UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Ingegneria - Dipartimento di Sistemi e Informatica

Ph.D. Course: Ingegneria Informatica e delle Telecomunicazioni
Curriculum: Architetture dei Sistemi di Elaborazione dell'Informazione

INTERVAL TEMPORAL LOGIC FOR REAL-TIME SYSTEMS: SPECIFICATION, EXECUTION AND VERIFICATION PROCESSES

Author:

Pierfrancesco Bellini

Supervisors:

Prof. Giacomo Bucci

Prof. Paolo Nesi

Coordinator:

Prof. Giacomo Bucci

Ciclo XIII, 1997-2000

Acknowledgements

I would like to thank Prof. Giacomo Bucci and Prof. Paolo Nesi, for their many suggestions and constant support during this research.
I wish to thank also the following people: Andrea Giotti, Andrea Mati, Michele Masolini and Giovanni Mugnai(for the work performed during their graduating studies to support this research).
Of course, I am grateful to my family for the patience demonstrated during these years.

Florence, Italy Pierfrancesco Bellini

Table of Contents

List of Tables

List of Figures

Summary

Formal methods should be adopted since the early phases of the system development to reduce failures in the final software product.

In the specification phase, many languages/formalisms could be used to specify the system under development. For real-time systems, where the specification is focussed on modelling system behavior in response of external stimuli, one of the possible approaches consists in using logical statements to describe what the system has to do.This is a different modalityof specification with respect to operational approaches which are based on producing descriptions focussed on how the system does certain operations.

For the specification of real-time systems formal frameworks are typically used. A formal framework is based on a formal model that allow representing the system under specification and a set of tools aids the model verification. Two kinds of specification verification can be present: verification by using property proof and verification by simulation.

Verification by using property-proof permits to prove properties for the system. For example, given the specification of a system it permits to demostrate if a deadlock will never happen. Property-proof is particularly important for safety-crytical systems where some critical situations have to be absolutelly avoided. In a framework based on a logical language, properties and the specification are expressed using the same logic language.

To prove a property means to demonstrate the property by using the logical specification. To this extent, theorem provers could be adopted to mechanically derive these proofs or at least to aid the user in such a work.

Temporal logics are extensions of propositional/first order logics to deal with time constraints, such as ordering of events (i.e., one event has to occur before another) or quantitative constraints (i.e., something has to occur within 10ms). In the past, many temporal logics were presented: in particular TILCO (Temporal Interval Logic with Compositional Operators) is specifically suitable for real-time systems specification.

Temporal logics are well suited to describe temporal constraints while they are not

particularly appropriate to describe system structure. For this reason, CTILCO (Communicating TILCO) has been introduced as an extension of TILCO. CTILCO describes the system as a set of communicating processes where each process can be furtherly decomposed or specified by using TILCO to express its behavior. This approach defines a composition/decomposition specification methodology supported by verification. CTILCO permits specification reuse (in the same project or in other projects). Moreover, a synchronous communication model has been defined in TILCO and communication theorems have been proved to permit to demonstrate properties concerning communicating processes.

Even if TILCO is well suited for real-time systems specification it has been furtherly enhanced with new operators to simplify the expression of temporal constraints (TILCOX). To this end, a new proof system has been developed within the Isabelle/HOL theorem prover.

As already mentioned property-proof is not the only validation support used during the specification, even simulation could be adopted. With system simulation the behavior of the system (or part of it) is tested to see its response to certain stimuli. In this case, only a small part of the system behavior is tested. Simulation is naturally used with operational specification languages (state machines, Petri nets, etc.), while simulation or execution is not natural for denotational formalisms such as logical specification languages. In these languages, given the history of inputs, execution means to find the history of outputs which satisfy the specification. Some problems may arise, (i) the history of outputs could not be univocally determined; (ii) the specification could be partial, that is the value of an output may not be specified in some instants; (iii) the value of an output may depend on the value of a future input (non-causal specification).

The execution of specifications formalized in temporal logics may not be limited to system simulation; in some cases, it may be used even for system implementation when some temporal constraints are satisfied and the specification is complete and causal. In traditional approaches, the system hardware executes a program written in a programming language as C, C++, Ada. This source-code is generated from an executable specification (i.e., Petri nets) and generally it uses services of the operating system for time scheduling of the actions to be done.

In our approach the system hardware executes a general inferential engine that given the specification and inputs, produces the outputs. The problem in this case is to produce the outputs at the correct time.

An executor for propositional TILCO and TILCOX has been developed. Specifications written in TILCO and TILCOX are translated in a simple temporal logic (BTL) with only *and*, *or*, *not* and *delay* operators and transformed in a temporal inference network. For

such a network, inference rules have been defined to infer from the input values the output values.

As regard real-time execution, it has been found that for a causal specification the time needed to produce the outputs at a time t knowing the inputs at time t, $t-1$, $t-2$,... is proportional to the number of arcs of the inferential network. Thus the real-time execution can be feasible depending on the network size and the system hardware. This is a strongly innovative result in the field of formal methods since there exist only few temptatives of defining execution engines for non trivial temporal logic specifications.

Chapter 1

Introduction

Techniques for specifying reactive as well as real-time systems are in the focus of interest of many researchers. In such systems, relevant failures are caused by violations of conditions on temporal constraints (i.e., deadline, timeouts, etc.). It has been often demonstrated that, even for large applications, the adoption of formal methods since the early phases of the development life-cycle reduces failures of the final product. Moreover, formal specification models are effective if they are supported by verification and validation techniques to ensure the correctness of system specification since the early phases of system development.

1.1 Reactive and Real-Time Systems

For specification models, three different aspects must be considered: the structural, the functional and the behavioral aspects [69], [146]. The *structural* aspect refers to the system decomposition into sub-systems. The *functional* aspect has to do with the transformational activities (on data) performed by individual software components. The *behavioral* aspect (i.e., the system dynamics) refers to the system reaction to external stimuli and internally generated events, either synchronously or asynchronously. The systems in which the behavioral aspect is relevant are usually denoted as reactive; real-time systems belong to this category. A reactive system has been defined in [88] as:

"A system that has some ongoing interaction with its environment".

Since reactive systems must maintain this continuous response rather than generating a final result on termination, they are considered safety-critical because an error, in some contexts, can cause security problems.

The reaction to external events is the primer purpose of the system requirements (the timing of these responses is also critical). This response consists in the sequence of states or

events that the system produces during its operation. The wide number of these sequences provokes that to trace requirements specifications is very difficult.

Reactive systems are often concurrent and distributed. A reactive system usually executes concurrently with its environment and, independently of the purpose of the total system requirements, is composed of different processes that run in a concurrent manner. Each component of the system must be specified and verified in terms of the interactions it has with the other components.

Real-time properties are often significant in reactive systems. Real-time systems are a subset of reactive systems because the reactiveness consideration does not necessarily include the explicit notion of time. A real-time system has been defined in [88] as:

"A system that must satisfy explicit (bounded) response time constraints or it will fail."

Real-time systems must generate correct responses subject to bounds on temporal conditions, producing events depending on a specific delay or deadline. Formal methods are advocated to satisfy safety-critical properties in real-time systems [138], for detecting and preventing incompleteness and inconsistencies of their external specification and communication subsystems (with special attention to timing constraints [139],[89] and providing the basis for a systematic approach [142], [35]), and being integrated by a methodology in all stages of the development process.

1.2 Formal Methods

A recent taxonomy for classifying formal methods is based on the extent to which they are *descriptive*, *operational* or *dual* (that is a mixture of descriptive and operational). *Operational* techniques are those which are defined in terms of states and transitions; therefore, they are intrinsically executable. *Descriptive* techniques are based on mathematical notations (axioms, clauses, etc.) and produce precise, rigorous specifications, giving an abstract view of the state space by means of algebraic or logic equations. These can be automatically processed for verifying the completeness and the consistency of the specification, by proving properties by means of automatic tools. *Dual* techniques tend to integrate both descriptive and operational capabilities, allowing the formal specification by means of clauses or other mathematical formalisms as well as the execution of specifications based on state diagrams or Petri nets. Descriptive methods usually fail in modeling structural and functional aspects, but they are suitable for describing system behavior. Operational methods are intrinsically suitable for modeling system behavior in detail, even if they lack in mathematical foundation for describing system behavior at the needed level of abstraction in order to allow validation (i.e., the proving of a required property) without simulation. For dual

techniques, the main problem is the formal relationship between operational and descriptive notations, which should be interchangeable.

In the literature, there are many other classifications according to which tools are divided in process-, data-, control-, and object-oriented [51] or in model-, and property-oriented approaches [146], [66]. In [150], Zave has made a classification by considering the degree of formalism with respect the degree of descriptiveness/operationality. This resulted in a plot having in the abscissa the formal-informal range, and in the ordinate the descriptive-operational range.

Some researchers [35], [12], have developed relevant criteria for evaluating and comparing formal specification methods, that can be helpful to select the most suitable method for an specific software application.

1.2.1 Mathematical Supports

In this section, the most frequently used mathematical supports for reasoning on communicating concurrent processes are briefly discussed. In the late 1970s, Hoare, with his work on CSP (Communicating Sequential Processes) [74], [76], and Milner, with his work on CCS (Calculus of Communicating Systems) [103], have posed the bases for the verification and validation of concurrent systems. The relationships among these two models have been discussed in [32]. Until [74], several methods for specifying communicating sequential processes were widely used, including semaphores [49], conditional critical regions [73], monitors and queues (concurrent Pascal) [31], etc. As observed in [74], '*most of these are demonstrably adequate for their purpose, but there is no widely recognized criterion for choosing between them*'. This consideration led Hoare to attempt to find a single simple solution to all those problems. In the light of the subsequent evolution, CSP is considered as a first rigorous approach to the specification of concurrent systems.

The mathematical bases of CSP have been widely used for defining and analyzing concurrent systems regarded as processes communicating via *channels* [76]. For this reason, the CSP model is denoted as process-oriented, and each process is modeled as a sequential machine. The communication mechanism is completely synchronous — i.e., the transmitter/receiver is blocked until the receiver/transmitter is ready to perform the communication. In the CSP notation, sending a message e on a channel c is denoted by $c!e$, while receiving a message e from a channel c is denoted by $c?e$. This syntax and communication model have been frequently used for defining programming languages (e.g., Occam) and specification tools. In CSP model constructs for modeling parallel ($\|$), sequential ($\gg$), and interleaved ($|||$) executions of processes are also defined [76].

Given its popularity, the original CSP model [74] has been expanded in many ways, resulting in a set of models of increasing complexity: the Counter Model, the Trace Model, the Divergence Model, the Readiness Model, and the Failure Model [106], [114], [75], [76], [104]. The Failure Model can be profitably used for reasoning about the safety and liveness conditions of the system under specification, even in the presence of divergent models (i.e., having an infinite number of states) and non-deterministic processes [15], [76]. The Trace Model can be used to analyze the history of events on the system channels, and for verifying if the system satisfies abstract descriptions of system behavior. For these reasons, CSP is an appropriate basis for both operational and descriptive approaches.

The CSP model does not comprise the concept of time and, thus, the system validation does not take into account timing constraints. For these reasons during the 1980s many extensions have been proposed for adding time support — e.g., CSP-R [84] (where time managing is added by means of *WAIT t* instruction), Timed CSP [129] (where time managing is added by means of the special function *delay()*), CSR (Communicating Shared Resources) [62] and in the CRSM (Communicating Real-time State Machines) [135] (where time is added by means of time bounds on executions and inputs/outputs), etc.

The syntax and semantics of CCS are based on the concept of *observation equivalence* between programs: a program is specified by describing its observation equivalent class which corresponds to the description of its behavior. This is given by means of a mathematical formalism in which variables, behavior-identifiers and expressions are defined. Behavior-identifiers are used in behavior expressions where the actions performed by the system are described. This makes the CCS model quite operational as pointed out in [103] and [113]. This model is based on an asynchronous communication mechanism. The CCS model provided the ground for several models proposed in the late 1980s — e.g., [25].

It should be noted that, the fact that the CSP model is strictly synchronous is not a limitation. In fact, by means of synchronous communicating state machines, asynchronous communications can also be defined. This is done through buffers of infinite capacity which are modeled as state machines as in [135]. In a similar manner, synchronous communications 1:1 (one sender and one receiver) can be expanded to 1:N communications (one sender and N receivers).

1.2.2 Operational Approaches

Operational approaches describe the system by means of an executable model [2], [149]. The model can be mathematically verified (for consistency and completeness) by using static

analysis, and validated by executing the model (i.e., simulation). Operational approaches can be divided in two categories.
The first category comprises languages and methods which are usually based on transition-oriented models, such as state machines [19] or Petri nets [130], that is, models naturally oriented towards the description of system behavior.
The second category includes methods which are based on abstract notations particularly suitable for supporting the system analysis and design (system (de)composition). In these cases, the notations are mainly oriented towards the description of system structure and/or functionality. For this reason, these notations are usually associated with guidelines for system analysis/design and are regarded as methodologies. However, most of them do not model system behavior and, thus, cannot be directly used for system simulation and specification execution. Moreover, since these methods have been developed by starting from visual notations, they lack in formalism and are usually considered as semi formal.

For example, SDL is an integrated operational tool for software specification used in the telecommunications field, providing both visual and textual representations of its syntax. In SDL, the system under specification is regarded as a block which can be decomposed in sub-blocks, thus modeling the structural aspects of the system. Blocks communicate asynchronously by means of strongly typed channels. The language provides support for defining new types of messages modeled as ADTs (Abstract Data Types) [64], [65]. A block can be decomposed in sub-blocks or in a set of communicating processes. A process is implemented as an extended state machine, where the communication semantics is defined by means of a single buffer of infinite length for each state machine. SDL state machines are a mixture of state diagrams and flow charts; in fact, they present states, transitions and selections (equivalent to the "if" statements of high-level languages). In SDL state diagrams, reading of inputs and writing of outputs, as well as the execution of assignments and procedures can be associated with each transition. Reading should always precede writing, since inputs usually represent the condition for transition. If no input reading is defined, the change of state is always performed. For example, the exiting from the initial state is usually performed without reading any input.

In SDL, timing constraints are modeled through *timers*. A timer can be considered as a separate process which is able to send messages. A process can have a set of timers, which can be set, read or reset. For example, an SDL state machine that must satisfy a timeout must set a timer and then wait for the message signalling its occurrence. To this end, the state machine must be able to receive input messages from the timer in all its states, and a transition from a state to the next cannot be interrupted. As a result, definition of deadlines may result in somewhat complicated expressions. SDL permits the dynamic generation of

processes. For the above reasons, the behavior of an SDL specification can be non completely deterministic. In order to manage the problems of real-time in telecommunications supporting functional, behavioral and structural aspects, several extensions of SDL, such as [30], [54], have been presented.

1.2.3 Descriptive Approaches

Descriptive approaches are based on mathematical notations (axioms, clauses, set theory, etc.) and produce precise, rigorous specifications, giving an abstract view of the system state space. The system is described by specifying its global properties, forcing the analyst to specify *what* must be done by the system rather than *how* it must be done. Descriptive specifications can usually be automatically processed for verifying their completeness and consistency. Moreover, a specification can also be validated by proving that high-level properties are verified by the specification itself. This is performed by means of theorem provers or Prolog engines. Since most of these are not enough efficient and predictable (from the performance point of view), descriptive approaches are not considered adequate for producing executable real-time specifications. Some descriptive languages have been enriched with primitives for dealing with time, making them suitable for specifying real-time systems.

Descriptive approaches can be divided in two categories:

• **Algebraic methods** are based on the concepts of Abstract Data Type (ADT) [64], [65]. Algebraic methods have been used for defining abstract data types in conventional applications [111]; later on, they have been employed for specifying reactive systems and communication protocols [141]. In these languages, state variables are modeled by means of axioms, which in turn are functions of the axioms of other ADTs. Most of the algebraic methods allow to specify the system at different levels of abstraction, starting from a coarse description and arriving at the most detailed one. For these methods, the system itself is regarded as an ADT, and its specification consists in describing its syntax and semantics. The syntax definition gives the description of the operator domains of the ADT, while the semantics is given by an implementation of these operators by means of mathematical expressions. Semantics is often defined by writing a set of axioms with a language based on first-order logic. Complex abstract data types are defined on the basis of simpler ones; hence, the semantics of complex types is specified by using the axioms of simple types, and thus the behavior of complex types can be again validated by using the axioms of the simple types. This allows to verify the specification correctness at each level of specification detail.

The completeness can be verified when it is proven that a defined property is verified

by the axioms of the system. This confers a descriptive rather than the operational nature to these approaches, although the ADT behavior can be in many cases translated in state machines. The operational descriptions are distributed among the operators and, therefore, they are not simply executable.

• **Logical methods** describe the system under specification by means of a set of logic rules, specifying how the system must evolve from certain conditions. Differently from the operational methods, the state space described by these specifications is limited and abstract. Rules can be given in the form of first-order clauses of Horn or higher-order logical expressions [91]. These languages are unsuitable for representing the structural aspect of a system, but are very appropriate for describing properties of the system under specification. Validation consists in proving high-level properties, which are also given in the form of logical expressions, by means of theorem solvers or Prolog engines. Simulation is also based on the same techniques.

In the literature, there are many examples of logic languages for the specification of relationships among times and actions. These are often integrated with other techniques addressing also the functional and/or the structural aspects of the system under specification — e.g., RT-ASLAN (Real-Time extension of ASLAN) integrates the first-order logic with the ADT [14], RTL (Real-Time Logic) is a formal language to describe the temporal relationships among events and actions [80], etc.

1.2.4 Dual Approaches

Dual methods [115], [55] try to integrate in a single approach the formal verifiability of descriptive approaches and the executability of operational approaches, though they are often in contrast, especially as regards the reuse and the verification of software specifications.

One of the first examples of dual approach can be considered the Transition Axiom Method proposed by Lamport [87], [86]. In this method, the specification is equivalent to a state machine, on which the proof of high-level properties given by means of axioms can be verified. ESM/RTTL is a dual approach obtained be the integration of ESM (Extended State Machine) language and RTTL (Real-Time Temporal Logic) [118], [115]. RTTL is a logic language based on the classical operator of temporal logic: *until* ($\sqcap$), and *next* ($\odot$). From these the more useful operators of: eventually ($\Diamond$), henceforth ($\Box$), etc., are derived. RTTL can be used to describe high-level properties of the system under specification by means of first-order logic formulae. The integration between RTTL and ESM is obtained by describing the high-level behavior of the system with first-order expressions in which conditions for transitions containing RTTL expressions can be also present. Both RTTL

Class sluice_gate
  visible go, position
  temporal domain integer
  TD Items
    Predicates go({up,down})
    vars position: { up, down, mvup, mvdown }
  TI Items
    vars Δ : integer
  axioms
    vars t: integer
    go_down: position=up $\wedge$ go(down) $\rightarrow$ Lasts(position=mvdown,Δ) $\wedge$ Futr(position=down,Δ)
    gp_up: position=down $\wedge$ go(up) $\rightarrow$ Lasts(position=mvup,Δ) $\wedge$ Futr(position=up,Δ)
    move_up: position=mvup $\wedge$ go(down) $\rightarrow$ $\exists$t (NextTime(position=up,t) $\wedge$
        Futr(Lasts(position=mvdown,Δ) $\wedge$ Futr(position=down,Δ),t)
    move_down: position=mvdown $\wedge$ go(up) $\rightarrow$ $\exists$t (NextTime(position=down,t) $\wedge$
        Futr(Lasts(position=mvup,Δ) $\wedge$ Futr(position=up,Δ),t)
end sluice_gate

Figure 1.1: Textual description of the class *sluice_gate* in TRIO+.

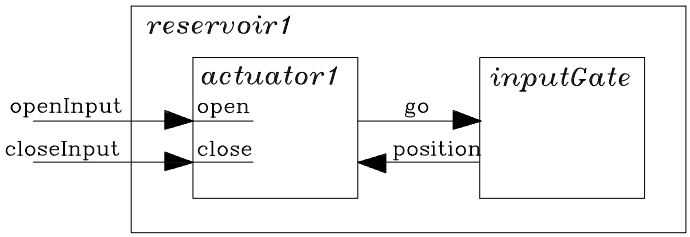

Figure 1.2: Visual description of class *reservoir1* comprised of *actuator1* and *inputGate* objects in TRIO+.

and ESM formulae can refer to the absolute time value.

TRIO+ (TRIO object-oriented) is a logical language for modular system specification [92], [93] extending TRIO (see with object-oriented capabilities. It is based on a first-order temporal language, providing support for a variety of validation activities, such as testing, simulation and property proof. TRIO+ is considered a dual language since it combines the use of visual notation, hierarchical decomposition (typically of operational approaches), with the rigor of the descriptive logical language. In Fig.1.1, the example of a pondage power station is reported, where the quantity of water held in the tank is controlled by means of a sluice gate. The gate is controlled by the commands: *up* and *down* which respectively open and close the gate. These are represented by a predicate *go* having a range { *up, down* }. The gate can be in one of the states: *up*, *down*, *mvUp*, *mvDown*. The state is modeled by a time-dependent variable named *position*. Since TRIO+ is based on logic programming, the object-oriented concept of an instance corresponds to a history of the Prolog interpreter (that is the history of the status of an object).

TRIO+ is endowed with a graphical notation that covers only the declarative part of the language. With this graphic interface the structural aspects can be described, by defining the components of a class and their relationships (see Fig.1.2). TRIO+ is an executable model which supports the executions of partially defined specifications.

TROL is an object-oriented dual specification language supported by the CASE tool TOOMS. TROL is one of the first dual specification languages and is based on an extension of timed CSP — i.e., the CRSM [135]. In TOOMS/TROL, the External Specification is expressed in terms of ports and clauses with temporal constraints. In TROL, the system under specification is hierarchically decomposed in objects and sub-objects. For these objects, the behavior can be specified by means of first-order clauses, since the early phases of decomposition. Moreover, also timing constraints at the external interface of each object can be defined according to [46]. These constraints can be associated with *Provided* and *Required* services of each class, and to *Clauses*.

1.3 Specifying with constraints

For real-time systems, both completeness and consistency of the formal specification should be guaranteed, as well as the satisfaction of system behavior with respect to the timing constraints defined and the high-level behavioral descriptions. These properties state the formal criteria of the specification correctness, which can be verified to reduce failures in requirements, (de)composition and implementation of the system. Toward this end, the use of formal verification criteria (independently of the verification technique) is recommended [117].

1.3.1 Completeness and Consistency Constraints

The *verification* of completeness and consistency is usually performed statically by controlling the syntax and semantics of the model without executing the specification. The system *validation* consists in controlling the conditions of liveness (i.e., absence of deadlock), safety, and the meeting of timing constraints (e.g., deadline, timeout, etc.). It is usually performed statically in descriptive approaches (i.e., by proving properties) [40] and dynamically in operational approaches (i.e., by simulation) [33]. The capability of the method for verifying and validating the system specification (by means of mathematical techniques or simulations) must be analyzed in order to establish if the tools are capable of guaranteeing that the specification produced exactly matches the behavior of the system under development, with safety, without deadlocks and by meeting all timing constraints. Please note that the two

terms verification and validation do not always receive the aforementioned meaning [142], [24]. For instance, in [142], the most frequently used definitions for *verification* are reported, while the term validation is mentioned as "final verification". A specification is complete to the extent that its parts are present and fully developed. A specification is consistent to the extent that its provisions do not conflict with each other or with the general objectives [24]. More precisely, in the system requirements and (de)composition, incompleteness must be detected and the consistency of the specification must be guaranteed.

The verification mechanism must also guarantee that the histories constituting external specifications are correct [71] and that requirements specifications are complete. In [79], several formal criteria for specification completeness in real-time systems have been presented. This research has been extended to verify completeness and consistency of systems requirements using Statecharts and tabular notations [70].

On the other hand, a partial specification of system requirements permit to define the completeness and consistency constraints of the internal and detailed specification. Please note that the use of verified and validated External Specifications is the first step for building the Internal Specification. For system (de)composition, the completeness can be verified by controlling that a component provide each service required by another component of the system, all elements are accounted in the specified interactions, and the requirements of the system assumed by each component are satisfied. Currently, most of the approaches for formally verifying the system (de)composition differ in the way that components and connections are specified. Wright [6] semantically states explicit connectors that are formally specified using CSP. In Rapide[90], events are used to characterize component interaction and simple connector types are formally verified.

1.3.2 Temporal Constraints

Since this work is focused on real-time systems specification, a particular attention is devoted to the expressivity in modeling the *temporal constraints* (timeout, deadline, etc.) [138]. At a high level, a formalism can deal with time either in an *explicit* or *implicit* manner. In the first case, the language allows the representation of time through variables. Explicit timing constraints can be expressed in relative or absolute form. When time is expressed in a relative manner, time durations and deadlines are given in time units. In this case, the relationship between these time units and the absolute measure of time expressed in seconds (or milliseconds) is not clear. However, the validation of specifications becomes almost hardware independent. When time is expressed in absolute form, time durations and deadlines are directly given in seconds or milliseconds (i.e., the absolute time of the

clock) and therefore the meeting of timing constraints depends on the context (machine type, number of processes, workload, etc.).

Real-time system requirements have been expressed quantitatively through three temporal constraints [46]:

- Duration: An event must occur for t amount of time.
- Minimum Rate: minimum t amount of time must elapse between two consecutive occurrences of the same event.
- Maximum Rate: maximum t amount of time must exactly elapse between two occurrences of the same event.

Please note that even if this description does not extensively cover all temporal constraints, it permits the reasoning about the most important temporal conditions on reactive systems. These temporal constraints are not mutually exclusive, i.e., both maximum and minimum time bounds can be associated with the occurrence of two events, which, in turn, can have specific durations. In turn, these event occurrences must precede or follow the occurrence of another event which is expressed as the ordering of constraints.

1.4 Methodology and Tools

Methodologies and tools that integrate formal methods with software engineering practices are playing a role of the major importance in software industry. Numerous formal languages and tools that have been reported in the previous section are still not an integral part of a development methodology. In particular, for dual approaches, requiring a massive verification and validation during the whole system development, there is not a life cycle defined.

In recent years, most of the formal specification languages, have been extended with object orientation. This has resulted in specification models which try to retain the best of the object-oriented paradigm and formal specification techniques.

For these reasons, several languages and tools for modeling the system under specification starting from its requirements have been proposed – early examples are [3], [27]. More recently, the Object-Oriented Paradigm (OOP) has also impacted on several formal specification languages, in the sense that these have been extended in order to support system structuring (i.e., (de)composition) and reuse – e.g., [35], [30].

1.4.1 Object Oriented Methodologies

Conventional methodologies, such as DFD, Entity-Relationships diagrams (ER) [38], etc., have been reinterpreted in the context of the object-oriented methodology — e.g., Coad and Yourdon [43], Rumbaugh et al. (OMT) [132], and Martin and Odell [96]. Of course, the resulting techniques are influenced by the functional view. More recently, Some "pure" object-oriented methodologies have also been proposed — e.g., Booch [26], and Wirsf-Brock et al. Pure object-oriented methodologies focus only on the definition of objects and relationships among them [105].

In many of the above-mentioned approaches, the system is decomposed into objects for representing the structural aspects of the system under specification. Object relationships are defined through extended Entity Relationship diagrams [43], [132] or by using the so-called Object Diagrams [26], [132], [77], [148]. To support all the features of the OOP, such as inheritance, polymorphism, aggregation, association, etc., special symbols for Entity Relationship diagrams or special diagrams, such as Class Hierarchy, have been defined. In most of the proposed methodologies, system behavior is encapsulated in the implementation of objects (more specifically in the implementation of class methods). The object behavior is usually described by means of extended state diagrams or state transition matrices. Shlaer and Mellor [136], [137] and Booch [26], use a Mealy model; Rumbaugh (OMT) [132] uses a notation similar to Statecharts, Coad and Yourdon [43] use a state event table.

Furthermore, though many of these methodologies are especially defined for the analysis and design of reactive systems, some of them are not completely satisfactory for specifying real-time systems. Usually, these notations only provide support for defining timing constraints of the system under analysis, but unfortunately they are not strongly supported by techniques for verifying the consistency and completeness of time relationships. This derives from the fact that these methods are not enough formal for supporting a formal semantics and for defining an executable model of the system.

1.4.2 External and Internal Specification

The adoption of OOP also constrains to better identify the External Interface of each system component. The specification of the External Interface is usually given in terms of the External Specification which consists of a set of statements describing the high-level behavior of the component under specification and the structure of its interface. It should be noted that the description of the class interface in terms of methods is not able to represent all the relationships that objects may have with respect to other objects, especially in a concurrent environment [36], [44], since this does not represent the services that a class of

objects *requires* from other objects [147], [144]. In fact, these requests are encapsulated into the methods body and, thus, they are hidden to the outer objects.

The External Specification is currently approached by specification languages for describing and analyzing system requirements. The External Specification can be defined since the early stages of the system development and can be very useful for checking the class/system/subsystem requirements; checking the system composition; evaluating costs of reuse; defining validated reference requirements, histories, traces, for the final validation.

According to OOP, a methodology (life-cycle) should supports both top-down and bottom-up approaches to development. Please note that in both cases the External Specification of classes can be adopted for helping the user to take the right direction in reusing classes (adopting, specializing, generalizing), and implementing classes (decomposing, describing as state machines). On the basis of the External Specification and the development context, the system (de)composition is described in terms of the Internal Specification .

The Internal Specification is comprised of attributes (communicating objects) and their connections. A class is decomposed in a set of subparts which in turn are instances of others classes.

1.4.3 Tool Support

Although many CASE tools support operational methods (SA, ROOM, etc), these must also support denotational methods as well. To date, model checking techniques are mostly used (algorithm to verify the trueness and falseness of a specific property) because they are more easy to use. In this case, there is a need of further research to automatically perform logical deduction of properties on denotational methods. The formal reasoning of system properties is a hard task that CASE tools need to automate.

For example, tools — e.g., GEODE [61] — cover all features defined in the so-called SDL 88 (version of 1988). Moreover, for supporting the configuration management, versioning, and report generator, other instruments are needed.

1.5 Thesis Contributions

This work extends the work done on TILCO temporal logic[97]. In particular, it

- presents a framework for the specification of communicating real-time systems/subsystems;
- extends TILCO with new operators as bounded happen and dynamic intervals to simplify formulæwriting, and presents a new deductive system;

- presents an executor of deterministic TILCO specifications, the executor is used for validation trought simulation or even (if it is possible) for system implementation.
- presents a set of tools developed to support the specification in TILCO and its extension.

1.6 Organization of the Thesis

This thesis is organized as follows:

- In chapter 2, temporal logics presented in literature in the last years are shortly reviewed to compare them.
- In chapter 3, TILCO temporal logic is presented in more details.
- In chapter 4, C-TILCO an extension of temporal logic TILCO is presented to deal with system (de)composition. A model of communication between system "parts" is presented. A deductive system for C-TILCO is presented.
- In chapter 5, a further extension of temporal logic TILCO is presented to enhance the expressiveness of the logic and to simplify formula
- In chapter 6, an executor of TILCO/TILCO-X specifications is presented.
- In chapter 6.5, conclusions are drawn.
- In Appendix A, tools developed for TILCO specification are presented.
- In Appendix B and C, Isabelle theories for TILCO-C and TILCO-X are reported.

Chapter 2

Temporal logics for real-time system specification

In this chapter classical logics are discussed in order to highlight their limitations in expressing temporal properties. Temporal logics are presented and the most important features that characterize the temporal logics are highlighted and some examples are presented. A selection of temporal logics presented in literature are briefly described and their suitability for the specification of real-time systems is higlighted. Finally the main features of the temporal logics considered are reported and discussed.

2.1 From Classical to Temporal Logics

The primary feature of a logic theory is its order, which defines the domain of all formulas described by the logic: (i) propositional, (ii) first order, (ii) higher order.

Formulae in *propositional logic* are built on the basis of a set of elementary facts (i.e., *atomic formulae*) by using a set of logic operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$). Their semantics can be defined in terms of *truth tables* or by inductive rules on the structure of the formula itself. Each formula can assume a logical value true ($\top$) or false ($\bot$).

The *First Order Logic*, FOL, adds several extensions to the propositional logic:

- There exists a domain of elements, $\mathcal{D}$, on the basis of which the logical formulae are built;
- n-ary $\mathcal{R}_i$ relationships on $\mathcal{D}$ can be defined, as subsets of $\mathcal{D}^n$;
- A n-ary predicate p_i is associated with each n-ary $\mathcal{R}_i$ relationship. The predicate is

a function that for each element of $\mathcal{D}^n$ gives a value $\top$, if it belongs to an n-ary $\mathcal{R}_i$ relationship, otherwise the value $\bot$ is assumed;

- The operators of FOL are those of propositional logic plus the universal quantifier $\forall$ (*for all*), and the existential quantifier $\exists$ (*exists*).

In FOL quantified variables must be elements of $\mathcal{D}$ and not full predicates. The presence of quantification increases the expressiveness of the logic, allowing the description of existential and generalization relationships.

The *Higher Order Logic*, HOL, extends the domain modeled by FOL by allowing the adoption of predicates as quantification variables. For example, the following HOL formula:

$$\forall P.\exists x.P(x),$$

cannot be written in FOL since it contains a quantifier varying over a predicate P. The higher expressiveness of HOL makes it suitable for formally describing lower order logics.

2.1.1 Deductive Systems

Classical logics can formalize the deductive process: given a set of true propositions, it is possible to verify if other propositions are a logical consequence of the early set.

Proving theorems by using formal logic is a process quite different from the human deductive process. *Deductive systems* are based on a formalized theory by means of a set of axioms and deduction rules. This makes possible to define a purely syntax-deductive system without adopting the concepts of validity and satisfiability, which are typical of the human deductive process.

In order to profitably adopt a deductive system for proving theorems it is mandatory to demonstrate that it is *complete* (i.e., it is possible the construction of a demonstration for all theorems of the theory), and *sound* (i.e., each theorem that can be demonstrable with the logic is a theorem of the logic) [47], [1], [22], [11].

Deductive systems tend to be minimal, as the set of axioms and deductive laws selected are usually just those strictly needed to describe the logic. Therefore, the process required to prove other theorems can be complex and long. On the other hand, it is often possible to use the deductive system to demonstrate new deduction laws and theorems that, in turn, can be used in other proof processes as the minimum initial set. In this way, a system of deduction laws that makes the process of proof easier can be built.

2.1.2 Classical Logic and Time

In general, assertions can be classified as either static or dynamic. Static assertions have a fixed and time-independent truth value, while the truth value of dynamic assertions is in some way time-dependent. For example, the proposition $1 < 2$ is always true, whereas the logical value of the proposition:

it is raining

is time-varying: sometimes it may be true while at others it may be false. Since the state of real system changes over time, logic predicates describing the behavior must provide propositions whose values vary over time. Classical logic can express only *atemporal* (non-time dependent) formulas whose validity and satisfiability do not depend on the instant in which they are evaluated. In other words, time has no role in classical logic; when a proposition presents a value that changes over time, the time must be modeled as an explicit variable. For example, if a proposition P has to be true in interval $[t+5, t+20]$ we have to write the formula as

$$\forall x \in [t+5, t+20].P(x).$$

This approach makes the writing of time-dependent propositions quite complex. In order to model the behavior of domains in which the logical value of propositions may vary, modal and temporal logics were introduced as extensions of classical logic. These approaches facilitate the specification of temporal relationships.

2.1.3 Modal Logic

In modal logic, the concepts of truth and falsity are not static and immutable, but are, on the contrary, relative and variable [78]. In modal logic, the classical concept of interpretation of a formula is extended, in the sense that every modal logic theory has associated with it, not just a single interpretation, but a set of interpretations called *worlds*. In each world, a truth value is assigned to the formulas, similarly to the interpretation of a formula in classical logic.

A modal logic system is defined by $< W, R, V >$ where: W is the set of worlds; $R \subseteq W \times W$ is the *reachability* relationship between worlds; and V is the evaluation function for formulas:

$$V : F \times W \rightarrow \{\top, \bot\},$$

where F is the set of the formulas of the modal theory. V assigns a truth value to every formula in F in every world in W.

The forms of W and V depend on other characteristics of the logic; for example, whether it is propositional or a FOL. Besides the operators and symbols of classical logic, modal logic introduces operators $\boldsymbol{L}$ (*necessary*) and $\boldsymbol{M}$ (*possibly*). These express the concept of the necessity and possibility of formulas in the set of worlds reachable from the world in which the main formula is evaluated.

The semantics of a modal logic can be formally given on the basis of an evaluation function V, which is inductively defined over the structure of the formula to be evaluated. Omitting the definition of the part about classical logic, V is defined over the modal operators $\boldsymbol{L}$ and $\boldsymbol{M}$ as follows:

- $V(\boldsymbol{M}f, w) = \top$ iff $\exists v \in W.wRv \Rightarrow V(f, v)$;
- $V(\boldsymbol{L}f, w) = \top$ iff $\forall v \in W.wRv \Rightarrow V(f, v)$.

In other words, formula $\boldsymbol{M}f$ is true in a world w if and only if there exists a world v reachable from w, where subformula f is true; formula $\boldsymbol{L}f$ is true in w if and only if in all worlds reachable from w subformula f is true. Modal operators $\boldsymbol{L}$ and $\boldsymbol{M}$ have a simple interpretation as quantifiers defined over the set of reachable worlds from the current world, namely: $\boldsymbol{M}$ is an existential quantifier, while $\boldsymbol{L}$ is a universal quantifier. It is easy to see that the following relation holds between operators $\boldsymbol{L}$ and $\boldsymbol{M}$:

$$\boldsymbol{L}f = \neg\boldsymbol{M}\neg f.$$

The features of a modal logic $< W, R, V >$ are strictly connected to the relationship that determines the structure of the set of worlds. The interpretations of relationship R may be several: R can represent how a set of classical theories are correlated; for example, in a non-monotonic logic the elementary truth and the deducible facts can change dynamically. In the context of temporal logics the most interesting interpretation for relationship R is the relation *next instant*. In this way, the worlds are the set of configurations that the system modeled may assume in successive time instants. In this case, the modal logic can be quite profitably used for the study of temporal properties of systems, and for this reason takes the name *temporal logic*.

2.1.4 Temporal Logic

Temporal logics are particular modal logics where the set of worlds W is interpreted as the set of all possible instants T of a temporal domain.

Usually temporal logics are built as extensions of classical logic by adding a set of new operators that hide quantification over the temporal domain. Temporal logics presented in the literature are principally obtained by extending propositional or FOL; rarely has the extension started with HOL.

As in modal logic, where the world in which the formula is evaluated is referenced, in temporal logic the evaluation instant of a formula is used. The value of a formula is a *dynamic concept*. Therefore, the concept of formula satisfiability must be modified to consider both the interpretation of a formula and the instant of the evaluation.

Generally temporal logics add four new operators with respect to classical logics [127]:

- $\boldsymbol{G}$, always in the future;
- $\boldsymbol{F}$, eventually in the future;
- $\boldsymbol{H}$, always in the past;
- $\boldsymbol{P}$, eventually in the past.

These can be formally defined:

- $V(\boldsymbol{G}f, t) = \top$ iff $\forall s \in T.t < s \Rightarrow V(f, s)$;
- $V(\boldsymbol{H}f, t) = \top$ iff $\forall s \in T.s < t \Rightarrow V(f, s)$;
- $\boldsymbol{F}f \equiv \neg\boldsymbol{G}\neg f$;
- $\boldsymbol{P}f \equiv \neg\boldsymbol{H}\neg f$.

These operators can express the concepts of necessity ($\boldsymbol{G}$, $\boldsymbol{H}$) and possibility ($\boldsymbol{F}$, $\boldsymbol{P}$) in the future and in the past, respectively. Often in temporal logics these operators are represented by other symbols: $\Box$ (*always*) denotes $\boldsymbol{G}$ and $\Diamond$ (*eventually*) denotes $\boldsymbol{F}$. For past operators (if they are present), symbol $\blacksquare$ denotes $\boldsymbol{H}$ and $\blacklozenge$ denotes $\boldsymbol{P}$.

If relation $<$ is transitive and non-reflexive, it is possible to introduce two other binary operators:

- **until** (in some cases represented with $\mathcal{U}$), with ϕ_1**until** ϕ_2 that is true if ϕ_2 will be true in the future and until that instant ϕ_1 will be always true;
- **since** (in some cases represented with $\mathcal{S}$), with ϕ_1**since** ϕ_2 that is true if ϕ_2 was true in the past and since that instant ϕ_1 has been true;

The semantic of these operators can be formally defined as follow:

- $V(f_1 \textbf{until} f_2, t) = \top$ iff $\exists s \in T.t < s \wedge V(f_2, s) \wedge \forall u \in T.t < u < s \Rightarrow V(f_1, u)$;
- $V(f_1 \textbf{since} f_2, t) = \top$ iff $\exists s \in T.s < t \wedge V(f_2, s) \wedge \forall u \in T.s < u < t \Rightarrow V(f_1, u)$.

Note that operator **until** (**since**) does not include the present instant in the future (past). The introduction of operators **until** and **since** is relevant since these operators can express concepts that cannot be expressed with the operators $\boldsymbol{G}$, $\boldsymbol{H}$, $\boldsymbol{F}$ and $\boldsymbol{P}$. On the contrary, these last operators can be defined in terms of **until** and **since**:

- $\boldsymbol{F}\phi \equiv \top \textbf{until} \phi$;
- $\boldsymbol{P}\phi \equiv \top \textbf{since} \phi$;

and

- $\boldsymbol{G}\phi \equiv \neg \boldsymbol{F} \neg \phi$;
- $\boldsymbol{H}\phi \equiv \neg \boldsymbol{P} \neg \phi$.

If the temporal logic has the *begin* property (e.g., stating that the temporal domain is bounded in the past as discussed in the sequel), the operator **until** is enough to complete the logic expressiveness: when the past is limited the operator **since** is not necessary. Relationships among events in the past can be expressed by using **until** starting from the beginning of time (from a fixed reference time instant).

Other common operators are *next* and *prev*, represented with ○ and ●, respectively. These operators are unary and can be defined in term of *until* and *since* operators:

- $\bigcirc \phi \equiv \bot \textbf{until} \phi$
- $\bullet \phi \equiv \bot \textbf{since} \phi$

These two operators assume different meanings depending on the time structure – e.g., discrete or continuous – or whether the logic is event-based.

The presence of distinct operators for past and future simplifies the specification model: since with their use formulas can be easily written – for instance, evaluating the past and describing the future. On the other hand, this distinction is only a convention, since in most temporal logics formulas can be easily shifted to the past or to the future.

2.2 Main Characteristics of Temporal Logics

This section presents the evaluation criteria used to compare the temporal logics discussed in the following sections. We provide a taxonomy to classify and evaluate the suitability of temporal logics used for specifying real-time systems. Temporal logics are typically used in the phases of *requirements analysis*, *advanced analysis*, *specification*, and more recently, even for *execution*. They focus on modeling system behavior rather that functional or structural aspects [35]. *Structural* aspect refers to system decomposition into subsystems (modular temporal logics). *Functional* aspect deals with the data transformation of the system. *Behavior* refers to the system reaction to external stimuli and internal events, a critical aspect of reactive and real-time systems.

To use temporal logics for real-time system specification, it is necessary to evaluate their expressiveness in modeling the typical requirements of such systems and of the constraints needed to express the specification. Typical temporal constraints can be divided in two main categories: (i) events and event orderings; (ii) quantitative temporal constraints.

The following paragraphs discuss the most important features of temporal logics and the criteria used to identify their general characteristics and properties.

2.2.1 Order of Temporal Logic

The order of a temporal logic is the order of classical logic on which the temporal logics is constructed. This characteristic dictates the set of formulas that the temporal logic can express. A higher order implies greater expressiveness but more complex formulas, and frequently, the logic itself is less complete and decidable. For instance, propositional temporal logics are less expressive than higher order logics, but often propositional temporal logics are decidable and their decision procedures have a tractable complexity; whereas higher order logics are more expressive but much more complex. First order temporal logics usually permit one to write quite expressive formulas without overly increasing the complexity of the logic.

2.2.2 Temporal Domain Structure

As stated in Section 2.1.3, the main properties of a modal logic, and then of a temporal logic, are related to the properties of relation R; the next section will show the structure of temporal domains derived from properties of relationship R. For temporal logics relation R is called a *precedence relation* and is denoted by $<$. Properties that bear on the temporal domain structure are:

transitivity	$\forall xyz.x < y \wedge y < z \Rightarrow x < z$
non-reflexivity	$\forall x.\neg x < x$
linearity	$\forall xy.x < y \vee x = y \vee y < x$
left linearity	$\forall xyz.y < x \wedge z < x \Rightarrow y < z \vee y = z \vee z < y$
right linearity	$\forall xyz.x < y \wedge x < z \Rightarrow y < z \vee y = z \vee z < y$
begin	$\exists x.\neg\exists y.y < x$
end	$\exists x.\neg\exists y.x < y$
predecessor	$\forall x.\exists y.y < x$
successor	$\forall x.\exists y.x < y$
density	$\forall xy.x < y \Rightarrow \exists z.x < z < y$
discreteness	$(\forall xy.x < y \Rightarrow \exists z.x < z \wedge \neg\exists u.x < u < z) \wedge$
	$(\forall xy.x < y \Rightarrow \exists z.z < y \wedge \neg\exists u.z < u < y)$

Usually $<$ is a transitive and non-reflexive relationship; hence it is a partial ordering on time instants.

