process the absolute time has been changing, however the databases define the future from the current time (t=0) to the future. The past is deleted since it has been already employed by the rules that are already activate or waiting to be fired. Now R_PD_C3 fires consecutively in small intervals (tsmall) and begins to update water level information. Here the database db_steam_level_md1 is used to measure the steam level. Assuming the steam level V = 0 for some time in the beginning the pumps change the level using this TPR in the following sequence relative to current time t=0 (time, Q):(0,5),((0,0.1],5.5),((0.1,0.2],6.0),((0.2,0.3],6.5),...,((0.9,1],10),...,((2.9,3],20),((3,3.1],20.5). At this point the level is between n1 and n2 and R_IM_OC5 is fired resulting in: db_program1: { [0,?):program_ready }. As a result R_PD_C5 from the active physical_device_object fires resulting db_program1: { [0,?):program_ready $\wedge$ [0,?):physical_units_ready }. Next the rule R_IM_OC6 of the active initialization_mode_operating_object fires resulting in resulting db_mode1: { [0,?):mode(normal)}. Due to ADS the initialization_mode_operating_object becomes disactivated and the normal_mode_operating_object becomes activated. The inherited rules R_OC1 and R_OC2 of the normal_mode_operating_object try to keep the system in working condition as long as the water level measuring unit is in good condition (R_OC3). So far we have assumed that the steam level has been zero. If this continues to be the case, then the water level evaluated by R_PD_C3 of the physical_devices_object will reach n2 causing R_OC1 to be fired turning the pumps off. However a scenario that also involves the steam can be created by introducing to the database of the steam measurement device db_steam_level_md1 a sequence of steam measurements caused by the heating of the water. For example at the beginning of the execution the future steam level measurement such as the following can be used db_steam_level_md1:{ [10, 15):steam(3) $\wedge$ [15, 20):steam(5) $\wedge$ [20, 25):steam(8) $\wedge$ [25, 30):steam(10) $\wedge$ [30, 35):steam(10) ...}. Such a scenario will show how the rules R_PD_C3, ROC_1 and R_OC2 act together first one simulating the physical system (i.e., water level) and the other two controlling the steam boiler in the normal operating mode. Note that when the program arrives t = 90 seconds from the beginning of the execution three consecutive stop signals are given. The stop signals after 90 seconds of execution are in the following form db_program1:{0:stop $\wedge$ 5:stop $\wedge$ 10:stop}. The first stop activates R_GC2 (rule of global_cl). Since all program objects share this rule, no matter in what mode the system is , the rule will be activated. After the rule is activated the rule watches in time whether other conditions are satisfied in given sequence. After the third stop message, this is the case and the program goes into emergency stop mode according to given scenario. Note that during the whole scenario the safety_liveliness_object is active and the TQ's check whether the given safety and liveliness conditions are satisfied.

6 Conclusions

The experience of specifying the steam boiler problem with TOP results in the following conclusions:

(i) The semantic gap between TOP formulation and informal specification is narrow, allowing quick development of the specification language that took about ten man-days time including understanding and analyzing the specifications and the debugging period. This time can be cut down with more coding experience with TOP. Since the method is an extension of rule based programming that many people are familiar with training time should be relatively short for people who are familiar with rule base programming, logic and object oriented program development. The fact that the language is declarative makes it easy to learn how to code. Declarative nature of the code and small semantic gap with informal specifications makes it also easy understanding of the code by the average programmer.

(ii) The use of temporal database as a communication mechanism allows a high level of abstraction for communication between modules (TPSO's). The databases can be in the same device or elsewhere, in this case there is a client server architecture.

(iii) Communication time has been assumed to be zero by the specification to ease the formal specification effort. However, in TPR's the different communication costs (in this case database access times) can be easily incorporated by changing TE's in RTLF's accordingly.

(iv) Rules and facts form the basis of logic and declarative programming. By extending the rules with time support, object-oriented concepts and distributed programming support we see that a rule based system can be a very valuable specification tool for complex real-time systems. Since a TPS expresses cause effect relationships in terms of sequences of events it provides a good mechanism to model the dynamics of physical systems (for example: given in difference equations). The data-driven execution mechanism with time support allows then simulation of such behavior in time.

(v) Support of object oriented programming allows structuring of the problem specification in an easily understandable, natural way that expresses the problem in terms of objects and shows how rules are shared between objects.

(vi) The approach presented here views objects as processes and shows that distributed production systems can be used for modeling action and reaction among different independent, autonomous objects. Message sending in this framework involves assertion of RTLF's to each others database. From the concurrency point of view traditional object-oriented programming can be restrictive for large objects with sequential behavior. The production system modeling of objects provide inherent concurrency since more than one TPR can be active at a given time.

Acknowledgement: I would like to thank DAAD (Deutscher Akademischer Austauschdienst) for supporting my stay in Germany during 1.2.95 - 31.2.96. I also would like to thank the department "Technische Informatik" in FH Albstadt-Sigmaringen for providing a very friendly atmosphere and support that is necessary for good research and teaching.