The property *begin* (*end*) states that the temporal domain is bounded in the past (future) [67], [101], whereas the property *predecessor* (*successor*) shows that the temporal domain is unlimited in the past (future). In fact, the following equivalencies hold:

$$(\exists x.\neg\exists y.y < x) \Leftrightarrow \neg(\forall x.\exists y.y < x)$$
$$(\exists x.\neg\exists y.x < y) \Leftrightarrow \neg(\forall x.\exists y.x < y)$$

A temporal domain is *dense* with respect to relationship $<$ if between two instants there is always a third. On the contrary, the temporal domain is *discrete* if there exist two instants between which a third cannot be determined.

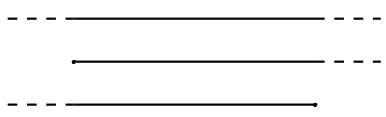

Figure 2.1: Representation of linear temporal domains.

If the precedence relation is *linear* then we have a *linear temporal structure* that corresponds to the intuitive notion of time. This is the simplest type of temporal structure. In this case, the precedence relation is a total order on time instants. Figure 2.1 shows the temporal domain for a linear temporal structure with a unlimited past and future, with only a unlimited future and with only an unlimited past. If the time structure is linear and discrete, a state of the system can be associated with each time instant. If the time is dense, the logic must be event-based to support a state-based semantics.

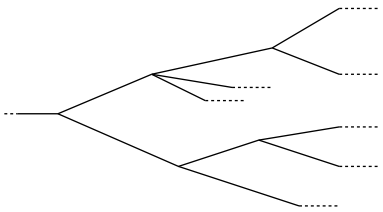

Figure 2.2: A non linear structure in the future.

When the precedence relation, $<$, is only *linear* on the left, the temporal structure is more complex: branches can exist in the future (in other words, more than one future can exist for each instant), but there exists only one past (see Figure 2.2). If the time is discrete and its structure is branched a next state exists but it cannot be unequivocally determined.

Temporal structures with branches in the past are also possible. If no hypotheses are made about linearity, branches in the future and in the past are possible.

The order relation of the structure is usually transitive and non-reflexive. The temporal domain may be limited in the past and/or in the future or unlimited, and it may be dense or discrete. Thus, the temporal structure may be linear or branched in the past and/or in the future. These properties have implications for the decidability of the logic, its executability, and the style used to write formulas.

2.2.3 Fundamental Entity of the Logic

A basic way to characterize temporal logics is whether points or intervals are used to model time. This also influences the expressiveness of the logic.

Point-based temporal logics express relationships among events in terms of points. Point-based logics define intervals as connected set of points. In point-based logics it is more difficult to express relationships between intervals in which certain events are verified. Time durations are expressed by using quantifications over time. Logics based on *time points* [94], [131] specify system behavior with respect to certain reference points in time; points are determined by a specific state of the system and by the occurrence of events marking state transition. In order to describe temporal relationships, the operators $\Box$ (*henceforth*) and $\Diamond$ (*eventually*) are usually adopted to specify *necessity* and *possibility*, respectively.

Interval-based temporal logics (interval logics) are more expressive since they are capable of describing events in time intervals and a single time instant is represented with

a time interval of one. Usually interval-based logics permit one to write formulas with a greater level of abstraction and so are more concise and easy to understand than point-based temporal logics. In the case of *time intervals* [133], [134], [110], [67], [68], [85], [101] [128], formulæ specify the temporal relationships among facts, events, and intervals, thus allowing a higher level of abstraction for system specification. Interval-based logics usually present specific operators to express the relationships between intervals (*meet*, *before*, *after* [4]), and/or operators for combining intervals (e.g., the *chop* operator [131]), or operators to specify the interval boundaries on the basis of the truth of predicates [101].

The qualitative relationships that may hold between intervals as classified by Allen in [5] are represented in Figure 2.3.

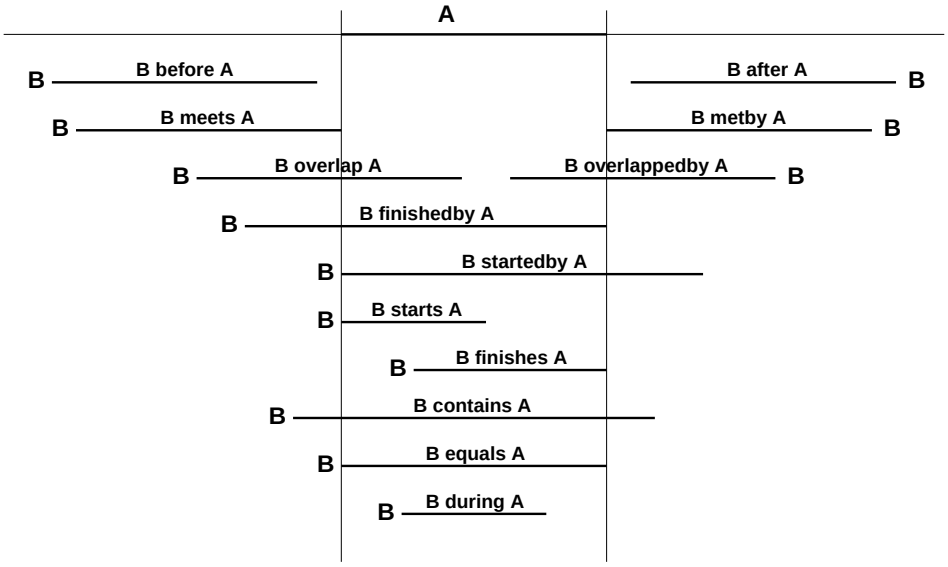

Figure 2.3: Possible relationships between two intervals.

The relationships among time points or intervals are typically qualitative, but quantitative temporal logics are preferable for the specification of real-time systems (e.g., RTL [80], MTL [82] and TRIO [63], TILCO [98]).

2.2.4 A Metric for Time and Quantitative Temporal Constraints

The presence of a metric for time determines the possibility of expressing temporal constraints in a quantitative form in the logic formulas; without a metric for time only temporal-order relations can be expressed (qualitative temporal logics).

The temporal operators presented in Section 2.1.4 are qualitative since it is not possible to give an exact measure (i.e., duration, timeout) for events and among events. Temporal logics without a metric for time adopt a time model for which the events are those that describe the system evolution (event-based temporal logics). Each formula expresses what the system does at each event, events are referred to other events, and so on: this results in specifying relationships of precedence and cause-effect among events.

Temporal logics with a metric for time allow the definition of quantitative temporal relationships, – such as distance among events and durations of events, in time units. The expression of quantitative temporal constraints is fundamental for real-time systems specification. It is necessary to have a metric for time if the temporal logic has to be used to express the behavior of hard or non-hard real-time systems. A typical way for adding a metric for time is to allow the definition of bounded operators – for example:

$$\Diamond_{[4,7]} A$$

for stating that A is eventually true from 4 to 7 time instants from the current time, or $\Diamond_{\leq 5} A$ which means that A is eventually true within 5 time units. A different method is based on the explicit adoption of a general system clock in the formulas (see section 2.2.6).

A different way to manage time quantitatively is to adopt the *freeze* quantifier [7], which allows only references to times that are associated with states. This means that freeze quantifier "$x.$" differs from the FOL quantification over time. For instance:

$$\Box x.\ \big(p \rightarrow \Diamond y.\ (q \vee y \leq x + 6)\big)$$

This means that in every state with time x, if p holds, then there is a future state with time y such that q holds and y is at most $x + 6$. Logics allowing *freeze* quantification are called half-order logics.

In specifying of real-time systems, the general behavior of the system is typically expressed by means of quantitative temporal constraints. The correct behavior of the system depends on the satisfiability of these temporal constrains.

In [82], a classification of temporal constraints with respect to event occurrences has been proposed. In particular, we can specify constraints for establishing relationships between the occurrence of

1. an event and a corresponding reaction (**reaction time**). Typical cases are:

   - maximum distance between event and reaction (e.g., *timeout*);

- exact distance between event and reaction (e.g., *delay*);

2. the same event (**period**). Typical cases are:

   - minimum distance between two occurrences of an event;
   - exact distance between occurrences of an event.

This classification can be simplified by reducing the types of temporal constraints to only two elementary constraints:

- universal temporal quantifier $\Box_i A$, that means that A is true in all time instants of interval i;
- existential temporal quantifier $\Diamond_i A$, that means that A is true in at least one time instant in interval i;

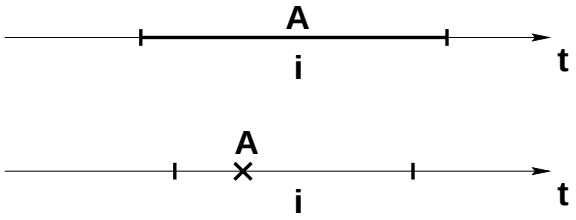

Figure 2.4: Quantitative temporal constraints.

where A is a temporal logic formula and i is an interval that can be either a set of points or a fundamental entity whose extremes are expressed quantitatively (see Figure 2.4). By using these two elementary operators most of the possible temporal requirements of real-time systems can be expressed. For example:

- when A happens, B must happen within t time units: $A \Rightarrow \Diamond_{[0,t)} B$
- when A happens, B must happen after t time units: $A \Rightarrow \Box_{[t,t]} B$
- the distance between two occurrences of event A is at least t time units: $A \Rightarrow \Box_{(0,t)} \neg A$
- the distance between two occurrences of event A is always equal to t time units:

$$A \Rightarrow (\Box_{(0,t)} \neg A) \wedge (\Box_{[t,t]} A)$$

where intervals are specified using the usual mathematical notation, with round and squared brackets used for excluding and including bounds, respectively. The intervals are defined relative to the instant in which formulas are evaluated, so the time is implicit.

The above two elementary temporal operators are sufficient for expressing *safeness* or *liveness*. For example, the classical safety conditions, such as $\Box_i A$ (where A is a positive property) must be satisfied by the system specification, where the interval i can be extended to the *specification temporal domain*, as well as to only a part of it. Liveness conditions, such as $\Diamond_i A$ (A will be satisfied within i) or deadlock-free conditions, such as $\Box_j(\Diamond_i \neg A)$ can also be specified.

If unbounded intervals are allowed operators $\Box$, $\Diamond$, $\blacksquare$ and $\blacklozenge$ can be defined as:

- $\Box\phi \equiv \Box_{(0,+\infty)}\phi$;
- $\Diamond\phi \equiv \Diamond_{(0,+\infty)}\phi$;
- $\blacksquare\phi \equiv \Box_{(-\infty,0)}\phi$.
- $\blacklozenge\phi \equiv \Diamond_{(-\infty,0)}\phi$.

Certain temporal logics also provide bounded versions of the operators **since** and **until**. These versions can be easily obtained from the unbounded operators **since** and **until** and the bounded operators **henceforth** and **always**.

Some other temporal logics are much more oriented towards presenting the behavior of predicates intended as signals. These logics have been frequently used for modeling digital signals and are typically based on intervals. In order to relate the definition of an interval for bounding predicates with the evolution of other predicates a special operator for capturing the time instant related to events is needed. This special function from $Predicate \rightarrow Time$ is frequently introduced by using special operators.

2.2.5 Events and Ordering of Events

Typical relationships of cause-effect can be specified by using the simple operators imply ($\Rightarrow$) and co-imply ($\Leftrightarrow$). Moreover, the simpler operators of temporal logic (i.e., $\Box$ and $\Diamond$) can be profitably used for describing facts and rules. A *fact* is a predicate that is true at least for a time (e.g., presence of an event), while a *rule* is a predicate that is true in all time instants. These operators are unsuitable for specifying relationships of ordering among events, such as:

(a) A precedes B;

(b) A follows B;

(c) A will be true until B will become true for the next time;

(d) A has been true since the last time that B was true for the last time;

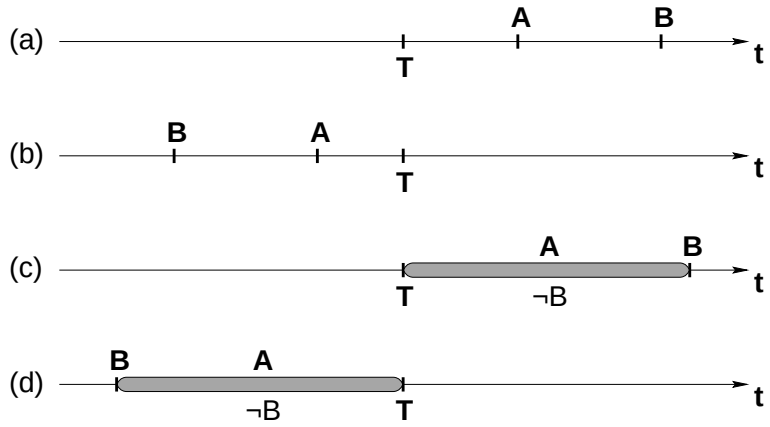

Figure 2.5: Constraints about the ordering between events.

where A and B are temporal logic formulas. In Figure 2.5 graphical representations of (a) through (d) are shown, where **T** represents the instant in which formulas are evaluated. Constraints (c) and (d) may be described by using operators **until** and **since**, respectively. The precedence relation in the future (past) may be defined with operator **until** (**since**), as is shown by Manna in [94], defining operators **precede** and **follow**:

$$\begin{aligned} A \textbf{ precede } B &\equiv \neg((\neg A)\textbf{until}B) \\ A \textbf{ follow } B &\equiv \neg((\neg A)\textbf{since}B). \end{aligned}$$

Therefore, in order to express the ordering between events the temporal logic has to provide the operators **until** and **since**.

In effect, several versions of until and since operators exist. The typical definition of the until/since operator is the "weak" definition:

- A **until**$_w$ B – is true if B will become true and until that instant A will be true, or if B will stay always false and A always true.

- A **since**$_w$ B – is true if B has been true since the instant in which A became true, or if B has been always false and A always true.

The strong version of these operators assumes the occurrence of the change of status for B. Therefore, they can be defined in terms of the above operators as follows:

- A **until** $B \equiv \Diamond B \wedge A$ **until**$_w$ B
- A **since** $B \equiv \blacklozenge B \wedge A$ **since**$_w$ B

Different versions can be defined, and the current time can also be included in the evaluation range of the operators. In this case, the so-called 0 version of the weak version of the operators can be defined as follows:

- A **until**$_{w0}$ $B \equiv B \vee (A \wedge A$ **until**$_w$ $B)$
- A **since**$_{w0}$ $B \equiv B \vee (A \wedge A$ **since**$_w$ $B)$

Other versions can be defined for combinations of the basic versions stated above.

2.2.6 Time Implicit, Explicit, Absolute

Time in temporal logics can be defined in an *implicit* or *explicit* manner. A time model is implicit when the meaning of formulas depends on the evaluation time, and this is left implicit in the formula. For instance, $\Box A$ means that:

$$\forall t \in [T_0, \infty].A(t)$$

where T_0 is the evaluation time (the so-called current time instant). When time is implicit, the formalism is able to represent the temporal ordering of events. Each formula represents what happens in the evaluation time (e.g., in the past or in the future of the evaluation time), which is the implicit current time:

$$\Diamond \Box_{[3,5]} A$$

means that A will be eventually true in the future for an interval of 3 to 5 time units later with respect to the evaluation time. If time is treated implicitly, the possibility of referring the specification to an absolute value of time is usually lost. Temporal logics with time implicit may or may not allow the quantification over time (e.g., TRIO allows quantification over time and adopts a implicit model of time).

On the contrary, when the time is explicit the language represents the time through a variable. In this way, it is possible to express any useful property of real-time. The explicit specification of time allows the specification of expressions that have no sense in the time domain – e.g., the activation of a predicate when the time is even.

The reference to time can be *absolute* or *relative*. It is considered absolute when the value of the current time is referenced to a general system clock (the clock is idealized in the sense that no drift is supposed). It is frequently represented with T; for example, in the following formula an absolute explicit model of time is used:

$$\forall t.\Box(E \wedge T = t) \rightarrow \Diamond(A \wedge T - t < 10ms)$$

where E is an event. When time is expressed in absolute form, time durations and deadlines are given directly in seconds or milliseconds (i.e., the absolute time on the clock). Therefore, the meeting of timing constraints depends on the context (machine type, number of processes, workload, etc.).

The formula that follows has a relative explicit model of time:

$$\forall t.\Box(E \wedge T = t) \rightarrow \Diamond(A \wedge T - t < 10)$$

Frequently, time is expressed in a relative manner – that is, time durations and deadlines are given in time units. In this case, the relationship between these time units and the absolute measure of time expressed in seconds (or milliseconds) is left until the implementation phase. However, the validation of specifications becomes almost implementation independent. A different definition for absolute and relative time has been reported in [82].

2.2.7 Logic Decidability

The decidability of a temporal logic is related to the concepts of validity and satisfiability. A formula is satisfiable if there exists an interpretation for the symbols in the formula for which the formula is true, whereas a formula is valid if for every interpretation the formula is true. This feature is strongly related to the order of the logic. First-order (discrete time) temporal logic is incomplete, and validity and satisfiability problems are undecidable in the general case. This is mainly due to the quantification of time dependent variables. The prohibition of this kind of quantification has often been shown to be a necessary condition for the existence of feasible automated verification mechanisms such as in TPTL [9].

Satisfiability (validity) is a decidable problem for a logic if there exists a decision procedure for the satisfiability (validity) of every formula of the logic. If one of these problems is

decidable for the logic then the proof of theorems may be automatic. This property is highly desirable because it increases the logic's usability, since automatic instruments to verify and validate specifications can be built. This property is much more useful for temporal logics that are based on property proofs for the verification and validation of system properties. The adoption of a theorem prover confers an absolute certainty about the behavior of the system.

Other temporal logics have a semantics defined in terms of state evolution. This makes their application much more operational than descriptive [35]. For these models, verification and validation activities are typically performed by using model-checking techniques. Unfortunately, for real systems, the verification of the system behavior in all its states can be infeasible because it is too complex and time consuming, even using symbolic model-checking algorithms. A semantics based on state is frequently associated with the presence of an event-based temporal logic or of a discrete linear model of time. In both these cases, the definition of an operational semantics for the temporal logic is quite simple.

2.2.8 Deductive System *sound* and *complete*

As expressed in Section 2.1.1, a deductive system is a formalization of the deduction process that is usually used to make proofs manually. A deductive system permits one to build proofs manually in simpler way and provides the basis for automating some simple rewriting of formulas. These mechanisms are typically used in automatic and semiautomatic theorem provers. Naturally it must be proved that this deductive system is *sound*, so that all proofs built are correct.

Another desirable but less "necessary" property, is the completeness of the deductive system; that is, the capacity to build a proof for *every* theorem true for the logic. It should be noted that it is never possible to build a *complete* deductive system: for example, the theory of natural numbers on FOL is sound but not complete; that is, there are non-provable true formulas [48].

2.2.9 Logic Specification Executability

The problem of executability of specifications given by means of temporal logics has often been misunderstood. This mainly depends on the meaning assigned to executability [60], [110], [18]. There are at least three different definitions of executability, as follows.

(i) Specification models are considered to be executable if they have a semantics defining an effective procedure, capable of determining for any formula of the logic theory,

whether or not that formula is a *theorem* of the theory [110]. In effect, this property corresponds to that of decidability of the validity problem rather than to that of system specification executability.

(ii) A second meaning refers to the possibility of generating a model for a given specification [56]. A detailed version of this concept leads to verifying if an *off-line* generated temporal evolution of inputs and outputs is compatible with the specification. This operation is usually called history checking.

(iii) The last meaning for executability consists of using the system specification itself as a prototype or implementation of the real-time system, thus allowing, in each time instant, the *on-line* generation of system outputs on the basis of present inputs and its internal state and past history. When this is possible, the specification can be directly executed instead of traducing it in a programming language.

In the literature, there exist only few executable temporal logics that can be used to build a system prototype according to meaning *(iii)* of executability. In general, the execution or simulation of logic specifications with the intent of producing system outputs in the correct time order by meeting the temporal constraints is a quite difficult problem. The difficulty mainly depends on the computational complexity of the algorithms proposed.

Moreover, while executing propositional temporal logics is a complex task, executing first order temporal logics is undecidable and highly complex [60], [102]. A solution for executing propositional temporal logics could be (a) to restrict the logic and providing an execution algorithm for the remaining part, or (b) to execute the complete logic by using specific inferential rules and/or backtracking techniques. For first order temporal logics the solution can be to apply the same approaches used for propositional temporal logics or to try to build a model for the formula as in *(i)* and *(ii)* above.

If a temporal logic is executable the system can be simulated and/or executed. Thus, it is possible to validate system behavior through simulation and to use the system specification as a prototype or as an implementation if the execution speed is high enough to satisfy temporal constraints of the system.

2.3 A Selection of Temporal Logics

This section presents a selection of the most interesting types of temporal logics for the specification of real time systems. There are many other temporal logics in the literature, but most of them can be regarded as generalizations or specializations of those discussed here in.

The order in which the logics are presented is quite close to the chronological, from the earliest to the latest, from the simplest to its more complex evolutions (if present). Several examples are given in order to make the comparisons among the temporal logics presented possible. The section concludes with a brief discussion of the logics and a table for comparison purposes.

2.3.1 PTL: Propositional Temporal Logic

The Propositional Temporal Logic (PTL) introduced by Pnueli [124], [125], [126] (see also [22]), extends the propositional logic introducing temporal operators $\Box$, $\Diamond$, $\bigcirc$, and $\mathcal{U}$. The propositions of PTL describe temporal relationships between states that characterize the temporal evolution of the system. PTL is an event-based logic and does not provide a metric for time.

System requirements are specified by describing a set of constraints on the event sequences that occur in the system modifying its state. Time consists of a sequence of instants corresponding with the sequence of states of the system. In a certain sense, the fundamental entity of the logic is the instant in which the state of the system changes. For these reasons it is particularly suitable for integration in operational models such as state machines [35].

The temporal structure of PTL is linear, bounded in the past (an initial instant exists), unbounded in the future (an infinite sequence of future states exists) and discrete (i.e., the set of instants is modeled with the set of natural numbers). For this reason, only temporal operators in the future are present. The temporal operators $\Box$, $\Diamond$ and $\mathcal{U}$ correspond to the operators $\boldsymbol{G}$, $\boldsymbol{F}$ and **until** described in Section 2.1.4. The formula $\bigcirc\phi$ is a valid formula if the formula ϕ is true in the *next* state. Operator **until** in PTL is equivalent to $\mathbf{until}_0$ presented in Section 2.2.5. Since PTL provides the operator **until** it is possible to specify real-time system requirements about the order of events in the future. The asymmetry of the logic (due to the boundary in the past) and the absence of the operator **since** does not permit specification of requirements about the order of events in the past. Moreover, the absence of a metric for time does not allow specification of any type of quantitative temporal constraint. Therefore, PTL is much more suitable for use with reactive and concurrent systems than with real-time systems. Reactive systems are typically event-driven and do not present quantitative temporal constraints such as timeouts or deadlines.

PTL is decidable (for example using a decision procedure based on the semantic tables method) and it is possible to build a *sound* and *complete* deductive system for the logic. In the literature, methods or instruments for executing PTL formulas have not been presented and, in general, these formulas are not executable.

In [95], Manna and Pnueli proved that for an extension of PTL built adding symmetric operators in the past for ■, ◆, ● and $\mathcal{S}$ it is possible to transform formulas of a particular class in finite state machines, thus permitting the execution of some formulas of this extension of PTL.

Table 2.1 shows some examples of the extended version of PTL. The table also shows a set of specifications that cannot be expressed by using this temporal logic. In the next subsections, similar tables are provided to allow comparison of the several temporal logics on the basis of a collection of equivalent specifications.

In [16], [17], [58], METAMEM is presented. METATEM includes an executable model and algorithm and can be considered to be based on an extended version of PTL.

Table 2.1: Some specifications in extended PTL.

meaning	PTL
Always A in the Past	●■A
Always A in the Future	○□A
Always A	■$A \wedge$ □A
A Since Weak B	●($A \mathcal{S} B \vee$ ■A)
A Until Weak B	○($A \mathcal{U} B \vee$ □A)
Lasts A up to t_1	–
Lasted A from $-t_1$	–
A Within $-t_1$ in the Past	–
A Within t_1 in the Future	–
A Within $(-t_1, t_2)$	–
A Was true in $(-t_1, -t_2)$	–
A Will be true in (t_1, t_2)	–
A Could be true in (t_1, t_2)	–
A Since B during $(-t_1, -t_2)$	–
A Until B during (t_1, t_2)	–

2.3.2 Choppy Logic

The Choppy Logic presented by Rosner and Pnueli [131] is an extension of PTL obtained by adding operator $\mathcal{C}$ (*chop*). This logic has all characteristics of PTL and enhances its expressiveness with operator $\mathcal{C}$, that permits one to concatenate state sequences. In the first approximation, the Chop operator can be regarded as an operator for dividing time intervals. In particular, a state sequence σ is a model for formula $\phi \mathcal{C} \psi$ if it can be divided in two sequences σ' and σ'' such that: σ' is a model for ϕ, and σ'' is a model for ψ. This logic has a greater expressiveness than PTL, but a more complex decision procedure is required. Thus, the Choppy Logic maintains all merits and problems of PTL.

2.3.3 BTTL: Branching Time Temporal Logic

The Branching Time Temporal Logic (BTTL) introduced by Ben-Ari, Pnueli and Manna [23] is an extension of PTL. It has a temporal structure with branches in the future, and thus could be used for describing the behavior of non-deterministic systems. PTL operators are enhanced to deal with branches. Four operators have been defined to quantify both on different evolution traces and states that are present on the selected traces:

- $\forall\Box$, for all traces π and for all states $s \in \pi$;
- $\exists\Box$, for at least one trace π and for all states $s \in \pi$;
- $\forall\Diamond$, for all traces π and at least one state $s \in \pi$;
- $\exists\Diamond$, for at least one trace π and for at least one state $s \in \pi$.

In several aspects BTTL is practically equivalent to PTL. Moreover, it adopts a temporal structure branched in the future. BTTL also presents a *complete* axiomatization; it is decidable; and the satisfiability of formulas can be determined by using a method based on semantic tables that also produces models for the BTTL formulas. The models for the formulas are finite and could be used to build finite state machines corresponding to the formulas. This makes the model operationally executable. Even with this improvement of PTL it is not possible to specify quantitative temporal constraints. Thus, this logic is also not suitable for real-time systems specification.

2.3.4 ITL: Interval Temporal Logic

The Interval Temporal Logic (ITL) introduced by Halpern, Manna and Moszkowski [67] and used/further studied by Moszkowski in [109], [108], [110] can be considered as an extension of PTL. ITL is a propositional logic with a temporal structure that is bounded in the past, unbounded in the future, discrete and linear. The fundamental entity of ITL is the interval made of a sequence of states. The length of an interval is defined as the number of states in the sequence. ITL does not provide a metric for time and can be considered an event-based logic. It has been applied for modeling the evolution of digital signals. ITL extends the propositional logic with the operators $\bigcirc$ (*next*), $\Box$, $\Diamond$ and ";" (*chop*, analogous to operator $\mathcal{C}$ of Choppy Logic). The semantics of all these operators is defined in terms of intervals rather then of states as in PTL. From the above basic operators a set of derived operators has been defined. The presence of operator *chop* makes the satisfiability of ITL formulas

undecidable; nevertheless, the satisfiability is decidable for a particular subclass of ITL formulas. It is possible to build a *sound* deductive system for ITL.

In [110] Tempura is presented. It is a subset of ITL formulas with some syntactic properties, for which the problem of building an execution for formula is tractable, even if unsolvable in the general case. In ITL only order properties showing qualitative relationships among the order of events can be specified. This makes this logic less powerful for specifying real-time systems. To specify order properties the operator *chop* must be used since ITL does not have operator **until**. As a surrogate of the metric for time a special operator $Len(n)$ is used to count the number of states in a sequence. This allows one to specify the exact duration in terms of number of transitions among events.

2.3.5 PMLTI: Propositional Modal Logic of Time Intervals

The Propositional Modal Logic of Time Intervals (PMLTI) presented by Halpern and Shoham [68] is a temporal logic that extends the propositional logic. The fundamental temporal entity is the interval and the temporal operators can express the possible relationships between intervals, as reported in Figure 2.3. The temporal structure requires only the total order of the points in the intervals. With this limitation, the time structure can be linear or branched, bounded or unbounded, dense or discrete. PMLTI does not provide an explicit metric for time. The selection of a specific temporal structure leads to implications about the complexity of the decision procedure for demonstrating the validity of formulas. The problem of validity and satisfiability of PMLTI formulas may be decidable or undecidable depending on the temporal structure chosen.

PMLTI uses a method of translating temporal logic formulas in FOL formulas of a specific deductive system to proof theorems of the logic. This approach enables application of all the techniques which are available for first order logic. To date, the problem of formula executability has not been addressed. The presence of operators for the specification of relationships between intervals permits one to easily express event order constraints. However, the absence of a metric for time makes the expression of quantitative temporal constraints impossible.

2.3.6 CTL: Computational Tree Logic

The Computational Tree Logic (CTL) presented by Clarke, Emerson and Sistla [41], [42], [140] is a propositional branching time temporal logic. The fundamental temporal entity is the point and presents specific operators for reasoning about the system behavior in terms of several futures, called sequences. It is very similar to BTTL. CTL does not provide

an explicit metric for time. For verifying CTL specification a model-checking approach is typically used since the specification can be modeled as a state machine [41]. In [52], [53] a real-time extension of CTL has been presented, RTCTL, presenting a metric for time. The satisfiability problem for this logic is doubly exponential. The model-checking has a polynomial time algorithm, [117]. In [81], a modular version of CTL has been presented, MCTL.

2.3.7 IL: Interval Logic

The Interval Logic (IL) presented by Schwartz, Melliar-Smith and Vogt [134], [133] is based on time interval and propositional logic. The temporal structure is linear, bounded in the past and unbounded in the future. IL does not present an explicit metric for time. Time intervals are bounded by events and by the changes of system state described by the formulas. Therefore, IL is an event-based logic. A typical IL formula is in the following form:

$$[\mathcal{I}]\alpha,$$

where α is a formula and $\mathcal{I}$ is the interval that is the context of which formula α has to be verified. This formula means that the next time the interval can be built then the formula α will hold in it. The most interesting feature of IL is the set of instruments that can be used for the determination and construction of time intervals. It presents bounded versions of operators $\Diamond$ and $\Box$. The bound is defined by means of the interval: $[\mathcal{I}]\Diamond\alpha$ means that α can be true in $\mathcal{I}$. The interval bounds can be defined by occurrence of events. Given an interval the initial and final intervals can be extracted. Moreover, the existence of an interval with certain characteristics is an event. Finally, to describe system behavior operators *at*, *in* and *after* have been defined; these specify the truth at the start, during and at the end of the interval, respectively. They may be used as events for construction of intervals. For instance: $A \Rightarrow$ as interval means that the interval starts when A starts and ends at the end of the context.

Results for testing the executability of IL do not exist. This is because IL has been introduced as a specification language and is verified by means of automatic instruments, without taking into consideration the possibility of simulating or executing the specifications.

IL permits one to easily write constraints about the order of events using the instruments for the construction of context intervals, but it cannot be used to specify quantitative temporal constraints, as can the extensions discussed in subsections 2.3.8 and 2.3.9 below.

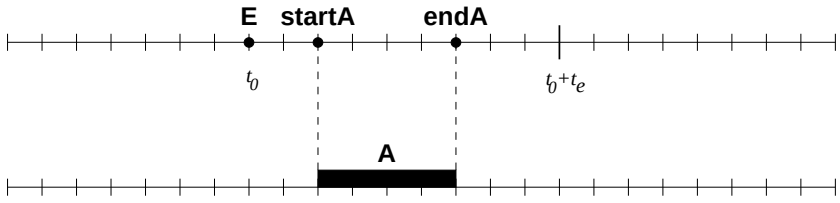

Figure 2.6: One example for several temporal logics.

2.3.8 EIL: Extended Interval Logic

Extended Interval Logic (EIL) was introduced by Melliar-Smith [101]. It extends IL by adding the possibility of specifying some types of quantitative temporal constraints. These extensions have been introduced to eliminate the incapacity of IL to express the typical requirements of real-time systems. The first extension is the possibility of defining an event from another event at a constant temporal distance (positive or negative): if E is an event then $E + 1\text{sec}$ is also an event. The second extension is the possibility of limiting the length of intervals. For example: formula $< 2\text{sec}$ is true if the interval in which it is evaluated has a duration of less than 2 seconds, while $> 10\text{min}$ is true if the interval has a duration greater than 10 minutes. The extensions introduced add the capability of expressing some of the quantitative temporal constraints that are needed to specify real-time systems. For instance:

$$\Box[E \Rightarrow *endA](< t_e \wedge *startA)$$

means that for each occurrence of event E predicates *startA* and *endA* (marking an interval in which A is true) hold and this interval is included from the occurrence of the E and t_e (see Figure 2.6, in which t_0 is time instant in which E occurs). In the above formula operator $*$ can be read as *exists an occurrence of*, while $\Rightarrow$ means that the left bound of the interval is defined by the occurrence of event E.

2.3.9 RTIL: Real-Time Interval Logic

The Real-Time Interval Logic (RTIL) presented by Razouk and Gorlick [128] is another extension of IL. In this case the goal was to permit the specification of real-time systems with the specific intention of verifying the consistency between the execution traces and the system specification itself. RTIL extends IL by introducing a metric for time. It can assign a temporal value to the extremes of the intervals and can construct intervals by assigning numerical values at interval bounds, not only by using events and state changes. Moreover, it is possible to measure the interval duration. This characteristic makes RTIL interesting

for the specification of real-time systems. For example, the specification described in Figure 2.6 can be written as:

$$\Box[\odot E \hookrightarrow t_e] * (\odot\ startA \Rightarrow \odot\ endA)$$

In this case, operator $*$ has to be read as *exists a subinterval*. The special operator $\odot A$ extracts the time instant in which A becomes true. *endA* and *startA* have the same meanings as in EIL. Instants can be specified absolutely or relative to the beginning of the current context. RTIL also permits quantification over finite domains. This feature does not enhance the expressiveness of the logic but simplifies the writing of complex and repetitive formulas.

2.3.10 LTI: Logic of Time Intervals

The Logic of Time Intervals (LTI) of Allen [5] is an interval temporal logic of the second order. It is also called Interval Time Logic (acronym ITL). To avoid confusion with the ITL presented it will be referred to in this paper as LTI. Intervals can be divided in subintervals. Intervals that cannot be further divided into subintervals constitute *moments*. The logic permits quantification of temporal intervals. The temporal structure is linear, without any further limitations – even the model of time can be either discrete or dense. LTI does not provide an explicit metric for time. Temporal propositions are made by declaring the order relationships between intervals (see Figure 2.3). In [85], it has been shown that LTI theory is incomplete and proposes a way to make it complete. Furthermore, it is shown that both the theories, the new and complete, and the early and incomplete versions, are decidable. An axiomatic system is provided for both, although there are not known results about logic executability. LTI does not present problems for ordering constraints regarding the expression of the typical temporal constraints of real-time systems. Specification of quantitative temporal constraints is impossible since the measure of the length of intervals is missing.

2.3.11 RTTL: Real-Time Temporal Logic

The Real-Time Temporal Logic (RTTL) presented by Ostroff and Wonham [118], [115], [119], [117] extends PTL with proof rules for real-time properties. The temporal structure is linear and discrete; the fundamental entity is the point. Time is limited in the past and unlimited in the future. Time is defined with both a sequence of state and a sequence of temporal instants. The presence of a state-based model makes RTTL particularly suitable for model-checking techniques; thus it can be used as a model to verify small systems. A

natural number is associated with each time instant; thus, RTTL is based on an explicit model of time. The *clock* of the system is periodically incremented and it is accessible for writing formulas. State changes can occur: (i) corresponding with the changes of time of system, or (ii) between two successive instants. In the case in which more events occur between two successive instants, these events are distinguishable only for the order in which they occur, and not for the temporal instant associated with the occurrences. For this reason, the metric for time is only partial: non-simultaneous events that occur for the same value of the system clock may exist. Operator **until** of PTL and operator **until** of RTTL are equivalent to $\mathbf{until}_0$ presented in Section 2.2.5. In RTTL, quantification of *rigid* variables is allowed. Rigid variables are variable in the set of possible executions but are constant for each execution. RTTL is a first order logic. For RTTL it is essential that the system clock (T) value be referenced in formulas to express some types of concepts, such as to establish relationships between different temporal contexts. Table 2.2 shows some RTTL specifications; no specifications involving the past are shown since RTTL presents only the future.

Table 2.2: Some specifications in RTTL.

meaning	RTTL
Always A in the Past	–
Always A in the Future	$\bigcirc\Box A$
Always A	–
A Since Weak B	–
A Until Weak B	$\bigcirc(A\mathcal{U}B) \vee \Box A$
Lasts A up to t_1	$t = T \rightarrow \Box((t < T \wedge T < t + t_1) \rightarrow A)$
Lasted A from $-t_1$	–
A Within $-t_1$ in the Past	–
A Within t_1 in the Future	$t = T \rightarrow \Diamond((t < T \wedge T < t + t_1) \wedge A)$
A Within $(-t_1, t_2)$	–
A Was true in $(-t_1, -t_2)$	–
A Will be true in (t_1, t_2)	$t = T \rightarrow \Box((t + t_1 < T \wedge T < t + t_2) \rightarrow A)$
A Could be true in (t_1, t_2)	$t = T \rightarrow \Diamond((t + t_1 < T \wedge T < t + t_2) \wedge A)$
A Since B during $(-t_1, -t_2)$	–
A Until B during (t_1, t_2)	$t = T \rightarrow A\mathcal{U}(B \wedge t + t_1 < T \wedge T < t + t_2)$

All global variables (e.g., t in the table) in formulas are assumed to be universally quantified [116]. The logic also presents **next** operator $\bigcirc$. "Lasts A up to t_1" can be also written in a more concise notation $\Diamond_{(0,t_1)}A$ while "A Until B during (t_1, t_2)" can be specified as $A\mathcal{U}_{(t_1,t_2)}B$. The situation described in Figure 2.6 can be specified by using:

$$\Box\left(E \rightarrow \bigcirc\left((\Diamond_{\leq t_e}\ endA) \wedge \neg(\neg startA\ \mathcal{U}\ endA)\right)\right)$$

considering predicates $startA$ and $endA$ as above. Note the adoption of bounded operator

$\Diamond$. The possibility of adopting (i) an explicit reference to the system clock value, and (ii) indirect quantifications on values assumed by the clock leads to the ability to write every type of ordering and quantitative constraints (the above example is implicitly quantified on t). This is extremely interesting for the specification of real-time systems. However, this flexibility leads to the production of formulas that are quite difficult to understand and manipulate with respect to other temporal logics that avoid quantification over time-dependent variables. A *sound* deductive system has been built for RTTL (extending a deductive system of PTL), but the satisfiability problem is undecidable. The suitability of RTTL for model checking and the presence of a deductive system makes RTTL a dual model according to the classification reported in [35]. No results about the executability of RTTL specification are available. TTM/RTTL is a dual approach obtained by the integration of a state machine model and RTTL [118], [115], [119]. TTM is an operational model based on communicating finite state machines in which variables with arbitrary domains are used. The *operations* allowed are variable assignment, send, and/or receive. The state machine follows a Mealy model in which conditions on transitions between states are equivalent to logic formulae on state variables, while the output is an assignment to state variables.

2.3.12 TPTL: Timed Propositional Temporal Logic

In [8], Alur and Henzinger presented the Timed Propositional Temporal Logic (TPTL) and in [9] they have shown the expressiveness and complexity of this logic. TPTL is an extension of PTL. Like PTL, TPTL is a propositional logic, where the instant is the fundamental temporal entity and the time is linear, discrete, limited in the past, unlimited in the future. An extension with respect to PTL is the presence of a metric for time: every instant corresponds to a natural number and a monotone function associates a temporal value with each state of the system, thus making timed state sequences possible. The presence of operator **until** permits one to specify order constraints. The possibility of specifying quantitative temporal constraints is one of the fundamental characteristics of the logic. For these reasons, this logic is suitable for specifying real-time systems requirements. Its theoretic bases that facilitate requirement verification and validation. Table 2.3 shows some specifications in TPTL. No specifications in the past are shown since TPTL presents only the future. TPTL adopts the *freeze* operator, thus x and y represent time instants. The specifications are quite similar to RTTL. Adoption of *freeze* operator can be very interesting to model system in which more than a real-time clock is present. A typical application is the specification of communicating systems in which distinct specifications have to be synchronized (see APTL in [145]).

Table 2.3: Some specifications in TPTL.

meaning	TPTL
Always A in the Past	–
Always A in the Future	$\bigcirc\Box A$
Always A	–
A Since Weak B	–
A Until Weak B	$\bigcirc(\mathcal{U}BA \vee \Box A)$
Lasts A up to t_1	$x.\Box y.(x < y < x + t_1) \rightarrow A$
Lasted A from $-t_1$	–
A Within $-t_1$ in the Past	–
A Within t_1 in the Future	$x.\Diamond y.(x < y < x + t_1) \wedge A$
A Within $(-t_1, t_2)$	–
A Was true in $(-t_1, -t_2)$	–
A Will be true in (t_1, t_2)	$x.\Box y.(x + t_1 < y < x + t_2) \rightarrow A$
A Could be true in (t_1, t_2)	$x.\Diamond y.(x + t_1 < y < x + t_2) \wedge A$
A Since B during $(-t_1, -t_2)$	–
A Until B during (t_1, t_2)	$x.\bigcirc\mathcal{U}\,(y.B \wedge (x + t_1 < y < x + t_2))\,A$

The situation described in Figure 2.6 is specified in TPTL by using:

$$\Box x.E \rightarrow (\Diamond y.\ endA \wedge y \leq x + t_e) \wedge \neg(\mathcal{U}\ endA\ \neg startA)$$

considering predicates *startA* and *endA* as above. In [9], it has been proven that the choice of the set of natural numbers for a temporal domain is essential to obtaining a temporal logic for which the satisfiability problem is decidable. In fact, for every temporal domain with a more complex structure than natural numbers, the problem of satisfiability is undecidable. PTL's deductive systems can be extended and transformed for TPTL by retaining the properties of *soundness* and *completeness*. Moreover, a decision procedure based on the semantic table algorithm and a model-checking algorithm has been presented. This facilitates the use of this logic for the specification and verification of real-time systems requirements.

2.3.13 RTL: Real-Time Logic

The Real-Time Logic of Jahanian and Mok [80] is a logic that extends the first-order logic with a set of elements for the specification of real-time systems requirements. RTL proposes a logic approach for the specification of real-time systems, but is not a temporal logic in the classical meaning. It presents an absolute clock to measure time progression. The value of this clock can be referenced in the formulas: function "@" permits one to assign a temporal value (execution instant) to an event occurrence. The temporal domain is the set of natural numbers, and is linear, discrete limited in the past, unlimited in the future, and

totally ordered. The fundamental entity is the time instant. In RTL, there are no problems in specifying ordering and quantitative temporal constraints, since it is possible to make explicit reference to time even through quantification. The main problem with RTL is the fact that absolute system time is referenced, with a low level of abstraction, leading to very complex formulas required to describe the system. The example of Figure 2.6 is specified in RTL by using:

$$\forall t.\forall i.@(\Omega E, i) = t \rightarrow (\exists j.(t \leq @(\uparrow A, j)) \wedge (@(\downarrow A, j) \leq t + t_e))$$

Operator ΩE states the occurrence of external event E; $\uparrow A$ the turning true from false of predicate/signal A; $\downarrow A$ the becoming false from true of A; i and j are the occurrences of the events marked with operator @; t is the time. Note the need of a quantification over time to specify the example. In [9], it has been shown that RTL is undecidable even when the syntax is restricted. In [80] a procedure to demonstrate the consistency of *safeness* assertions relative to real-time system specification is proposed. A deductive system for RTL has not been presented, but it seems to be feasible by extending a system for FOL with laws for the new operators. There are no known results regarding the executability of RTL. In [13] an approach based on RTL and Statechart was presented. In that case, the formal verification was provided by using a theorem prover.

2.3.14 TRIO: Tempo Reale ImplicitO

TRIO is a logic language for real-time system specification (Tempo Reale ImplicitO - Implicit Real Time). It has been presented by Ghezzi, Mandrioli and Morzenti [63], [57]. TRIO extends FOL with specific predicates for real-time system specification. The temporal structure is linear and totally ordered: possible temporal domains are the natural numbers, the integers, the real numbers, or an interval of one of these set. The fundamental temporal entity is the point and a metric for time is available. On that basis, it is possible to measure the distance of two points and the length of an interval. Since TRIO is an extension of FOL, which is undecidable, then TRIO is also an undecidable logic. TRIO presents only two temporal operators: **Futr**(A, t) and **Past**(A, t) for specifying that A occurs at time instant t in the future and past, respectively (more recently it has been demonstrated that both these operators can be defined in terms of a unique operator). Moreover, in TRIO, based on these operators, several other operators can be defined as parametric predicates. This is frequently allowed by many temporal logics - e.g., TILCO, MTL. The temporal operators introduced by TRIO, with the possibility of quantification on temporal variables without any restriction, permit the expression of order and quantitative temporal constraints as needed for real-time systems specification. It is necessary

to use quantification over the time domain, so formulas are often complex and difficult to read and manipulate. Table 2.4 shows some specifications in TRIO, the table presents three columns. The middle column shows the specification written on the basis of TRIO's elementary operators, while the column on the right shows the version of the specification in a derived form. This derived form can be obtained by defining a new temporal operator (special parameterized predicate) with the specification reported in the middle column or by using already defined operators. It is possible to define new "temporal operators" by means of special functions: on the one hand, this keeps the size of formulas low, but on the other hand, it makes the language harder to understand. A large number of operators can create confusion during the specification process, especially when these specifications have to be understood by other analysts who do not know the definitions of the same predicates implementing complex temporal operators.

Table 2.4: Some TRIO simple temporal specifications.

meaning	TRIO	TRIO derived
Always Past	$\forall t(t > 0 \rightarrow \mathbf{Past}(A, t))$	AlwP(A)
Always Future	$\forall t(t > 0 \rightarrow \mathbf{Futr}(A, t))$	AlwF(A)
Always	$\forall t(t > 0 \rightarrow \mathbf{Futr}(A, t)) \wedge A \wedge \forall t(t > 0 \rightarrow \mathbf{Past}(A, t))$	Alw(A)
Since Weak	$\forall t''(t'' > 0 \rightarrow \mathbf{Past}(A, t'')) \vee$	
	$\exists t(t > 0 \wedge \mathbf{Past}(B, t) \wedge \forall t'(0 < t' < t \rightarrow \mathbf{Past}(A, t)))$	Since$_w(B, A)$
Until Weak	$\forall t''(t'' > 0 \rightarrow \mathbf{Futr}(A, t'')) \vee$	
	$\exists t(t > 0 \wedge \mathbf{Futr}(B, t) \wedge \forall t'(0 < t' < t \rightarrow \mathbf{Futr}(A, t)))$	Until$_w(B, A)$
Lasts	$\forall t'(0 < t' < t \rightarrow \mathbf{Futr}(A, t'))$	Lasts(A, t)
Lasted	$\forall t'(0 < t' < t \rightarrow \mathbf{Past}(A, t'))$	Lasted(A, t)
Within Past	$\exists t'(0 < t' < t \wedge \mathbf{Past}(A, t'))$	WithinP(A, t)
Within Future	$\exists t'(0 < t' < t \wedge \mathbf{Futr}(A, t'))$	WithinF(A, t)
Within	$\exists t'(0 < t' < t_1 \wedge \mathbf{Past}(A, t')) \vee \ A \vee$	
	$\exists t''(0 < t'' < t_2 \wedge \mathbf{Futr}(A, t''))$	Within(A, t_1, t_2)
Was	$\mathbf{Past}(\forall t'(0 < t' < t_1 - t_2 \rightarrow \mathbf{Futr}(A, t')), t_1)$	$\mathbf{Past}(\text{Lasts}(A, t_1 - t_2), t_1)$
Will be	$\mathbf{Futr}(\forall t'(0 < t' < t_2 - t_1 \rightarrow \mathbf{Futr}(A, t')), t_1)$	$\mathbf{Futr}(\text{Lasts}(A, t_2 - t_1), t_1)$
Could be	$\mathbf{Futr}(\neg\forall t'(0 < t' < t_2 - t_1 \rightarrow \mathbf{Futr}(\neg A, t')), t_1)$	$\mathbf{Futr}(\neg\text{Lasts}(\neg A, t_2 - t_1), t_1)$
A Since B	$\exists t(0 < t_2 < t < t_1) \wedge \mathbf{Past}(B, t) \wedge$	
during $(-t_1, -t_2)$	$\forall t'(0 < t' < t \rightarrow \mathbf{Past}(A, t'))$	Since$_B(B, A, t_1, t_2)$
A Until B	$\exists t(0 < t_1 < t < t_2) \wedge \mathbf{Futr}(B, t) \wedge$	
during (t_1, t_2)	$\forall t'(0 < t' < t \rightarrow \mathbf{Futr}(A, t'))$	Until$_B(B, A, t_1, t_2)$

For TRIO, the example of Figure 2.6 is obtained by using:

$$Alw\,(E \rightarrow \exists t\,((0 < t < t_e) \wedge \mathbf{Futr}(endA, t) \wedge WithinF(startA, t)))$$

In this case, the specification has been obtained by using a user-defined operator $WithinF()$; its definition is provided in Table 2.4. Even in this case quantification over time is needed. The same specification could be given by using operator *until* without the adoption of the quantification over time:

$$Alw\,(E \rightarrow WithinF(endA, t_e) \wedge \neg Until(endA, \neg startA))$$

A deductive system for TRIO has been presented. This system has been used to prove theorems for TRIO and to build a deductive system for Timed Petri Nets. TRIO has been used mainly for the validation and verification of system requirements through testing activity (history checking) and not by means of the proof of system properties. TRIO has been described as an executable logic language in the general sense. It can be used to build a model of the system under specification as TRIO formulas. Histories of system variables can be checked against the specification in order to verify whether they satisfy the specification. Therefore, TRIO must be considered a specific case of model checking and not a full execution according to the classification of [60].

2.3.15 MTL: Metric Temporal Logic

In [82] Koymans presented Metric Temporal Logic (MTL) that extends FOL with temporal operators from modal logic:$\boldsymbol{G}$, $\boldsymbol{F}$, $\boldsymbol{H}$, $\boldsymbol{P}$. MTL includes a metric for time according to some properties that describe the structure of the temporal domain. One of these properties states that the order of the temporal structure has to be total, thus leading to a linear temporal structure. The fundamental entity of the logic is the temporal point. The presence of the metric for time permits one to modify the temporal operators making temporal versions of most of the above-discussed temporal operators: $\boldsymbol{G}$, $\boldsymbol{F}$, $\boldsymbol{H}$, $\boldsymbol{P}$. This allows one to reduce the needs of using quantifications on temporal domain. The operators **until** and **since** can be obtained on the basis of the other operators as depicted in Table 2.5. These provide support for avoiding the adoption of quantification over time. In Table 2.5 some MTL specifications are given. MTL presents both past and future operators. The three columns in Table 2.5 have the same meaning as the table presented for TRIO (see Table 2.4).

The example shown in Figure 2.6 for MTL can be obtained by using:

$$E \rightarrow \exists t\,(0 \lll t \lll t_e \wedge \mathbf{F}_t endA \wedge \mathbf{F}_{<t} startA)$$

The same specification could be written without the adoption of the quantification over time

$$E \rightarrow \mathbf{F}_{<te} endA \wedge \neg\,(\neg startA\ until\ endA)$$

As stated in [9] MTL is undecidable, but a deductive system is available. The MTL operators permit one to specify constraints on event order (**until**, **since**) and quantitative

Table 2.5: Some specifications in MTL.

meaning	MTL	MTL Derived
Always A in the Past	$\mathbf{H}A$	
Always A in the Future	$\mathbf{G}A$	
Always A	$\mathbf{H}A \wedge A \wedge \mathbf{G}A$	
A Since Weak B	$\mathbf{H}A \vee \exists t(t \gg 0 \wedge \mathbf{P}_t B \wedge \mathbf{H}_{<t} A)$	A *since* B
A Until Weak B	$\mathbf{G}A \vee \exists t(t \gg 0 \wedge \mathbf{F}_t B \wedge \mathbf{G}_{<t} A)$	A *until* B
Lasts A up to t	$\mathbf{G}_{<t}A$	
Lasted A from $-t$	$\mathbf{H}_{<t}A$	
A Within $-t$ in the Past	$\mathbf{P}_{<t}A$	
A Within t in the Future	$\mathbf{F}_{<t}A$	
A Within $(-t_1, t_2)$	$\mathbf{P}_{<t_1}A \wedge A \wedge \mathbf{F}_{<t_2}A$	
A Was true in $(-t_1, -t_2)$	$\mathbf{P}_{t_1}(\mathbf{H}_{<(t_1-t_2)}A)$	
A Will be true in (t_1, t_2)	$\mathbf{F}_{t_1}(\mathbf{G}_{<(t_2-t_1)}A)$	
A Could be true in (t_1, t_2)	$\mathbf{G}_{t_1}(\mathbf{F}_{<(t_2-t_1)}A)$	
A Since B during $(-t_1, -t_2)$	$\exists t(t_2 \ll t \ll t_1 \wedge \mathbf{P}_t B \wedge \mathbf{H}_{<t} A)$	$\mathbf{P}_{t_2}(A\ \mathbf{since}_{<(t_1-t_2)}\ B) \wedge \mathbf{H}_{<t_2}A$
A Until B during (t_1, t_2)	$\exists t(t_1 \ll t \ll t_2 \wedge \mathbf{F}_t B \wedge \mathbf{G}_{<t} A)$	$\mathbf{F}_{t_1}(A\ \mathbf{until}_{<(t_2-t_1)}\ B) \wedge \mathbf{G}_{<t_1}A$

temporal constraints (***G***, ***F***, ***H***, ***P***). The executability of MTL has not been discussed in the literature.

2.3.16 TILCO: Time Interval Logic with Compositional Operators

In [97], [99], and [98], Mattolini and Nesi presented TILCO (Time Interval Logic with Compositional Operators), a temporal logic for real-time system specification. TILCO extends the FOL and uses as a fundamental temporal entity the interval even if the interval is defined in terms of a couple of time instants. The temporal structure is linear and presents a metric for time that associates an integer number to every temporal instant; no explicit temporal quantification is allowed. In TILCO, the same formalism used for system specification is employed for describing high-level properties that should be satisfied by the system itself. These must be proven on the basis of the specification in the phase of system validation. Since TILCO operators quantify over intervals, instead of using time points, TILCO is more concise in expressing temporal constraints with time bounds, as is needed in specifying real-time systems. The basic temporal operators of TILCO are the existential and universal temporal quantifiers (**@** and **?**, respectively), and operators **until** and **since**). These operators permit a concise specification of temporal requirements, relationships of ordering and quantitative distance among events; thus TILCO fully supports the specification of real-time systems. TILCO is also characterized by its compositional operators that work with intervals: *comma* ",", which corresponds to $\wedge$, and *semicolon* ";", which corresponds to $\vee$, between intervals. Compositional operators "," and ";" assume

different meanings if they are associated with operators "**@**" or "**?**":

$$\begin{aligned} A\mathbf{@}i,j &\equiv (A\mathbf{@}i) \wedge (A\mathbf{@}j), \\ A\mathbf{?}i,j &\equiv (A\mathbf{?}i) \wedge (A\mathbf{?}j), \\ A\mathbf{@}i;j &\equiv (A\mathbf{@}i) \vee (A\mathbf{@}j), \\ A\mathbf{?}i;j &\equiv (A\mathbf{?}i) \vee (A\mathbf{?}j). \end{aligned}$$

Other operators among intervals, such as intersection, "$\cap$", and union, "$\cup$", have been defined by considering time intervals as sets. Table 2.6 shows some specifications in TILCO. In this case, the table has only two columns; even in TILCO, special functions can be easily written for defining new temporal operators, such as in TRIO and MTL. However, in TILCO this is less necessary since TILCO specifications are quite concise, as can be noted by comparing Tables 2.4, 2.5, and 2.6.

Table 2.6: Some specifications in TILCO.

meaning	TILCO
Always A in the Past	$A\mathbf{@}(-\infty, 0)$
Always A in the Future	$A\mathbf{@}(0, \infty)$
Always A	$A\mathbf{@}(-\infty, \infty)$
A Since Weak B	$\mathbf{since}(B, A)$
A Until Weak B	$\mathbf{until}(B, A)$
Lasts A up to t	$A\mathbf{@}(0, t)$
Lasted A from $-t$	$A\mathbf{@}(-t, 0)$
A Within $-t$ in the Past	$A\mathbf{?}(-t, 0)$
A Within t in the Future	$A\mathbf{?}(0, t)$
A Within $(-t_1, t_2)$	$A\mathbf{?}(-t_1, t_2)$
A Was true in $(-t_1, -t_2)$	$A\mathbf{@}(-t_1, -t_2)$
A Will be true in (t_1, t_2)	$A\mathbf{@}(t_1, t_2)$
A Could be true in (t_1, t_2)	$A\mathbf{?}(t_1, t_2)$
A Since B during $(-t_1, -t_2)$	$B\mathbf{?}(-t_1, -t_2) \wedge \mathbf{since}(B, A)\mathbf{@}[-t_2, -t_2] \wedge A\mathbf{@}[-t_2, 0)$
A Until B during (t_1, t_2)	$B\mathbf{?}(t_1, t_2) \wedge \mathbf{until}(B, A)\mathbf{@}[t_1, t_1] \wedge A\mathbf{@}(0, t_1]$

For TILCO, the condition depicted in Figure 2.6 can be specified by using:

$$E \rightarrow \ endA\mathbf{?}(0, t_e] \wedge \neg\mathbf{until}(endA, \neg startA)$$

In [97] and [99] a sound deductive system for TILCO has been presented. This system is used in the context of the general theorem prover Isabelle [122] to provide an assisted support for proving TILCO formulas. Using this formalization, a set of fundamental theorems has been proven and a set of tactics has been built for supporting the semi-automatic demonstration of properties of TILCO specifications. Causal TILCO specifications are also executable by using a modified version of the Tableaux algorithm. Since TILCO has aspects typical of both descriptive and operational semantics, it can be considered a dual approach

following the classification reported in [35]. Since TILCO extends FOL, it is undecidable in the general case. However, the subset of formulas that presents only quantifications on finite sets is decidable. Causal TILCO specifications can be executed with a modified version of a tableaux algorithm.

2.4 Discussion

In Table 2.7, the main characteristics of the temporal logics reviewed in the previous sections have been collected. The following discussion considers two main aspects of the logics: the intrinsic power of expressiveness in terms of logic order and quantification over time variable; and the readability/ understandability of the logics.

Table 2.7: Comparative table regarding the features of the temporal logics examined.

Logic	Logic order[1]	Fundamental time entity[2]	Temporal structure[3]	Metric for time/-Quantitative temporal constraints[4]	Logic decidability[4]	Deductive system[4]	Logic executability[4]	Ordering events[4]	Implicit, Explicit[5]
PTL	P	P	L	N	Y	Y	Y	Y	I
Choppy	P	P	L	N	Y	(Y)	(Y)	Y	I
BTTL	P	P	B	N	Y	Y	Y	Y	I
ITL	P	I	L	N	(Y)	(Y)	(Y)	Y	I
PMLTI	P	I	L/B	N	(Y)	NA	NA	Y	I
CTL	P	P	B	N	Y	NA	NA	Y	I
IL	P	I	L	N	Y	NA	NA	Y	I
EIL	P	I	L	Y	Y	NA	NA	Y	I
RTIL	P	I	L	Y	Y	NA	NA	Y	(I)
LTI	2^{nd}	I	L	N	Y	Y	NA	Y	(I)
RTTL	1^{st}	P	L	(Y)	N	Y	NA	Y	E
TPTL	P	P	L	Y	Y	Y	NA	Y	(E)
RTL	1^{st}	I	L	Y	N	NA	NA	Y	E
TRIO	1^{st}	P	L	Y	N	Y	(Y)	Y	I
MTL	1^{st}	P	L	Y	(N)	(Y)	NA	Y	I
TILCO	1^{st}	I	L	Y	(Y)	Y	(Y)	Y	I

[1] P= propositional, 1^{st} = first order, 2^{nd} = second order;
[2] P= point, I= interval;
[3] L= linear, B= branching;
[4] N= no, (N)=no in the general case, Y= yes, (Y)=yes in some specific case, NA= not available;
[5] I= implicit, E= esplicit.

The temporal logics discussed can be divided into two main categories: temporal logics without a metric for time and those with a metric for time. PTL, Choppy Logic, BTTL, ITL, PMLTI, IL, CTL and LTI belong to the first category. These logics are less satisfactory for the specification of real-time systems since quantitative temporal constraints cannot be

specified. In the second category, lie the following temporal logics: EIL, RTIL, RTTL, TPTL, RTL, TRIO, MTL, and TILCO. Some of these logics are characterized by the fact that they permit explicit quantification on the variable *time*, whereas for the others it is not permitted. In [8], it has been observed that not permitting explicit quantification on time brings about a more natural specification style. Moreover, in [9] the impossibility of explicit quantification on time was demonstrated to be a necessary condition for the existence of a practically usable verification method, such as the techniques based on tableaux. In fact, a logic that allows quantification over time has the expressive power of FOL and is undecidable. For this reason, in many cases, logics as EIL, RTIL, TPTL, and TILCO, are typically preferable to RTTL, RTL, TRIO, and MTL that permit quantifications over time. When a temporal logic allows the possibility of quantification on non-temporal variables (even with some limitations) it can be considered a first order temporal logic. This is a great advantage since it leads to a more expressive specification language and has a greater power of abstraction. Among the logics examined, only RTIL, RTL, TRIO, MTL, and TILCO permit quantification on non-time dependent variables. More specifically, only RTIL and TILCO seem to present the most complete collection of interesting characteristics for real-time systems specification (metric for time, expression of quantitative and events order temporal constraints, no quantification over time). Both of these logics do not permit quantification on time but permit the quantification on non-time dependent variables with finite domains. RTIL permits one only to reference the absolute time, and then only indirectly in a relative manner. Moreover, the order of events is not complete, since events having a relationship of *successor* or *predecessor* can occur for the same value of the system clock. TILCO does not have these problems and has a sound deductive system that supports the assisted proof of theorems and execution of formulas.

From the point of view of readability and understandability of the temporal logic it is highly relevant to evaluate two aspects: the number of elementary operators, and the structure of the syntax. The first of these aspects is quite objective, since a lower number of temporal operators is typically preferred. Temporal logics that have a high number of operators are, like programming languages, typically hard to learn and hard to understand. Their expressiveness can be high, since a wide collection of operators or temporal predicates can be very useful for specifying complex systems, but ease of learning and readability are low. It has been previously shown that all the most useful specifications can be expressed by using a very low number of temporal operators. If these operators support a metric for time, their expressiveness is even higher. As a case limit, all the operators can be defined in terms of a measuring operator or modeled with delay. On the other hand, having too low a number

of temporal operators can produce the same effects, since complex specifications have to be built by using elementary operators even for very simple specifications. This means that a balance between the power of the temporal logic and its number of temporal operators is needed. The number of operators also influences the syntax of the temporal logic. In some cases, the verbosity of temporal logic depends on the presence of a neat distinction between past and future – e.g., extended PTL, TRIO, MTL. This distinction typically leads to duplication of the number of operators in order to have specific operators for past and future. When this distinction is not made, time can considered only in the future – e.g., RTTL, TPTL – or more general and flexible operators capable of working continuously from past to future are defined – e.g., TILCO. In evaluating temporal logics, other interesting features can be the availability of a graphical representation for the visual specification. The visual representation of temporal specifications has frequently been addressed by researchers who have neglected the capabilities of temporal logics. Visual representation may make the readability of the specifications easier, but their real expressiveness is given by the above-mentioned features of the temporal logics. An interesting integrated approach can be seen in [50], [107].

Chapter 3

TILCO

In this chapter temporal logic TILCO is presented in more detail. The syntax and semantics of the language is provided. Some examples are provided to highlight how to specify typical requirements. The deductive system of TILCO is shortly presented.

3.1 Definition of TILCO

TILCO extends FOL to create a logic language that can specify both relationships between events and time, and data domain transformations. TILCO can be used to specify temporal constraints among events in either a qualitative or quantitative manner. Therefore, interval boundaries, which specify the length of intervals and actions, can be expressed relative to other events (qualitatively) or with an absolute measure (quantitatively). This allows definition of expressions of ordering relationships among events or delays and time-outs. These features are mandatory for specifying the behavior of real-time systems. TILCO deductive approach is sound, and thus consistent. It forces the user to write formulæ without using direct quantifications over the temporal domain, thus preventing them from writing specifications that are overly intricate or difficult to understand [9].

TILCO includes the concepts of typed variables and constants; it provides a set of basic types and lets users define new types using the mechanisms of enumerated collection and type constructors. A type-checking mechanism is automatically extended to these new types. The predefined types are: **nat** for natural numbers, **int** for integer numbers, **bool** for Booleans, **char** for text characters, and **string** for character strings. The usual arithmetic operators: $+$, $-$, $*$, $/$, mod , $\sim$ (change sign), are defined for integers and natural numbers. String manipulation functions are defined for strings. Comparative operators: $=$, $<$, $>$, $\geq$, $\leq$, $\neq$, can be used with integers, naturals, characters and strings.

A system specification in TILCO is a tuple

$$\{\mathcal{U}, \mathcal{T}, \mathcal{F}, \mathcal{P}, \mathcal{V}, \mathcal{W}, \mathcal{C}, \mathcal{J}\},$$

where $\mathcal{U}$ is a set of TILCO formulæ, $\mathcal{T}$ a set of type definitions, $\mathcal{F}$ a set of functions, $\mathcal{P}$ a set of predicates, $\mathcal{V}$ a set of typed time-dependent variables, $\mathcal{W}$ a set of typed time-independent variables, $\mathcal{C}$ a set of typed constants (also called time invariant parameters), and $\mathcal{J}$ is a set of integer intervals. $\mathcal{U}$ specifies the rules defining the specified system's behavior. $\mathcal{T}$ defines the specification types. Functions and predicates have their usual meaning and are used to manipulate predefined and user-defined data-types. Time-dependent variables model the specified system's inputs (read-only), outputs (write-only), and auxiliary variables (read/write). Time-dependent variables can assume any value in their corresponding domain. Time-independent variables are used to build parametric formulæ that operate on structured data types (i.e., arrays, lists, etc.) through quantification. Constants are used for modeling system parameters. Integer intervals – which are connected sets of integers – are used for specifying quantitative temporal relationships.

A system is specified in TILCO according to the following rules:

- a system is characterized by its input and output ports, which communicate with the external environment, and by its auxiliary variables representing a part of its internal state;
- inputs, outputs and auxiliary variables can assume only one value at each time instant. Each of them is defined by a unique name;
- an input is a typed variable whose value can change due to external events;
- an output is a typed variable that can be forced to assume a value by some predicates through an assignment, which leads to a change in the external environment;
- an auxiliary variable can be forced to a value by an assignment and it can be read as an input variable; and
- a system is described as a set of formulæ that define its behavior and the data transformation.

3.1.1 Syntax and semantics of TILCO

TILCO extends *temporal operators* to FOL by leaving their evaluation time implicit. Therefore, the meaning of a TILCO formula is given with respect to the current time such as in

other logic languages such as [63], [55]. In TILCO, the time is discrete and linear, and the temporal domain is $\mathbb{Z}$, the set of integers; the minimum time interval corresponds to 1 time unit. The current time instant is represented by 0, whereas positive (negative) number represent future (past) time instants. TILCO formulæ can be time dependent or independent; the latter are those that do not present any TILCO *temporal operator*, and are comprised only of time-independent subformulæ. A time independent formula can be regarded as a constraint that must be satisfied in each time instant.

The basic temporal entity in TILCO is the interval. Intervals can be quantitatively expressed by using the notation with round brackets for excluding interval boundaries, "(", ")", or squared brackets for including them, "[", "]". Time instants are regarded as special cases that are represented as closed intervals composed of a single point (e.g., $[a, a]$). Symbols $+\infty$ and $-\infty$ can be used as interval boundaries – if the extreme is open – to denote infinite intervals. For example, $[a, +\infty)$ represents set $\{x \in \mathbb{Z} | a \leq x\}$. In this way, TILCO lets users specify both *facts* in intervals and *events* in time instants. Classical operators of temporal logic (eventually, $\Diamond$, and henceforth, $\Box$) can be easily obtained by using TILCO operators with infinite intervals. For these reasons, TILCO can be regarded as a generalization of most of the interval logics presented in the literature in the past – such as [134], [80], [124] – but with the addition of a metric to measure time.

The basic TILCO *temporal operators* are:

- "**@**", bounded universal temporal quantification over an interval;
- "**?**", bounded existential temporal quantification over an interval;
- **until**, to express that either a predicate will always be true in the future, or it will be true until another predicate will become true;
- **since**, to express that either a predicate has always been true in the past, or it has been true since another predicate has become true.

Operators "**@**" and "**?**" are called temporal quantifiers. $A@i$ is true if formula A is true in every instant in the interval i, with respect to the current time instant. Therefore, if t is the current time instant, $(A@i)^{(t)} \equiv \forall x \in i.A^{(x+t)}$ holds. In particular, $A@[t_1, t_2)$ evaluated in t means:

$$\forall x \in [t_1, t_2).A^{(x+t)}.$$

Obviously t_1 and t_2 can be either positive or negative and thus the interval can be in the past or in the future. If the interval's lower bound is greater than the upper bound, then

it is null (that is, it is equal to the empty set). Operators "**@**" and "**?**" correspond, in the temporal domain, to FOL quantifiers $\forall$ and $\exists$, respectively; hence, they are related by a duality relationship analogous to that between $\forall$ and $\exists$. "**@**" and "**?**" are used to express delays, time-outs and any other temporal constraint that requires a specific quantitative bound. Concerning the other *temporal operators*, **until** A B (evaluated in t) is true if B will always be true in the future with respect to t, or if B will be true in the interval $(t, x+t)$ with $x > 0$ and A will be true in $x+t$. This definition of **until** does not require the occurrence of A in the future, so the **until** operator corresponds to the *weak until* operator defined in PTL [22]. The operators **until** and **since** express the same concept for future and past, respectively; they are related by a relationship of *temporal duality*. **until** and **since** can be effectively used to express ordering relationships among events without the need of specifying any numeric constraint.

Given $\mathcal{F}$, $\mathcal{P}$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{C}$, $\mathcal{J}$, the syntax of TILCO formulæ is defined by the following BNF-like definitions:

$$
\begin{array}{rcl}
\text{interval} & ::= & (a,b)|(a,b]|[a,b)|[a,b] \quad \textit{for each } a,b \in \mathbb{Z} \\
\text{interval\_list} & ::= & \text{interval} \\
 & | & \text{interval interval\_op interval} \\
\text{interval\_op} & ::= & ,|; \\
\text{variable} & ::= & w \quad \textit{for each } w \in \mathcal{W} \\
\text{term} & ::= & v \quad \textit{for each } v \in \mathcal{V} \\
 & | & \text{variable} \\
 & | & c \quad \textit{for each } c \in \mathcal{C} \\
 & | & f(\text{term\_list}) \quad \textit{for each } f \in \mathcal{F} \\
\text{term\_list} & ::= & \text{term} \\
 & | & \text{term, term\_list} \\
\text{atomic\_formula} & ::= & p(\text{term\_list}) \quad \textit{for each } p \in \mathcal{P} \\
\text{formula} & ::= & \top|\bot|\text{atomic\_formula} \\
 & | & \neg\text{formula} \\
 & | & \text{formula op formula} \\
 & | & v := \text{term} \quad \textit{for each } v \in \mathcal{V} \\
 & | & \text{quantifier variable. formula} \\
 & | & \text{formula temporal\_quantifier interval\_list}
\end{array}
$$

	|	temporal_op formula formula
	|	(formula)
op	::=	$\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow \mid \Rrightarrow \mid \Lleftarrow$
quantifier	::=	$\forall \mid \exists \mid \exists!$
temporal_quantifier	::=	**@**|**?**
temporal_op	::=	**until**|**since**

The use of parentheses in TILCO expressions is reduced by using the operators' precedence relationships reported in Tab. 3.1.

prec.	operators
1	$\sim$
2	$*$ / mod
3	$+$ $-$
4	$=$ $>$ $<$ $\geq$ $\leq$ $\neq$
5	$\neg$, ;
6	$:=$
7	**@** **?**
8	$\wedge$
9	$\vee$
10	$\Leftrightarrow$ $\Rightarrow$ $\Rrightarrow$ $\Lleftarrow$
11	$\forall$ $\exists$ $\exists!$
12	**until** **since**

Table 3.1: Precedences among TILCO operators.

Before defining the semantics of TILCO, it is important to introduce the concept of *interpretation* of a TILCO formula. This concept is also used to define the validity and the satisfiability of TILCO formulæ.

Given a syntactically correct TILCO formula A, with $\{t_1, \ldots, t_h\}$ set of types used in A, $\{p_1, \ldots, p_k\}$ predicates, $\{f_1, \ldots, f_l\}$ functions, $\{v_1, \ldots, v_m\}$ time-dependent variables, $\{c_1, \ldots, c_q\}$ constants, and $\{j_1, \ldots, j_r\}$ intervals present in A, then an *interpretation* $\mathcal{I}$ is a tuple

$$(\{D_1, \ldots, D_h\}, \{R_1, \ldots, R_k\}, \{F_1, \ldots, F_l\}, \{V_1(t), \ldots, V_m(t)\}, \{C_1, \ldots, C_q\}, \{J_1, \ldots, J_r\})$$

where:

- $\{D_1, \ldots, D_h\}$ assigns a domain D_i to each type t_i;

- $\{R_1, \ldots, R_k\}$ assigns an n-ary relation R_i over $D_{i_1} \times \ldots \times D_{i_n}$ to each n-ary predicate p_i with arguments of type $t_{i_1}, \ldots, t_{i_n}$;
- $\{F_1, \ldots, F_l\}$ assigns an n-ary function F_i over $D_{i_1} \times \ldots \times D_{i_n}$ to each n-ary function f_i with arguments of type $t_{i_1}, \ldots, t_{i_n}$;
- $\{V_1(t), \ldots, V_m(t)\}$ assigns a function of time $V_i(t) : \mathbb{Z} \to D_n$ to each time-dependent variable v_i of type t_n, specifying the history of that variable in every time instant (where t is the absolute time);
- $\{C_1, \ldots, C_q\}$ assigns a value $C_i \in D_n$ to each constant c_i of type t_n;
- $\{J_1, \ldots, J_r\}$ assigns an interval value J_i to each integer interval j_i.

Given a TILCO formula A and an interpretation $\mathcal{I}$ for A, notation

$$\mathcal{I}, t \models A$$

expresses that $\mathcal{I}$ is a *model* for A evaluated in the time instant t. The evaluation of $\mathcal{I}, t \models A$, stating the semantics of TILCO, is inductively defined on the structure of A by the following rules:

- $\mathcal{I}, t \models \top$;
- $\mathcal{I}, t \not\models \bot$;
- $\mathcal{I}, t \models \neg A$ iff $\mathcal{I}, t \not\models A$;
- $\mathcal{I}, t \models A_1 \wedge A_2$ iff $\mathcal{I}, t \models A_1$ and $\mathcal{I}, t \models A_2$;
- $\mathcal{I}, t \models A_1 \vee A_2$ iff either $\mathcal{I}, t \models A_1$ or $\mathcal{I}, t \models A_2$;
- $\mathcal{I}, t \models A_1 \Rightarrow A_2$ iff $\mathcal{I}, t \models \neg A_1 \vee A_2$;
- $\mathcal{I}, t \models A_1 \Rrightarrow A_2$ iff either $\mathcal{I}, t \models \neg A_1$ or $\mathcal{I}, t+1 \models A_2$;
- $\mathcal{I}, t \models A_1 \Lleftarrow A_2$ iff either $\mathcal{I}, t \models \neg A_1$ or $\mathcal{I}, t-1 \models A_2$;
- $\mathcal{I}, t \models A_1 \Leftrightarrow A_2$ iff $\mathcal{I}, t \models A_1 \Rightarrow A_2 \wedge A_2 \Rightarrow A_1$;
- $\mathcal{I}, t \models x := exp$ iff there exists a constant $k \in D_x$ such that $\mathcal{I}, t \models x = k$ and $\mathcal{I}, t-1 \models exp = k$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;

- $\mathcal{I}, t \models \forall x.A(x)$ iff, for each $y \in D_x$ it is true that $\mathcal{I}, t \models A(y)$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;
- $\mathcal{I}, t \models \exists x.A(x)$ iff, there exists a $y \in D_x$ such that $\mathcal{I}, t \models A(y)$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;
- $\mathcal{I}, t \models \exists! x.A(x)$ iff, there exists one and only one $y \in D_x$ such that $\mathcal{I}, t \models A(y)$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;
- $\mathcal{I}, t \models A\mathbf{@}i$ iff, for each $s \in i$, $\mathcal{I}, s + t \models A$ is true;
- $\mathcal{I}, t \models A\mathbf{?}i$ iff, there exists an $s \in i$ such that $\mathcal{I}, s + t \models A$;
- $\mathcal{I}, t \models \mathbf{until}\ A_1\ A_2$ if either $\mathcal{I}, t \models A_2\mathbf{@}(0, +\infty)$ or there exists $\tau > 0$ such that $\mathcal{I}, t + \tau \models A_1$ and $\mathcal{I}, t \models A_2\mathbf{@}(0, \tau)$;
- $\mathcal{I}, t \models \mathbf{since}\ A_1\ A_2$ if either $\mathcal{I}, t \models A_2\mathbf{@}(-\infty, 0)$ or there exists $\tau < 0$ such that $\mathcal{I}, t + \tau \models A_1$ and $\mathcal{I}, t \models A_2\mathbf{@}(\tau, 0)$;
- $\mathcal{I}, t \models A\mathbf{@}i, j$ iff $\mathcal{I}, t \models (A\mathbf{@}i) \wedge (A\mathbf{@}j)$;
- $\mathcal{I}, t \models A\mathbf{?}i, j$ iff $\mathcal{I}, t \models (A\mathbf{?}i) \wedge (A\mathbf{?}j)$;
- $\mathcal{I}, t \models A\mathbf{@}i; j$ iff $\mathcal{I}, t \models (A\mathbf{@}i) \vee (A\mathbf{@}j)$;
- $\mathcal{I}, t \models A\mathbf{?}i; j$ iff $\mathcal{I}, t \models (A\mathbf{?}i) \vee (A\mathbf{?}j)$;
- $\mathcal{I}, t \models p_i(e_1, \ldots, e_n)$, iff $(E_1, \ldots, E_n) \in R_i$, where R_i is the relation assigned by $\mathcal{I}$ to p_i and E_j, for each $j = 1, \ldots, n$, are the results of the expressions e_j when the values assigned by $\mathcal{I}$ are substituted for the constants and variables, and the variables are evaluated in t.

The semantics of predicates also includes that of functions, variables and constants.

Remark 3.1.1 In the case where the interval is null, it holds:

$$\begin{aligned} A\mathbf{@}\emptyset &= \top; \\ A\mathbf{?}\emptyset &= \bot. \end{aligned}$$

□

Some useful definitions follow.

Definition 3.1.1 *Given an interpretation*

$$\mathcal{I} = \left\{ \begin{array}{l} \{D_1, \ldots, D_h\} \\ \{R_1, \ldots, R_k\} \\ \{F_1, \ldots, F_l\} \\ \{V_1(t), \ldots, V_m(t)\} \\ \{C_1, \ldots, C_q\} \\ \{J_1, \ldots, J_r\} \end{array} \right\},$$

its temporal translation by $s \in \mathbb{Z}$ time units is defined by:

$$\tau(\mathcal{I}, s) \stackrel{def}{=} \left\{ \begin{array}{l} \{D_1, \ldots, D_h\} \\ \{R_1, \ldots, R_k\} \\ \{F_1, \ldots, F_l\} \\ \{V_1(t+s), \ldots, V_m(t+s)\} \\ \{C_1, \ldots, C_r\} \\ \{J_1, \ldots, J_r\} \end{array} \right\}.$$

Definition 3.1.2 *A TILCO formula A is said to be* satisfiable *if there exists an interpretation $\mathcal{I}$ and a value $t \in \mathbb{Z}$ such that $\mathcal{I}, t \models A$.*

Definition 3.1.3 *A TILCO formula A is said to be* valid in an interpretation $\mathcal{I}$ *if for each $t \in \mathbb{Z}$ it is true that $\mathcal{I}, t \models A$. The notation used is $\mathcal{I} \models A$.*

Definition 3.1.4 *A TILCO formula A is said to be* valid *if for each interpretation $\mathcal{I}$ and for each $t \in \mathbb{Z}$ it is true that $\mathcal{I}, t \models A$. The notation used is $\models A$.*

Definition 3.1.5 *Given a set of TILCO formulæ, $U = \{A_1, \ldots, A_n\}$, U is said to be* satisfiable *if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models A_1, \ldots \mathcal{I} \models A_n$. $\mathcal{I}$ is said to be a* model *for U. U is said to be* unsatisfiable *if for each $\mathcal{I}$ there exists an i such that $\mathcal{I} \not\models A_i$.*

Definition 3.1.6 *Given a set of TILCO formulæ U and a TILCO formula A, if every model $\mathcal{I}$ for U is such that $\mathcal{I} \models A$, then A is said to be a* logic consequence *of U. The notation used is $U \models A$.*

Definition 3.1.7 *Given $\mathcal{S}(U) = \{A | U \models A\}$, $\mathcal{S}(U)$ is called* theory *of U and the elements of $\mathcal{S}(U)$ are called* theorems *of U. The elements of U are called* axioms *of $\mathcal{S}(U)$.*

3.1.2 Comments

- In a TILCO specification, a system is described by a formula consisting of the conjunction of all the formulæ of $\mathcal{U}$, each describing a different aspect of the system. A specification is defined in a *specification temporal domain* by means of operator "**@**". For example, if $\mathcal{U} = \{F_1,\ F_2,\ F_3\}$ and the temporal domain is i, then the system is described by:

$$(F_1 \wedge F_2 \wedge F_3)\mathbf{@}i,$$

  which means that all properties F_1, F_2, F_3 must be valid in each time instant of i.

- Each TILCO formula used in a system specification must be closed, in the sense that each time-independent variable in a formula must be quantified. For instance, formula $\exists s.\ f(k,s) \Rightarrow P$ is open, while $\exists s.\ \exists k.\ f(k,s) \Rightarrow P$ is closed. If a TILCO formula is open, it is replaced by its universal closure (that is, an external universal quantifier is introduced for each of the time independent variables which are not quantified). According to the syntax definition, each quantified variable must be time independent, otherwise (i) it would be possible to write higher order formulæ and (ii) time could not be left implicit because the formula's meaning would change during system evolution.

- In a TILCO specification, predicates and functions with typed parameters can also be defined. Predicates are functions that return a value of type **bool**. Predicates and other functions define operations and relationships over predefined and user-defined types. Predicates and other functions are incrementally defined using predefined functions and predicates over the basic data types and type constructors. The body of each predicate must be specified by means of a TILCO formula, in which the only non-quantified variables are the predicate parameters. Predicates are only instruments used to simplify the writing; hence, more complex temporal expressions and formulæ can be hidden in predicates. Predicates are functions also extend the number of temporal operators of TILCO, since they can be used to constitute a user-defined library, thus improving the specifications reusability. For example, a predicate for specifying that A occurs only once in an interval i could be defined as:

$$\text{OnlyOnce}(A : \text{int} \rightarrow \text{bool}, i : \text{interval}) : \text{bool} \stackrel{\text{def}}{=} \exists! m.A(m)\mathbf{?}i,$$

  where each occurrence of A is characterized by a different value of m:

$$\models \forall m.A(m) \Rightarrow \neg(A(m)\mathbf{?}(-\infty, 0)),$$

so that $\exists!m.A(m)\mathbf{?}i$ specifies that the event A happens only once during the interval i. m can be regarded as a *time-stamp*. The adoption of *time-stamps* for distinguishing different occurrences of events has been introduced in [83], to overcome the temporal logics' limitation in recognizing different occurrences of an event. Since TILCO is an extension of FOL, the use of *time-stamps* in specifications is simply obtained by adding them to predicates whose different occurrences must be distinguished.

- The two predicates

$$\begin{aligned} \text{rule}(A : \text{bool}) &\stackrel{\text{def}}{=} A\mathbf{@}(-\infty, +\infty), \\ \text{fact}(A : \text{bool}) &\stackrel{\text{def}}{=} A\mathbf{?}(-\infty, +\infty), \end{aligned}$$

  express that a predicate A is always or sometimes true, respectively. These predicates are often used in specifications to express the concepts of necessity and possibility over the whole temporal domain.

- The classical *henceforth* operator, $\Box$, can be expressed in terms of TILCO operator "**@**": $A\mathbf{@}[0, +\infty)$, which means that A will be true forever from the current time instant. Analogously, the *eventually* operator, $\Diamond$, can be expressed by $A\mathbf{?}[0, +\infty)$.

- Operator "**?**" could also be defined in terms of operator "**@**" by using the duality relationship:

$$A\mathbf{?}i \equiv \neg(\neg A\mathbf{@}i).$$

- In order to simplify writing specifications the symbol $=\!\!\triangleright\!\!\triangleright$ ($=\!\!\triangleleft\!\!\triangleleft$) has been introduced to express that a formula implies that another formula will be (has been) true at the next (previous) time instant:

$$\begin{aligned} A =\!\!\triangleright\!\!\triangleright B &\equiv A \Rightarrow B\mathbf{@}[1, 1], \\ A =\!\!\triangleleft\!\!\triangleleft B &\equiv A \Rightarrow B\mathbf{@}[-1, -1]. \end{aligned}$$

- TILCO is also characterized by its compositional operators that work with intervals: *comma* ",", which corresponds to $\wedge$, and *semicolon* ";", which corresponds to $\vee$, between intervals. Compositional operators "," and ";" assume different meanings if they are associated with operators "**@**" or "**?**". Other operators among intervals, such as intersection, "$\cap$", and union, "$\cup$", could be defined by considering time intervals as sets. However, the introduction of $\cup$ is problematic because the set of intervals is not closed over this operation.

- **until** $A\ B$ operator does not consider the evaluation time instant as an instant where A could happen, then operator $\mathbf{until_0}$ has been introduced. It is defined as:

$$\mathbf{until_0}\ A\ B \equiv A \vee (B \wedge \mathbf{until}\ A\ B)$$

and also a "strong" **until** is sometime needed, for this reason the operator $\mathbf{until}'$ has been defined as:

$$\mathbf{until}'\ A\ B \equiv A?(0, +\infty) \wedge \mathbf{until}\ A\ B$$

for completeness also the $\mathbf{until}'_\mathbf{0}$ has been defined as:

$$\mathbf{until}'_\mathbf{0}\ A\ B \equiv A?[0, +\infty) \wedge \mathbf{until_0}\ A\ B$$

in a similar manner $\mathbf{since_0}$, $\mathbf{since}'$ and $\mathbf{since}'_\mathbf{0}$ operators have been defined.

3.1.3 Short examples

Tab. 3.2 provides examples of TILCO formulæ. To provide a clearer view of TILCO's expressiveness the formulæ are accompanied by an explanation of their meaning. In Tab. 3.2, t stands for a positive integer number.

A more complex example is a formula that specifies a system with an input I_1 : **int** and an output O_1 : **bool**. The system produces an output signal for t_1 time instants with a delay of t_2 time instants every time that the input assumes the value val:

$$I_1 = val \Rightarrow O_1\mathbf{@}[t_2, t_1 + t_2].$$

The same system is also specified by the formula:

$$I_1 = val \Rightarrow O_1\mathbf{@}[0, t_1]\mathbf{@}[t_2, t_2].$$

Another example is the specification of a system for generating periodic events:

$$(\neg B\mathbf{@}(0, 10] \Leftrightarrow (B\mathbf{@}(10, 20] \wedge \neg B\mathbf{@}(20, 30])) \wedge ((\neg B\mathbf{?}[-1, -1] \wedge B) \Leftrightarrow (A\mathbf{@}[0, 2] \wedge \neg A\mathbf{@}(2, 20))).$$

This TILCO formula specifies that signal B is periodic with a duty-cycle of 50 percent and a period of 20 time units while, being associated with each transition of B (from false to true) signal A stays true for 2 time units. Fig. 3.1 depicts the histories of signals A and B. Once system behavior is specified by means of a set of TILCO formulæ, the specification can be validated to verify its correspondence to system requirements. In TILCO, system validation is performed by proving that high-level properties (such as, safety, liveness, and

$A\mathbf{@}[1,1]$	A will be true at the next time instant
$A\mathbf{@}[0,t)$	A is true from now for t time instants
$A\mathbf{@}(-\infty,+\infty)$	A has been, is and will be always true
$A\mathbf{@}(0,+\infty)$	A will be always true in the future
$A\mathbf{?}(0,+\infty)$	A will be sometimes true in the future
$A\mathbf{@}[t_1,t_2]$	A is true in $[t_1,t_2]$
$A\mathbf{?}[t_1,t_2)$	A is true in an instant of $[t_1,t_2)$
$\neg(A\mathbf{@}(-\infty,+\infty))$	A is not always true
$\neg A\mathbf{@}(-\infty,+\infty)$	A is always false
$A\mathbf{@}[t_1,t_1],(t_2,t_3]$	A is true at t_1, and in $(t_2,t_3]$
$A\mathbf{?}[t_1,t_1],(t_2,t_3]$	A is true at t_1, and is true at least once in $(t_2,t_3]$
$A\mathbf{@}[t_1,t_1];(t_2,t_3]$	A is true at t_1, or in $(t_2,t_3]$
$A\mathbf{@}[t,t]\wedge\neg A\mathbf{@}(0,t)$	t is the next time instant in which A will be true
$A\mathbf{@}[-t,-t]\wedge\neg A\mathbf{@}(-t,0)$	$-t$ is the last time instant in which A has been true
$A\mathbf{?}[0,t_1]\mathbf{@}[0,+\infty)$	A will become true within t_1 for each time instant in the future (response)
$A\mathbf{@}[0,t_1]\mathbf{?}[0,+\infty)$	A will be true, and since then it will remain true for t_1 time units (persistence)
$(A\Rightarrow B)\mathbf{@}[0,+\infty)$	A causes B always in the future
$(A\Rightarrow B)\mathbf{?}[0,t]$	if A is true within t, then also B will be true at the same time
$(A\Rightarrow B\mathbf{?}i)\mathbf{@}j$	A leads to an assertion of B in i for each time instant of j
$(A\Rightarrow B\mathbf{@}i)\mathbf{@}j$	A leads to the assertion of B in the whole interval i for each time instant of j
$(A\Rightarrow B\mathbf{@}i)\mathbf{?}j$	A leads to the assertion of B in the whole interval i in at least a time instant of j

Table 3.2: Examples of TILCO formulæ.