References

Abrial Jean-Raymond (1994a). "Steam boiler control specification problem," Report for the Dagstuhl-Seminar 9523: Methods for Semantics and Specification.

Abrial Jean-Raymond (1994b). "Additional information concerning the physical behavior of the steam boiler," available via http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html .

Agha G.A. (1986). "Actors," MIT Press:MA,USA.

Allen J. F. (1984). "Toward a General Theory of Action and Time." *Artificial Intelligence*, vol.23, pp.123-154.

Barringer H., Fisher M., Gabbay D., Gough G., Owens R. (1991). "METATEM: A Framework for Programming in Temporal Logic." *In Stepwise Refinement of Distributed Systems*, Bakker J.W., de Roever W.P (eds.), Springer Verlag:NY, USA, pp.94-129.

Fisher M., Owens R. (eds.) (1995). "Executable Modal and Temporal Logics", Springer Verlag: Heidelberg, Germany.

Dean T.L., Wellman M.P. (1991). *Planning and Control*.. Morgan Kaufmann, CA.

Degl'Innocenti M., Ferrari G.L., Pacini G., Turini F. (1990). "RSF: A Formalism for Executable Requirement Specifications," *IEEE Transactions on Software Engineering*, vol. 16, no.11, pp. 1235-1246.

Gabbay, D. (1987). "The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems," *Temporal Logic in Specification* , Springer Verlag, pp.409-448.

Hoare, C.A.R. (1985). Communicating Sequential Processes, NY:Prentince Hall.

Ishikawa Y., Tokuda H., Mercer C.W. (1990). "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," OOPSLA/ECOOP 90, pp. 289-298.

Kabakçıoglu A. Mete (1995a). TiEMPO: Temporal Production System. Proceedings of IJCAI workshop on Artificial Intelligence and Software Engineering, Montreal, Canada, Aug. 1995.

Kabakçıoglu A. Mete (1995b). TiEMPO: Temporal Production System. Technical Report No. 182, 44 pages, Universidad de Los Andes, CIFI, Bogotá, Colombia, *January 1995.*

Kabakçıoglu A. Mete (1992a). "Autonomous Robots and Temporal Reasoning". Proceedings of *AAAI Fall Symposium on Applications of Artificial Intelligence to Real-World Autonomous Mobile Robots,* AAAI Technical Report FS-92-02, Cambridge, Massachusetts, pp. 76-82, October 1992.

Kabakçıoglu A. Mete. (1992b). "Temporal Production Systems for Medical Knowledge Representation." *In the Proceedings of AAAI Spring Symposium on Artificial Intelligence in Medicine (AIM)*, Stanford University, Stanford, CA, March, pp.63-67.

Kabakçıoglu A. Mete, Stabler E.P. (1991). "Automatic Verification of Temporal Logic Specifications about Sequential Hardware Designs using Symbolic Simulation." *International Journal on Computer Simulation*, special issue in digital simulation, vol.1, no.3, pp. 311-321.

Lifschitz, V. (1986). "On the semantics of Strips." *Reasoning about Actions & Plans: Proceedings of the 1986 Workshop*, Oregon, (Morgan Kaufmann, CA, 1987), pp.1-10.

McCabe, F. (1992). "Logic and Objects." NY:Prentince Hall, USA.

Nota G., Pacini G. (1992). "Querying of Executable Software Specifications," IEEE Transactions on Software Engineering," vol. 18, no.8, pp. 705-715, August 1992.

Ostroff J.S., Wonham W.M.. (1990). "A Framework for Real-Time Discrete Event Control." *IEEE Transactions on Automatic Control*, vol.35, no.4, (April): 386-397.

Pnueli A. (1977) "The Temporal Logic of Programs. "*Proceedings of the 18th Symposium on Foundations of Computer Science*, Providence, pp. 46-57.

Reiter, R. (1978). "On Closed World Databases." *In Logic and Databases*, Gallaire H. and Minker J. (eds.), Plenum Press, pp. 55-76. Also in *Readings in Nonmonotonic Reasoning*, Ginsberg M. (ed.), Morgan Kaufmann:CA, USA (1987), pp. 300-310.

Shoham Y. (1988). "Reasoning about Change." The MIT Press: MA., USA.

Shoham Y. (1987). "Temporal Logics in AI: Semantical and Ontological Considerations." *Artificial Intelligence*, vol. 33, pp. 89-104.

Stoyenko A.D, (1992)."The Evolution and State-of-the Art of Real-Time Languages*". The Journal of Systems and Software*, pp.61-84, April 1992.

van Benthem J. (1988). "Time, Logic and Computation." Bakker J.W.et al. (eds.) *School/Workshop on Linear & Branching Time and Partial Order in Logics and Models for Concurrency*, Springer Verlag: Heidelberg, Germany, pp. 1-49.

Wooldridge M.J., Jennings N. R (eds.) (1995). " Intelligent Agents", Proceedings of *ECAI-94 Workshop on Agent Theories, Architectures, and Languages* (Amsterdam, Netherlands, August), Springer Verlag: Heidelberg, Germany.