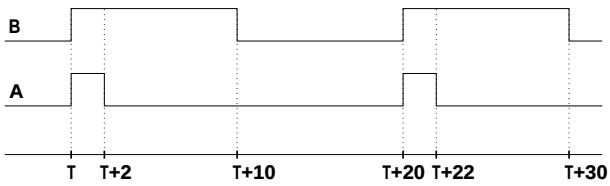

Figure 3.1: Histories of signals A and B.

so on) are satisfied by the TILCO system specification. These properties can be expressed by other TILCO formulæ, and thus TILCO specifies both the system and its high-level properties. As a result, the classical safety conditions, such as $A@i$ (where A is a positive property), and $\neg B@i$ where B is a negative condition) must be satisfied by the system specification, where the interval i can be extended to all or part of the *specification temporal domain.*

Moreover, users can also specify liveness conditions, such as $A?i$ (A will be satisfied within i) or deadlock-free conditions, such as $(\neg A?i)@j$. During the specification validation, if a desired property (constituting a system requirement) cannot be deduced from the system specification given in terms of TILCO formulæ, then the specification is incomplete. If the system must satisfy that property, a new TILCO formula should be added to the system specification, provided that this formula does not contradict any existing specification formula. This formula may itself be the desired property or a formula that completes the system specification to prove the desired property, thus allowing the incremental system specification.

3.1.4 A more complex example

In this subsection, a more significant real-time system example is presented: an allocator that serves a set of client processes by sharing a resource according to several temporal constraints – [16], [57]. In every time instant and for every process a, the resource is assigned to process a ($\mathrm{gr}(a)$) if and only if, since the last time the resource was granted ($\mathrm{gr}(b)$) the resource has been released (fr) and

- a requested the resource ($\mathrm{rq}(a, \delta)$) and that request has not already expired;
- since the request was issued, the resource has not already been assigned to a;
- there are no a' and δ' such that:
  - $a \neq a'$
  - a' requested the resource ($\mathrm{rq}(a', \delta')$) and that request has not already expired;
  - since when the request was issued, the resource has not already been assigned to a';
  - a' requested the resource before a (i.e.: a' did not request the resource after a).

Equation (3.1) shows the TILCO system specification.

$$\left(\begin{array}{l} \forall a.\mathrm{gr}(a) \leftrightarrow \left(\begin{array}{ll} \textbf{since}(\neg\exists b.\mathrm{gr}(b), \mathrm{fr}) & \wedge \\ \exists\delta. \left(\begin{array}{ll} \mathrm{rq}(a,\delta)\textbf{?}[-\delta,-5] & \wedge \\ \textbf{since}(\mathrm{rq}(a,\delta), \neg\mathrm{gr}(a)) & \wedge \end{array}\right) & \\ \neg\exists a'\delta'. \left(\begin{array}{ll} a' \neq a & \wedge \\ \mathrm{rq}(a',\delta')\textbf{?}[-\delta',-5] & \wedge \\ \textbf{since}(\mathrm{rq}(a',\delta'), \neg\mathrm{gr}(a')) & \wedge \\ \textbf{since}(\mathrm{rq}(a,\delta), \neg\mathrm{rq}(a',\delta')) & \end{array}\right) & \end{array}\right) \end{array}\right) \textbf{@}(-\infty,\infty) \quad (3.1)$$

3.2 Deductive System

FOL's deductive system in natural deduction style has been enhanced with added rules for introduction and eliminating TILCO *temporal operators*. In stating rules

- $\vdash A$, means that A is provable in every time instant;
- $\vdash_t A$, means that A is provable in the time instant t.

$\top I \quad \dfrac{}{\vdash \top}$ $\qquad$ $\bot E \quad \dfrac{\vdash \bot}{\vdash P}$

$\wedge I \quad \dfrac{\vdash P \quad \vdash Q}{\vdash P \wedge Q}$ $\qquad$ $\wedge E1 \quad \dfrac{\vdash P \wedge Q}{\vdash P}$ $\qquad$ $\wedge E2 \quad \dfrac{\vdash P \wedge Q}{\vdash Q}$

$\vee E \quad \dfrac{\vdash P \vee Q \quad P \vdash R \quad Q \vdash R}{\vdash R}$ $\qquad$ $\vee I1 \quad \dfrac{\vdash P}{\vdash P \vee Q}$ $\qquad$ $\vee I2 \quad \dfrac{\vdash Q}{\vdash P \vee Q}$

$\Rightarrow I \quad \dfrac{P \vdash Q}{\vdash P \Rightarrow Q}$ $\qquad$ $\Rightarrow E(\mathrm{MP}) \quad \dfrac{\vdash P \Rightarrow Q \quad \vdash P}{\vdash Q}$

$\forall I \quad \dfrac{\vdash P(x)}{\vdash \forall x.P(x)}$ x free only in P $\qquad$ $\forall E \quad \dfrac{\vdash \forall x.P(x)}{\vdash P[t/x]}$

$\exists I \quad \dfrac{\vdash P[t/x]}{\vdash \exists x.P(x)}$ $\qquad$ $\exists E \quad \dfrac{\vdash \exists x.P(x) \quad P(x) \vdash Q}{\vdash Q}$ x free only in P

Rule $\Rightarrow I$ is also called *deduction rule* (DR), rule $\forall I$ is also called *generalization rule* (GEN), and $\exists E$ is called *existential instantiation rule* (EI). An alternative and often simpler to use way of stating rule EI is:

$$\frac{\vdash \exists x.P(x)}{\vdash P(a)} \quad \text{where } a \text{ is a new constant}$$

The following introduction and elimination rules have been specifically proven for TILCO operators:

$:=I$ $\dfrac{\vdash_t v = k \quad \vdash_{t-1} e = k}{\vdash_t v := e}$ k constant $\qquad$ $:=E$ $\dfrac{\vdash_t v := e \quad \vdash_{t-1} e = k}{\vdash_t v = k}$ k constant

$\mathbf{@}I$ $\dfrac{x \in i \vdash_{x+t} P}{\vdash_t P\mathbf{@}i}$ x not free in any assump. $\qquad$ $\mathbf{@}E$ $\dfrac{\vdash_t P\mathbf{@}i \quad \vdash x \in i}{\vdash_{x+t} P}$

$\mathbf{?}I$ $\dfrac{\vdash_{x+t} P \quad \vdash x \in i}{\vdash_t P\mathbf{?}i}$ $\qquad$ $\mathbf{?}E$ $\dfrac{\vdash_t P\mathbf{?}i \quad \dfrac{\vdash_{x+t} P \quad \vdash x \in i}{\vdash R}}{\vdash R}$ x not free in any assump.

$\text{until}I1$ $\dfrac{\vdash_{t+x} P \quad \vdash_t Q\mathbf{@}(0,x) \quad \vdash 0 < x}{\vdash_t \text{until } P\ Q}$ $\qquad$ $\text{until}I2$ $\dfrac{\vdash_t Q\mathbf{@}(0,+\infty)}{\vdash_t \text{until } P\ Q}$

$\text{since}I1$ $\dfrac{\vdash_{t+x} P \quad \vdash_t Q\mathbf{@}(x,0) \quad \vdash x < 0}{\vdash_t \text{since } P\ Q}$ $\qquad$ $\text{since}I2$ $\dfrac{\vdash_t Q\mathbf{@}(-\infty,0)}{\vdash_t \text{since } P\ Q}$

$\text{until}E$ $\dfrac{\vdash_t \text{until } P\ Q \quad \dfrac{\vdash_{t+x} P \quad \vdash_t Q\mathbf{@}(0,x) \quad \vdash 0<x}{\vdash R} \quad \dfrac{\vdash_t Q\mathbf{@}(0,+\infty)}{\vdash R}}{\vdash R}$

$\text{since}E$ $\dfrac{\vdash_t \text{since } P\ Q \quad \dfrac{\vdash_{t+x} P \quad \vdash_t Q\mathbf{@}(x,0) \quad \vdash x<0}{\vdash R} \quad \dfrac{\vdash_t Q\mathbf{@}(-\infty,0)}{\vdash R}}{\vdash R}$

For certain operators introduction and elimination rules have not been proven. Therefore, a set of equivalencies are used to arrive at formulæ that contain only operators for which introduction and elimination rules are available:

$(\neg P)$	$\equiv$	$P \Rightarrow \bot$	$\neg$-equiv
$(P \Leftrightarrow Q)$	$\equiv$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$	$\Rightarrow$-equiv
$(\exists! x.P(x))$	$\equiv$	$(\exists x.P(x) \wedge (\forall y.P(y) \Rightarrow y = x))$	$\exists!$-equiv
$A \Rrightarrow B$	$\equiv$	$A \Rightarrow (B\mathbf{@}[1,1])$	$\Rrightarrow$-equiv
$A \Lleftarrow B$	$\equiv$	$A \Rightarrow (B\mathbf{@}[-1,-1])$	$\Lleftarrow$-equiv
$A\mathbf{@}i, j$	$\equiv$	$(A\mathbf{@}i) \wedge (A\mathbf{@}j)$	$\mathbf{@}$,-equiv
$A\mathbf{?}i, j$	$\equiv$	$(A\mathbf{?}i) \wedge (A\mathbf{?}j)$	$\mathbf{?}$,-equiv
$A\mathbf{@}i; j$	$\equiv$	$(A\mathbf{@}i) \vee (A\mathbf{@}j)$	$\mathbf{@}$;-equiv
$A\mathbf{?}i; j$	$\equiv$	$(A\mathbf{?}i) \vee (A\mathbf{?}j)$	$\mathbf{?}$;-equiv

Since each TILCO formula is defined with respect to the implicit time, a formula specifies a behavior that holds in different contexts – even if the interpretation is translated in the temporal domain[1]. Thus, the following theorem, called the *translation rule*, has been proven:

Theorem 3.2.1 *If A is a TILCO formula, $\mathcal{I}$ is an interpretation for A and $s, t \in \mathbb{Z}$ then*

$$\mathcal{I}, t \models A \quad \textit{if and only if} \quad \tau(\mathcal{I}, s), t - s \models A.$$

where τ is the temporal translation – see definition 2.1.

Corollary 3.2.1 *From the previous theorem it follows as a corollary that*

$$\exists \mathcal{I} t.\mathcal{I}, t \models A \quad \textit{if and only if} \quad \forall t.\exists \mathcal{I}.\mathcal{I}, t \models A.$$

As in [143], [55], [72], in TILCO the *generalization rule* cannot be applied to time-dependent variables and predicates, thus having a *time generalization* rule (TG). This is due to the implicit model of time. Therefore, TILCO needs a different kind of rule to permit generalization over the temporal domain. Thus, as a consequence of TG, the following rule has been proven:

$$\frac{\vdash A}{\vdash A@(-\infty, +\infty)}.$$

This rule states that, if formula A is provable in every time instant, then formula $A@(-\infty, +\infty)$ is true in every time instant. Formula A in the rule's premise must be provable in *every* time instant, otherwise – because a formula is true in a given time instant, it could be deduced that the same formula is always true, which is clearly unacceptable. Moreover, it can be easily shown that $A \Rightarrow (A@(-\infty, +\infty))$ is not provable.

3.3 Property Proof

To support the validation of TILCO system specifications, TILCO theory has been formalized in Isabelle [120], [122], an automatic theorem-proving environment. It allows the definition of new theories and the demonstration of theorems by using either manual or automatic techniques. Isabelle is written in Standard ML [121] language, which is also used for constructing functions and tools for automatic theorem proving.

TILCO theory was built atop Isabelle/HOL [123], an implementation of Church's High Order Logic [39]. The use of HOL to construct FOL theories has been justified in [28],

[1] This is possible since the specification of temporal constraints is given with respect to events and actions.

[29], where this is shown that it not only demonstrates theorems *in* the object logic (i.e., TILCO), but also theorems *about* the object logic.

Intervals are implemented as connected sets of integers using HOL's set theory. Interval theory provides constructors for using intervals with the usual mathematical notation with round and square brackets. TILCO theory is based on integer and interval theories; it defines the TILCO syntax and semantics using Isabelle/HOL as a metalogic for defining operator semantics. A comprehensive set of TILCO operator theorems were proven to simplify the construction of theorems either in manual or semi-automatic manner.

TILCO-theory support in Isabelle/HOL permits an absolute degree of confidence in a proven theorem's truth. It is much safer than using a *pencil and paper* approach, because the use of Isabelle ensures that the demonstrations built are, in fact, correct. In general, with logical approaches, the problem is to demonstrate high-level properties by using low-level specifications, which usually describe uncorrelated elementary system properties. This can be simplified by using an incremental approach to specification through theorem proving, thus allowing either a top-down or a bottom-up approach:

- Top-Down Approach – A high-level specification is refined into a lower-level specification using theorem proving techniques to validate the refinement, until a detailed specification of the system is achieved;
- Bottom-Up Approach - Theorem proving techniques are used on low-level specifications to prove higher-level properties. This process is repeated until the desired top-level properties are proven.

This approach easily supports the validation of high-level properties – constituting a high-level specification with respect to the system specification, where intermediate lemmas can also be viewed as intermediate system specifications. The approach also supports the reuse of specifications from commonly used systems with proven characteristic properties to specify more complex systems.

Chapter 4

An extension of TILCO for the specification of complex systems

In this chapter, an extension of TILCO to support the specification of complex systems is presented. In TILCO, the specification of a system is given as a set of TILCO formulæ that represent the system behavior in response to external stimuli, that are modeled as temporal variables. This approach is feasible for systems of little/medium size, for large/complex systems where to specify the system behavior there are hundreds or even thousands of formulæ, this approach is unmanageable. Typically not all the formulæ are related each other, some specify the behavior of parts while other specify other features, therefore it is natural to divide the complex system in parts.

4.1 Introduction

Composition/decomposition techniques are mechanisms to cope with the general system complexity. Most of software development methodologies address the structural composition/decomposition of the systems. A composite object is defined in terms of its sub-object/components and their relationships. Object-based and object-oriented approaches include and formalize composition/decomposition concepts. Different communication mechanisms among components: shared variables, synchronous or asynchronous communications are chosen. Components can be separately developed, tested and then combined for modeling the whole system. Problems arise when the combination of components produces unexpected and, thus, difficulty controllable and verifiable behavior for the presence of communication among components. To this end, verification and validation criteria for

compositional methods are used [34], [20]. These must address the verification and validation of composition of components and their relationships with the requirements of the composite object.

For complex and large systems, the compositional approaches are typically accompanied by the availability of a layering support. The verification of consistency between composite object and its components at each level of the structural hierarchy guarantees the satisfactory of the abstract specification and thus of system requirements – for example, [57],[92], [34], [20].

4.2 CTILCO Overview

A system specification in C-TILCO is a hierarchy of communicating processes whose specifications are written in TILCO. Many instances of the same process can be present in the specification. Processes can have some general static parameters and every instance could have different values.

The communication between processes is based on typed synchronous input/output ports connected through channels. The connection is 1:1, each output port is connected to at most one input port and vice versa. In the following, the way in which processes are modeled in C-TILCO is introduced and in the next sections the formalization of communication between processes in TILCO and the way that could be used for reasoning about communicating processes are presented.

In the following a *process* represents a class according to object-based formalism. In C-TILCO a process is represented by two views:

1. the *external view* that basically describes the input/output behavior of the process;
2. the *internal view* that describes the process decomposition into subprocesses or a low-level formalization of the process behavior if it cannot be furtherly decomposed.

A C-TILCO process is *externally* characterized by:

- a set of external *input* ports used to acquire information from the outside;
- a set of external *output* ports used to produce information to the outside;
- a set of external *variables* used to give some general information about the process state or to simplify the external behavior specification;
- a set of external *parameters* used to permit general process specification to make easy process reuse, since different process instances may have different parameters;

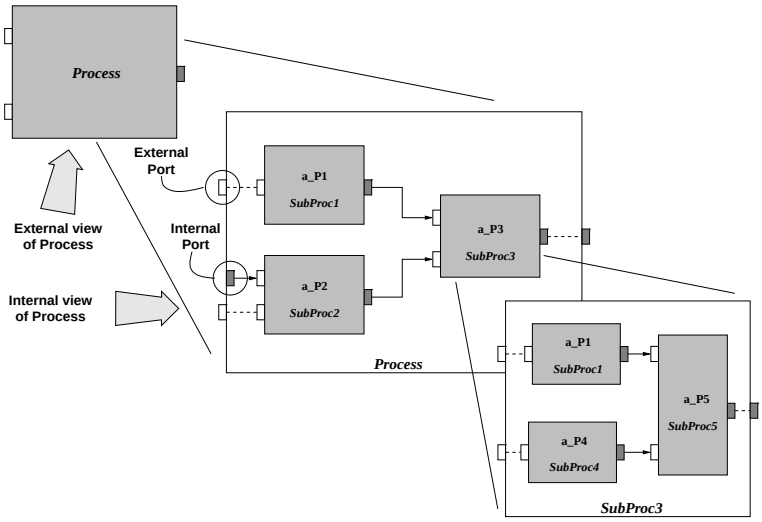

Figure 4.1: External and internal representation of a C-TILCO process

- a set of external TILCO *formulæ* that describe the external process behavior by means of the messages exchanged and constraints on the external variables,

while is *internally* characterized by:

- a set of C-TILCO *subprocesses*;
- a set of internal *input* ports, used to get information from subprocesses;
- a set of internal *output* ports used to send information to subprocesses;
- a set of internal *variables*;
- a set of internal TILCO *formulæ*, that describe the internal behavior of the process.

The ports of subprocesses can be directly connected to the containing process ports (of the same type, input to input and output to output) or can be connected through channels to the complementary internal ports (output to input and input to output). In Figure 4.1, a decomposition is exemplified. The use of internal ports permits even a *partial* decomposition, when not all the process behavior is partitioned in subprocesses and some interactions with the subprocesses is done in the internal specification TILCO formulæ.

In TILCO formulæ, to access process components the *dot* notation is used, for example if p is a process with a variable v then $p.v$ is used to refer to the variable of p. Moreover, if process p has a subprocess s with a variable v then $p.s.v$ is used to access to the subprocess variable.

Since many instances of the same process can be present in the system, its specification is valid for all of them. For example, if in the internal specification of a process with a variable *ivar* we have the following formula:

$$:ivar = 1 \Rightarrow (:ivar = 0)@[20, 20]$$

it means that if *ivar* is equal to 1, then after 20 time units *ivar* will be equal to 0. This will be true in each process independently.

With the *colon* operator, within a formula, process and local variables can be easily identified. This leads to a great readability of formulæ.

Since in TILCO the time axis is infinite in both directions there is not a time instant that can be regarded as the *start* instant of execution process. In the specification of a system it is natural to think at a reference time instant in which the process starts its work and before that the signals are stable. For this reason a boolean variable *process_start* has been introduced to each process. This variable is true only in one instant for each process. It should be noted that each process has its own start instant and a formula of the internal specification is used to define the start time instant of its subprocesses, typically when a process starts all its subprocesses start.

4.3 CTILCO Communication Model

The communication between two processes is structured in two layers: the *low-level* communication model for transmission of typed messages and of acknowledgements (ACKs); and the *high-level* communication model that uses the low-level to realize a synchronous communication protocol.

4.3.1 Low-level communication

Properties assumed for the low-level are:

- **no data creation**: a message (or ACK) arrived has been surely sent;
- **no data loss**: a message (or ACK) sent will be received;
- **constant delay**: a message (or ACK) sent will be received after a constant delay greater or equal than zero.

From these assumptions the *no reorder* property can be derived (messages arrive in the same order as they are sent). The *no data creation* assumption is fundamental (without

this assumption communications have no sense). The *no data loss* and *constant delay* assumptions have been introduced to have a deterministic behavior.

In this layer, the following four low-level temporal predicates are defined:

`<outPort>`.send(`<expr>`)
: is true when output port `<outPort>` sends the value obtained by evaluating expression `<expr>`.

`<outPort>`.receiveAck
: is true when an ACK has been received by output port `<outPort>`.

`<inPort>`.receive(`<expr>`)
: is true when a message has been received by input port `<inPort>` with the value indicated by `<expr>`.

`<inPort>`.sendAck
: is true when input port `<inPort>` sends an acknowledgement.

There is also a *connection* predicate between ports:

$$outP \xrightarrow{d} inP$$

that asserts that output port $outP$ is connected to input port inP and messages (and ACKs) sent are delayed of d time units.
Please note that *connections* are static assertions, design-fixed. With a little definition effort C-TILCO may be extended to permit dynamic connections.

The rules introduced to manage low-level communication are reported in the following:

message transmission:

$$(outP \xrightarrow{d} inP) \Rightarrow \text{rule}(outP.\,\text{send}(k) \iff inP.\,\text{receive}(k)@[d,d])$$

This rule states: if port $outP$ is connected to port inP then in every time instant, $outP$ sends a message if and only if inP receives the same message after d time units. From this rule, we have that the message sent is received after d time units (no data loss) and that the message received has been sent d time units ago (no creation).

ack transmission:

$$(outP \xrightarrow{d} inP) \Rightarrow \text{rule}(inP.\,\text{sendAck} \iff outP.\,\text{receiveAck}\,@[d,d])$$

This rule is similar to the previous except that it deals with the ACKs and that the direction is opposite (from input port to output port).

send one value:

$\text{rule}(outP.\,\text{send}(k) \wedge outP.\,\text{send}(v) \Rightarrow k = v)$

This rule states: if at the same time instant two values are sent on the same port these values have to be equal.

receive one value:

$\text{rule}(inP.\,\text{receive}(k) \wedge inP.\,\text{receive}(v) \Rightarrow k = v)$

This rule states: if at the same time instant two values are received on the same port these values have to be equal.

4.3.2 High-level Communication

The high-level layer introduces synchronous ports, the basic operators on these ports are *Send* (!!) and *Receive* (??) these are easy to remember due to their similarity with CSP:

`<outPort>` !! `<expr>` [`<whileExpr>`] ; ; `<thenExpr>` sends through output port `<outPort>` the value obtained by evaluating expression `<expr>`. When the communication ends TILCO expression `<thenExpr>` is asserted, while during the waiting the temporal expression `<whileExpr>` is asserted.

`<inPort>` ?? [`<whileExpr>`] ; ; `<thenExpr>` waits for a message (if not already arrived) from input port `<inPort>`. When the message arrives TILCO expression `<thenExpr>` is evaluated as a function of the value received, while during the waiting the expression `<whileExpr>` is asserted.

In order to specify that a process has not to send a message on a port or that the process has not to ask for a message other two operators: $outP\,\overline{!!}$ and $inP\,\overline{??}$ have been introduced. These conditions cannot be specified by using $\neg(inP\ !!\ v\ [P]\,;;W)$ which has a different meaning.

High-level synchronous operators are defined in TILCO by using the low-level predicates as reported in the following. In Figure 4.2 the two cases of synchronous communication are reported: (i) the emitting process sends a message, and after the receiving process asserts that wants to receive a message; (ii) the receiving process waits for a message and after the emitting process sends the message.

- operator **Send** emits the message and waits for an ACK. While it is waiting wait formula, W_s, is asserted and no other messages are sent. When the ACK arrives the "end of communication" formula, P_s, is asserted. The behavior of Send can be

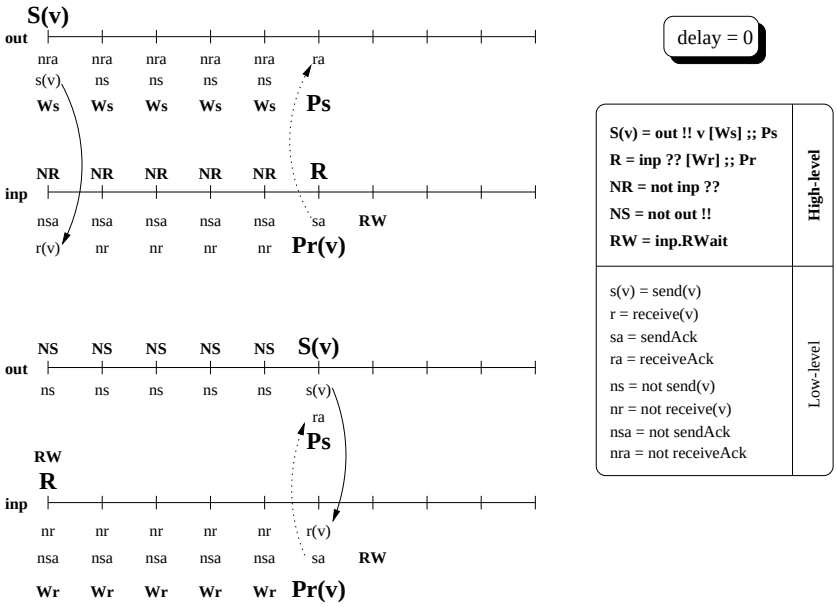

Figure 4.2: Examples of synchronous communications with no delay.

specified in TILCO with the following axioms:

$$\begin{aligned}
&\text{rule}((outP\ !!\ v\ [W_s]\,;;\,P_s) \Longrightarrow outP.\,\text{send}(v) \wedge \\
&\quad \mathbf{until_0}(outP.\,\text{receiveAck} \wedge P_s) \\
&\qquad (\neg outP.\,\text{receiveAck} \wedge W_s) \wedge \\
&\quad (outP.\,\text{receiveAck} \vee \\
&\qquad (\neg outP.\,\text{receiveAck} \wedge \\
&\qquad\quad \mathbf{until}(outP.\,\text{receiveAck}) \\
&\qquad\qquad (\neg outP.\,\text{receiveAck} \wedge outP\,\overline{!!}))))
\end{aligned}$$

$$\text{rule}(outP\,\overline{!!} \Longrightarrow \neg\exists k.outP.\,\text{send}(k))$$

the $\mathbf{until_0}$ formula is used to state that P_s is true when the ACK is received and W_s is true until this time instant. The other part of the formula states that during the waiting for the ACK no message is sent.

- operator **Receive** has two possible situations. If there exists a message received in the past that was not acknowledged the ACK must be sent and the "end of communication" formula, P_r, is asserted with the value received. In the other case, a new message has to be waited asserting wait formula W_r. When a message is received (if any) the "end of communication" formula, P_r, is evaluated with the value received.

The behavior of Receive can be specified in TILCO with the following axioms:

$$\begin{aligned}
&\text{rule}((inP\,??\,[W_r]\,;;\,P_r) \wedge inP.\text{RValue}\ v \implies \\
&\qquad inP.\,\text{sendAck} \wedge P_r(v)) \\
&\text{rule}((inP\,??\,[W_r]\,;;\,P_r) \wedge inP.\text{RWait} \implies \\
&\qquad \mathbf{until_0}\,(\exists k.inP.\,\text{receive}(k) \wedge inP.\,\text{sendAck} \wedge P_r(k)) \\
&\qquad\qquad (\neg\exists k.inP.\,\text{receive}(k) \wedge W_r) \wedge \\
&\qquad (\exists k.inP.\,\text{receive}(k) \vee \\
&\qquad\quad (\neg\exists k.inP.\,\text{receive}(k) \wedge \neg\ inP.\,\text{sendAck} \wedge \\
&\qquad\qquad \mathbf{until}(\exists k.inP.\,\text{receive}(k)) \\
&\qquad\qquad\qquad (\neg\exists k.inP.\,\text{receive}(k) \wedge inP\,\overline{??}))))
\end{aligned}$$

$$\text{rule}(inP\,\overline{??} \implies \neg\ inP.\,\text{sendAck})$$

$$\text{rule}(inP\,??\,[W_r]\,;;\,P_r \wedge inP\,\overline{??} \implies \bot)$$

where:

$$inP.\text{RValue}\ v\ = \mathbf{since'}(inP.\,\text{receive}(v) \wedge \neg inP.\,\text{sendAck})\ (\neg inP.\,\text{sendAck})$$

is a formula indicating that there exists a pending v message; and formula

$$inP.\text{RWait}\ = \neg\exists v.\ inP.\text{RValue}\ v$$

states the absence of a pending message to be elaborated (the current instant is not considered).

In Figure 4.3, the more complex case in which there is a delay in transmission is shown. Even in this case there are two situations: (i) when the distance from the Send and the subsequent Receive is greater than the delay, thus the message is received prior to the Receive action; and the opposite case (ii) when the Send action is performed after the Receive or before it but with a distance lower then the delay.

4.4 CTILCO Communication Theorems

In the definition, many properties have been proved about the communication operators, in order to validate the definitions of operators and to aid the construction of proofs involving these operators. The proofs were made by using a formalization of TILCO and C-TILCO in Isabelle/HOL.

Theorems proved can be divided in two groups:

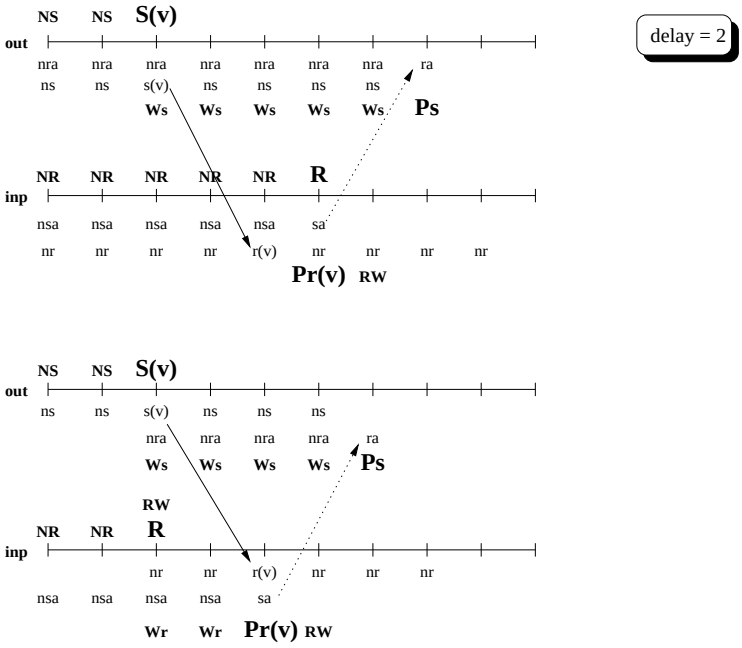

Figure 4.3: Examples of synchronous communications with delay.

- theorems used to prove internal properties of a process. They substitute operators Send and Receive with their semantics;

- theorems used to prove properties involving connected processes.

In the first group, there are the theorems that can be used to eliminate a Send from the assumptions of a goal.

$$\frac{\begin{array}{c}[\vdash_t p.\,\mathrm{send}(v)]\\ \vdots\\ \vdash_t p \;!!\; v\;[W_s]\,;;\,P_s \quad \vdash_t p.\,\mathrm{receiveAck?}[0,+\infty)\end{array}}{\vdash_t \mathbf{until}'_0\; P_s\; W_s}$$

$$\frac{\vdash_t p \;!!\; v\;[W_s]\,;;\,P_s}{\vdash_t \mathbf{until}_0\; P_s\; W_s}$$

The first theorem states that: if the process wants to send a message at time t and the message is sent receiving the ACK, then a time instant exists in which P_s is true and until that time instant, predicate W_s is true. This theorem can be used to substitute the Send with a strong until in the assumptions of the goal within the backward proofs of Isabelle.

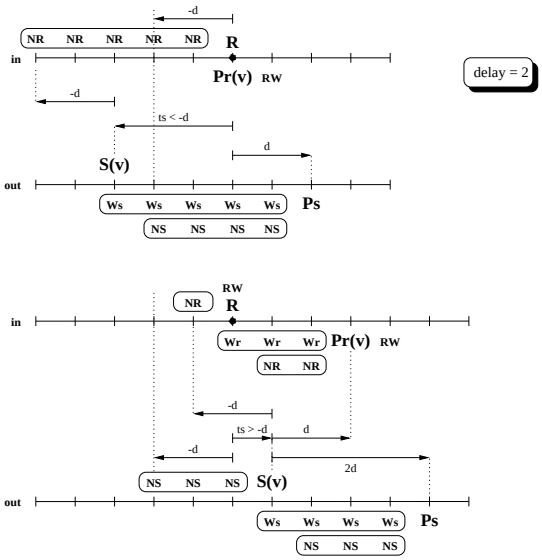

Figure 4.4: Theorems for synchronous communication

The second theorem is similar to the previous one but without the assumption that *if a message is sent an ACK will be received*, in this weaker condition the same condition, but with the weak-until, can be derived.

For the Receive, similar theorems have been proved:

$$\frac{\vdash_t p \,??\, [W_r]\,;;\, P_r \quad \vdash_t \exists k.\ p.\,\mathrm{receive}(k)\ ?\ [0,+\infty)}{\vdash_t \exists v.\ \ \mathbf{until}'_{\mathbf{0}}\, P_r(v)\ W_r}$$

$$\frac{\vdash_t p \,??\, [W_r]\,;;\, P_r}{\vdash_t \exists v.\, \mathbf{until_0}\, P_r(v)\ W_r}$$

The first theorem of Receive states that, if a message will be received the operator Receive may be substituted with a strong until. The other theorem substitutes the Receive operator with a weak until, making no assumptions about the message arrival.

In Figure 4.4, the visual descriptions of the next two theorems proved are reported. The assumptions of the theorems are depicted over the time axis while consequences are below.

In the theorems used to prove properties for connected processes, the *RWait* operator

plays an important role. It summarizes the communication status:

$$\frac{\begin{array}{l}\vdash out \xrightarrow{d} in \\ \vdash_t in\,??\,[W_r]\,;;\,P_r \\ \vdash_{t+t_s} out\,!!\,v\,[W_s]\,;;\,P_s \\ \vdash_t in\,\overline{??}\,@[t_s-d,0) \\ t_s < -d\end{array}}{\begin{array}{l}\vdash_t P_r(v) \\ \vdash_{t+d} P_s \\ \vdash_t W_s@[t_s,d) \\ \vdash_t out\,\overline{!!}\,@(t_s,d) \\ \vdash_{t+1} in.\mathrm{RWait}\end{array}}$$

This means in the premises: if two ports are connected with a delay d, a Receive is asserted at time t, and a Send is asserted t_s instants before the Receive. In the implication: the message is received at time t, P_s is true after d time instants, the wait formula of Send is true from the Send time instant to the end of communication time instant, and at $t+1$ *RWait* is true stating that no message is pending.

The following theorem covers the opposite case, that is when there is not a pending message, and the Send is done after the Receive or within the delay.

$$\frac{\begin{array}{l}\vdash_t in.\mathrm{RWait} \\ \vdash out \xrightarrow{d} in \\ \vdash_t in\,??\,[W_r]\,;;\,P_r \\ \vdash_{t+t_s} out\,!!\,v\,[W_s]\,;;\,P_s \\ \vdash_t in\,\overline{??}\,@[t_s-d,0) \\ \vdash_t out\,\overline{!!}\,@[-d,t_s) \\ -d \le t_s\end{array}}{\begin{array}{l}\vdash_{t+t_s+d} P_r(v) \\ \vdash_{t+t_s+2d} P_s \\ \vdash_t W_r@[0,t_s+d) \\ \vdash_{t+t_s} W_s@[0,2d) \\ \vdash_t in\,\overline{??}\,@(0,t_s+d) \\ \vdash_{t+t_s} in\,\overline{!!}\,@(0,2d) \\ \vdash_{t+t_s+d+1} in.\mathrm{RWait}\end{array}}$$

Using these theorems the following ones can be derived for communication with no delay, it should be noted that some premises have been removed since in this case it refers to an empty interval:

$$\frac{\vdash out \to in \quad \vdash_t in\,??\,[W_r]\,;;\,P_r \quad \vdash_{t+t_s} out\,!!\,v\,[W_s]\,;;\,P_s \quad \vdash_t in\,\overline{??}\,@[t_s,0) \quad t_s < 0}{\vdash_t P_r(v) \quad \vdash_t P_s \quad \vdash_t W_s@[t_s,0) \quad \vdash_t out\,\overline{!!}\,@(t_s,0) \quad \vdash_{t+1} in.\mathrm{RWait}}$$

$$\frac{\vdash_t in.\mathrm{RWait} \quad \vdash out \to in \quad \vdash_t in\,??\,[W_r]\,;;\,P_r \quad \vdash_{t+t_s} out\,!!\,v\,[W_s]\,;;\,P_s \quad \vdash_t out\,\overline{!!}\,@[0,t_s) \quad 0 \le t_s}{\vdash_{t+t_s} P_r(v) \quad \vdash_{t+t_s} P_s \quad \vdash_t W_r@[0,t_s) \quad \vdash_t in\,\overline{??}\,@(0,t_s) \quad \vdash_{t+t_s+1} in.\mathrm{RWait}}$$

Other theorems consider the case in which a process is sending/receiving on a port and the other process will never receive/send on the connected port. In this case, the waiting formula is always true in the future and the process will never send/receive on the related ports:

$$\frac{\vdash_t in.\text{RWait} \quad \vdash out \xrightarrow{d} in \quad \vdash_t in\,??\,[W_r]\,;;\,P_r \quad \vdash_t out\,\overline{!!}\,@[-d,+\infty)}{\vdash_t W_r@[0,+\infty) \quad \vdash_t (in\,\overline{??})@(0,+\infty)}$$

$$\frac{\vdash out \xrightarrow{d} in \quad \vdash_t out\;!!\;v\;[W_s]\,;;\,P_s \quad \vdash_t in\,\overline{??}\,@[-d,+\infty)}{\vdash_t W_s@[0,+\infty) \quad \vdash_t (out\,\overline{!!})@(0,+\infty)}$$

Other theorems have been proved, some about the *RWait* operator. In particular, the followin theorem:

$$\frac{\vdash_t in.\text{RWait} \quad \vdash out \xrightarrow{d} in \quad \vdash_t out\,\overline{!!}\,@[-d,s-d) \quad 0 \leq s}{\vdash_{t+s} in.\text{RWait}}$$

permits to deduce that if *RWait* is true for an input port and the connected emitting process is not sending, then *RWait* will remain true.

Another case is when the emitting process has never sent a value, from this we can deduce that RWait is true:

$$\frac{\vdash out \xrightarrow{d} in \quad \vdash_t out\,\overline{!!}\,@[-\infty,-d)}{\vdash_t in.\text{RWait}}$$

4.5 CTILCO Specification Validation

In order to validate a CTILCO specification, properties can be proved by using the Isabelle/HOL theorem prover with the formalization of TILCO and CTILCO. In that environment, theorems reported in the previous section and many others facilitate the proofs of properties manually or automatically. It should be noted that in this environment properties can be proved for the entire system as well as for single processes with generic parameters.

Properties proved are typically safeness (nothing bad will never happen) or liveness properties (something good will happen). Other kind of properties that can be proved is the verification of the process decomposition, that is, the proof of the external properties stated for a process by means of its internal specification (decomposition).

Since TILCO specifications can be executed by using a causal inferential engine (see chapter 6) even a C-TILCO specification can be executed. Obviously, not all the specifications can be executed, quantifications have to be done on finite domains, the specifications

have to be deterministic and no generic parameters have to be present. However, the specification can be time incomplete, that is the behavior of the system can be partially specified for all the time instants.

4.6 An Example

In this section, an example to highlight the composition and reuse capabilities of C-TILCO is presented together with some validations.

The system under specification is an abstraction of a train system that connects a set of stations. Every train passes from a fixed set of stations with a cyclic path. A train needs bounded time duration to go from a station to the next one. In order to enter in a station the train has to ask the permission; once the permission is granted the train remains in the station for a constant time duration and then it leaves the station for the next one. Every station can have only one train inside as the same time. As an example we consider the system shown in Figure 4.5. The system is decomposed with three types of processes:

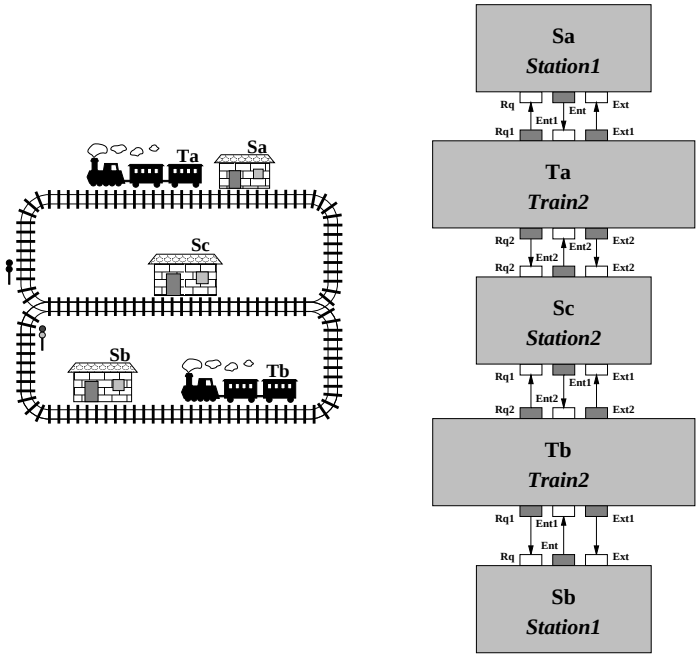

Figure 4.5: The railway system and its decomposition.

- process *Station1* (**Sa** and **Sb**) manages the access of only one train.
- process *Station2* (**Sc**) manages the access of two trains.

- process *Train2* (**Ta** and **Tb**) models a train that accesses to two stations.

Please note that the specification at system level only consists in the definition of process relationships and a global start predicate.

In order to mange the access to a station, three ports are needed, one for the request to enter in the station (Rq), another to give access to the station when the station is free (Ent), and the last to notify at the station that the train has left the station (Ext).

4.6.1 Process *Station1*

Process *Station*1 has three ports (Rq, Ent, Ext) to communicate with the train and three Boolean internal variables:

- *hasTrain* stating that the station has a train inside.
- *waitRq* that is true when the process has to wait for a request of the train.
- *waitExt* that is true when the process has to wait for the notification of the train exiting.

When the process starts, it has to wait for a request and before the starting the station has no train inside and no communication has been issued:

$$\begin{aligned} &:process\_start \implies :waitRq \wedge (\neg :hasTrain)@(-\infty, 0) \\ &:process\_start \implies (:Rq\,\overline{??} \wedge :Ent\,\overline{!!} \wedge :Ext\,\overline{??})@(-\infty, 0) \end{aligned}$$

The general behavior is specified with the following formula:

$$\begin{aligned} :waitRq \implies\ & :Rq\,??\,[\neg :hasTrain \wedge :Ent\,\overline{!!} \wedge :Ext\,\overline{??}]\,;; \\ & :Ent\,!!\,enter\,[\neg :hasTrain \wedge :Rq\,\overline{??} \wedge :Ext\,\overline{??}]\,;; \\ & :Ext\,??\,[:hasTrain \wedge :Rq\,\overline{??} \wedge :Ent\,\overline{!!}]\,;; \\ & :waitRq \end{aligned}$$

stating that if the process has to wait for a request a Receive is performed on port *Rq* and when a request is received the grant is immediately sent. During the waiting of the Receive on *Rq* and the Send on *Ent*, the train is not in the station ($\neg :hasTrain$). When the grant is received the process waits for the exit notification. In this while, it asserts that the train is in the station. When the notification is received the *waitRq* variable is newly asserted to begin the waiting for a new request. It should be noted that during the waiting on a certain port the waiting predicate states that the process is not sending/receiving on the other ports. This is given for granted in the following.

4.6.2 Process *Station2*

This kind of process has six ports (Rq1, Ent1, Ext1, Rq2, Ent2, Ext2) to communicate with the two trains and two Boolean variables *hasTrain1* and *hasTrain2* stating that the station has train 1 or 2 inside, respectively.

A general requirement is that only one train can be inside the station at the same time instant:

$$(\neg(:hasTrain1 \wedge :hasTrain2))@(-\infty, +\infty)$$

For the internal specification of process *Station*2 the following Boolean variables have been used:

- $free$ states that the station in free;
- $waitRq1$ and $waitRq2$ when true indicate that the process has to wait for an access request of train 1 or 2, respectively;
- $req1$ and $req2$ indicate the receipt of an access request for train 1 or 2, respectively. It remains true until the train has access to the station;
- $sendEnt1$ and $sendEnt2$ when true indicate that the process has to send to train 1 or 2 the enter notification and to wait for the exit notification.

For the specification the following shortcuts have been used:

$$\begin{aligned} A \Rrightarrow B &\equiv A \Rightarrow B@[1,1] \\ \mathrm{inv}(A) &\equiv A \Leftrightarrow A@[-1,-1] \end{aligned}$$

The $free$ variable is defined as:

$$:free \Longleftrightarrow \neg :hasTrain1 \wedge \neg :hasTrain2$$

When the process starts, it has to wait for the requests, until a request is received $req1/2$ is false and when the request is received $req1/2$ becomes true:

$$\begin{aligned} :process\_start &\Rightarrow :waitRq1 \wedge :waitRq2 \wedge :free\,@\,(-\infty, 0] \\ :process\_start &\Rightarrow (:Rq1\,\overline{??} \wedge :Ent1\,\overline{!!} \wedge :Ext1\,\overline{??})\,@\,(-\infty, 0) \\ :process\_start &\Rightarrow (:Rq2\,\overline{??} \wedge :Ent2\,\overline{!!} \wedge :Ext2\,\overline{??})\,@\,(-\infty, 0) \\ :waitRq1 &\Rightarrow :Rq1?? \; [\neg :req1 \wedge :Ent1\,\overline{!!} \wedge :Ext1\,\overline{??}]\,;; \\ &\quad\; :req1 \wedge \neg :hasTrain1 \wedge :Ent1\,\overline{!!} \wedge :Ext1\,\overline{??} \end{aligned}$$

$$:waitRq2 \quad \Rightarrow \quad :Rq2?? \; [\neg :req2 \wedge :Ent2\,!!\wedge :Ext2\,\overline{??}]\,;;$$
$$:req2 \wedge \neg :hasTrain2 \wedge :Ent2\,!!\wedge :Ext2\,\overline{??}$$

When no requests have been received $hasTrain1/2$ remains stable (with the same value):

$$\neg :req1 \quad \Rrightarrow \quad \mathrm{inv}(:hasTrain1)$$
$$\neg :req2 \quad \Rrightarrow \quad \mathrm{inv}(:hasTrain2)$$

When the station is free and a request is received for a train but not for the other the enter notification is sent:

$$:free \wedge :req1 \wedge \neg :req2 \quad \Rrightarrow \quad :sendEnt1$$
$$:free \wedge :req2 \wedge \neg :req1 \quad \Rrightarrow \quad :sendEnt2$$

If two requests are contemporaneously received train 1 has the precedence:

$$:free \wedge :req1 \wedge :req2 \quad \Rrightarrow \quad :sendEnt1 \wedge :req2 \wedge \neg :hasTrain2$$

If the station is not free and a request is received the request is maintained active and no communication is issued:

$$\neg :free \wedge :req1 \quad \Rrightarrow \quad :req1 \wedge \neg :hasTrain1 \wedge :Rq1\,\overline{??}\wedge :Ent1\,!!\wedge :Ext1\,\overline{??}$$
$$\neg :free \wedge :req2 \quad \Rrightarrow \quad :req2 \wedge \neg :hasTrain2 \wedge :Rq2\,\overline{??}\wedge :Ent2\,!!\wedge :Ext2\,\overline{??}$$

When $sentEnt1/2$ is true the enter notification is sent and then the exit notification is waited, in this while $hasTrain1/2$ is true and no requests have to be received. When the exit notification is received, $hasTrain1/2$ becomes false and, at the next time instant, the process begins to wait for a new request (to leave the chance for a pending request to be served):

$$:sendEnt1 \quad \Rightarrow \quad :Ent1\,!!\,enter\;[:hasTrain1 \wedge \neg :req1 \wedge :Rq1\,\overline{??}\wedge :Ext1\,\overline{??}]\,;;$$
$$:Ext1??\;[:hasTrain1 \wedge \neg :req1 \wedge :Rq1\,\overline{??}\wedge :Ent1\,!!]\,;;$$
$$\neg :hasTrain1 \wedge \neg :req1 \wedge :waitRq1\;@\,[1,1]$$
$$:sendEnt2 \quad \Rightarrow \quad :Ent2\,!!\,enter\;[:hasTrain2 \wedge \neg :req2 \wedge :Rq2\,\overline{??}\wedge :Ext2\,\overline{??}]\,;;$$
$$:Ext2??\;[:hasTrain2 \wedge \neg :req2 \wedge :Rq2\,\overline{??}\wedge :Ent2\,!!]\,;;$$
$$\neg :hasTrain2 \wedge \neg :req2 \wedge :waitRq2\;@\,[1,1]$$

4.6.3 Process *Train2*

Process Train2 managing the access to two stations is decomposed in two kind of processes connected as depicted in Figure 4.6. Processes of type *TrainAtStation* manage the access

to a station while processes of type *MinMaxDelay* are used to model the time spent by the train to reach the next station. A deterministic delay can be fixed depending on the railway path length.

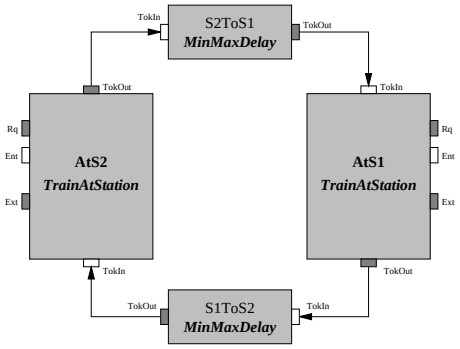

Figure 4.6: Train2 decomposition.

Ports $TokIn$ and $TokOut$ are used to sequentially activate the processes. When a message is received from port $TokIn$ the process is activated and when the process has finished a message is sent via the $TokOut$ port. It is a sort of token passing.

Reusing the above processes strongly more complex configurations can be defined and validated.

4.6.4 Process *TrainAtStation*

Process TrainAtStation manages the access to the station, the presence in the station and finally the departure from the station. It can be decomposed with three processes as shown in Figure 4.7. Process *EnterStation* manages the request of access to the station and the wait for enter notification. Process $MinMaxDelay$ (already presented in the upper levels) is reused to model the time spent by the train in the station. Process *ExitStation* states the exit from the station.

4.6.5 Process EnterStation

This process has to wait for the token, then it sends the access request, waits the enter notification, sends the token and waits for the token again.

$$
\begin{aligned}
:process\_start \quad &\Rightarrow \quad :waitTok \wedge (:TokIn\,\overline{??} \wedge :Ent\,\overline{??} \wedge :TokOut\,\overline{!!})\ @(-\infty, 0) \\
:waitTok \quad &\Rightarrow \quad :TokIn\ ??\ [\neg :waiting \wedge :Rq\,\overline{!!} \wedge :Ent\,\overline{??} \wedge :TokOut\,\overline{!!}]\ ;; \\
&\qquad\quad :Rq\ !!\ request\ [:waiting \wedge :TokIn\,\overline{??} \wedge :Ent\,\overline{??} \wedge :TokOut\,\overline{!!}]\ ;;
\end{aligned}
$$

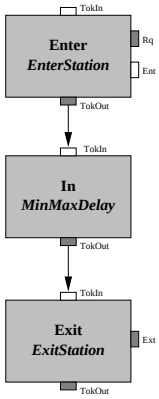

Figure 4.7: TrainAtStation decomposition.

$$:Ent\ ??\ [:waiting \wedge :TokIn\ \overline{??} \wedge :Rq\ \overline{!!} \wedge :TokOut\ \overline{!!}]\ ;;$$
$$:TokOut\ !!\ token\ [\neg :waiting \wedge :TokIn\ \overline{??} \wedge :Rq\ \overline{!!} \wedge :Ent\ \overline{??}]\ ;;$$
$$:waitTok$$

4.6.6 Process ExitStation

This process has to wait for the token, then it sends the exit notification, sends the token and waits for the token again.

$$\begin{aligned}
:process_start &\Rightarrow :waitTok \wedge (:TokIn\ \overline{??} \wedge :Ext\ \overline{!!} \wedge :TokOut\ \overline{!!})\ @(-\infty, 0) \\
:waitTok &\Rightarrow :TokIn\ ??\ [\neg :waiting \wedge :Ext\ \overline{!!} \wedge :TokOut\ \overline{!!}]\ ;; \\
&\quad :Ext\ !!\ exit\ [:waiting \wedge :TokIn\ \overline{??} \wedge :TokOut\ \overline{!!}]\ ;; \\
&\quad :TokOut\ !!\ token\ [\neg :waiting \wedge :TokIn\ \overline{??} \wedge :Ext\ \overline{!!}]\ ;; \\
&\quad :waitTok
\end{aligned}$$

4.6.7 Process MinMaxDelay

This process has to wait for the token and to send the token to the next process after a delay between $MinDelay$ and $MaxDelay$.

$$\begin{aligned}
:process_start &\Rightarrow :waitTok \wedge (:TokIn\ \overline{??} \wedge :TokOut\ \overline{!!})\ @(-\infty, 0) \\
:waitTok &\Rightarrow :TokIn\ ??\ [\neg :waiting \wedge :TokOut\ \overline{!!}]\ ;; \\
&\quad (\neg :sendTok\ @[0, :MinDelay) \wedge \\
&\quad :sendTok\ ?[:MinDelay, :MaxDelay] \wedge
\end{aligned}$$

$$
\begin{array}{rcl}
 & & \mathbf{until_0}\ {:}sendTok\ (\neg\,{:}waiting \wedge\, {:}TokIn\,\overline{??} \wedge\, {:}TokOut\,\overline{!!})) \\
{:}sendTok & \Rightarrow & {:}TokOut\ !!\ token\ [\neg\,{:}waiting \wedge\, {:}TokIn\,\overline{??}]\,;; \\
 & & {:}waitTok
\end{array}
$$

4.6.8 Validation

Using the proved rules reported in the previous sections several properties have been proved.

The specification has been formally validated with success by using Isabelle theorem prover. In addition, each single process can be tested with the TILCO executor. In this case, several typical histories for inputs and outputs can be generated and viewed by using a signal editor.

For example, for process $Station2$, the external mutual exclusion requirement has been derived from the internal specification, this can be considered as a decomposition verification and is also a safeness property proof.

For example, for the train **Ta**, the following liveness property has been proved:

$$
\begin{array}{l}
\mathrm{up}(:Ta.inStation1) \Longrightarrow \\
\quad \mathrm{up}(:Ta.inStation1)\,?\,[min_{\mathbf{Ta}}, max_{\mathbf{Ta}}]
\end{array}
$$

that is, the distance between two successive time instants in which the train enters in the first station is bounded. In the best case, the minimum time needed to across the path is:

$$
\begin{array}{rcl}
min_{\mathbf{Ta}} & = & Ta.timeInS1 + Ta.minS1ToS2 + \\
 & & Ta.timeInS2 + Ta.minS2ToS1
\end{array}
$$

while in the worst case we have:

$$
\begin{array}{rcl}
max_{\mathbf{Ta}} & = & Ta.timeInS1 + Ta.maxS1ToS2 + \\
 & & Tb.timeInS2 + Ta.timeInS2 + \\
 & & Ta.maxS2ToS1
\end{array}
$$

where: *timeInS1*, *timeInS2*, *maxS1ToS2*, *maxS2ToS1*, *minS1ToS2* and *minS2ToS1* are generic parameters of process $Train2$ expressing the time spent in each station and the maximum/minimum time to pass from a station to the next. *inStation1* is a Boolean variable indicating that the train is in the first station of its path.

Chapter 5

An extension of TILCO to simplify formulæ

In this chapter a further extension of TILCO is presented. With this extension Dynamic Intervals and Bounded Happen have been added. The first permits to specify intervals not only with constant integer values but also using other formulæ. The second permits to quantify the number of times a formula is true within an interval. The new syntax an semantics is presented and the new deductive system is introduced. An example is used to highlight the language expressivity.

5.1 TILCO-X, TILCO eXtension

TILCO, MTL and TRIO are first order temporal logics for real-time system specifications. They have a metric of time and thus can be profitably used for specifying qualitative and quantitative temporal constraints. Many other logics produce specifications structurally similar to TRIO and MTL or have similar operators; while, other logics can be difficult to use in comparison to those of TRIO, MTL and TILCO, since they are based on elementary operators that lead to the production of overly complex specifications.

For the specification of real-time systems, the conciseness, readability and understandability of the temporal logic used are strongly relevant. These features depend on the expressiveness of the logic, on the number of operators, on the structure of formulas (nesting levels, number of calls to special functions/parameterised predicates, presence of quantifiers), on the needs of user defined special operators, on and the need of adopting temporal quantifications.

A formal proof of TILCO's conciseness with respect to other temporal logics would

be difficult, mainly due to the lack of a formal definition of conciseness or readability. Moreover, an examination of the elementary specifications for TILCO, TRIO and MTL, has demonstrated that the typical specifications produced in TILCO use fewer distinct operators than in TRIO and MTL [99]. In addition, some specifications can be written in TRIO and/or MTL only by using nested operators, while simple direct operators are used in TILCO.

TILCO specifications are based upon four fundamental operators. A greater number of operators are used in TRIO/MTL-like formulas. Thus, complexity increases and conciseness decreases for both TRIO and MTL; and this leads to a higher cognitive complexity (or comprehensibility complexity) as in programming language (demonstrated by the validation of several cognitive metrics [100], [37], [151], [45], [112]).

TRIO and MTL are both based on points and present a sharp distinction between past and future. MTL and TRIO have distinct operators for past and future (e.g., MTL: **G**, **H**; TRIO: **Past()** and **Futr()**). In contrast, TILCO is an interval temporal logic with a uniform model for time from past to future. TILCO specifications can describe facts and events without to use different operators for past and future [99].

In the analysis of TILCO and several other temporal logics performed by the authors [21], two specific fields of improvement have been identified for the specification of real time systems:

- TRIO, MTL and many other temporal logics adopt **since** and **until** operators to specify dependencies between events. Also TILCO [99] adopts **since** and **until** operators for the same purpose. These operators make a strong distinction between past and future and, thus, their adoption frequently makes the specification complex and hard to read. The adoption of a unique operator for defining ordering relationships between events reduces in several cases the needs of the adoption of nested **since** and **until** operators.

- The needs of specifying the occurrence of one event from the repeated occurrence of another is quite frequent (operators for events counting are needed). For instance, *A has to start after the arrival of 5 messages on channel B within interval I.* Specifications with these constrains are quite complex to understand and difficult to realise by using classical temporal operators such as those proposed in TILCO, IL, TRIO, MTL, RTL, etc. This is the reason for which some temporal logics present specific operators for this purpose. In RTL, a special operator capable of recovering *at which instant the particular occurrence of an event happens* has beeen proposed [80]. These constraints can be specified in FOL and thus also in first order temporal logic but

the specification results to be strongly complex with respect to the complexity of the concept under specification and involves several quantifications.

These facts are a limitation for specification conciseness and readability and thus for the adoption of logical languages for the specification of real-time systems. To this end, the operators called *Dynamic Intervals* and *Bounded Happen* have been defined in TILCO-X by extending TILCO to enhance its readability and conciseness, especially for the expression of order relations. The new operators can be combined allowing the definition of very powerful real-time constraints in a strongly concise manner.

5.1.1 Dynamic Intervals

Dynamic Intervals have been introduced to: (i) avoid the needs of distinguishing between past and future for ordering relationships; (ii) avoid in several cases the nesting of **since** and **until** operators; (iii) reduce the number of quantifications; (iv) allow the combination of order and quantitative relationships.

These capabilities have been introduced in TILCO-X by making possible to write temporal intervals not only as constant integer sets, but also by using a formula as an interval bound.

For example, the following TILCO-X formula

$$A\,@[10, +B)$$

states that *A is true from 10 time units in the future until B is true for the first time*, where $+B$ identifies the first future instant in which B is true (from the evaluation time instant), if such an instant does not exist A is true forever in interval $[10, +\infty)$. These two conditions are represented in Figure 5.1.

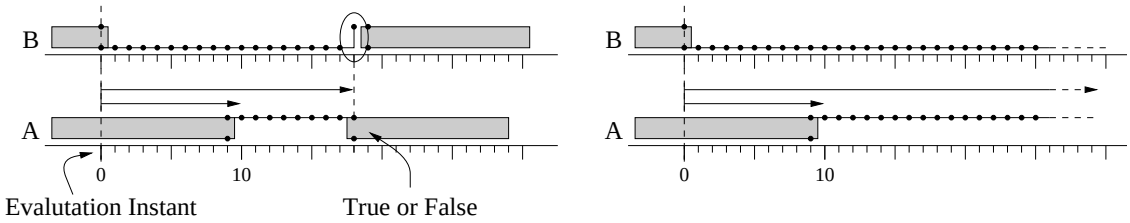

Figure 5.1: Example of Dynamic Interval: $A\,@[10, +B)$

In a similar way, an interval bound can be located in the past; for example, formula

$$A\,@(-B, 0]$$

states that *A is true since the last time instant in which B is true until the current instant.* Where $-B$ identifies the last instant where B is true.

It should be noted that, **until** and **since** operators can be defined by means of the following formulas:

$$\begin{aligned} \textbf{until}\ A\ B &\equiv B\,@(0, +A), \\ \textbf{since}\ A\ B &\equiv B\,@(-A, 0). \end{aligned}$$

In the following, the above mentioned applications of the Dynamic Interval solution are presented and discussed.

Distinction between past and future

The following specification is a typical case in which a distinction between past and future has to be performed for adopting of **since** and **until** operators: *since the last occurrence of C and until the first occurrence of D, for every occurrence of A there will be an occurrence of B at the same time.* In TILCO, it can be formalised as:

$$(\textbf{since}\ C\ (A \Rightarrow B)) \wedge (A \Rightarrow B) \wedge (\textbf{until}\ D\ (A \Rightarrow B)).$$

In MTL and TRIO structurally similar formulas are obtained.

With TILCO-X, it is possible to write intervals starting from the past and ending in the future; thus, the above specification results to be strongly simplified:

$$(A \Rightarrow B)\,@(-C, +D).$$

This TILCO-X formula can be read as: $A \Rightarrow B$ is true from the last occurrence of C in the past and the first occurrence of D in the future, with respect to the evaluation time instant. Another example in TILCO-X is the following formula specifying that *A or B happened in the last ten instants or will happen until C and D are true*:

$$(A \vee B)\,?[-10, +(C \wedge D))].$$

This example shows, how it is possible to write specifications in which the interval has a fixed lower bound in the past and a dynamic upper bound in the future. This last example can be written in TILCO in the following way:

$$(A \vee B)\,?[-10, 0] \vee \neg\,\textbf{until}(C \wedge D)(\neg(A \vee B)).$$

Nesting levels

The definition of intervals with dynamic bounds (identified by the validity of a generic formula) avoids in many cases the adoption of nesting temporal quantifiers. Thus, TILCO-X produces more concise and readable formulas.

An example could be the following TILCO-X formula

$$A\ @[+B, +\infty)$$

stating that *after the next occurrence of B, A is always true.* The same behaviour can be specified in TILCO by using nested **until** and **@** operators such as in:

$$\boxed{\textbf{until}\,(B \wedge \boxed{A\ @[0, +\infty)})\ (\neg B)}.$$

The box around formulas highlights the nesting levels of the temporal operators.

A more complex example can be the following TILCO-X formula

$$A\ ?[+B, +C].$$

This formula states that *A happens between the next occurrence of B and the next occurrence of C.* The interval boundaries are included; therefore, A may happen even at the same time instant as B or C. Without the avalability of the new construct the same formula has to be written by using two nested **until** operators.

$$B\ ?(0, +\infty) \wedge (\textbf{until}\ B\ \neg C) \wedge \boxed{\textbf{until}\,(B \wedge (A \vee \neg\,(\boxed{\textbf{until}\,(C \wedge \neg A)\ \neg A}\,)))\ (\neg B)}$$

B must happen in the future because otherwise the interval $[+B, +C]$ is empty and A cannot happen on an empty interval.

Reduction of the number of quantifications

Temporal quantifications are not allowed by TILCO language since their prohibition has been shown to be a necessary condition for the existence of feasible automated verification mechanisms [10], [99]. Therefore, in TILCO is very complex to specify certain constraints without the adoption of a direct temporal quantification. For example:

> *After every occurrence of event S, a signal A has to be true after the first occurrence of event E (if it happens) and A remains true until a certain deadline d (value relative to event S).*

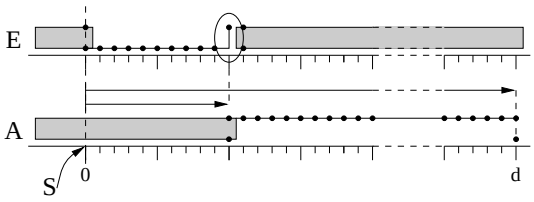

Figure 5.2: A complex constraint.

Figure 5.2 reports a representation of the constraint proposed which is complex to be specified without temporal quantification.

The specification complexity is due to the fact that, the above constraint is comprised of two parts: one relating S to next occurrence of E; and the second, relating S to the end of the temporal interval d (that is constant). In order, to specify the second part an @ operator could be used. The identified time instant in which E will occurs has to be considered as the starting bounds for its interval. This dependency, between the two parts, can be only defined by using a quantification.

A way to specify this requirement in TILCO is to introduce a clock variable "Ck" with the following property:

$$((\mathrm{Ck} = 0 \wedge \neg S)\,@(-\infty, 0] \wedge (\exists\, v.\ \mathrm{Ck} = v \ \Rightarrow\ (\mathrm{Ck} = v + 1)\,@[1, 1])\,@(0, +\infty))?(-\infty, +\infty)$$

stating that an instant (the initial) exists before Ck is zero and S does not happen, and that Ck is incremented by one at every time instant, after the initial time instant.

Using Ck, the above reported constraint can be written as follows in TILCO:

$$S \wedge E\,?(0, d) \Rightarrow (\exists v.\ \mathrm{Ck} = v \wedge \mathbf{until}\,(E \wedge \mathbf{until}\,(\mathrm{Ck} \geq v + d)\ A)\ (\neg E). \tag{5.1}$$

A little bit more complex formulas, but structurally similar, can be obtained by using MTL or TRIO. This writing modality for constraint specification should be avoided since it produces poorly readable specifications. This fact is much more relevant for complex specifications.

In other temporal logics that support time quantification such as TRIO, the above reported constraint can be rewritten as follows:

$$\forall\, t\ t'.\ S \wedge \mathbf{Futr}(E, t) \wedge (0 < t < d) \wedge (t < t' < d) \longrightarrow \mathbf{Futr}(A, t').$$

A structurally similar formula can be obtained in MTL.

In TILCO-X, the above constraint is simply stated by the following formula:

$$S \Rightarrow A \,@(+E, d),$$

that is simpler than both TRIO and TILCO versions.

Therefore, it has been shown that the adoption of new TILCO-X operators reduces the number of quantifications and operators, thus, increasing the readability and coinciseness of formulas.

Combination of order and quantitative relationships

Using TILCO-X is easy to write ordering relations between events, especially those that combine order and quantitative relationships.

For example, the following TILCO-X formula states that *A happens after B within 100 time units*:

$$A \,?(+B, 100].$$

In Figure 5.3, the visual representation of this condition is reported.

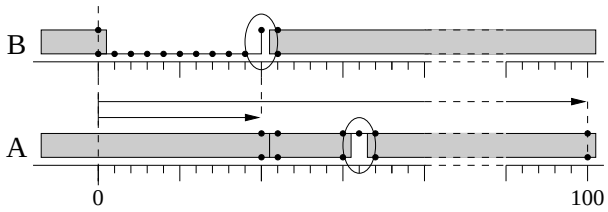

Figure 5.3: Example of Dynamic Interval: $A \,?(+B, 100]$

The above formula is quite *similar* to TILCO formula

$$A \,?(0, 100] \wedge \neg \textbf{until}\, A \; (\neg B).$$

These two formulas are not equivalent (the second implies the first); because according to TILCO-X formula, A may be true or not before B is true. This is not possible for the TILCO formula, as depicted in Figure 5.4.

In order to have the equivalence, A has to be false until B is true, thus the correct TILCO formula is:

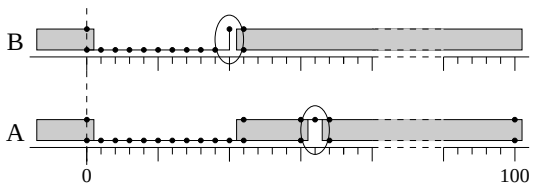

Figure 5.4: A model for formula $A\,?(0,100] \wedge \neg\,\textbf{until}\,A\;(\neg B)$

$$A\,?(0,100] \wedge \neg\,\textbf{until}\,A\;(\neg B) \Longleftrightarrow (\neg A)\,@(0,+B] \wedge A\,?(+B,100].$$

This is a further example of the TILCO-X conciseness with respect to TILCO. As demonstrated in [99], quite structurally similar formulas can be written for TRIO and MTL, but they result even less concise than TILCO formulas.

5.1.2 Bounded Happen

Bounded Happen has been defined to increase the readability of constraints which includes the dependence from the counting of occurrences. Sometimes constraint implies the counting the number of occurrences of an event or in general the number of times that a formula is true in a given time interval. In TILCO, as well as in TRIO, MTL and other temporal logics, such a requirement can be specified by using a variable to count the number of times that a formula is true from an instant to another. This can be performed by using formula $(C_{\rightarrow}(A, n_A))$ stating that n_A is a variable that counts the occurrences of A from the evaluation time instant.

For example, to state that an event E occurs at most five times in $[2, 10)$, the following formula can be used:

$$C_{\rightarrow}(E, n_E)\,@[2,2] \wedge (n_E \leq 5)\,@[2,10),$$

where:

$$\begin{aligned} C_{\rightarrow}(A, n_A) \stackrel{\text{def}}{=}\ & (\neg A \rightarrow n_A = 0) \wedge \\ & (A \rightarrow n_A = 1) \wedge \\ & ((\forall\, k.\ A \wedge (n_A = k)\,@[-1,-1] \rightarrow n_A = k + 1) \wedge \\ & (\forall\, k.\ \neg A \wedge (n_A = k)\,@[-1,-1] \rightarrow n_A = k))\,@(0,+\infty). \end{aligned}$$

When E occurs at least three times in $[2, 10)$, the formula is:

$$C_{\rightarrow}(E, n_E)\,@[2,2] \wedge (n_E \geq 3)\,?[2,10).$$

Therefore, a formula stating that *the above formula is true in every time instant* has to manage the variability of distinct counters that should be activated in each time instant:

$$(C_{\rightarrow}(E, n_E)\,@[2,2] \wedge (n_E \geq 3)\,?[2,10))\,@[0,+\infty).$$

A different solution can be based on the adoption of a unique counter for the whole constraint and an existential quantification:

$$C_{\rightarrow}(E, n_E) \wedge (\exists k.(n_E = k)\,@[2,2] \wedge (n_E \geq k+3)\,?[2,10))\,@[0,+\infty).$$

Bounded Happen operator has been introduced to specify the family of the above presented constraints in a concise manner. It can be used to state that a formula is true in an interval from a minimum to a maximum number of times. For example, TILCO-X formula:

$$A\,?_2\,[1,15)$$

states that A is true at least two times in interval $[1, 15)$. While TILCO-X formula

$$A\,?^3\,[1,15)$$

states that A is true up to three times in $[1, 15)$.

With the combination of these operators, it can be stated that a formula has to be true in the interval from a minimum to a maximum number of times; as it is shown with the following TILCO-X example:

$$A\,?_2^3\,[1,15).$$

Bounded happen can be used with *Dynamic Interval operator*. The following formula states that A happens two or three times until B happens:

$$A\,?_2^3\,[0,+B).$$

The bounded happen may be used to state that a formula *becomes* true a limited number of times in an interval, this can be achieved with the derived operator up ($\uparrow$) defined as

$$\uparrow A \stackrel{\text{def}}{=} A \wedge (\neg A)\,@[-1,-1].$$

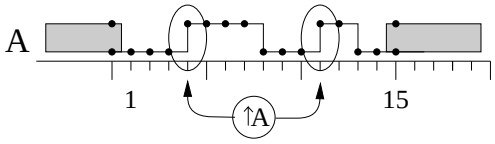

Figure 5.5: Example of Bounded Happen: $(\uparrow A)\,?_2^2\,[1, 15)$

Therefore, formula

$$(\uparrow A)\,?_2^2\,[1, 15)$$

states that A becomes true exactly two times in $[1, 15)$. A possible model for this formula is reported in Figure 5.5.

Additional interesting examples are:

$$(A \wedge B\,?_2^2\,(-100, 0)) \Rightarrow \neg A\,@(0, 10)$$

which states that *if A is true and in the last 100 time units there were two occurrences of B, then A will be false for 10 time units*; and

$$A\,?_1^3\,[0, +(B \wedge C)]$$

stating that *A will happen from one to three times until B and C are true.*

5.2 TILCO-X Syntax & Semantics

Given $\mathcal{F}$, $\mathcal{P}$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{C}$ as defined for old TILCO, the syntax of TILCO-X formulæ is defined by the following BNF-like definitions:

$$\begin{array}{rcl}
\text{interval} & ::= & \text{open limit , limit close} \\
& | & [\,\text{limit}\,] \\
& | & \text{open limit} , +\infty) \\
& | & (-\infty , \text{limit close} \\
& | & (-\infty , +\infty) \\
\text{limit} & ::= & i \quad \textit{for each } i \in \mathbb{Z} \\
& | & +\text{formula} \mid -\text{formula}
\end{array}$$

$$
\begin{array}{rcl}
\text{open} & ::= & (\mid [\\
\text{close} & ::= &] \mid) \\
\text{interval\_list} & ::= & \text{interval} \\
 & \mid & \text{interval interval\_op interval} \\
\text{interval\_op} & ::= & , \mid ; \\
\text{variable} & ::= & w \quad \textit{for each } w \in \mathcal{W} \\
\text{term} & ::= & v \quad \textit{for each } v \in \mathcal{V} \\
 & \mid & \text{variable} \\
 & \mid & c \quad \textit{for each } c \in \mathcal{C} \\
 & \mid & f(\text{term\_list}) \quad \textit{for each} f \in \mathcal{F} \\
\text{term\_list} & ::= & \text{term} \\
 & \mid & \text{term}, \text{term\_list} \\
\text{atomic\_formula} & ::= & p(\text{term\_list}) \quad \textit{for each } p \in \mathcal{P} \\
\text{formula} & ::= & \top \mid \bot \mid \text{atomic\_formula} \\
 & \mid & \neg\text{formula} \\
 & \mid & \text{formula op formula} \\
 & \mid & v := \text{term} \quad \textit{for each } v \in \mathcal{V} \\
 & \mid & \text{quantifier variable. formula} \\
 & \mid & \text{formula temporal\_quantifier interval\_list} \\
 & \mid & (\text{formula}) \\
\text{op} & ::= & \vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow \mid =\!\!\gg \mid =\!\!\ll \\
\text{quantifier} & ::= & \forall \mid \exists \mid \exists! \\
\text{temporal\_quantifier} & ::= & @ \mid ? \mid ?_m \mid ?^M \mid ?_m^M \quad \textit{for each } m, M \in \mathbb{N}
\end{array}
$$

Before defining the semantics of TILCO-X, it is important to introduce the concept of *interpretation* of a TILCO-X formula. This concept is also used to define the validity and the satisfiability of TILCO-X formulæ and has been derived from the corresponding concept of TILCO [99].

Given a syntactically correct TILCO-X formula A, with $\{t_1, \ldots, t_h\}$ set of types used in A, $\{p_1, \ldots, p_k\}$ predicates, $\{f_1, \ldots, f_l\}$ functions, $\{v_1, \ldots, v_m\}$ time-dependent variables, $\{c_1, \ldots, c_q\}$ constants then an *interpretation* $\mathcal{I}$ is a tuple

$$(\{D_1, \ldots, D_h\}, \{R_1, \ldots, R_k\}, \{F_1, \ldots, F_l\}, \{V_1(t), \ldots, V_m(t)\}, \{C_1, \ldots, C_q\})$$

where:

- $\{D_1, \ldots, D_h\}$ assigns a domain D_i to each type t_i;
- $\{R_1, \ldots, R_k\}$ assigns an n-ary relation R_i over $D_{i_1} \times \ldots \times D_{i_n}$ to each n-ary predicate p_i with arguments of type $t_{i_1}, \ldots, t_{i_n}$;
- $\{F_1, \ldots, F_l\}$ assigns an n-ary function F_i over $D_{i_1} \times \ldots \times D_{i_n}$ to each n-ary function f_i with arguments of type $t_{i_1}, \ldots, t_{i_n}$;
- $\{V_1(t), \ldots, V_m(t)\}$ assigns a function of time $V_i(t) : \mathbb{Z} \to D_n$ to each time-dependent variable v_i of type t_n, specifying the history of that variable in every time instant (where t is the absolute time);
- $\{C_1, \ldots, C_q\}$ assigns a value $C_i \in D_n$ to each constant c_i of type t_n;

Given a TILCO-X formula A and an interpretation $\mathcal{I}$ for A, notation

$$\mathcal{I}, t \models A$$

expresses that $\mathcal{I}$ is a *model* for A evaluated in the time instant t.

To properly define the TILCO-X temporal operators (@ and ?) a function, to interpret an interval I, is needed:

$$[\![I]\!]_{\mathcal{I},t} \subseteq \mathbb{Z}.$$

This represents the set of time instants corresponding to instant t where a formula defined over I has to be evaluated. The definition of this function, which makes TILCO-X strongly different with respect to TILCO, is reported later.

Moreover, to formally define the *Bounded Happen* operator, a function ($\mathrm{N}_{\mathcal{I},t}(I, A)$) to count the number of time instants where formula A is true in an interval I is needed. Its definition is

$$\mathrm{N}_{\mathcal{I},t}(I, A) \stackrel{\text{def}}{=} |\{i \in I \mid \mathcal{I}, i + t \models A\}|,$$

where $|\cdot|$ gives the number of elements in a set if the set is finite, or $+\infty$ if it is infinite.

The evaluation of $\mathcal{I}, t \models A$, stating the semantics of TILCO-X, is inductively defined on the structure of A by the following rules:

- $\mathcal{I}, t \models \top$;
- $\mathcal{I}, t \not\models \bot$;
- $\mathcal{I}, t \models \neg A$ iff $\mathcal{I}, t \not\models A$;
- $\mathcal{I}, t \models A_1 \wedge A_2$ iff $\mathcal{I}, t \models A_1$ and $\mathcal{I}, t \models A_2$;
- $\mathcal{I}, t \models A_1 \vee A_2$ iff either $\mathcal{I}, t \models A_1$ or $\mathcal{I}, t \models A_2$;
- $\mathcal{I}, t \models x := exp$ iff there exists a constant $k \in D_x$ such that $\mathcal{I}, t \models x = k$ and $\mathcal{I}, t-1 \models exp = k$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;
- $\mathcal{I}, t \models \forall x.A(x)$ iff, for each $y \in D_x$ it is true that $\mathcal{I}, t \models A(y)$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;
- $\mathcal{I}, t \models \exists x.A(x)$ iff, there exists a $y \in D_x$ such that $\mathcal{I}, t \models A(y)$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;
- $\mathcal{I}, t \models \exists! x.A(x)$ iff, there exists one and only one $y \in D_x$ such that $\mathcal{I}, t \models A(y)$, where D_x is the domain assigned to the type of x by $\mathcal{I}$;
- $\mathcal{I}, t \models A \,@\, I$ iff, for each $s \in [\![I]\!]_{\mathcal{I},t}$, $\mathcal{I}, s+t \models A$ is true;
- $\mathcal{I}, t \models A \,?\, I$ iff, there exists an $s \in [\![I]\!]_{\mathcal{I},t}$ such that $\mathcal{I}, s+t \models A$;
- $\mathcal{I}, t \models A \,?_m I$ iff, $m \leq \mathrm{N}_{\mathcal{I},t}([\![I]\!]_{\mathcal{I},t}, A)$;
- $\mathcal{I}, t \models A \,?^M I$ iff, $\mathrm{N}_{\mathcal{I},t}([\![I]\!]_{\mathcal{I},t}, A) \leq M$;
- $\mathcal{I}, t \models A \,?_m^M I$ iff, $\mathcal{I}, t \models (A \,?_m I) \wedge (A \,?^M I)$;
- $\mathcal{I}, t \models p_i(e_1, \ldots, e_n)$, iff $(E_1, \ldots, E_n) \in R_i$, where R_i is the relation assigned by $\mathcal{I}$ to p_i and E_j, for each $j = 1, \ldots, n$, are the results of the expressions e_j when the values assigned by $\mathcal{I}$ are substituted for the constants and variables, and the variables are evaluated in t.

The semantics of predicates also includes that of functions, variables and constants.

In TILCO-X, the definition of $[\![I]\!]_{\mathcal{I},t}$ depends on two functions: $l^+_{\mathcal{I}}(A,t)$ and $l^-_{\mathcal{I}}(A,t)$. They are used to locate the next/previous time instant, corresponding to time instant t, where a formula A is true. These functions return $+\infty/-\infty$, if such an instant does not exist. Their formal definition is:

$$l^+_{\mathcal{I}}(A,t) = \begin{cases} x & \text{if } 0 < x \text{ and } \mathcal{I}, x+t \models A \text{ and } \mathcal{I}, t \models (\neg A)\,@(0,x) \\ +\infty & \text{if } \mathcal{I}, t \models (\neg A)\,@(0,+\infty) \end{cases}$$

$$l^-_{\mathcal{I}}(A,t) = \begin{cases} -x & \text{if } 0 < x \text{ and } \mathcal{I}, -x+t \models A \text{ and } \mathcal{I}, t \models (\neg A)\,@(-x,0) \\ -\infty & \text{if } \mathcal{I}, t \models (\neg A)\,@(-\infty,0) \end{cases}$$

All the possible typologies of intervals that can be written in TILCO-X are reported in Figure 5.6, with their corresponding semantics. For example, to interval $[+A, b]$ is associated the set of integer values lower or equal to b and greater or equal to $l^+_{\mathcal{I}}(A,t)$ that represent the next time where formula A is true. If b is negative the set is empty.

$$[\![\,[a,b]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a \le x \le b\} \quad (5.1)$$
$$[\![\,[+A,b]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) \le x \le b\} \quad (5.2)$$
$$[\![\,[-A,b]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) \le x \le b\} \quad (5.3)$$
$$[\![\,[a,+B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a \le x \le l^+_{\mathcal{I}}(B,t)\} \quad (5.4)$$
$$[\![\,[a,-B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a \le x \le l^-_{\mathcal{I}}(B,t)\} \quad (5.5)$$
$$[\![\,[+A,+B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) \le x \le l^+_{\mathcal{I}}(B,t)\} \quad (5.6)$$
$$[\![\,[-A,+B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) \le x \le l^+_{\mathcal{I}}(B,t)\} \quad (5.7)$$
$$[\![\,[-A,-B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) \le x \le l^-_{\mathcal{I}}(B,t)\} \quad (5.8)$$
$$[\![\,[+A,-B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) \le x \le l^-_{\mathcal{I}}(B,t)\} \quad (5.9)$$
$$[\![\,(a,b]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a < x \le b\} \quad (5.10)$$
$$[\![\,(+A,b]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) < x \le b\} \quad (5.11)$$
$$[\![\,(-A,b]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) < x \le b\} \quad (5.12)$$
$$[\![\,(a,+B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a < x \le l^+_{\mathcal{I}}(B,t)\} \quad (5.13)$$
$$[\![\,(a,-B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a < x \le l^-_{\mathcal{I}}(B,t)\} \quad (5.14)$$
$$[\![\,(+A,+B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) < x \le l^+_{\mathcal{I}}(B,t)\} \quad (5.15)$$
$$[\![\,(-A,+B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) < x \le l^+_{\mathcal{I}}(B,t)\} \quad (5.16)$$
$$[\![\,(-A,-B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) < x \le l^-_{\mathcal{I}}(B,t)\} \quad (5.17)$$
$$[\![\,(+A,-B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) < x \le l^-_{\mathcal{I}}(B,t)\} \quad (5.18)$$
$$[\![\,[a,b)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a \le x < b\} \quad (5.19)$$
$$[\![\,[+A,b)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) \le x < b\} \quad (5.20)$$
$$[\![\,[-A,b)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) \le x < b\} \quad (5.21)$$
$$[\![\,[a,+B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a \le x < l^+_{\mathcal{I}}(B,t)\} \quad (5.22)$$
$$[\![\,[a,-B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a \le x < l^-_{\mathcal{I}}(B,t)\} \quad (5.23)$$
$$[\![\,[+A,+B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) \le x < l^+_{\mathcal{I}}(B,t)\} \quad (5.24)$$
$$[\![\,[-A,+B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) \le x < l^+_{\mathcal{I}}(B,t)\} \quad (5.25)$$
$$[\![\,[-A,-B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) \le x < l^-_{\mathcal{I}}(B,t)\} \quad (5.26)$$
$$[\![\,[+A,-B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) \le x < l^-_{\mathcal{I}}(B,t)\} \quad (5.27)$$
$$[\![\,(a,b)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a < x < b\} \quad (5.28)$$
$$[\![\,(+A,b)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) < x < b\} \quad (5.29)$$
$$[\![\,(-A,b)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) < x < b\} \quad (5.30)$$
$$[\![\,(a,+B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a < x < l^+_{\mathcal{I}}(B,t)\} \quad (5.31)$$
$$[\![\,(a,-B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a < x < l^-_{\mathcal{I}}(B,t)\} \quad (5.32)$$
$$[\![\,(+A,+B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) < x < l^+_{\mathcal{I}}(B,t)\} \quad (5.33)$$
$$[\![\,(-A,+B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) < x < l^+_{\mathcal{I}}(B,t)\} \quad (5.34)$$
$$[\![\,(-A,-B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) < x < l^-_{\mathcal{I}}(B,t)\} \quad (5.35)$$
$$[\![\,(+A,-B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) < x < l^-_{\mathcal{I}}(B,t)\} \quad (5.36)$$
$$[\![\,[a,+\infty)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a \le x\} \quad (5.37)$$
$$[\![\,[+A,+\infty)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) \le x\} \quad (5.38)$$
$$[\![\,[-A,+\infty)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) \le x\} \quad (5.39)$$
$$[\![\,(a,+\infty)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid a < x\} \quad (5.40)$$
$$[\![\,(+A,+\infty)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^+_{\mathcal{I}}(A,t) < x\} \quad (5.41)$$
$$[\![\,(-A,+\infty)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid l^-_{\mathcal{I}}(A,t) < x\} \quad (5.42)$$
$$[\![\,(-\infty,b]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid x \le b\} \quad (5.43)$$
$$[\![\,(-\infty,+B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid x \le l^+_{\mathcal{I}}(B,t)\} \quad (5.44)$$
$$[\![\,(-\infty,-B]\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid x \le l^-_{\mathcal{I}}(B,t)\} \quad (5.45)$$
$$[\![\,(-\infty,b)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid x < b\} \quad (5.46)$$
$$[\![\,(-\infty,+B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid x < l^+_{\mathcal{I}}(B,t)\} \quad (5.47)$$
$$[\![\,(-\infty,-B)\,]\!]_{\mathcal{I},t} = \{x \in \mathbb{Z} \mid x < l^-_{\mathcal{I}}(B,t)\} \quad (5.48)$$
$$[\![\,(-\infty,+\infty)\,]\!]_{\mathcal{I},t} = \mathbb{Z} \quad (5.49)$$

Figure 5.6: Definition of $[\![\,\cdot\,]\!]_{\mathcal{I},t}$

Other TILCO-X operators are treated with the following definitions:

$$
\begin{array}{rcl}
A_1 \Rightarrow A_2 & \stackrel{\text{def}}{=} & \neg A_1 \vee A_2 \\
A_1 \Leftrightarrow A_2 & \stackrel{\text{def}}{=} & A_1 \Rightarrow A_2 \wedge A_2 \Rightarrow A_1 \\
A_1 =\!\!\!\Rightarrow\!\!\!\Rightarrow A_2 & \stackrel{\text{def}}{=} & A_1 \Rightarrow A_2 \,@[1,1] \\
A_1 =\!\!\!\Leftarrow\!\!\!\Leftarrow A_2 & \stackrel{\text{def}}{=} & A_1 \Rightarrow A_2 \,@[-1,-1] \\
A \,@\, I, J & \stackrel{\text{def}}{=} & (A \,@\, I) \wedge (A \,@\, J) \\
A \,?\, I, J & \stackrel{\text{def}}{=} & (A \,?\, I) \wedge (A \,?\, J) \\
A \,@\, I; J & \stackrel{\text{def}}{=} & (A \,@\, I) \vee (A \,@\, J)
\end{array}
\qquad
\begin{array}{rcl}
A \,?\, I; J & \stackrel{\text{def}}{=} & (A \,?\, I) \vee (A \,?\, J) \\
A \,?_m I, J & \stackrel{\text{def}}{=} & (A \,?_m I) \wedge (A \,?_m J) \\
A \,?_m I; J & \stackrel{\text{def}}{=} & (A \,?_m I) \vee (A \,?_m J) \\
A \,?^M I, J & \stackrel{\text{def}}{=} & (A \,?^M I) \wedge (A \,?^M J) \\
A \,?^M I; J & \stackrel{\text{def}}{=} & (A \,?^M I) \vee (A \,?^M J) \\
A \,?^M_m I, J & \stackrel{\text{def}}{=} & (A \,?^M_m I) \wedge (A \,?^M_m J) \\
A \,?^M_m I; J & \stackrel{\text{def}}{=} & (A \,?^M_m I) \vee (A \,?^M_m J)
\end{array}
$$

Where the TILCO interval composition operators "," and ";" are extended to bounded happen in a way similar to regular happen.

In the case where the interval is empty, it holds:

$$
\begin{array}{rcl}
A \,@\, \emptyset & \equiv & \top; \\
A \,?\, \emptyset & \equiv & \bot.
\end{array}
$$

5.3 Deductive system

In this section, the deductive system used to prove properties in TILCO-X is introduced.

The FOL's deductive system in natural deduction style has been enhanced by adding rules for introduction and elimination of TILCO/TILCO-X temporal operators.

The deduction rules for basic logical operators are the following:

$$
\top I \quad \frac{}{\vdash \top} \qquad\qquad \bot E \quad \frac{\vdash \bot}{\vdash P}
$$

$$
\wedge I \quad \frac{\vdash P \quad \vdash Q}{\vdash P \wedge Q} \qquad\qquad \wedge E1 \quad \frac{\vdash P \wedge Q}{\vdash P} \qquad\qquad \wedge E2 \quad \frac{\vdash P \wedge Q}{\vdash Q}
$$

$$
\vee E \quad \frac{\vdash P \vee Q \quad P \vdash R \quad Q \vdash R}{\vdash R} \qquad\qquad \vee I1 \quad \frac{\vdash P}{\vdash P \vee Q} \qquad\qquad \vee I2 \quad \frac{\vdash Q}{\vdash P \vee Q}
$$

$$
\Rightarrow I \quad \frac{P \vdash Q}{\vdash P \Rightarrow Q} \qquad\qquad \Rightarrow E(\text{MP}) \quad \frac{\vdash P \Rightarrow Q \quad \vdash P}{\vdash Q}
$$

$$
\forall I \quad \frac{\vdash P(x)}{\vdash \forall x.P(x)} \; x \text{ free only in } P \qquad\qquad \forall E \quad \frac{\vdash \forall x.P(x)}{\vdash P[t/x]}
$$

$$
\exists I \quad \frac{\vdash P[t/x]}{\vdash \exists x.P(x)} \qquad\qquad \exists E \quad \frac{\vdash \exists x.P(x) \quad P(x) \vdash Q}{\vdash Q} \; x \text{ free only in } P
$$

The introduction and elimination rules for @ and ? are:

$$@I \quad \frac{\frac{\vdash_t x \,\mathbf{in}\, I}{\vdash_{x+t} P}}{\vdash_t P @ I} \quad x \text{ not free in any assump.} \qquad @E \quad \frac{\vdash_t P @ I \quad \vdash_t x \,\mathbf{in}\, I}{\vdash_{x+t} P}$$

$$?I \quad \frac{\vdash_{x+t} P \quad \vdash_t x \,\mathbf{in}\, I}{\vdash_t P ? I} \qquad ?E \quad \frac{\vdash_t P ? I \quad \frac{\vdash_{x+t} P \quad \vdash_t x \,\mathbf{in}\, I}{\vdash R}}{\vdash R} \quad x \text{ not free in any assump.}$$

These rules are similar to the ones provided for TILCO except that operator **in** replaces the standard $\in$ set operator. The **in** operator establishes if an integer value is in a Dynamic Interval and its evaluation depends on the evaluation time t.

The **in** operator applied to a Dynamic Interval can be defined in the following form:

$$x \,\mathbf{in}\, I \quad = \quad \begin{cases} x \succeq b1 \wedge x \preceq b2 & \text{if } I = [b1, b2] \\ (x-1) \succeq b1 \wedge (x+1) \preceq b2 & \text{if } I = (b1, b2) \\ x \succeq b1 \wedge (x+1) \preceq b2 & \text{if } I = [b1, b2) \\ (x-1) \succeq b1 \wedge x \preceq b2 & \text{if } I = (b1, b2] \\ x \succeq b \wedge x \preceq b & \text{if } I = [b] \end{cases}$$

This definition uses two new operators, $\succeq$ and $\preceq$, to check if an integer value is after or before an interval bound, where a bound can be an integer value, plus or minus infinity or a dynamic bound as $+A$ or $-A$. Since a dynamic bound depends on the evaluation instant also $\succeq$ and $\preceq$ operators depend on the evalutation instant.

These operators ($\succeq$, $\preceq$ and **in**) have been introduced to avoid the specification of two rules (introduction and elimination) for each of the 49 possible combinations of intervals, and to permit to prove generic properties about intervals.
For example:

$$12 \succeq +B$$

is true in the evalutation time instant if there exists an instant in the future where B is true, B is false upto that instant, and this instant is distant less or equal to 12 time units from the evaluation time instant.
Formula

$$8 \preceq +B$$

is true in the evaluation time instant in two cases: if B will not be true in the future, and if the first time when B will be true is after 8 time units from the current time.
These examples are reported in Figure 5.7

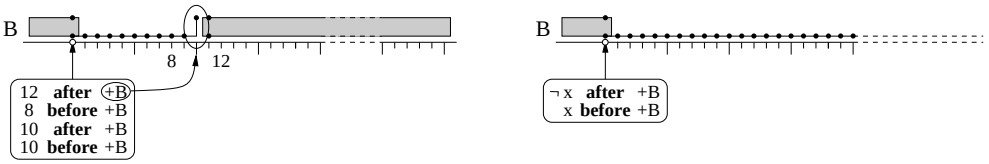

Figure 5.7: Examples of $\succeq$ (after) and $\preceq$ (before)

$\succeq$ and $\preceq$ operators are defined in the following way:

$$x \succeq b \quad = \quad \begin{cases} b \leq x & \text{if } b \text{ is an integer} \\ \bot & \text{if } b = +\infty \\ \top & \text{if } b = -\infty \\ \exists\, t.0 < t \wedge t \leq x \wedge N(t) \wedge & \\ \quad \forall\, t'.0 < t' < t \rightarrow \neg N(t') & \text{if } b = +N \\ (\exists\, t.t < 0 \wedge t \leq x \wedge P(t) \wedge & \\ \quad \forall\, t'.t < t' < 0 \rightarrow \neg P(t')) \vee & \text{if } b = -P \\ \forall\, t'.t' < 0 \rightarrow \neg P(t') & \end{cases}$$

$$x \preceq b \quad = \quad \begin{cases} x \leq b & \text{if } b \text{ is an integer} \\ \top & \text{if } b = +\infty \\ \bot & \text{if } b = -\infty \\ (\exists\, t.0 < t \wedge x \leq t \wedge N(t) \wedge & \\ \quad \forall\, t'.0 < t' < t \rightarrow \neg N(t')) \vee & \text{if } b = +N \\ \forall\, t'.0 < t' \rightarrow \neg N(t') & \\ \exists\, t.t < 0 \wedge x \leq t \wedge P(t) \wedge & \\ \quad \forall\, t'.t < t' < 0 \rightarrow \neg P(t') & \text{if } b = -P \end{cases}$$

where $P(t)$ (or $N(t)$) is true if expression P (or N) is true t time instants from the evaluation instant (in the future or in the past, it depends on the sign of t).

It should be noted that $\succeq$ and $\preceq$ operators are not strict, since the bound value is also considered to satisfy the relation (so they are an extension of $\geq$ and $\leq$).

The introduction and elimination rules for $\succeq$ operator are:

$$\succeq vI \quad \frac{\vdash v \leq x}{\vdash_t x \succeq v} \qquad \succeq vE \quad \frac{\vdash_t x \succeq v}{\vdash v \leq x}$$

$$\succeq -\infty I \quad \frac{}{\vdash_t x \succeq -\infty} \qquad \succeq +\infty E \quad \frac{\vdash_t x \succeq +\infty}{\vdash \bot}$$

$$\succeq nextI \quad \frac{\vdash_{t+x'} N \quad \vdash_t \neg N\,@(0,x') \quad \vdash 0 < x' \quad \vdash x' \leq x}{\vdash_t x \succeq +N}$$

$$\succeq nextE \quad \frac{\vdash_t x \succeq +N \quad \dfrac{\vdash_{t+x'} N \quad \vdash_t \neg N\,@(0,x') \quad \vdash 0<x' \quad \vdash x' \leq x}{\vdash R}}{\vdash R}$$

$$\succeq prevI1 \quad \frac{\vdash_{t+x'} P \quad \vdash_t \neg P\,@(x',0) \quad \vdash x' < 0 \quad \vdash x' \leq x}{\vdash_t x \succeq -P}$$

$$\succeq prevI2 \quad \frac{\vdash_t \neg P\,@(-\infty,0)}{\vdash_t x \succeq -P}$$

$$\succeq prevE \quad \frac{\vdash_t x \succeq -P \quad \dfrac{\vdash_{t+x'} P \quad \vdash_t \neg P\,@(x',0) \quad \vdash x' < 0 \quad \vdash x' \leq x}{\vdash R}}{\vdash R}$$

Introduction and/or elimination rules are reported for each kind of bound: integer value, plus and minus infinity, next ($+N$) and prev ($-P$).

Rules for $\preceq$ operator are similar to the previous:

$$\preceq vI \quad \frac{\vdash x \leq v}{\vdash_t x \preceq v} \qquad \preceq vE \quad \frac{\vdash_t x \preceq v}{\vdash x \leq v}$$

$$\preceq +\infty I \quad \frac{}{\vdash_t x \preceq +\infty} \qquad \preceq -\infty E \quad \frac{\vdash_t x \preceq -\infty}{\vdash \bot}$$

$$\preceq nextI1 \quad \frac{\vdash_{t+x'} N \quad \vdash_t \neg N\,@(0,x') \quad \vdash 0 < x' \quad \vdash x \leq x'}{\vdash_t x \preceq +N}$$

$$\preceq nextI2 \quad \frac{\vdash_t \neg N\,@(0,+\infty)}{\vdash_t x \preceq +N}$$

$$\preceq nextE \quad \frac{\vdash_t x \preceq +N \quad \frac{\vdash_{t+x'} N \quad \vdash_t \neg N\,@(0,x') \quad \vdash 0<x' \quad \vdash x \leq x'}{\vdash R}}{\vdash R}$$

$$\preceq prevI \quad \frac{\vdash_{t+x'} P \quad \vdash_t \neg P\,@(x',0) \quad \vdash x' < 0 \quad \vdash x \leq x'}{\vdash_t x \preceq -P}$$

$$\preceq prevE \quad \frac{\vdash_t x \preceq -P \quad \frac{\vdash_{t+x'} P \quad \vdash_t \neg P\,@(x',0) \quad \vdash x'<0 \quad \vdash x \leq x'}{\vdash R}}{\vdash R}$$

Introduction and elimination rules for **in** operator have been provided for each combination of interval parenthesis (open/close):

$$\mathbf{in}\,ccI \quad \frac{\vdash_t x \succeq l \quad \vdash_t x \preceq u}{\vdash_t x\,\mathbf{in}\,[l,u]} \qquad \mathbf{in}\,ccE \quad \frac{\vdash_t x\,\mathbf{in}\,[l,u] \quad \frac{\vdash_t x \succeq l \quad \vdash_t x \preceq u}{\vdash R}}{\vdash R}$$

$$\mathbf{in}\,ocI \quad \frac{\vdash_t (x-1) \succeq l \quad \vdash_t x \preceq u}{\vdash_t x\,\mathbf{in}\,(l,u]} \qquad \mathbf{in}\,ocE \quad \frac{\vdash_t x\,\mathbf{in}\,(l,u] \quad \frac{\vdash_t (x-1) \succeq l \quad \vdash_t x \preceq u}{\vdash R}}{\vdash R}$$

$$\mathbf{in}\,coI \quad \frac{\vdash_t x \succeq l \quad \vdash_t (x+1) \preceq u}{\vdash_t x\,\mathbf{in}\,[l,u)} \qquad \mathbf{in}\,coE \quad \frac{\vdash_t x\,\mathbf{in}\,[l,u) \quad \frac{\vdash_t x \succeq l \quad \vdash_t (x+1) \preceq u}{\vdash R}}{\vdash R}$$

$$\mathbf{in}\,ooI \quad \frac{\vdash_t (x-1) \succeq l \quad \vdash_t (x+1) \preceq u}{\vdash_t x\,\mathbf{in}\,(l,u)} \qquad \mathbf{in}\,ooE \quad \frac{\vdash_t x\,\mathbf{in}\,(l,u) \quad \frac{\vdash_t (x-1) \succeq l \quad \vdash_t (x+1) \preceq u}{\vdash R}}{\vdash R}$$

Bounded Happen

For bounded happen, a more complex formalization has been provided. Happen-min, $?_m$, has been defined using a list datatype, in the following way:

$$P\,?_m I = \exists f.length(f) = m \wedge (\forall i.i < m \rightarrow P(f[i]) \wedge f[i]\,\mathbf{in}\ I) \wedge (\forall 0 < j < m \rightarrow f[j-1] < f[j])$$

stating that exists a list f that enumerates m instants in I where P is true, and these instants are strictly monotone[1]. Note that the [ ] operator is used to access to the elements of the list starting from 0. Happen-max,$?^M$, has been defined, using happen-min, as:

$$P\,?^M I = \neg(P\,?_{M+1} I)$$

stating that P happens at most M times in I if P does not happen more than $M+1$ times in I.

The introduction/elimination rules of bounded happen are based on inductive properties of happen-min and happen-max:

$$m \neq 0 \Rightarrow P\,?_m[a,b] \Leftrightarrow \exists x \in [a,b].P@[x,x] \wedge (P\,?_{m-1}[x+1,b] \vee P\,?_{m-1}[a,x-1])$$

meaning that P happens at least $m > 0$ times in interval [a,b] iff exists a time instant x in the interval where P is true and P happens at least $m-1$ times before or after time instant x. Moreover property $P\,?_0 I = \top$ is used to terminate the recursion.

Similar properties hold for happen-max:

$$M \neq 0 \Rightarrow P\,?^M[a,b] \;\Leftrightarrow\; \exists x \in [a,b].P@[x,x] \;\wedge\; ((\neg P@[a,x-1] \wedge P\,?^{M-1}[x+1,b]) \vee (P\,?^{M-1}[a,x-1] \wedge \neg P@[x+1,b]))$$

In this case, the additional constraint that P must not happen before or after time instant x has been added and property $P\,?^0 I = \neg P@I$ is used to terminate recursion.

The previous properties have been presented for a constant interval while similar ones hold for any type of interval. To avoid providing specific introduction/elimination rules for each kind of interval, functions sublft() and subrgt() have been introduced. sublft() and subrgt() are functions that from an interval and a value in it give the left or right subinterval excluding the given value - e.g., $\text{sublft}([+B,10),x) = [+B,x)$ and $\text{subrgt}([+B,10),x) = (x,10)$ so $I = \text{sublft}(I,x) \cup \{x\} \cup \text{subrgt}(I,x)$ holds.

[1]The strict monotone condition is not strictly necessary. Instants have only to be different, but in that case there exists a monotone list of instants where P is true. Therefore this definition has been used to have simpler proofs.

Using the definition of happen-min and these functions the following inductive introduction/elimination rules have been proved:

$$?_0 I \qquad \frac{}{\vdash_t P\, ?_0 I}$$

$$?_{min} rgtI \qquad \frac{\vdash_{t+x} P \quad \vdash m \neq 0 \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t P\, ?_{m-1} \mathrm{subrgt}(I,x)}{\vdash_t P\, ?_m I}$$

$$?_{min} rgtE \qquad \frac{\vdash_t P\, ?_m I \quad \dfrac{\vdash_{t+x} P \quad \vdash m \neq 0 \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t P\, ?_{m-1} \mathrm{subrgt}(I,x)}{\vdash R}}{\vdash R}$$

$$?_{min} lftI \qquad \frac{\vdash_{t+x} P \quad \vdash m \neq 0 \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t P\, ?_{m-1} \mathrm{sublft}(I,x)}{\vdash_t P\, ?_m I}$$

$$?_{min} lftE \qquad \frac{\vdash_t P\, ?_m I \quad \dfrac{\vdash_{t+x} P \quad \vdash m \neq 0 \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t P\, ?_{m-1} \mathrm{sublft}(I,x)}{\vdash R}}{\vdash R}$$

Note that, there are different introduction/elimination rules for considering the interval split on the right or on the left.

Similarly for happen-max, the following rules have been derived:

$$?^{max} I1 \qquad \frac{\vdash_t \neg P \,@\, I}{\vdash_t P\, ?^M I}$$

$$?^{max} rgtI2 \qquad \frac{\vdash_{t+x} P \quad \vdash M \neq 0 \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t \neg P \,@\, \mathrm{sublft}(I,x) \quad \vdash_t P\, ?^{M-1} \mathrm{subrgt}(I,x)}{\vdash_t P\, ?^M I}$$

$$?^{max} lftI2 \qquad \frac{\vdash_{t+x} P \quad \vdash M \neq 0 \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t \neg P \,@\, \mathrm{subrgt}(I,x) \quad \vdash_t P\, ?^{M-1} \mathrm{sublft}(I,x)}{\vdash_t P\, ?^M I}$$

$$?^{max} rgtE \qquad \frac{\vdash_t P\, ?^M I \quad \frac{\vdash_t \neg P \,@\, I}{\vdash R} \quad \frac{\vdash_{t+x} P \quad \vdash M \neq 0 \quad \vdash_t \neg P \,@\, \mathrm{sublft}(I,x) \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t P\, ?^{M-1} \mathrm{subrgt}(I,x)}{\vdash R}}{\vdash R}$$

$$?^{max} lftE \qquad \frac{\vdash_t P\, ?^M I \quad \frac{\vdash_t \neg P \,@\, I}{\vdash R} \quad \frac{\vdash_{t+x} P \quad \vdash M \neq 0 \quad \vdash_t \neg P \,@\, \mathrm{subrgt}(I,x) \quad \vdash_t x \,\mathbf{in}\, I \quad \vdash_t P\, ?^{M-1} \mathrm{sublft}(I,x)}{\vdash R}}{\vdash R}$$

Properties

Using these deduction rules and the induction principle some properties have been proved:

Intervals

$$x \succeq b \Rightarrow (x+1) \succeq b$$
$$\neg(x \succeq b) \Rightarrow (x+1) \preceq b$$
$$x \,\mathbf{in}\, [b1, b3] \Rightarrow x \,\mathbf{in}\, [b1, b2] \vee x \,\mathbf{in}\, (b2, b3]$$
$$y \,\mathbf{in}\, \mathrm{sublft}(I, x) \wedge x \,\mathbf{in}\, I \Rightarrow y < x$$
$$y \,\mathbf{in}\, \mathrm{subrgt}(I, x) \wedge x \,\mathbf{in}\, I \Rightarrow x \,\mathbf{in}\, \mathrm{sublft}(I, y)$$
$$x \,\mathbf{in}\, I \wedge y \,\mathbf{in}\, \mathrm{sublft}(I, x) \Rightarrow \mathrm{sublft}(\mathrm{sublft}(I, x), y) = \mathrm{sublft}(I, y)$$

TILCO-X

The following properties hold for happen-min:

$$\begin{array}{rcll}
A\,?_1\,I & \Leftrightarrow & A\,?\,I & \\
A\,?_m\,I & \Rightarrow & A\,?_{m-1}\,I & \text{if } m > 0 \\
A\,?_m\,[a,b] & \Leftrightarrow & \bot & \text{if } m > b - a + 1 \\
A \wedge A\,?_m\,[0,b] & \Rightarrow\!\!\!\Rightarrow & A\,?_{m-1}\,[0, b-1] & \text{if } m > 0 \\
\neg A \wedge A\,?_m\,[0,b] & \Rightarrow\!\!\!\Rightarrow & A\,?_m\,[0, b-1] & \\
A\,?_m\,[a,0] \wedge A\,@[1,1] & \Rightarrow\!\!\!\Rightarrow & A\,?_{m+1}\,[a-1, 0] & \\
A\,?_m\,[a,0] \wedge \neg A\,@[1,1] & \Rightarrow\!\!\!\Rightarrow & A\,?_m\,[a-1, 0] & \\
\forall m.\, A\,?_m[0,+\infty) & \Leftrightarrow & (A?(0,+\infty))@[0,+\infty) & \\
A@[0,m) & \Rightarrow & A\,?_m[0,m) &
\end{array}$$

and the following properties hold for happen-max:

$$\begin{array}{rcll}
A\,?^0\,I & \Leftrightarrow & \neg A\,@\,I & \\
A\,?^M\,I & \Rightarrow & A\,?^{M+1}\,I & \\
A\,?^M\,[a,b] & \Leftrightarrow & \top & \text{if } M \geq b - a + 1 \\
A \wedge A\,?^M\,[0,b] & \Rightarrow\!\!\!\Rightarrow & A\,?^{M-1}\,[0, b-1] & \text{if } M > 0 \\
\neg A \wedge A\,?^M\,[0,b] & \Rightarrow\!\!\!\Rightarrow & A\,?^M\,[0, b-1] & \\
A\,?^M\,[a,0] \wedge A\,@[1,1] & \Rightarrow\!\!\!\Rightarrow & A\,?^{M+1}\,[a-1, 0] & \\
A\,?^M\,[a,0] \wedge \neg A\,@[1,1] & \Rightarrow\!\!\!\Rightarrow & A\,?^M\,[a-1, 0] & \\
A\,?^{M+1}[0,M] & \Leftrightarrow & \top &
\end{array}$$

and

$$A\,?_{min}^{max}\,I \Leftrightarrow \bot \qquad \text{if } min > max$$

5.4 Specification example

In this section, a specification example to highlight the use of TILCO-X is presented. It is considered a system where a process has to respond to an external stimulus within 100ms. If the process does not respond within the given time, a controller has to retry up to 3 more times. If after all the temptatives the process does not respond the operation is aborted. After an abort the process cannot be started again for 500ms and an eventual request has to be ignored. If three consecutive operations are aborted the system is blocked until system reset by the user.

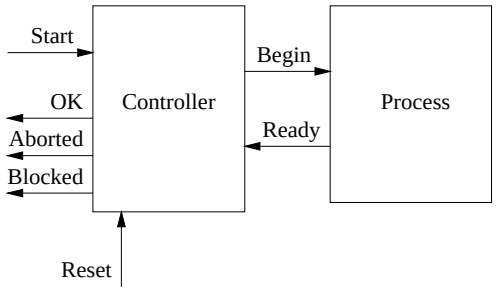

Figure 5.8: An example of a process control system

In Figure 5.8 the system structure is reported. *Process* is the process under control that has to respond to the *Begin* signal with a Ready signal being true within 100ms. The *Controller* is specified by using TILCO-X with the formulæ reported in the following. An internal signal *Enabled* is used to state that the Controller is enabled to consider the *Start* signal externally issued.

The response of the Controller to the *Start* signal is specified with the formulæ:

$$\begin{aligned} Start \wedge \neg Blocked \wedge Enabled &\Rightarrow Begin \\ Start \wedge \neg Enabled &\Rightarrow (\neg Begin \wedge \neg Aborted \wedge \neg Ok)\,@[0, +Start) \\ Start \wedge Blocked &\Rightarrow (\neg Begin \wedge \neg Aborted \wedge \neg Ok)\,@[0, +Reset) \end{aligned}$$

The first condition states that if the *Start* signal is true, the system is not Blocked and is Enabled the signal *Begin* is asserted. The second formula specifies the behavior of the

system if it is not Enabled. In this case signals *Begin*, *Aborted* and *Ok* are false until the next Start. The last formula specifies the response to the *Start* signal if the system is Blocked. In this case signals *Begin*, *Aborted* and *Ok* are false until the next *Reset*.

When signal *Begin* is true, then the system has to wait for *Ready* within 100ms. If it happens signal Ok is true. This behavior can be specified with:

$$\begin{aligned} Begin \wedge Ready?(0,100] &\Rightarrow (\neg Begin \wedge \neg Aborted \wedge \neg Enabled \wedge (Ready \Leftrightarrow Ok))\,@(0,+Ready] \\ Begin \wedge \neg Ready@(0,100] &\Rightarrow (\neg Begin \wedge \neg Aborted \wedge \neg Ok \wedge \neg Enabled)\,@(0,100) \\ Begin &\Rightarrow \neg Aborted \wedge \neg Ok \end{aligned}$$

The condition on the repetition of the *Begin* signal is specified using Bounded Happen:

$$\begin{aligned} Begin\,@[-100] \wedge \neg Ready\,@(-100,0] \wedge Begin\,?^{3}[-(Start \wedge Enabled),0) &\Rightarrow Begin \wedge \neg Enabled \\ Begin\,@[-100] \wedge \neg Ready@(-100,0] \wedge Begin\,?_{4}[-(Start \wedge Enabled),0) &\Rightarrow Aborted \wedge \neg Ok \end{aligned}$$

The first formula specifies that, if the last *Begin* has failed and the *Begin* has been issued up to 3 times, since the last enabled Start, then the *Begin* has to be retried. The second formula specifies the case in which *Begin* has been retried more than 3 times, and in this case, the signal *Aborted* is asserted.

The behavior of the *Enabled* signal is specified with formulæ:

$$\begin{aligned} Aborted &\Rightarrow \neg Begin \wedge \neg Enabled\,@[0,500) \wedge (Enabled\,@[0,+Start])@[500] \\ Ok \vee (Reset \wedge Enabled) &\Rightarrow Enabled\,@(0,+Start] \end{aligned}$$

The first formula specifies that, when *Aborted* is true the system is not enabled to satisfy a Start request for 500ms and after this period the system is enabled until a Start is received. Similarly, when Ok or Reset is true the system is enabled until the next Start.

The system is Blocked if and only if *Aborted* is true more than 3 times since last *Ok* or last *Reset*, in formula:

$$Blocked \Leftrightarrow Aborted\,?_{3}(-(Ok \vee Reset),0)$$

After *Ok* or *Aborted* are true or after a *Reset* signals *Begin*, *Aborted* and *Ok* are false:

$$\begin{aligned} Ok \vee Aborted &\Rightarrow (\neg Begin \wedge \neg Aborted \wedge \neg Ok)\,@(0,+(Start \wedge Enabled)) \\ Reset \wedge Enabled &\Rightarrow (\neg Begin \wedge \neg Aborted \wedge \neg Ok)\,@[0,+Start) \end{aligned}$$

Validation

This specification has been firstly validated by using the TILCO-X executor (presented in the next chapter). In Figure 5.9 temporal traces of the system execution are reported. A *Reset* at time 0 has been issued for the proper initialization, after that, signal *Start* is asserted and since the *Ready* signal is false the signal Begin is issued four times and then the signal *Aborted* is trued. The signal Start is asserted other two times and since the Process does not response the operations are aborted. The failure of three operations brings up the *Blocked* signal. The *Reset* is issued and enables the system to the receipt for the *Start* signal. The *Start* is issued again and at the second temptative the *Ready* signal is received and *Ok* asserted.

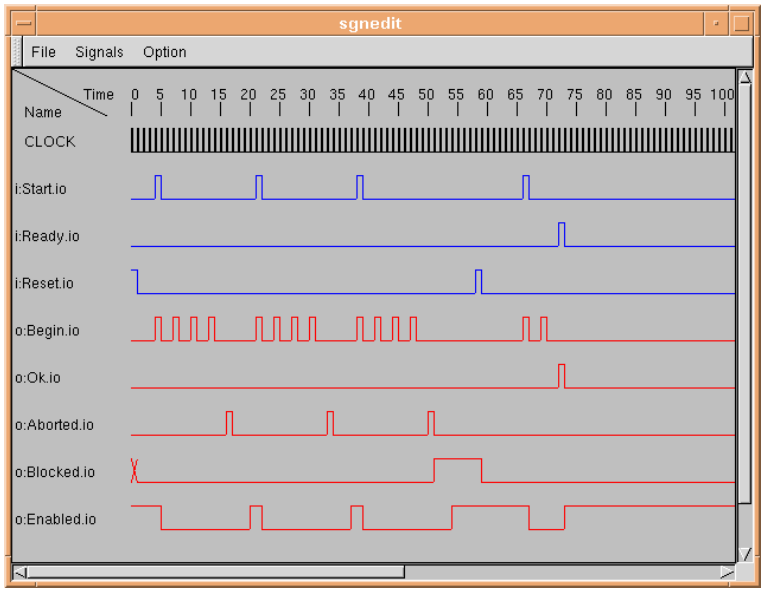

Figure 5.9: Execution of the TILCO-X specification

Moreover, some properties has been proved using the TILCO-X theory within Isabelle:

$$Start \wedge Enabled \Rightarrow \neg Enabled\,@(0, +(Ok \vee Aborted)]$$
$$Start \wedge Enabled \Rightarrow Enabled\,?(0, +\infty)$$
$$Start \wedge Enabled \Rightarrow (Ok \vee Aborted)\,?(0, +\infty)$$
$$Start \wedge Enabled \Rightarrow Ok\,?(0, +500) \vee Aborted\,@[500]$$

The first property states that the controller is not enabled until the termination (with

success or failure); the second property states that, if the process is started there will be a future instant in which it will be enabled to be started again; the third property states that, if the process is started it will terminate with success or failure; the last property gives more details on when signals Ok and Aborted will be true.

Chapter 6

Executing TILCO

In this chapter a new way of executing temporal logics is presented. In order to simplify the execution of TILCO specifications a basic temporal logic (BTL) with lower number of operators has been introduced, this is capable of expressing all complex operators of TILCO. In the following it is briefly described how to translate the propositional part of TILCO and TILCO-X to BTL. Inference rules for BTL are presented and it is described how to graphically represent the BTL specifications as Temporal Inference Networks. An algorithm for executing temporal inference network is presented and the possibility of real-time execution is highlighted.

6.1 Execution of temporal logics

The validation of a logic specification usually consists in proving high-level properties, which are also given in the form of logical expressions, by means of theorem provers [98]. Other techniques are based on the so-called history-checking which is a restricted version of the model-checking technique, which is typical of operational approaches [57]. In these cases, the time ordering is not satisfied, history-checking is mainly oriented to validating specifications against off-line generated histories of system inputs and outputs.

The problem of executability of specifications given by means of temporal logics has often been misunderstood. This mainly depends on the meaning assigned to executability [59], [60], [110], [18]. There are at least three different definitions of executability, as follows.

(i) In many cases, specification models are considered as executable if they have a semantics defining an effective procedure capable of determining for any formula of the logic theory, whether or not that formula is a *theorem* of the theory [110]. In effect, this

property corresponds to that of decidability of the validity problem rather than to that of system specification executability.

(ii) Another meaning for the property of executability refers to the possibility of generating a model for a given specification [56]. A detailed version of this concept leads to verify if an *off-line* generated temporal evolution of inputs and outputs is compatible with the specification. This operation is usually called history checking.

(iii) The last meaning for executability consists in using the system specification itself as a prototype or implementation of the real-time system, thus allowing, in each time instant, the *on-line* generation of system outputs on the basis of present inputs and its internal state and past history. When this is possible, the specification can be directly executed instead of traducing it in a programming language.

In the literature, there are only few executable temporal logics which can be used to build a system prototype according to the *(iii)* meaning of executability. In general, the execution or simulation of logic specifications with the intent of producing system outputs in the correct time order by meeting the temporal constraints is a quite difficult problem. The difficulty mainly depends on the computational complexity of the algorithms proposed in the literature which makes their adoption for executing logic specifications in real-time impossible. In many cases, the computational complexity depends on language semantics and on the specification itself. Usually, the computational complexity is $O(D^h)$ where D is the domain size of the nested quantifications (for both time dependent and independent variables), and h is number of these quantifications.

A temporal logic specification is generally composed of:

1. a set of *input signals*, that represent information acquired from outside (i.e., water level, temperature, etc.);

2. a set of *output signals*, that is the information produced to outside (i.e., alarm to be switched on/off, pump to be switched on, etc.);

3. a set of *internal signals/variables*, that represents information produced internally;

4. a *logic formula* that represents the temporal behavior of the specified system (i.e., if the water level is over the alarm level for three time instants then switchs on the alarm);

To execute such a specification means, given the facts acquired from the outside (input signals) and the specification, to deduce new facts to be sent outside (output signals). Or in a logic view, it means, given interpretation $\mathcal{I}_i$ of input signals, to find the interpretations $\mathcal{I}_o$,$\mathcal{I}_{int}$ of outputs and internal signals that satisfy the specification formula S:

$$\mathcal{I}_i \cup \mathcal{I}_{int} \cup \mathcal{I}_o \models S$$

For example in a control system to monitor the level of a reservoir, the input signal is

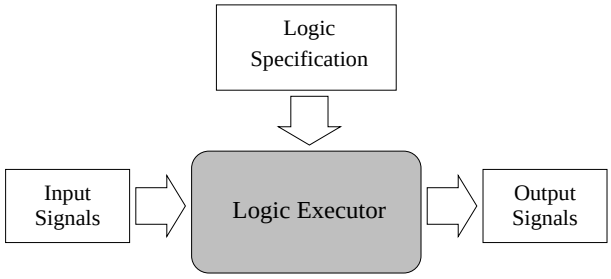

Figure 6.1: Execution of a logic specification

the water level ($wlevel$) and the output signal is the alarm. The system behavior can be specified with the following TILCO formula:

$$up(wlevel > 150) \longrightarrow alarm@[1, 100] \wedge \textbf{until}\ \ wlevel < 150\ \ alarm$$

meaning that if the water level becomes grater than 150 the alarm is switched on for at least 100 time units and until the level becomes lower than 150. To execute this specification means to deduce the history of signal $alarm$ knowing the history of signal $wlevel$, as depicted in Figure 6.2. As can be seen in this example, specification can be partial, in fact nothing is said about the falseness of the alarm. For the instants in which the water level is lower than 150 nothing can be derived from the specification, so the logical value of alarm in such instants is unknown.

The executor can be deterministic or non deterministic, in the sense that it can execute

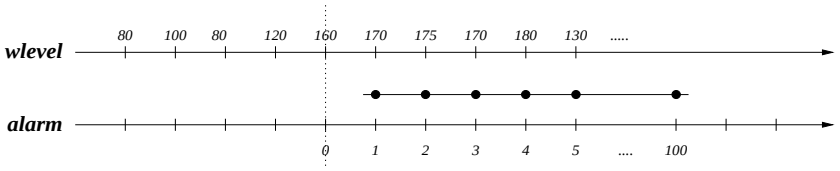

Figure 6.2: Example of execution

even a non deterministic specification. For example, the formula:

$$up(alarm)?(-200, 0) \wedge \textbf{since}\ \ up(alarm)\ \ (\neg reset \wedge \neg pump) \longrightarrow up(pump)?(0, 10]$$

means that if the alarm is switched on in the last 200 time units and since this time instant the reset button has not been pressed and the pump is off; then within 10 time units the pump has to be switched on. This specification cannot be executed deterministically, in fact, if the condition is satisfied, it is not specified in which instant of the interval the pump is turned on.

6.2 Basic Temporal Logic

Basic Temporal Logic (BTL) is a propositional 3-value logic, with And ($\wedge$), Or ($\vee$) and with the Delay (∂) operator, where signals associate a true/false/unknown value for each time instant. The BTL syntax is the following:

$$\begin{aligned} formula \ &:= \ signal \\ &\mid \ \neg signal \\ &\mid \ formula \wedge formula \\ &\mid \ formula \vee formula \\ &\mid \ \partial\, formula \\ &\mid \ \partial^k formula \\ &\mid \ (formula) \end{aligned}$$

As can be noted from the syntax the $\neg$ operator can be applied only to signals and not to complex formulæ.

A BTL formula can be evaluated at an instant t with @ operator

- $(F@t) = \top$ means that formula F is true at time t;
- $(F@t) = \bot$ means that formula F is false at time t;
- $(F@t) =?$ means that formula F is unknown at time t;

This operator is inductively defined in the following manner:

$s@t$	is the value of signal s at instant t,
$(\neg s)@t$	is $\top$ if $s@t = \bot$, is $\bot$ if $s@t = \top$, is ? otherwise
$(A \wedge B)@t$	is $\top$ if $A@t = \top$ and $B@t = \top$, is $\bot$ if $A@t = \bot$ or $B@t = \bot$, is ? otherwise
$(A \vee B)@t$	is $\top$ if $A@t = \top$ or $B@t = \top$, is $\bot$ if $A@t = \bot$ and $B@t = \bot$, is ? otherwise
$(\partial\, A)@t$	is the value of $(\partial^1\, A)@t$
$(\partial^k\, A)@t$	is $\top$ if $A@(t-k) = \top$, is $\bot$ if $A@(t-k) = \bot$, is ? otherwise

A BTL specification is a formula always true, that is written as $\bowtie S$. For example, if a and b are signals then

$$\bowtie \partial(\neg a \vee \neg b) \vee b$$

that means: if a and b were true at the previous time instant then b is true at the current time instant.

6.2.1 TILCO in BTL

TILCO temporal operators can be translated to BTL. The TILCO @ operator can be translated by using the following formula:

$$A@[m, M] = \partial^{-m}\, A \wedge \partial^{-(m+1)}\, A \wedge \ldots \wedge \partial^{-M}\, A = \bigwedge_{i=m}^{M} \partial^{-i}\, A$$

Since $A@[m, M]$ is true if A is true in all time interval instants.
Similarly, TILCO ? operator is translated with:

$$A?[m, M] = \partial^{-m}\, A \vee \partial^{-(m+1)}\, A \vee \ldots \vee \partial^{-M}\, A = \bigvee_{i=m}^{M} \partial^{-i}\, A$$

Since $A?[m, M]$ is true if A is true in one time interval instant.
The until operator is translated using the following property:

$$\mathbf{until}(A, B) \Rightarrow A \vee (B \wedge \partial^{-1}\, \mathbf{until}(A, B))$$

and a similar one for since operator:

$$\mathbf{since}(A, B) \Rightarrow A \vee (B \wedge \partial^{1}\, \mathbf{since}(A, B))$$

In order to translate a propositional TILCO formula to BTL some setps are needed. As a first step, the $\Rightarrow$ and $\Leftrightarrow$ operators are translated by using the following rewriting rules:

$$\begin{aligned} A \Rightarrow B &\mapsto \neg A \vee B \\ A \Leftrightarrow B &\mapsto (A \Rightarrow B) \wedge (B \Rightarrow A) \end{aligned}$$

after that, the $\neg$ operator has to be propagated down to the signals using the following rewriting rules:

$$\begin{aligned}
\neg(A \wedge B) &\mapsto \neg A \vee \neg B \\
\neg(A \vee B) &\mapsto \neg A \wedge \neg B \\
\neg(\neg A) &\mapsto A \\
\neg(A?I) &\mapsto \neg A@I \\
\neg(A@I) &\mapsto \neg A?I
\end{aligned}$$

Since there is no rule to propagate $\neg$ operator for until and since operators, the $\neg\,\mathbf{until}$ and $\neg\,\mathbf{since}$ operators are translated by using the following properties:

$$\begin{aligned}
\neg\,\mathbf{until}(\neg A, \neg B) &\Rightarrow A \wedge (B \vee \partial^{-1}\,\neg\,\mathbf{until}(\neg A, \neg B)) \\
\neg\,\mathbf{since}(\neg A, \neg B) &\Rightarrow A \wedge (B \vee \partial^{1}\,\neg\,\mathbf{since}(\neg A, \neg B))
\end{aligned}$$

Moreover, since the $\neg\,\mathbf{until}$ and $\neg\,\mathbf{since}$ operators generally are not in the form needed to use these properties, the following rewite rules can be used:

$$\begin{aligned}
\neg\,\mathbf{until}(A, B) &\mapsto \neg\,\mathbf{until}(\neg(\neg A), \neg(\neg B)) \\
\neg\,\mathbf{until}(\neg A, B) &\mapsto \neg\,\mathbf{until}(\neg A, \neg(\neg B)) \\
\neg\,\mathbf{until}(A, \neg B) &\mapsto \neg\,\mathbf{until}(\neg(\neg A), \neg B) \\
\neg\,\mathbf{since}(A, B) &\mapsto \neg\,\mathbf{since}(\neg(\neg A), \neg(\neg B)) \\
\neg\,\mathbf{since}(\neg A, B) &\mapsto \neg\,\mathbf{since}(\neg A, \neg(\neg B)) \\
\neg\,\mathbf{since}(A, \neg B) &\mapsto \neg\,\mathbf{since}(\neg(\neg A), \neg B)
\end{aligned}$$

For example, TILCO formula:

$$A?(-3, 0) \wedge \mathbf{since}(A, \neg B) \Rightarrow \mathbf{until}(A, B)$$

after $\Rightarrow$ substitution and $\neg$ propagation becomes:

$$\neg A@(-3, 0) \vee \neg\,\mathbf{since}(\neg(\neg A), \neg B) \vee \mathbf{until}(A, B)$$

then the substitution of the temporal operators $(@, \neg\,\mathbf{since}, \mathbf{until})$ is made, leading to the BTL formula:

$$\begin{aligned}
&((\partial^{-2}\,\neg A \wedge \partial^{-1}\,\neg A) \vee nsince \vee until) \wedge \\
&(\neg nsince \vee (\neg A \wedge (B \vee \partial\, nsince))) \wedge \\
&(\neg until \vee A \vee (B \wedge \partial^{-1}\, until))
\end{aligned}$$

where signals *nsince* and *until* have been introduced to represent the $\neg$ **since** and **until** formulæ.

6.2.2 TILCO-X in BTL

The translation of TILCO-X formulæ to BTL is a complex task, since the translations of at, happen and bounded happen operators have to be made for all the possible combinations of dynamic intervals (see section 5.2). Generally, TILCO-X formulæ are translated in TILCO by using at, happen (on constant intervals), until and since and then translated to BTL. For example formula:

$$E \mathbin{@} [-A, b] \qquad (b >= 0)$$

is substitututed with a signal U and the following condition is added:

$$U \Rightarrow \partial \neg \textbf{since}\ (\neg E, \neg A) \wedge E \mathbin{@} [0, b]$$

While formula:

$$E \mathbin{@} [+A, +B)$$

is substituted with a signal U' and the following condition is added to the specification:

$$U' \wedge \partial^{-1} \neg \textbf{until}\ (B, \neg A) \Rightarrow \partial^{-1} \neg \textbf{until}\ (\neg(A \Rightarrow \textbf{until}\ (B, E)), \neg A)$$

Similar translations are used for the happen operator over dynamic intervals.

For bounded happen the conditions are more complex, for example formula:

$$E \mathbin{?_m} (-A, b] \qquad (b < 0)$$

is substituted with signal U'' and the following conditions are added:

$$\begin{aligned}
U'' &\Rightarrow \neg A \mathbin{@} [b - m + 1, -1] \wedge \partial^{-b} E_m \\
E_m &\Rightarrow \neg A \mathbin{@} [-m + 1, 0] \wedge (\neg E \Rightarrow \partial^1 E_m) \wedge (E \Rightarrow \partial^1 E_{m-1}) \wedge E_{m-1} \\
E_{m-1} &\Rightarrow \neg A \mathbin{@} [-m + 2, 0] \wedge (\neg E \Rightarrow \partial^1 E_{m-1}) \wedge (E \Rightarrow \partial^1 E_{m-2}) \wedge E_{m-2} \\
&\dots \\
E_2 &\Rightarrow \neg A \mathbin{@} [-1, 0] \wedge (\neg E \Rightarrow \partial^1 E_2) \wedge (E \Rightarrow \partial^1 E_1) \wedge E_1 \\
E_1 &\Rightarrow \neg A \wedge \neg \textbf{since}\ (A, \neg E)
\end{aligned}$$

where signal E_i indicate that i occurrences of E happened from the last occurrence of A.

6.3 Temporal inference with BTL

An inference process produces new information applying inference rules to known facts. These rules generally are composed of an antecedent that represents the condition to be satisfied, and the consequent that represents the new information inferred.
For example

$$(A \wedge B)@t \vdash A@t, B@t$$

is an inference rule that indicates that if $A \wedge B$ is true at time t then A and B are true at time t. The inference rules of BTL are the following:

$$\begin{array}{rcll}
(A \wedge B)@t & \vdash & A@t, B@t & \downarrow \top \\
\neg(A \wedge B)@t, A@t & \vdash & \neg B@t & \downarrow \bot \\
\neg(A \wedge B)@t, B@t & \vdash & \neg A@t & \downarrow \bot \\
\neg A@t & \vdash & \neg(A \wedge B)@t & \uparrow \bot \\
\neg B@t & \vdash & \neg(A \wedge B)@t & \uparrow \bot \\
A@t, B@t & \vdash & (A \wedge B)@t & \uparrow \top \\
 & & & \\
(A \vee B)@t, \neg A@t & \vdash & B@t & \downarrow \top \\
(A \vee B)@t, \neg B@t & \vdash & A@t & \downarrow \top \\
\neg A@t, \neg B@t & \vdash & \neg(A \vee B)@t & \uparrow \bot \\
\neg(A \vee B)@t & \vdash & \neg A@t, \neg B@t & \downarrow \bot \\
A@t & \vdash & (A \vee B)@t & \uparrow \top \\
B@t & \vdash & (A \vee B)@t & \uparrow \top \\
 & & & \\
(\partial^k A)@t & \vdash & A@(t-k) & \downarrow \top \\
\neg(\partial^k A)@t & \vdash & \neg A@(t-k) & \downarrow \bot \\
A@t & \vdash & (\partial^k A)@(t+k) & \uparrow \top \\
\neg A@t & \vdash & \neg(\partial^k A)@(t+k) & \uparrow \bot \\
 & & & \\
l@t & \vdash & \neg(\neg l)@t & \rightarrow \bot \\
\neg(l)@t & \vdash & (\neg l)@t & \rightarrow \top \\
(\neg l)@t & \vdash & \neg(l)@t & \rightarrow \bot \\
\neg(\neg l)@t & \vdash & l@t & \rightarrow \top
\end{array}$$

This set of rules have been classified in the following types of rules:

$\downarrow \top$ rules that propagate the truth from a parent formula to a child formula;

$\uparrow \top$ rules that propagate the truth from a child formula to a parent formula;

$\downarrow \bot$ rules that propagate the falseness from a parent formula to a child formula;

$\uparrow \bot$ rules that propagate the falseness from a child formula to a parent formula;

$\rightarrow \top$ rules that propagate the falsity of a signal to truth of the complementary signal;

$\rightarrow \bot$ rules that propagate the truth of a signal to falsity of the complementary signal;

This set of inference rules is correct but not complete, in the sense that not all the true statements can be inferred. For example, from $(A \vee A)@t$ we cannot deduce that $A@t$ is true using the presented inference rules.

It should be noted that for all operators, after applying a $\uparrow$ rule a $\downarrow$ rule cannot produce new information. For example if $A@t$ it true then rule $A@t \vdash (A \vee B)@t$ ($\uparrow \top$ rule) can be applied and if $\neg B@t$ is true then from $(A \vee B)@t, \neg B@t \vdash A@t$ ($\downarrow \top$ rule) can be found that $A@t$ is true, but it was already known. Moreover if we consider all the $\bot$ expressions at a given time instant these were not found using $\downarrow \bot$ rules. This is true if we consider that all the apriori information is known true (the specification). For this reason this kind of rules can be dropped. Also the $\uparrow \top$ rules are not needed since to their result cannot be applied a $\uparrow \bot$ rule (the rule produces a $\top$ while this rule needs a $\bot$) nor a $\downarrow \top$ rule (no new information can be inferred), only another $\uparrow \top$ rule can be applied but this rules will never produce new knowledge about a signal. For this reason also the $\uparrow \top$ rules have been dropped. As a result also the $\rightarrow \top$ rules have been dropped since their result could be used only with $\uparrow \top$ rules that are anymore present.

Therefore, the final set of inference rules for BTL is:

$$
\begin{array}{rcll}
(A \wedge B)@t & \vdash & A@t, B@t & \downarrow \top \\
\neg A@t & \vdash & \neg(A \wedge B)@t & \uparrow \bot \\
\neg B@t & \vdash & \neg(A \wedge B)@t & \uparrow \bot \\
 & & & \\
(A \vee B)@t, \neg A@t & \vdash & B@t & \downarrow \top \\
(A \vee B)@t, \neg B@t & \vdash & A@t & \downarrow \top \\
\neg A@t, \neg B@t & \vdash & \neg(A \vee B)@t & \uparrow \bot \\
 & & & \\
(\partial^k A)@t & \vdash & A@(t-k) & \downarrow \top \\
\neg A@t & \vdash & \neg(\partial^k A)@(t+k) & \uparrow \bot \\
 & & & \\
l@t & \vdash & \neg(\neg l)@t & \rightarrow \bot \\
(\neg l)@t & \vdash & \neg(l)@t & \rightarrow \bot
\end{array}
$$

6.3.1 A graphical view of temporal inference

Given a BTL specification a graphical view of the formula can be built from the syntax tree of the formula. For each operator a component presented in Figure 6.3 can be used.

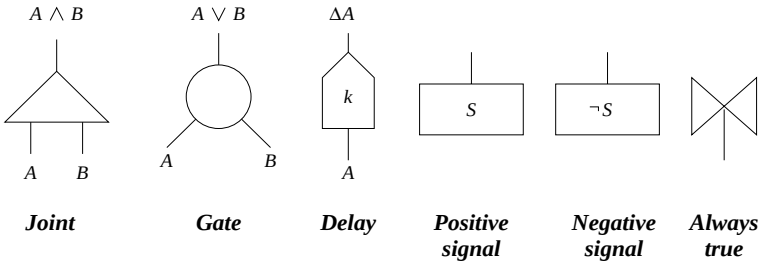

Figure 6.3: The basic components

For example formula:

$$\bowtie(\partial\, \neg s \wedge s \Rightarrow up)$$

states that signal up is true if at the previous time instant signal s was false and now s is true. This formula is not written in BTL since there is the imply ($\Rightarrow$) operator, but using the rewriting rules: $A \Rightarrow B \equiv \neg A \vee B$, $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$,$\neg(A \vee B) \equiv (\neg A \wedge \neg B)$, $\neg\neg A \equiv A$ and $\neg\, \partial^k\, A \equiv \partial^k\, \neg A$ it can be translated to BTL. Then it is obtained:

$$\bowtie(\partial\, s \vee \neg s \vee up)$$

The syntax tree of this formula is depicted in Figure 6.4.

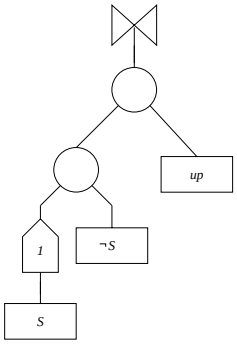

Figure 6.4: The basic components

A direction can be associated with the arcs connecting a parent to a sub-tree, in the following way:

- an arrow from the parent to the sub-tree if the sub-expression is true;
- an arrow from the sub-tree to the parent if the sub-expression is false.

This association depends on the evaluation time instant, thus for each time instant different arrow directions may exist. For example, in the up example, if S if known false at time

t there is an arrow from the S signal to the delay component, moreover since the always operator means that the expression is always true there is also an arrow from the always component to the gate as depicted in Figure 6.5. With the association of true/false with

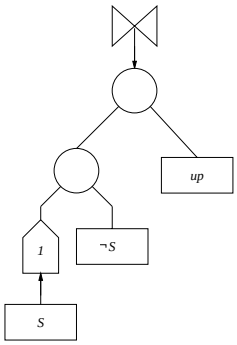

Figure 6.5: The up example, arrow configuration

the arc direction, the inference rules reported in the previous section can be represented in a graphical manner, see Figure 6.6 where the black arrows represent the antecedents of the rules and the white arrows the consequences.

Returning to the *up* example, applying rule (D2) to the configuration presented, an arrow from the delay to the parent gate can be drawn at the following time instant. If at this time instant signal S is true, an arrow from the $\neg S$ component to its parent gate can be added. At this point, rule (G3) can be applied to draw an arrow from this gate to the upper one. Since the always component imposes an arrow from it to the upper gate then rule (G1) can be applied to draw an arrow from the gate to the *up* signal that can be interpreted as *up* being true. This is correct since S is true and it was false at the previous time instant. This deduction sequence is depicted in Figure 6.7. The inference rules can be applied also to the cases in which signal *up* is known false and the signal S is known true, in fact using these rules can be derived that S has to be true at the previous time instant.

6.4 Temporal inference networks

A temporal inference network, is built from the syntax tree of a specification appling the simplifications reported in Figure 6.8 to eliminate the $\bowtie$ component. The simplification (1a) is based on the following property:

$$\forall t.(A \wedge B)@t \Longleftrightarrow (\forall t.A@t) \wedge (\forall t.B@t),$$

That permits the propagation of $\bowtie$ to the lower levels.

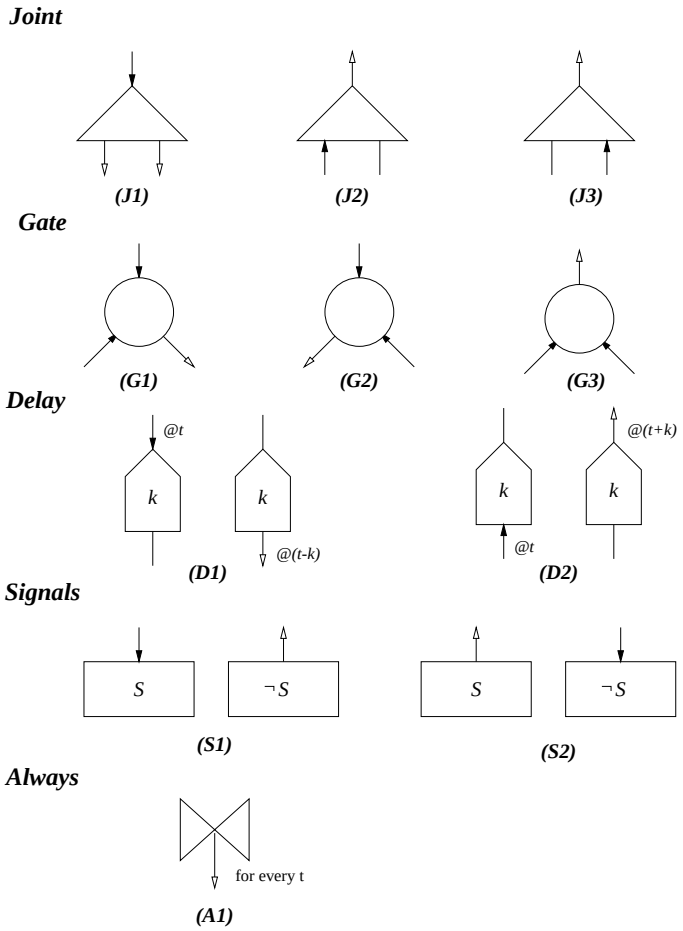

Figure 6.6: Graphical inference rules

Simplification (1b) is based on property:

$$\forall t.(\partial^k A)@t \iff (\forall t.A@(t-k)) \iff (\forall t.A@t),$$

which also permits the propagation of $\bowtie$.

Simplification (1c) has been derived by considering the two $\downarrow \top$ inference rules of $\vee$ operator:

$$\begin{aligned}(A \vee B)@t, \neg A@t &\vdash B@t \\ (A \vee B)@t, \neg B@t &\vdash A@t\end{aligned}$$

if $(A \vee B)@t$ is true for all t, if sub-expression A is false then B is true, it means that if there is an arrow from A to the gate then there is an arrow from the gate to B, the same in the

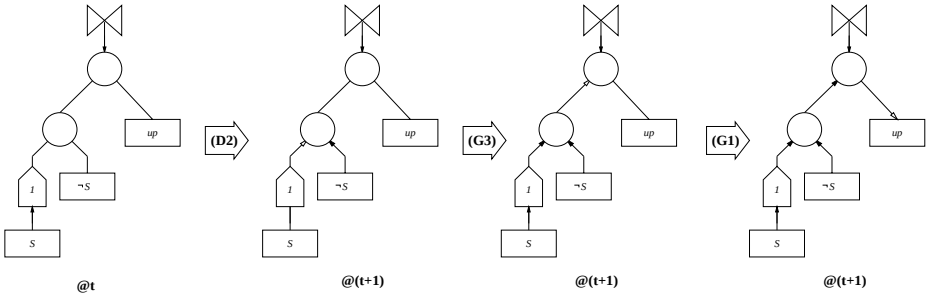

Figure 6.7: The up example execution

reverse order (from B to A) as reported in Figure 6.6. The same behavior is accomplished connecting directly sub-tree A to sub-tree B.

Other substitutions to eliminate also signals components have been found. These are reported in Figure 6.9. Simplification (2a) permits to substitute two signal nodes of the same type (same signal and both positive or negative) with one signal node, the behavior is the same. If the signal (positive or negative) is false, the arrow is from the signal node to the joint node, the application of rule (J1) permits to derive the correct directions of arrows for the connections of the two signals. And, if a signal (positive or negative) is derived as true from the upper levels, using rule (J2) or (J3) the signal node is derived as true. Using this substitution rule, all the signals of the same type can be replaced with one signal node.

Simplification (2b) permits to substitute two complementary signal nodes (one positive and one negative) with a wire connecting its parents. In fact, if the signal is known true, an arrow from the positive to the negative can be drawn (since the negative signal is false); in the other direction if it is known false (since the positive signal is false). Moreover, if the signal is derived as true then an arrow from the positive parent to the negative one can be drawn, in the opposite sense if the signal is derived as false. In this way a signal is represented in the network by an arc labeled with the signal name and with an arrow in the middle meaning the direction to be given when the signal is true.

If for a signal s there exist only positive or only negative nodes then tautology $(s \vee \neg s)$ can be added to the specification to allow the substitution for this signal.

Using these simplifications the resulting graph has only the three basis types of nodes (joint, gate and delay), and it can be executed by using the defined inference rules.

The Temporal inference network of the *up* example can be derived in the following way: Have to be noted that the signal *up* is present only as positive, for this reason the tautology $up \vee \neg up$ has to be added to the specification. The BTL specification is:

$$\bowtie((\partial\, s \vee \neg s \vee up) \wedge (up \vee \neg up))$$

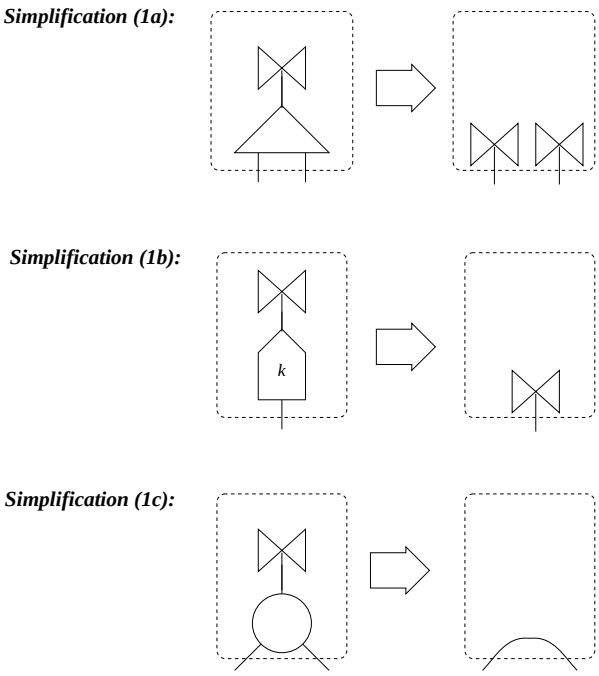

Figure 6.8: Simplifications to eliminate $\bowtie$

The syntax tree of this formula is reported in Figure 6.10 (the up left) with the application of the simplifications. Simplifications (1a) and (1c) can be applied to eliminate the $\bowtie$ node. Simplification (2b) is applied to eliminate signal S, for signal up simplification (2a) is applied to have only one up signal node and then simplification (2b) is used to eliminate the up signals.

6.4.1 The execution algorithm

A temporal inference network $\mathcal{N}$ is formally defined as:

$$\mathcal{N} =< N, t, p, l, r >$$

where:

- N is the set of nodes;
- $t : N \rightarrow \{\wedge, \vee, \partial^k\}$ is a function that gives for each node its type;
- $p : N \rightarrow N$ is a function that gives for each node its parent;
- $l : N \rightarrow N$ is a function that gives for each node its left son;

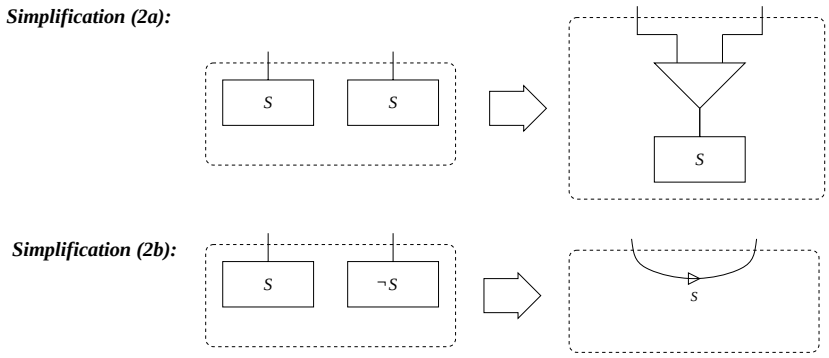

Figure 6.9: Simplifications to eliminate signals

- $r : N \to N$ is a function that gives for each node its right son;

An arrow from node n_1 to node n_2 for the network is represented as a couple (n_1, n_2), but at each time instant the network can have different arrow directions, thus each arrow has a time instant. This is represented as the 3-ple (n_1, n_2, t) or in a more readable way $(n_1, n_2)@t$ meaning that at time t there is an arrow from node n_1 to node n_2, such an element is also called an event.

The set $T = [0, T_{max}] \subset \mathbb{Z}$ represents all time instants. The known information of a network is a subset of $D = N \times N \times T$. The inference process can be seen as a sequence of subsets of D, $\Omega_0 \subset \Omega_1 \subset \ldots \subset \Omega_n \subseteq D$. The elements of Ω_{k+1}/Ω_k represent the new information, that can be inferred from Ω_k using the inference rules or acquired from the outside via inputs.

A general schema for the algorithm is:

repeat forever
  acquire information from outside;
  if there is information to be processed
    select an event to be processed;
    infer new events from the event selected;
  end if
  add new information (inferred or acquired) to information to be processed;
end repeat

This kind of process can be more detailed.
In the following algorithm where:

- $\Omega_0 \subset D$ is the initial knowledge;

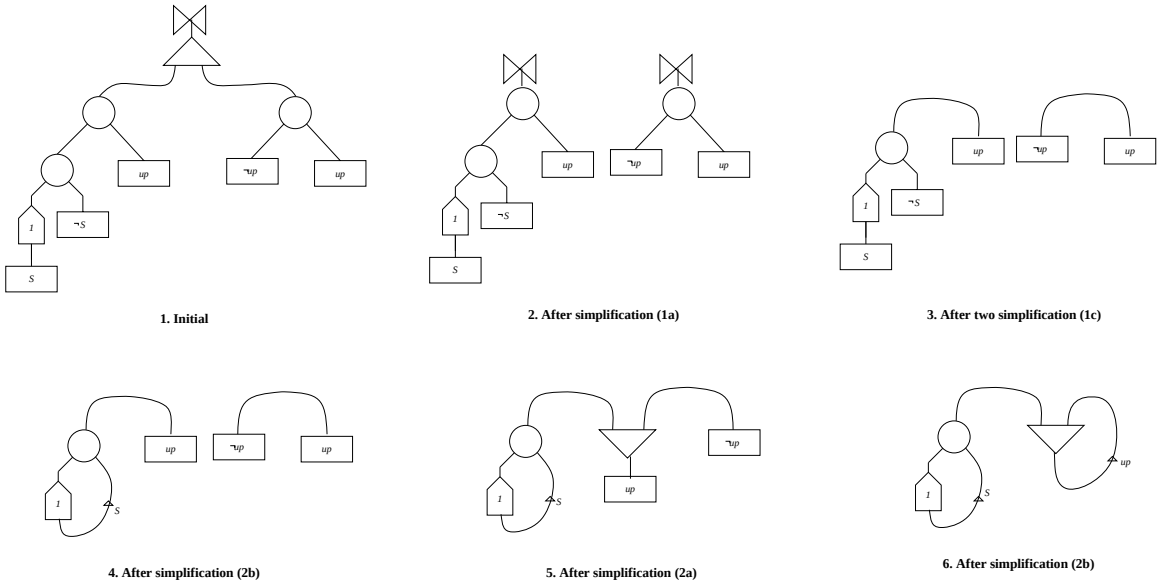

Figure 6.10: Simplifications of the up example

- $\omega()$ is a function that for each evaluation gives a subset of D representing information acquired from the outside;
- $\lambda_{\mathcal{N}}(e@t, S) \subset D$ is a function that determines the events derivable from $e@t$ for the network $\mathcal{N}$ based on knowledge S.
- variable $\Phi \subset D$ represents the set of available knowledge;
- variable $\Delta \subset D$ is the set of acquired information not yet elaborated;
- variable Ψ is the set of knowledge found in the current iteration;

$\Phi \leftarrow \Omega_0$; // initial information
$\Delta \leftarrow \Omega_0$; // initial information that has to be processed
repeat forever
    $\Psi \leftarrow \omega() \setminus \Phi$; // acquire new information from outside
    if $\Delta \neq \emptyset$ **then** // if there is something to process...
        choose $e@t \in \Delta$ with the minimum t;
        $\Psi \leftarrow \Psi \cup \lambda_{\mathcal{N}}(e@t, \Phi \setminus \Delta) \setminus \Phi$; // determine the new information inferred by $e@t$
        $\Delta \leftarrow \Delta \setminus \{e@t\}$; // drop $e@t$ from information to be processed
    endif
    $\Delta \leftarrow \Delta \cup \Psi$; // add the new information as to be processed

$\Phi \leftarrow \Phi \cup \Psi$; // add the new information to the current knowledge
endrepeat

The function $\Lambda = \lambda_{\mathcal{N}}((n_i, n_o)@t, \Phi)$ can be computed with the following algorithm using the temporal inference rules presented in the previous section.

function $\lambda_{\mathcal{N}}((n_i, n_o)@t, S)$
  if $t(n_o) = \wedge$ **then**
    if $n_i = p(n_o)$ **then**
      $\Lambda \leftarrow \{(n_o, l(n_o))@t, (n_o, r(n_o))@t\}$; // rule (J1)
    else
      $\Lambda \leftarrow \{(n_o, p(n_o))@t\}$; // rules (J2)/(J3)
    endif
  elseif $t(n_o) = \vee$ **then**
    if $(n_i = p(n_o) \wedge (l(n_o), n_o)@t \in S) \vee (n_i = l(n_o) \wedge (p(n_o), n_o)@t \in S)$ **then**
      $\Lambda \leftarrow \{(n_o, r(n_o))@t\}$; // rule (G1)
    elseif $(n_i = p(n_o) \wedge (r(n_o), n_o)@t \in S) \vee (n_i = r(n_o) \wedge (p(n_o), n_o)@t \in S)$ **then**
      $\Lambda \leftarrow \{(n_o, l(n_o))@t\}$; // rule (G2)
    elseif $(n_i = r(n_o) \wedge (l(n_o), n_o)@t \in S) \vee (n_i = l(n_o) \wedge (r(n_o), n_o)@t \in S)$ **then**
      $\Lambda \leftarrow \{(n_o, p(n_o))@t\}$; // rule (G3)
    else $\Lambda \leftarrow \emptyset$;
    endif
  elseif $t(n_o) = \partial^k$ **then**
    if $n_i = p(n_o)$ **then**
      if $t - k \in T$ **then** $\Lambda \leftarrow \{(n_o, l(n_o))@(t-k)\}$; // rule (D1)
      else $\Lambda \leftarrow \emptyset$;
      endif
    else
      if $t + k \in T$ **then** $\Lambda \leftarrow \{(n_o, p(n_o))@(t+k)\}$; // rule (D2)
      else $\Lambda \leftarrow \emptyset$;
      endif
    endif
  endif
return Λ

In order to evaluate the complexity of the execution algorithm, data structures and the algorithm have to be better defined.

The main data structures that have to be considered are those needed to represent the sets Φ and Δ.
In order to manipulate the Φ and Δ sets the following operations are needed:

On set Φ: (i) to add an element to the set, (ii) check if an element is in the set. These two operations can be performed in a constant execution time for example using a vector of boolean variables.

On set Δ: (i) add an element to the set, (ii) drop an element from the set, (iii) find an element with the minimum t. For the last operation the representation with a boolean vector is not convenient, we used a list where each element contains a list of the arcs to be elaborated, all with the same time instant and the main list is ordered with increasing time instants. For example:

$$\Delta = [[(n_1, n_3), (n_2, n_5)]@5, [(n_1, n_3)]@6, [(n_5, n_2), (n_3, n_1)]@7]$$

Considering the complexity of operations on Δ an element can be added with O(card T) or in costant time if the time instant of the event to be added is always lower than those in Δ. The event with minimum t is found in constant time (is the top of the list) and the top of the list can be dropped in a constant time d duration.

Functions *add*, *choose* and *drop* are used to add an element to the list, to get the head of the list and drop the top from the list. These functions can be specified in a functional-style programming language where $x \,|\, L$ represents the list with head x and list tail L, $[\,]$ is the empty list and $[x]$ is equivalent to $x \,|\, [\,]$.

function $add(X@t' \,|\, L, e@t)$
    if $t < t'$ **then**
        return $[e]@t \,|\, (X@t' \,|\, L)$;
    elseif $t = t'$ **then**
        return $(e \,|\, X)@t \,|\, L$;
    else
        return $X@t' \,|\, add(L, e@t)$;
    endif
endfunction

function $choose((e \,|\, X)@t \,|\, L)$
return $e@t$;

function $drop((e \,|\, X)@t \,|\, L)$
    if $X \neq []$ **then**
        return $X@t \,|\, L$;
    else

```
        return L;
    endif
endfunction
```

The execution algorithm can be written as:

```
algorithm execute_N(Ω_0, ω)
    Φ ← ∅;
    S ← ∅;
    Δ ← [ ];
    foreach s ∈ Ω_0
        Φ ← Φ ∪ {s};
        Δ ← add(Δ, s);
    endfor
    repeat forever
        if Δ ≠ [ ] then
            e@t ← choose(Δ);
            foreach s ∈ λ_N(e@t, S) // first cycle
                if s ∉ Φ then
                    Φ ← Φ ∪ {s};
                    Δ ← add(Δ, s);
                endif
            endfor
            S ← S ∪ {e@t};
            Δ ← drop(Δ);
        endif
        foreach s ∈ ω() //second cycle
            if s ∉ Φ then
                Φ ← Φ ∪ {s};
                Δ ← add(Δ, s);
            endif
        endfor
    endrepeat
endalgorithm
```

The evaluation of the asimpthotical complexity for the "repeat" body that in the following is also called a deduction step or simply a step has to be discussed.

Functions *choose* and *drop* do not contribute to the complexity, since they can be executed in a constant time and are not dominant instructions, complexity of function *add* depends on the event instant or better on the distance of this event from the head. The complexity of a deduction step is the sum of complexity of the first cycle and the second cycle. In the first **foreach** cycle, the *add* is repeated a number of times up to the maximum size of $\lambda_{\mathcal{N}}(e@t, S)$ that is 2 (as it can be seen from its definition). The maximum distance of events time in $\lambda_{\mathcal{N}}(e@t, S)$ from t is the maximum of the absolute value of delays present in the network (D_{max}). The evaluation of λ can be done in a constant time, thus the contribution to complexity of the first cycle is $O(2\ D_{max})$ then an $O(D_{max})$. For the second cycle, the number of elements in $\omega()$ is at most equal to the number of inputs (card I) and the time of these events is at most equal to the size of the temporal domain (card T) thus the complexity of the second cycle is an $O(\text{card } I \text{ card } T)$ giving a global complexity of $O(D_{max} + \text{card } I \text{ card } T)$. If the time instants of the input events is less than the time of events in Δ then complexity of the second cycle is an $O(\text{card } I)$ because function *add* adds the element to the top of the list, and then the complexity is an $O(D_{max} + \text{card } I)$. It should be noted that $O(D_{max} + \text{card } I)$ is the complexity of a deduction step, not that to produce all the inferable information for a time instant, to infer this information a certain number of deduction steps are needed.

To obtain an evaluation of the time needed to produce the whole information for a given time instant, the inference function λ has to be causal. This means that it infers events in the present or in the future with respect to event $e@t$:

$$\forall e@t, S, d@t'.\ \ d@t' \in \lambda_{\mathcal{N}}(e@t, S) \Rightarrow t' \geq t$$

With this assumption, and assuming $\omega() = \emptyset$ (meaning that no inputs are acquired) to drop from Δ all the events with time t (where t is the minimum time in Δ) in the worst case a number of steps equal to the number of the network arcs are needed. In fact, at each step the algorithm chooses an arc with time t (for causality *lambda* cannot introduce events previous than t) and if an arc is processed it will be never reprocessed in the same time instant. From this we can deduce that in the worst case all the arcs of the network are processed. In this case, the number of steps of the algorithm is equal to the number of arcs. From this considerations we can deduce that the time needed to infer all the information for a given time instant t is an:

$$O((D_{max} + \text{card } I)\ N_a)$$

where N_a is the number of arcs of the network and assuming λ a causal inference function.

6.5 Real-time execution

The execution algorithm has been developed even for on-line execution. Let us consider a sampled time system where continous input signals are sampled at a given rate and outputs are produced.

If the specification is strictly causal, meaning that outputs depend only on past samples of inputs and internal variables, then at a certain discrete time instant, knowing the current/past inputs, using the inference process, the output values for the next time instant can be desumed as depicted in Figure 6.11.

The time duration between two consecutive output generations or input acquisitions may be not enough to produce the information about outputs.

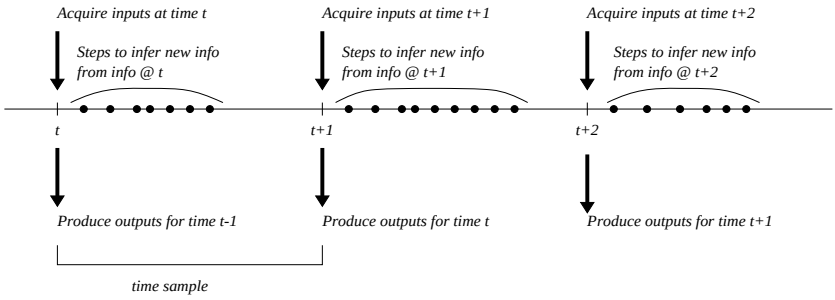

Figure 6.11: Real-time execution

As showed in the previous section, if the inference function is causal, N_a inference steps are needed to produce all the inferrable information for time t, where N_a is the number of network arcs.

The maximum time needed for one step of the algorithm can be evaluated depending on the machine (clock, CPU timings, etc.) and the inference network (D_{max}), multiplying this time duration for the number of arcs of the inference network the maximum time needed to produce the outputs for an instant can be found. If this time is less then the time sample then real-time execution can be done, in formula:

$$T_s > T_{max}\ N_a$$

where: T_s is the duration of the time sample and T_{max} is the maximum time needed to execute an inferential step.

In order to have a more precise evaluation of the time needed, T_{max} can be split in two times:

1. T_{input} the time needed to add inputs to set Δ;

2. T_{infer} the time needed to infer information.

Considering that during execution, only in the first step after the acquisition of inputs there is new information, then condition on the time sample becomes:

$$T_s > T_{input} + T_{infer}\ N_a$$

Even condition on N_a can be slightly reduced. Considering that the elimination of input signals from the syntax tree produced a situation similar to that depicted in Figure 6.12.

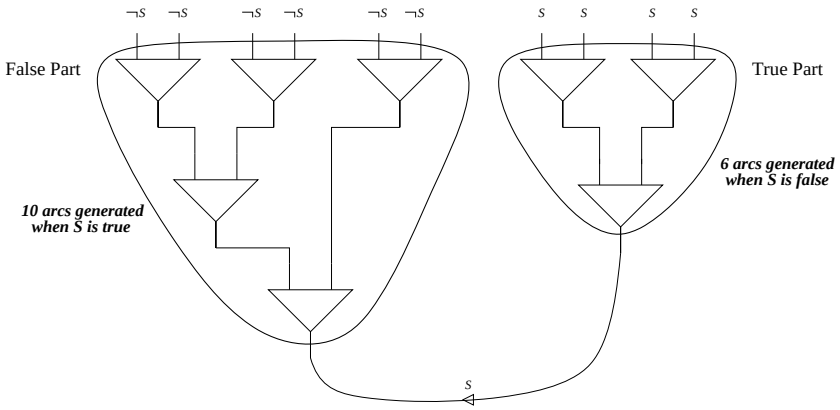

Figure 6.12: Inputs of the system

When the input S is true at time t, using rule J1, all the arcs of the *false part* are generated ($\neg S$ is false): one for each inference step. It should be noted that the arcs connected to the *true part* are not generated (for time t) and therefore they will be never inserted in Δ (for time t). Assuming that the signal is true, the maximum number of inference steps are N_a minus the number of arcs in the *true part* that will be not generated. In the worst case, considering an input signal i, a number of inference steps equal to N_a minus the number of arcs of the part with less arcs are at most needed. Considering all the inputs, N_a can be decreased of the minimum number of arcs of each part for each input.

The number of arcs connected to the n occurrences of the same input (positive or negative) is:

$$2\ (n-1).$$

If i is an input, $n^+(i)$ is the number of positive occurrences of input i in the specification formula and $n^-(i)$ is the number of negative ones, then the maximum number of steps to produce the information for time t is:

$$N_a - 2\ \sum_{i \in I} (\min(n^+(i), n^-(i)) - 1).$$

Finally, the condition on the sampling time becomes:

$$T_s > T_{input} + T_{infer}\ (N_a - 2\ \sum_{i \in I}(\min(n^+(i), n^-(i)) - 1))$$

The conditions presented do not change the asimphotical complexity of the algorithm evaluated in the previous section. While, these more detailed conditions can be useful when the evaluation of condition $T_s > T_{max}\ N_a$ fails. In this case, if the difference $T_{max}\ N_a - T_s$ is very small, the following condition should be checked:

$$T_s > T_{max}\ (N_a - 2\ \sum_{i \in I}(\min(n^+(i), n^-(i)) - 1)),$$

and if it fails, this more detailed condition:

$$T_s > T_{input} + T_{infer}\ (N_a - 2\ \sum_{i \in I}(\min(n^+(i), n^-(i)) - 1))$$

will be surely useful for the verification of T_s.

Conclusions

In this work two extensions of temporal logic TILCO have been presented: C-TILCO for the specification of complex/large systems and TILCO-X to simplify the writing of temporal expressions. Moreover, a new way of executing temporal logics has been presented.

C-TILCO
C-TILCO is well suited for system composition/decomposition, it permits the construction of hierarchical specifications in a top-down approach, it permits to reuse the specification of components within the same system or the development of other systems. Communication mechanisms between processes have been specified using TILCO. C-TILCO has been formalized within Isabelle/HOL theorem prover. Properties for the whole system as well as for a single process can be proved. This logical framework permits also the validation of the decomposition of a process. The possibility to execute the specification is an important feature since well-known conditions can be quickly tested. The language used for the specification is expressive, simple and concise with a limited "time to learn" since it has inherited conciseness from TILCO [21]. C-TILCO can be profitably used for the formal specification of critical complex real-time systems.

TILCO-X
TILCO-X is a temporal logic for the specification, validation and verification of real-time systems. The introduction of the Bounded Happen and Dynamic Interval operators enhanced the expressive power of TILCO. TILCO-X enhances the readability and conciseness of formulas with respect to TILCO, especially for requirements that include the event ordering. In this case, it removes the differences between past and future maintaining at the same time the implicit time specifications.

In summary, TILCO-X differs from other temporal logics proposed in the literature. TILCO-X is a first order interval logic that (i) provides a metric for time (thus allowing

specification of qualitative and quantitative timing constraints); (ii) presents a linear implicit time model; (iii) adopts a uniform manipulation of intervals from past to future for actions, events, event ordering; (iv) present specific operators for defining temporal constraints including the counting the occurrence of events; (iv) provides decidability for a wide set of formulæ (non-temporal quantifications must bind only variables with types over finite domains); (v) TILCO-X allows execution of specifications for validation and also for system implementation. In TILCO-X no explicit quantification over the temporal domain is allowed and with the new operators this limitation is strongly less relevant since the demand of quantification has been reduced with respect to the old TILCO version; and

TILCO-X is particularly suitable for requirements analysis and the incremental specification of real-time systems. TILCO-X supports validation during all phases of the system life-cycle by means of its formalisation in the automatic theorem prover Isabelle/HOL. This allows validation for refinement and the proof of properties. Moreover, the final operational validation is also supported by a *TILCO-X Executor*, which allows execution and the model-checking of systems specifications.

TILCO/TILCO-X Executor

The execution of temporal logics has been studied, in particular the possibility of real-time execution has been analyzed in order to define and implement an executor for TILCO and TILCO-X specifications.

To this end, the specifications written in TILCO or TILCO-X are translated to BTL a temporal logic with a low number of operators. A visual representation of BTL formulæ has been given, and inference rules for the deduction of logical values of output signals has been presented. Also a visual representation of the deduction process has been proposed.

In the case of causal inference, the complexity of the time needed to deduce the value of outputs from the value of the inputs is proportional to the number of arcs of the inference network an therefore is linear with the "size" of the specification and proportional to the maximum delay in the network. From this result, conditions for the feasibility of real-time execution has been found.

Appendix A

Tools for TILCO specification

A.1 Architecture

For the validation of TILCO, TILCO-X and C-TILCO specifications some tools have been developed. These tools are used to aid the validation of specifications by means of theorem-proving (using Isabelle) and/or by simulation using the temporal logic executor. In Figure A.1, the relationships between the developed tools are represented. In the following, a

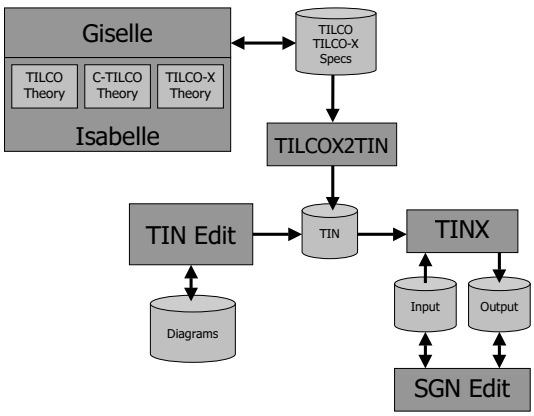

Figure A.1: Tools for TILCO specification

short description of each tool is reported:

- **Isabelle** is the validation environment, via property proof, for TILCO specifications, Isabelle theories for TILCO, TILCO-X and C-TILCO have been developed and are used for the validation of specifications.

- **Giselle** is a graphic user interface for Isabelle, its aim is to aid the interaction with Isabelle that is a text oriented application;
- **TILCOX2TIN** is a translator, it reads textual specifications written in TILCO/TILCO-X and produces a temporal inference network (TIN) for executing the specification.
- **TIN Edit** is a graphic editor for the visual specification of system properties. It permits the direct production of TIN files, it supports the basic components of TINs (*joint, gate, delay, signal*) and moreover other components such as *at* and *happen* (over constant intervals) can be used and also generic sub-nets can be defined and used.
- **TINX** is the executor. Basically, it gets in input the TIN file that contains the specification. Then it is capapble of reading the inputs and producing the outputs according to the specification.
- **Sgn Edit** is a signal editor. It permits to create, view, and change the files representing the inputs/outputs of the system.

A.2 Isabelle TILCO theories

For the validation of TILCO specifications within Isabelle, some Isabelle theories have been developed.

- **TILCO theory**
  - **Interval.thy** extends the integers theory to define constant intervals.
  - **Tilco.thy** extends Interval.thy to define all the operator of logic TILCO.
- **C-TILCO theory** (see Appendix B)
  - **PPorts.thy** extends the TILCO theory to provide primitive ports that permit to send/receive messages. It represents the low-level communication layer.
  - **Ports.thy** extends PPorts.thy to provide synchronous ports that permit to send/receive messages synchronously. It represents the high-level communication layer.
  - **Process.thy** extends Ports.thy to provide the concept of process with its variables, parameters, sub-processes.
- **TILCO-X theory** (see Appendix C)

- **Time.thy** extends the integers theory to provide the basic logical operators (and, or, not, forall, exists, etc.) over temporal variables.
- **IntervalX.thy** extends Time.thy to provide the concept of dynamic intervals, it defines the after and before operators.
- **TilcoX.thy** extends IntervalX.thy to provide at and happen operators as well as the other temporal operators with exception of Bounded Happen.
- **TilcoXX.thy** extends TilcoX.thy to provide Bounded Happen operators.

A.3 Giselle

Giselle is a graphic user interface for Isabelle. Isabelle is a text oriented application, then to increase the usability the user interface called Giselle has been developed. Other user interfaces for Isabelle can be found (Proof General, XIsabelle) but are plug-ins for Emacs/X-Emacs or use tcl/tk.

The main window of Giselle, presented in Figure A.2, is divided in tree parts: on the left, there is the current proof state; on the right, the tactics applied to the goal (the property to be proved) that produced the current proof state are reported; on the bottom, there are the tactics that can be applied to the current proof state. Clicking on a button a dialog appears where arguments of the tactic can be chosen. The interface to tactics is completely

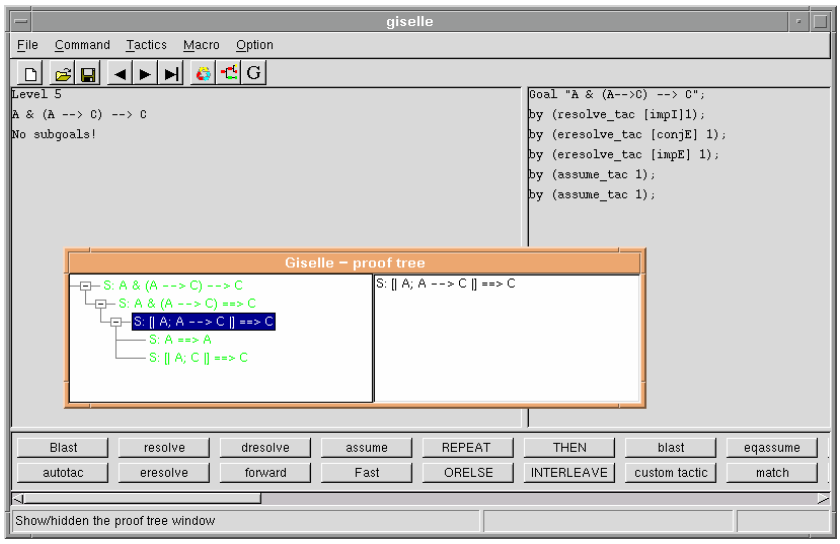

Figure A.2: Giselle main window and proof tree window

configurable. In a textual configuration file, there is, for each tactic, its textual form and the corresponding user inputs are highlighted.

In Giselle, it is also possible to see the proof tree, where is reported how a particular sub-goal has been proved.

A.4 TILCOX2TIN

TILCOX2TIN is a translator that is used to generate from a textual specification containing TILCO/TILCO-X formulæ the `.tin` file for the executor TINX. In Table A.1, the translations used from the symbolic mathematical language to the textual language are reported.

name	symbolic	textual
and	$\wedge$	`&`
or	$\vee$	`|`
not	$\neg$	`~`
imply	$\rightarrow$	`-->`
coimply	$\leftrightarrow$	`<-->` or `==`
delay-k	∂^k	`#k`
delay	∂	`#`
at	@	`@`
happen	?	`?`
until	**until** A B	`until(A,B)`
since	**since** A B	`since(A,B)`
happen-min	$?_m$	`?_ m`
happen-max	$?^M$	`?^ M`
happen-minmax	$?_m^M$	`?_^ m M`
infinity	∞	`inf`

Table A.1: TILCOX2TIN: Translation from symbols to text

A specification file `.tlc` starts with a prologue where the signals used as inputs and outputs and eventually the initial conditions for some signals are reported. After the prologue a list of TILCO/TILCO-X formula separated with a semicolon is reported. All these formula are considered in *and*, and are assumed always true.
For example:

```
inputs: S
outputs: A
```

```
inits:

upS == S & #~S; //upS is true iff S is true now and
                // false in the prec. instant
upS --> A @ [0,+upS); //if upS is true then A is true
                      //upto the next time upS is true
```

A.5 TIN Edit

The TIN Editor is a graphic editor that permits the visual creation of temporal inference networks. It can be regarded as a visual specification interface for TILCO. In Figure A.3, the main window is depicted, where the user arranges the temporal logic elements, connecting them together in order to create the desired network. To this end, one can select the required element from the toolbar.

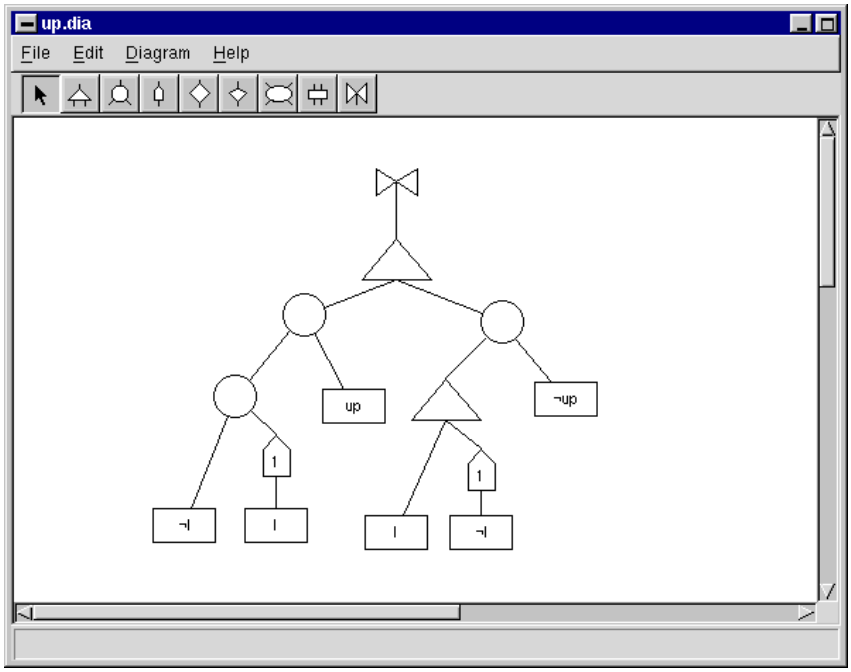

Figure A.3: TIN Edit main window

The main characteristics of the TIN editor are:

- it permits the construction of the TIN using the basic elements as *joint, gate, delay,*

always and *signals*, connecting them together with arcs;

- it permits to use non basic elements as the *at* and the *happen* operators;
- it permits to use TIN networks within another network, giving the possibility of decomposition/composition of the system specification;
- it permits to save/load/print the networks;
- it checks if some information is missing (the value of a delay, or the intervals of the at component, etc.).
- it permits to generate the `.tin` file for the TINX executor.

A.6 TINX

TINX is the temporal logic executor, it executes the specification written in a `.tin` file describing the temporal inference network of the system. Each input/output signal is associated with a file `.io` where the values of the signal are stored. TINX reads the `.io` files of input signals and produces `.io` files of output signals.

A `.tin` file is a textual representation of the network. In the file the list of nodes of the network is reported. For each node, there is a line with the following form:

```
<node name>: <node type>; <parent node>,<left son>[,<right son>]
```

where: `<node name>` is the unique node name, `<parent node>`,`<left son>` and `<right son>` are name references to other nodes. `<node type>` can be: `G` for a gate node, `J` for a joint node, `D` for a delay node eventually followed by an integer representing the delay value.

After the lines representing the nodes of the network, there are the lines where arcs are bound to input/output files. Each line has the following form:

```
<io mode> <filename> (<node name>,<node name>)
```

where: `<io mode>` can be `!` for an input signal and `?` for an output signal; `<filename>` is the name of the file associated with the signal (without .io) and the two `<node name>` represent the extremes of the arc.

For example the following is a valid `.tin` file:

```
g0: G ; j2, j0, d0
j0: J ; g0, j1, d1
j1: J ; j2, d0, j0
j2: J ; j1, d1, g0
d0: D ; g0, j1
d1: D ; j0, j2

! 1 (j1,j2)
? up (g0,j0)
```

A `.io` file is a sequence of chars `0`, `1` (false/true) and *unknown* value represented with a char with ASCII code 0. The file is ended with char '`.`'.

The executor can generate also a log where the events generated during execution are reported. Optionally the execution can be made by using only causal inference.

A.7 SGN Edit

SGN Edit is a signal editor used to create, view and modify signals. It allows reading the signals associated with a TIN file. Figure A.4 shows the main window where the signals of the *up* example are reported.

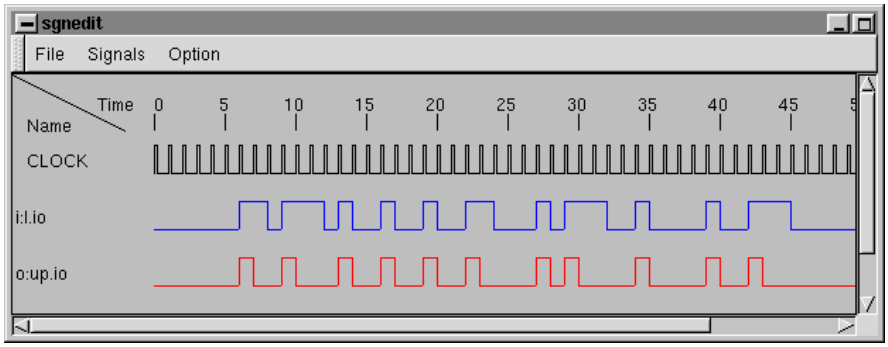

Figure A.4: SGN Edit main window

The values of a signal can be changed by clicking on the desired time instant and the logical value toggles form true to false, from false to undefined and from undefined to true. The values can be changed also selecting a portion of the signal and these instants can be forced to assume value all true, false, undefined, randomly true or false, or alternating true/false

values. The executor can be directly invoked from the menu to quickly see the response of the system when an input signal is changed.

Appendix B

Isabelle C-TILCO Theory

B.1 PPorts.thy

```
(*
    File: PPorts.thy
    Theory Name: PPorts
    Logic Image: HOL
*)

PPorts  =  Tilco +

classes
  Port    < term
  inPort  < Port
  outPort < Port
types
  (* typed primitive ports *)
  'a p_inPort
  'a p_outPort
arities
  p_inPort  :: (term) inPort
  p_outPort :: (term) outPort

consts
(* transmission and reception primitives *)
  p_send       :: ['a p_outPort, 'a] => tbool ("_.psend _" [50,50] 50)
  p_receiveAck ::  'a p_outPort => tbool      ("_.preceiveAck" [50] 50)

  p_receive :: ['a p_inPort,'a] => tbool ("_.preceive _" [50,50] 50)
  p_sendAck ::  'a p_inPort => tbool     ("_.psendAck" [50] 50)
```

```
  connectd  :: ['a p_outPort,nat,'a p_inPort] => bool
    ("_ ->_>- _" [50,50,50] 50)

syntax
  connect0  :: ['a p_outPort,'a p_inPort] => bool ("_ ->- _"  [50,50] 50)
  connect1  :: ['a p_outPort,'a p_inPort] => bool ("_ ->>- _" [50,50] 50)
translations
  "outP ->- inP"  == "outP ->0>- inP"
  "outP ->>- inP" == "outP ->1>- inP"
rules
  (* rules for transmission of Msg and Ack*)
  p_tr_msg_rule " (outP ->d>- inP) -->   \
\                 rule((outP.psend k) #= (inP.preceive k)@single($#d))"
  p_tr_ack_rule " (outP ->d>- inP) -->   \
\                 rule((inP.psendAck) #= (outP.preceiveAck)@single($#d))"

  (* in each instant only one value can be sent/received *)
  p_uni_send_rule " rule( ((outP.psend k) #& (outP.psend v)) #-->   \
\                         ((const v)#=(const k)))"
  p_uni_rec_rule  " rule( ((inP.preceive k) #& (inP.preceive v)) #-->   \
\                         ((const v)#=(const k)))"

end
```

B.2 Ports.thy

```
(*
    File:  Ports.thy
    Theory Name: Ports
    Logic Image: HOL
*)

Ports  =  PPorts +

types
  (* high level ports *)
  'a s_inPort  (* synchornous reception *)
  'a s_outPort (* synchronous emission *)
arities
  s_inPort  :: (term) inPort
  s_outPort :: (term) outPort
```

```
consts
(* synchronous emission *)
  s_out    :: 'a s_outPort => 'a p_outPort
  s_SendW  :: ['a s_outPort, 'a, tbool,tbool] => tbool
    ("_.!! (_)/ [ (_) ] ;;/ (_)" [50,50,10,10] 9)
  s_noSend :: 'a::outPort => tbool ("_.~!!" [50] 50)

(* synchronous reception *)
  s_in        :: 'a s_inPort => 'a p_inPort
  s_ReceiveW  :: ['a s_inPort, tbool, 'a => tbool] => tbool
    ("_.?? [ (_) ] ;;/ (_)" [50,10,10] 9)
  s_Receive :: 'a::inPort => tbool ("_.~??" [50] 50)
  s_RWait     :: 'a s_inPort => tbool ("_.RWait" [50] 50)
  s_RValue     :: ['a s_inPort, 'a ]=> tbool ("_.RValue _" [50,50] 50)
syntax
  s_Send :: ['a s_outPort, 'a, tbool] => tbool
    ("_.!! (_) ;;/ (_)" [50,50,10] 9)
  s_Receive :: ['a s_inPort, 'a => tbool] => tbool
    ("_.?? ;; (_)" [50,10] 9)
  send :: ['a::outPort, 'b] => tbool
    ("_.(send (_))" [50,50] 50)
  receive :: ['a::inPort, 'b] => tbool
    ("_.(receive (_))" [50,50] 50)
  receiveAck :: 'a::outPort => tbool
    ("_.receiveAck" [50] 50)
  sendAck :: 'a::inPort => tbool
    ("_.sendAck" [50] 50)
translations
  "p.send v"         == "(s_out p).psend v"
  "p.receive v"      == "(s_in p).preceive v"
  "p.receiveAck"     == "(s_out p).preceiveAck"
  "p.sendAck"        == "(s_in p).psendAck"

  "p.!! v ;; P"  == "p.!! v [ true ] ;; P"
  "p.?? ;; P" == "p.?? [ true ] ;; P"
defs
  s_RValue_def "(siP.RValue v) == (since' (siP.receive v #& #~ siP.sendAck)
                                          (#~ siP.sendAck))"
  s_RWait_def "(siP.RWait) == (#~ (#? k. siP.RValue k))"
  s_aSend_def "(soP.!!) == (#? v W Q. (soP.!! v [ W ] ;; Q))"
  s_aReceive_def "(siP.??) == (#? W Q. (siP.?? [ W ] ;; Q)) "
rules
  s_SendW_rule
```

```
    " rule((soP.!! v [ W ] ;; Q) #-->
              (soP.send v) #&
              (until0 (soP.receiveAck #& Q)
                      (#~ soP.receiveAck #& W)) #&
              (soP.receiveAck #|
              (#~ soP.receiveAck #&
                  until (soP.receiveAck)
                        (#~ soP.receiveAck #& soP.~!!)))))"
  s_noSend_rule
    " rule( soP.~!! #--> (#! k. #~ soP.send k)) "

  s_SendW_unique
    " rule((soP.!! v1 [W1];;P1)#&(soP.!! v2 [W2];;P2) #-->
      ((const v1) #= (const v2)) #&
      ((W1#=W2) #& (P1#=P2))@<-inf,inf>)"
  s_SendW_noSend_rule
    " rule((soP.!! v1 [W1];;P1) #& (soP.~!!) #--> false)"

  s_ReceiveW1_rule
    " rule((siP.?? [ W ] ;; Q) #& siP.RWait #-->
              (until0 (#? k. siP.receive k #&
                             siP.sendAck #& Q k)
                      (#~(#? k. siP.receive k) #& W)) #&
              ((#? k. siP.receive k) #|
              (#~ (#? k. siP.receive k) #& #~siP.sendAck #&
                  until ((#? k. siP.receive k))
                        (#~ (#? k. siP.receive k) #& #~siP.??))))"
  s_ReceiveW2_rule
    " rule((siP.?? [ W ] ;; Q) #& siP.RValue v #-->
              siP.sendAck #& Q v) "
  s_noReceive_rule
    " rule(siP.~?? #--> (#~ siP.sendAck)) "

  s_ReceiveW_unique
    " rule((siP.?? [W1];;P1) #& (siP.?? [W2];;P2) #-->
      ((W1#=W2) #& (#! v. (P1 v) #= (P2 v)))@<-inf,inf>)"
  s_ReceiveW_noReceive_rule
    " rule( (siP.?? [W1];;P1) #& siP.~?? #--> false )"
end
```

B.3 Process.thy

```
(*
    File:  Process.thy
    Theory Name: Process
    Logic Image: HOL
*)

Process  =  Ports +

classes
  process    < term
  tComponent < term
types
  ('a, 'b) tVar
       'a  tProp
  ('a, 'b) tSubproc
  ('a, 'b) tSInPort
  ('a, 'b) tSOutPort
arities
  tVar      :: (process,term)    tComponent
  tProp     :: (process)         tComponent
  tSubproc  :: (process,process) tComponent
  tSInPort  :: (process,term)    tComponent
  tSOutPort :: (process,term)    tComponent
consts
  pvar::['a::process, ('a, 'b) tVar]     => 'b tfun      ("_.var _" [50,50] 55)
  ppro::['a::process, 'a tProp]          => tbool        ("_.pro _" [50,50] 55)
  psub::['a::process, ('a, 'b) tSubproc] => 'b           ("_.sub _" [50,60] 55)
  psip::['a::process, ('a, 'b) tSInPort] => 'b s_inPort  ("_.sip _" [51,51] 55)
  psop::['a::process, ('a, 'b) tSOutPort]=> 'b s_outPort ("_.sop _" [51,51] 55)

  pstart :: 'a::process => tbool ("_.process'_start" [50] 55)
  system_start :: tbool
syntax
  pdot::['a::process, 'b::tComponent ] => 'c::term       ("_._" [50,60] 55)
  pcolon::'b::tComponent =>'c::term                      (":_" [60] 55)
  cstart :: tbool (":process'_start")
rules
  uni_start
    "rule (system_start ?! <-inf,inf>)"
  uni_pstart
    "rule (p.process_start ?! <-inf,inf>)"
```

```
  pstart_noReceive
    "rule (p.process_start #--> ((p.sip inP).~??) @ <-inf,$#0>)"
  pstart_noSend
    "rule (p.process_start #--> ((p.sop outP).~!!) @ <-inf,$#0>)"
end
ML
  ...
```

Appendix C

Isabelle TILCO-X Theory

C.1 Time.thy

```
(*
    File:  Time.thy
    Theory Name: Time
    Logic Image: HOL
*)

Time  =  IntArith +
types
  time    = int
  tbool   = (time => bool)
  'a tfun = (time => 'a)
consts
  T        :: time
  now      :: [tbool, time] => bool
  true     :: tbool
  false    :: tbool

  "#~"     :: tbool => tbool                                    ("#~ _" [40] 40)
  "#&"     :: [tbool, tbool] => tbool                               (infixr 35)
  "#|"     :: [tbool, tbool] => tbool                               (infixr 30)
  "#="     :: ['a tfun, 'a tfun] => tbool                           (infixr 25)
  "#~="    :: [tbool, tbool] => tbool                               (infixr 33)
  "#-->"   :: [tbool, tbool] => tbool                               (infixr 25)

  TIF      :: [tbool, 'a tfun, 'a tfun] => ('a tfun)
                                          ("(IF (_)/ THEN (_)/ ELSE (_))" 10)
```

```
  TAll     :: ('a => tbool) => tbool                          (binder "#! "  10)
  TEx      :: ('a => tbool) => tbool                          (binder "#? "  10)
  TEx1     :: ('a => tbool) => tbool                          (binder "#?! " 10)
  TBall    :: ['a set, 'a => tbool] => tbool
  TBex     :: ['a set, 'a => tbool] => tbool

  const    :: 'a => ('a tfun)
  unary    :: ['a => 'b, 'a tfun] => ('b tfun)
  binary   :: [['a,'b] => 'c,'a tfun,'b tfun] => ('c tfun)

syntax
  "*TBall" :: [idts,'a set,tbool]=>tbool  ("(3#! _:_./ _)" 10)
  "*TBex"  :: [idts,'a set,tbool]=>tbool  ("(3#? _:_./ _)"  10)

translations
  "A #~= B"          == "#~ (A #= B)"
  "#! x:A. P"        == "TBall A (%x. P)"
  "#? x:A. P"        == "TBex A (%x. P)"

defs
  now_def      "now A t      == A(T+t)"

  Ttrue_def    "true         == (%x. True)"
  Tfalse_def   "false        == (%x. False)"

  Tnot_def     "#~ A         == (%x. ~ A x)"
  Tand_def     "A #& B       == (%x. A x & B x)"
  Tor_def      "A #| B       == (%x. A x | B x)"
  Tequiv_def   "A #= B       == (%x. A x = B x)"
  Timply_def   "A #--> B     == (%x. A x --> B x)"

  TIF_def      "TIF P x y    == (%z. if (P z) then (x z) else (y z))"

  TAll_def     "TAll P       == (%y. ! x. P x y)"
  TEx_def      "TEx P        == (%y. ? x. P x y)"
  TEx1_def     "TEx1 P       == (%z. ?! x. P x z)"
  TBall_def    "TBall A P    == (%y. ! x:A. P x y)"
  TBex_def     "TBex A P     == (%y. ? x:A. P x y)"

  const_def    "const K      == (%x. K)"
  unary_def    "unary f B    == (%t. f (B t))"
  binary_def   "binary f A B == (%t. f (A t) (B t))"
end
```

C.2 IntervalX.thy

```
(*
    File:        IntervalX.thy
    Theory Name: IntervalX
    Logic Image: HOL
*)

IntervalX = Time +

datatype bound =  Val   int ("# _") |
                  Pinf  ("+inf") |
                  Minf  ("-inf") |
                  Next  tbool ("++ _") |
                  Prev  tbool ("-- _")
datatype interval = oo  bound bound ("<_,_>") |
                    cc  bound bound ("[_,_]") |
                    oc  bound bound ("<_,_]") |
                    co  bound bound ("[_,_>") |
                    sng bound ("[_]")

consts
  win      :: [int, interval] => tbool
  after    :: [int, bound] => tbool
  before   :: [int, bound] => tbool
  sublft   :: [interval, int] => interval
  subrgt   :: [interval, int] => interval
  eq       :: [interval, interval] => bool (infixr 20)

primrec
  after_val  "(after x (# v))  = const (v <= x)"
  after_pinf "(after x (+inf)) = false"
  after_minf "(after x (-inf)) = true"
  after_next "(after x (++ N)) =
        (%v. (? t. #0 < t & t <= x & N (v+t) &
                (! t'. (#0 < t' & t'< t) --> ~N(v+t'))))"
  after_prev "(after x (-- P)) =
        (%v. (? t. t < #0  & t <= x & P (v+t) &
(! t'. (t < t' & t'< #0) --> ~P(v+t'))) |
     (! t'. t'<#0 --> ~P(v+t')))"
primrec
  before_val  "(before x (# v))  = const (x <= v)"
  before_pinf "(before x (+inf)) = true"
```

```
  before_minf "(before x (-inf)) = false"
  before_next "(before x (++ N)) =
        (%v. (? t. #0 < t & x <= t & N (v+t) &
(! t'. (#0 < t' & t'< t) --> ~N(v+t'))) |
     (! t'. #0<t' --> ~N(v+t')))"
  before_prev "(before x (-- P)) =
        (%v. ? t. t < #0  & x <= t & P (v+t) &
       (! t'. (t < t' & t'< #0) --> ~P(v+t')))"
primrec
  win_cc  "(win v [l,u]) = ((after v l) #& (before v u))"
  win_oo  "(win v <l,u>) = ((after (v-#1) l) #& (before (v+#1) u))"
  win_oc  "(win v <l,u]) = ((after (v-#1) l) #& (before v u))"
  win_co  "(win v [l,u>) = ((after v l) #& (before (v+#1) u))"
  win_sng "(win v [b])   = ((after v b) #& (before v b))"

primrec
  sublft_cc  "(sublft [l,u] v) = [l,#v>"
  sublft_oo  "(sublft <l,u> v) = <l,#v>"
  sublft_oc  "(sublft <l,u] v) = <l,#v>"
  sublft_co  "(sublft [l,u> v) = [l,#v>"
  sublft_sng "(sublft [b] v)   = [b,#v>"

primrec
  subrgt_cc  "(subrgt [l,u] v) = <#v,u]"
  subrgt_oo  "(subrgt <l,u> v) = <#v,u>"
  subrgt_oc  "(subrgt <l,u] v) = <#v,u]"
  subrgt_co  "(subrgt [l,u> v) = <#v,u>"
  subrgt_sng "(subrgt [b] v)   = <#v,b]"

defs
  eq_def "(i1 eq i2) == (!x. (win x i1)=(win x i2))"

end
```

C.3 TilcoX.thy

```
(*
    File:        TilcoX.thy
    Theory Name: TilcoX
    Logic Image: HOL
*)
```

```
TilcoX = IntervalX +
consts
  at       :: [tbool, interval] => tbool                ("_ @ _"  [36,37] 36)
  happen   :: [tbool, interval] => tbool                ("_ ? _"  [36,37] 36)
  happen1  :: [tbool, interval] => tbool                ("_ ?! _" [36,37] 36)

  "->>"    :: [tbool, tbool] => tbool                                (infixr 25)
  "-<<"    :: [tbool, tbool] => tbool                                (infixr 25)
  ":="     :: ['a tfun, 'a tfun] => tbool                            (infixr 38)
  until    :: [tbool, tbool] => tbool
  since    :: [tbool, tbool] => tbool

  up       :: tbool => tbool
  down     :: tbool => tbool
  tinv      :: ('a tfun) => tbool
  rule     :: tbool => bool
  fact     :: tbool => bool

syntax
  "&&"     :: [interval, interval] => interval                       (infixr 40)
  "||"     :: [interval, interval] => interval                       (infixr 40)
  "@tEval":: 'a tfun => 'a tfun ("tEval[_]")
translations
  "A @   (i &&  j)"  => "(A @   i) #& (A @   j)"
  "A ?   (i &&  j)"  => "(A ?   i) #& (A ?   j)"
  "A ?!  (i &&  j)"  => "(A ?!  i) #& (A ?!  j)"
  "A @   (i ||  j)"  => "(A @   i) #| (A @   j)"
  "A ?   (i ||  j)"  => "(A ?   i) #| (A ?   j)"
  "A ?!  (i ||  j)"  => "(A ?!  i) #| (A ?!  j)"

defs
  at_def       "A @  i       == (%x. ! t. (win t i) x --> A(x+t))"
  happen_def   "A ?  i       == (%x. ? t. (win t i) x & A(x+t))"
  Tnext_def    "A ->> B      == A #--> (B @ [##1])"
  Tprev_def    "A -<< B      == A #--> (B @ [##-1])"
  Tassign_def  "A := exp     == (%x. A x = exp (x+#-1))"
  until_def    "until P Q    == Q @ <##0,++P>"
  since_def    "since P Q    == Q @ <--P,##0>"
  up_def       "up A         == (%t. ~A(t+#-1) &  A t)"
  down_def     "down A       == (%t.  A(t+#-1) & ~A t)"
  tinv_def     "tinv A       == (%x.  A x = (A (x+#-1)) )"
  rule_def     "rule A       == (! t. now A t)"
  fact_def     "fact A       == (? t. now A t)"
```

```
end
```

C.4 TilcoXX.thy

```
(*
    File:        TilcoXX.thy
    Theory Name: TilcoXX
    Logic Image: HOL
*)

TilcoXX = TilcoX + List +
consts
  happenmn :: [tbool, nat, interval] => tbool        ("_ ?-_ _" [36,30,37] 36)
  happenmx :: [tbool, nat, interval] => tbool        ("_ ?^_ _" [36,30,37] 36)
  bhappen  :: [tbool, nat, nat, interval] => tbool
                                                ("_ ?[_ _] _" [36,30,30,37] 36)
  happeneq :: [tbool, nat, interval] => tbool
                                                ("_ ?=_ _" [36,30,37] 36)
  monotone :: int list => bool
translations
  "P ?= n I" => "P ?[n n] I"

primrec
  mono_nil "(monotone [])  = True"
  mono_seq "(monotone (a # l)) = (if l=[] then True
      else ((a < (hd l)) & monotone l))"

defs
  happenmn_def "(P ?- m I) ==
    (%t. ? (f::int list). (length f)=m &
               (!i. i<m --> P (t+(nth f i)) & (win (nth f i) I) t) &
               monotone f)"
  happenmx_def "(P ?^ M I) == (#~(P ?-(Suc M) I))"
  bhappen_def  "(P ?[m M] I) == ((P ?-m I) #& (P ?^M I))"

end
```

Bibliography

[1] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum. *Handbook of Logics in Computer Science, Vol.1.* Oxford Science Publications, 1992.

[2] M. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Transactions on Software Engineering*, 3(1), Jan. 1977.

[3] M. Alford. Srem at the age of eight; the distributed computing design system. *Computer*, April 1985.

[4] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, Nov. 1983.

[5] J. F. Allen and G. Ferguson. Actions and events in interval temporal logic. Technical report, University of Rochester Computer Science Department, TR-URCSD 521, Rochester, New York 14627, July 1994.

[6] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(9):213–249, July 1997.

[7] R. Alur. Logics and models of real time: A survey. Technical report, Dept. of Computer Science Cornell University, Ithaca, New York, USA, Jan. 1992.

[8] R. Alur and T. A. Henzinger. A really temporal logic. In *30th IEEE FOCS*, 1989.

[9] R. Alur and T. A. Henzinger. Real time logics: complexity and expressiveness. In *Proc. of 5th Annual IEEE Symposium on Logic in Computer Science, LICS 90*, pages 390–401, Philadelphia, USA, June 1990. IEEE.

[10] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. Technical report, Dept. of Comp. Science and Medicine STAN-CS-90-1307, Stanford University, Stanford, California, USA, March 1990.

[11] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth Through Proof.* Academic Press, Inc., Orlando, USA, 1986.

[12] M. A. Ardis, J. A. Chaves, L. J. Jagadeesan, P. Mataga, C. Puchol, M. G. Staskauskas, and J. VonOlnhausen. A framework for evaluating specification methods for reactive systems – experience report. *IEEE Transactions on Software Engineering*, 22(6):378–369, June 1996.

[13] J. Armstrong and L. Barroca. Specification and verification of reactive system behavior: The railroad crossing example. *Journal of Real-Time Systems*, 10:143–178, 1996.

[14] B. Auernheimer and R. A. Kemmerer. Rt-aslan: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 12(9):879–889, Sept. 1986.

[15] H. Barringer. *A Survey of Verification Techniques for Parallel Programs.* Lecture Notes in Computer Science 191, Springer Verlag, New York, 1985.

[16] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. Metatem: A framework for programming in temporal logic. In *in Proc. of REX Workshop on Stepwise Refinement of Distributed Systems: Models Formalism, Correctness*, Mook, Netherlands, June 1989. Springer Verlag, LNCS n.430.

[17] H. Barringer, M. Fisher, D. Gabbay, and A. Hunter. Meta-reasoning in executable temporal logic. In *Proc. of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, April 1991. Morgan Kaufmann.

[18] H. Barringer, M. Fisher, D. Gabbay, R. Owens, and M. Reynolds. *The Imperative Future: Principles of Executable Temporal Logic.* Research Studies Press, LTD, John Wiley & Sons Inc., New York, 1996.

[19] Z. Bavel. *Introduction to the Theory of Automata.* Reston Publishing Company, Prentice-Hall, Reston, Virginia, 1983.

[20] P. Bellini, M. Bruno, and P. Nesi. Verification criteria for a compositional model for reactive systems. *Proc. of the IEEE International Conference on Complex Computer Systems*, Sept. 11-15 2000.

[21] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *in press, ACM Computing Surveys*, December 1999.

[22] M. Ben-Ari. *Mathematical Logic for Computer Science.* Prentice Hall, New York, 1993.

[23] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20, 1983.

[24] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, Jan. 1984.

[25] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*, 14(1):25–29, 1987.

[26] G. Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, California, USA, 1991.

[27] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specifications. *Computer*, pages 82–91, April 1985.

[28] J. P. Bowen and M. J. C. Gordon. Z and hol,. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, pages 141–167. Workshops in Computing Series, Springer-Verlag, 1994.

[29] J. P. Bowen and M. J. C. Gordon. A shallow embedding of z in hol. *Information and Software Technology*, 37(5-6):269–276, May/June 1995.

[30] R. Braek and O. Haugen. *Engineering Real Time Systems: An object-oriented methodology using SDL.* Prentice Hall, New York, London, 1993.

[31] P. Brinch-Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[32] S. D. Brookes. On the relationship of ccs and csp. In J. Diaz, editor, *Proc. of the 10th Colloquium on Automata Languages and Programming, Lecture Notes in Computer Science LNCS N.154*, pages 83–96, Barcellona, Spain, July 1983. Springer Verlag.

[33] G. Bruno and R. Agarwal. Validating software requirements using operational models. In *Second Symposium on Software Quality Techniques and Acquisition Criteria*, Florence, Italy, 1995.

[34] M. Bruno and P. Nesi. Verification of external specifications of reactive systems. *IEEE Transactions on System Man and Cybernetics*, pages submitted for publication, also

Technical Report of Department of Systems and Informatics, University of Florence, 1998.

[35] G. Bucci, M. Campanai, and P. Nesi. Tools for specifying real-time systems. *Journal of Real-Time Systems*, 8:117–172, March 1995.

[36] G. Bucci, M. Campanai, P. Nesi, and M. Traversi. An object-oriented case tool for reactive system specification. In *Proc. of 6th International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE)*, Le CNIT, Paris la Defense, France, 15-19 Nov. 1993.

[37] S. N. Cant, D. R. Jeffery, and B. Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. Technical report, University of New South Wales, Information Technology Research Centre, n.57, New South Wales 2033, Australia, Oct. 1991.

[38] P. P. Chen. The entity relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9, March 1976.

[39] A. Church. A formulation of the simple theory of types. *JOURNAL of Symbolic Logic*, (5):56–68, 1940.

[40] E. Clarke. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc of IBM Workshop on Logic of Programs*. Springer Verlag, 1981.

[41] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, April 1986.

[42] E. M. Clarke and O. Grumberg. The model checking problem for concurrent systems with many similar processes. In B. Banieqbal, H. Barringer, and P. Pnueli, editors, *Proc. of the International Conference on Temporal Logic in Specification*, pages 188–201, Altrincham, UK, 8-10 April 1987. Springer Verlag, LNCS n.398.

[43] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, New Jersey, USA, 1991.

[44] D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, Jan. 1992.

[45] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjaming/Cumminggs, Publ. Co., 1986.

[46] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, Jan. 1985.

[47] L. S. Davis, D. DeMenthon, T. Bestul, D. Harwood, H. V. Srinivasan, and S. Ziavras. Rambo - vision and planning on the connection machine. In *Proc. of 6th Scandinavian Conference on Image and Application, Oulu, Finland*, pages 1–14, 19-22 June 1989.

[48] R. E. Davis. *Truth, Deduction, and Computation: Logic and Semantics for Computer Science*. Computer Science Press, New York, 1989.

[49] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, NY, USA, 1968.

[50] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[51] M. Dorfman. System and software requirements engineering. In H.Thayer and M.Dorfman, editors, *System and Software Requirements Engineering*, pages 4–16. IEEE Compute Society Press, Los Alamitos CA, 1990.

[52] E. A. Emerson and J. Y. Halpern. Sometimes and not never revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, Jan. 1986.

[53] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proc of the Workshop on Finite-State Concurrency*, Grenoble, 1989.

[54] V. Encontre, E. Delboulbe, P. Gabaud, P. Leblanc, and A. Baussalem. Combining services, message sequence charts and sdl: Formalism methods and tools. Technical report, Verilog, 1990.

[55] M. Felder, D. Mandrioli, and A. Morzenti. Proving properties of real-time systems through logical specifications and petri net models. Technical report, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 91-072, Piazza Leonardo da Vinci 32, Milano, Italy, 1991.

[56] M. Felder and A. Morzenti. Validating real-time systems by history-checking trio specifications. In *Proc. of 14th International Conference on Software Engineering*, pages 199–211, Melbourne, Australia, 11-15 May 1992. IEEE press, ACM.

[57] M. Felder and A. Morzenti. Validating real-time systems by history-checking trio specifications. *ACM Transactions on Software Engineering and Methodology*, 3(4):308–339, Oct. 1994.

[58] M. Finger, M. Fisher, and R. Owens. Metamem at work: Modeling reactive systems using executable temporal logic. In *Proc. of the 6th International Conference on Industrial and Engineering Applications if Artificial Intelligence and Expert Systems (IEA-AIE*, Edinburg, UK, June 1993. Gordon and Breach.

[59] M. Fisher and R. Owens. From the past to the future: Executing temporal logic programs. In *Proc. of the Conference on Logic Programming and Automated Reasoning (LPAR)*, St. Petersberg, Russia, July 1992. Springer Verlarg, LNCS N.624.

[60] M. Fisher and R. Owens. An introduction to executable modal and temporal logics. In M. Fisher and R. Owens, editors, *Executable Modal and Temporal Logics: Proceedings of the IJCAI '93 Workshop, Chamberry, France, August 1993*, pages 1–20. Lecture Notes in Artificial Intelligence, Springer Verlag LNCS 897, 1995.

[61] GEODE. Age/geode editor, user's manual, ver.1.4. Technical report, Verilog, avenue Artistide Briand, 52, 92220 Bagneaux, France, 1992.

[62] R. Gerber and I. Lee. Communicating shared resources: A model for distributed real-time systems. In *Proc. of the 10th IEEE Real-Time Systems Symposium*, pages 68–78. IEEE Computer Society Press, Dec. 1989.

[63] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.

[64] J. Guttag. Abstract data types and development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[65] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *ACTA Informatica*, 10, 1978.

[66] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[67] J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In J. Diaz, editor, *Proc. of the 10th Colloquium on Automata Languages and Programming, Lecture Notes in Computer Science LNCS N.154*, pages 278–291, Barcellona, Spain, July 1983. Springer Verlag.

[68] J. Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. In *Proc. of the 1st IEEE Symp. on Logic in Computer Science*, pages 274–292. IEEE Press, 1986.

[69] D. Harel. On visual formalism. *Communications of the ACM*, 31(5):514–530, May 1988.

[70] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, jUNE 1996.

[71] C. Heitmeyer and D. *Formal Methods for Real-Time Computing.* John Wiley & Sons., England, 1996.

[72] H. B. Henderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[73] C. A. R. Hoare. Towards a theory of parallel programming,. In *Operating Systems Techniques*, pages 61–71. Academic Press, NY, USA, 1972.

[74] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[75] C. A. R. Hoare. A calculus of total correctness for communicating processes. *Sci. Comput. Program.*, 1:49–72, 1981.

[76] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, NY, USA, 1985.

[77] HOOD. An overview of the hood toolset. Technical report, Software Sciences, 1988.

[78] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen, London, 1968.

[79] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdhal, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[80] F. Jahanian and A. K.-L. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, Sept. 1986.

[81] B. Josko. Mctl: An extension of ctl for modular verification of concurrent systems. In B. Banieqbal, H. Barringer, and P. Pnueli, editors, *Proc. of the International Conference on Temporal Logic in Specification*, pages 165–187, Altrincham, UK, 8-10 April 1987. Springer Verlag, LNCS n.398.

[82] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems Journal*, 2:255–299, 1990.

[83] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Number 651. Lecture Notes in Computer Science,Springer-Verlag, 1992.

[84] R. Koymans, R. K. Shyamasundar, W. P. deRoever, R. Gerth, and A. Arun-Kumar. Compositional semantics for real-time distributed computing. In *Proc. of Logics of Programs, Lecture Notes in Computer Sciences, LNCS 193*, New York, 1985. Springer Verlag. vedi levi90.

[85] P. Ladkin. Models of axioms for time intervals. In *Proc. of the 6th National Conference on Artificial Intelligence, AAAI'87*, pages 234–239, USA, 1987.

[86] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, Jan. 1989.

[87] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1993.

[88] P. Laplante. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, New York, 1992.

[89] S.-T. Levi and A. K. Agrawala. *Real-Time System Design*. McGraw-Hill Publishing Company, New York, USA, 1990.

[90] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept. 1995.

[91] D. Maier and D. S. Warren. *Computing with Logic*. The Benjamin/Cummings, Inc., Menlo Park, CA, USA, 1988.

[92] D. Mandrioli. The specification of real-time systems: a logical object-oriented approach. In *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS'92*, 1992.

[93] D. Mandrioli. The object-oriented specification of real-time systems. In *Tutorial Note of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Europe '93*, Versailles, France, 8-11 March 1993.

[94] Z. Manna and A. Pnueli. Proving precedence properties: The temporal way. In J. Diaz, editor, *Proc. of the 10th Colloquium on Automata Languages and Programming, Lecture Notes in Computer Science LNCS N.154*, pages 491–512, Barcelona, Spain, July 1983. Springer Verlag.

[95] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th Symposium on Principles of Distributed Computing, Quebec, Canada, Aug. 1990*, pages 377–408, 1990.

[96] J. Martin and J. Odell. *Object Oriented Analysis and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[97] R. Mattolini. *TILCO: a Temporal Logic for the Specification of Real-Time Systems (TILCO: una Logica Temporale per la Specifica di Sistemi di Tempo Reale).* PhD thesis, 1996.

[98] R. Mattolini and P. Nesi. Using tilco for specifying real-time systems. *Proc. of the 2nd IEEE Intl. Conference on Engineering of Complex Computer Systems, Montreal (Quebec, Canada)*, pages 18–25, 21-25 Oct. 1996.

[99] R. Mattolini and P. Nesi. An interval logic for real-time system specification. *IEEE Transactions on Software Engineering, in press*, 1999.

[100] R. A. McCauley and W. R. Edwards. Analysis and measurement techniques for logic-based languages. In *Proc. of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, Florence, Italy, 8-11 March 1998. IEEE Press.

[101] P. M. MelliarSmith. Extending interval logic to real time systems. In *Proc. of Temporal Logic Specification United Kingdom, (B. Banieqbal, H. Barringer, A. Pnueli, eds)*, pages 224–242. Springer Verlag, Lecture Notes in Computer Sciences, LNCS 398, April 1987.

[102] S. Merz. Efficiently executable temporal logic programs. In M. Fisher and R. Owens, editors, *Executable Modal and Temporal Logics: Proceedings of the IJCAI '93 Workshop, Chamberry, France, August 1993*, pages 1–20. Lecture Notes in Artificial Intelligence, Springer Verlag LNCS 897, 1995.

[103] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92, Springer Verlag, New York, 1980.

[104] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[105] D. E. Monarchi and G. I. Puhr. A research typology for object oriented analysis and design. *Communications of the ACM*, 35(9):35–47, Sept. 1992.

[106] A. P. Moore. The specification and verified decomposition of system requirements using csp. *IEEE Transactions on Software Engineering*, 16(9):932–948, Sept. 1990.

[107] L. E. Moser, Y. S. Ramakrishna, G. Kutty, Melliar-Smith, and L. K. Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology*, 6(1):31–79, Jan. 1997.

[108] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, pages 10–19, Feb. 1985.

[109] B. Moszkowski and Z. Manna. Reasoning in interval logic. In *Proc. of ACM/NSF/ONR workshop on LOgics of Programs LNCS 164, Lecture Notes in Computer Science*, pages 371–384. Springer Verlag, 1984.

[110] B. C. Moszkowski. *Executing Temporal Logic Programs*. PhD thesis, Cambridge University, 1986.

[111] D. R. Musser. Abstract data type specification in the affirm system. *IEEE Transactions on Software Engineering*, 6(1):24–32, Jan. 1980.

[112] P. Nesi and M. Campanai. Metric framework for object-oriented real-time systems specification languages. *The Journal of Systems and Software*, 34:43–65, 1996.

[113] E.-R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. In J. Diaz, editor, *Proc. of the 10th Colloquium on Automata Languages and Programming, Lecture Notes in Computer Science LNCS N.154*, pages 561–572, Barcelona, Spain, July 1983. Springer Verlag.

[114] E. R. Olderog and C. A. R. Hoare. Specification oriented semantics for communicating sequential process. *ACTA Informatica*, 23:9–66, 1986.

[115] J. S. Ostroff. *Temporal Logic for Real-Time Systems.* Research Studies Press LTD., Advanced Software Development Series, 1, Taunton, Somerset, England, 1989.

[116] J. S. Ostroff. Verification of safety critical systems using ttm/rttl. In *Proc. of the REX Workshop on Real-Time: Theory in Practice, LNCS 600.* Springer Verlag, 1991.

[117] J. S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, pages 33–60, April 1992.

[118] J. S. Ostroff and W. Wonham. Modeling and verifying real-time embedded computer systems. In *Proc. of the 8th IEEE Real-Time Systems Symposium*, pages 124–132. IEEE Computer Society Press, Dec. 1987.

[119] J. S. Ostroff and W. M. Wonham. A framework for real-time discrete event control. *IEEE Transactions on Automatic Control*, 35(4):386–397, April 1990.

[120] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Ochifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[121] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[122] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Lecture Notes in Computer Science, Springer Verlag LCNS 828, 1994.

[123] L. C. Paulson. Isabelle's object-logics. Technical Report 286, Computer Laboratory, University of Cambridge U.K., 1994.

[124] A. Pnueli. The temporal logic of programs. In *18th IEEE FOCS*, 1977. mattolini.

[125] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13, 1981.

[126] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency.* Lecture Notes in Computer Science, Springer Verlag LNCS 224, 1986.

[127] A. Prior. *Past, Present and Future.* Oxford University Press, London, 1967.

[128] R. R. Razouk and M. M. Gorlick. A real-time interval logic for reasoning about executions of real-time programs. In *Proc. of the ACM/SIGSOFT'89, Tav.3*, pages 10–19. ACM Press, Dec. 1989.

[129] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Proc. of ICALP'86, Lecture Notes in Computer Science, LNCS N. 226*, pages 314–323. Springer Verlag, 1986.

[130] W. Reisig. *Petri Nets. An introduction.* EATCS Monographs on Theoretical Computer Science, Springer Verlag, New York, 1985.

[131] R. Rosner and A. Pnueli. A choppy logic. In *Proc. of the 1st IEEE Symp. on Logic in Computer Science*, pages 306–313. IEEE Press, 1986.

[132] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, Englewood Cliffs, New Jersey, 1991.

[133] R. L. Schwartz and P. M. Melliar-Smith. From state machines to temporal logic: Specification methods for protocol standards. *IEEE Transactions on Communications*, 30(12):2486–2496, Dec. 1982.

[134] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. A interval logic for higher-level temporal reasoning. In *Proc. of the 2nd Annual ACM Symp. Principles of Distributed Computing, Lecture Notes in Computer Science, LNCS N. 164*, pages 173–186, Montreal Canada, 17-19 Aug. 1983. Springer Verlag, ACM NewYork.

[135] A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, Sept. 1992.

[136] S. Shlaer and S. J. Mellor. *Object Oriented Analysis: Modeling the World in Data.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1988.

[137] S. Shlaer and S. J. Mellor. *Object Life Cycles: Modeling the World in States.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[138] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, pages 10–19, Oct. 1988.

[139] J. A. Stankovic and K. Ramamritham. *Advances in Real-Time Systems.* IEEE Computer Society Press, Washington, 1992.

[140] C. Stirling. Comparing linear and branching time temporal logics. In B. Banieqbal, H. Barringer, and P. Pnueli, editors, *Proc. of the International Conference on Temporal Logic in Specification*, pages 1–20, Altrincham, UK, 8-10 April 1987. Springer Verlag, LNCS n.398.

[141] C. A. Sunshine, D. H. Thompson, R. W. Erickson, S. L. Gerhart, and D. Schwabe. Specification and verification of communication protocols in affirm using state transition models. *IEEE Transactions on Software Engineering*, 8(5):460–489, Sept. 1982.

[142] H. Thayer and M. Dorfman. *System and Software Requirements Engineering.* IEEE Compute Society Press, Los Alamitos CA, 1990.

[143] J. Thistle and W. M. Wonham. Control problems in a temporal logic framework. *International Journal of Control*, 44(4), 1986.

[144] I. J. Walker. Requirements of an object-oriented design method. *Software Engineering Journal*, pages 102–113, March 1992.

[145] F. Wang, A. K. Mok, and A. Emerson. Distributed real-time system specification. *ACM Transactions on Software Engineering and Methodology*, 2(4):346–378, Oct. 1993.

[146] J. M. Wing. A specifier's introduction for formal methods. *Computer*, pages 8–24, Sept. 1990.

[147] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: a responsibility-driven approach. In *Proc. OOPSLA'89, Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 71–75, New Orleans, Louisiana,, Oct. 1989. SIGPLAN NOT, ACM Press.

[148] R. J. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object Oriented Software.* Prentice Hall, Englewood Cliffs, N.J., USA, 1990.

[149] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, May 1982.

[150] P. Zave. A comparison of the mayor approaches to software specification and design. In H.Thayer and M.Dorfman, editors, *System and Software Requirements Engineering*, pages 197–199. IEEE Compute Society Press, Los Alamitos CA, 1990.

[151] H. Zuse. *Software Complexity: measures and methods.* Walter de Cruyter, Berlin, New-York, 1991.