The ForSpec Temporal Logic: A New Temporal Property-Specification Language*

Roy Armoni[1], Limor Fix[1], Alon Flaisher[1], Rob Gerth[2], Boris Ginsburg, Tomer Kanza[1], Avner Landver[1], Sela Mador-Haim[1], Eli Singerman[1], Andreas Tiemeyer[1], Moshe Y. Vardi**[3], and Yael Zbar[1]

[1] Intel Strategic CAD Labs
[2] Intel Israel Development Center
[3] Rice University

Abstract. In this paper we describe the *ForSpec Temporal Logic* (FTL), the new temporal property-specification logic of *ForSpec*, Intel's new formal specification language. The key features of FTL are as follows: it is a linear temporal logic, based on Pnueli's LTL, it is based on a rich set of logical and arithmetical operations on bit vectors to describe state properties, it enables the user to define temporal connectives over time windows, it enables the user to define regular events, which are regular sequences of Boolean events, and then relate such events via special connectives, it enables the user to express properties about the past, and it includes constructs that enable the user to model multiple clock and reset signals, which is useful in the verification of hardware design.

1 Introduction

One of the most significant recent developments in the area of formal verification is the discovery of algorithmic methods, called *model checking*, for verifying temporal-logic properties of *finite-state* systems. Model-checking tools have enjoyed a substantial and growing use over the last few years, showing ability to discover subtle flaws that result from extremely improbable events. While until recently these tools were viewed as of academic interest only, they are now routinely used in industrial applications [Kur97]. Several model-checking tools are widely used in the semiconductor industry: SMV, a tool from Carnegie Mellon University with many industrial incarnations (e.g., IBM's RuleBase); VIS, a tool developed at the University of California, Berkeley FormalCheck, a tool developed at Bell Labs and marketed by Cadence; and Forecast, a tool developed in Intel and is used for property and equivalence formal verification.

A key issue in the design of a model-checking tool is the choice of the formal specification language used to specify properties, as this language is one of the *primary* interfaces to the tool. [Kur97]. (The other primary interface is the modeling language, which is typically the hardware description language used by the designers). In designing a formal specification language one has to balance several competing needs:

* A longer version of this paper can be found at `www.cs.rice.edu/~vardi/papers/`

** Supported in part by NSF grants CCR-9700061, CCR-9988322, IIS-9908435, IIS-9978135, and EIA-0086264, by BSF grant 9800096, and by a grant from the Intel Corporation.

J.-P. Katoen and P. Stevens (Eds.): TACAS 2002, LNCS 2280, pp. 296–311, 2002.

- Expressiveness: The language has to be expressive enough to cover most properties likely to be used by verification engineers. This should include not only properties of the unit under verification but also relevant properties of the unit's environment.
- Usability: The language should be easy to understand and to use for verification engineers. This rules out expressive languages such as the μ-calculus, [Koz83], where alternation of fixpoints is notoriously difficult to understand. At the same time, it is important that the language has a rigorous formal semantics to ensure correct compilation and optimization and enable formal reasoning.
- Compositionality: The language should enable the expression of complex properties from simpler one. This enables maintaining libraries of properties and property templates. Thus, we believe that the language should be closed under *all* of its logical connectives, both Boolean and temporal, enabling property reuse. The language should also enable modular reasoning, since current semiconductor designs are exceedingly complex and are amenable only to modular approaches.
- Implementability: The design of the language needs to go hand-in-hand with the design of the model-checking tool. In considering various language features, one should balance their importance against the difficulty of ensuring that the implementation can handle these features.
- Dynamic Validation: Once a property is specified, it should be used by both the formal verification capabilities and by the dynamic validation, i.e., simulation, capabilities. Thus, a property must have the same semantics (meaning) in both dynamic and formal validation and should also be efficient for both.

In spite of the importance of the formal specification language, the literature on the topic is typically limited to expressiveness issues (see discussion below). In this paper we describe FTL, the temporal logic underlying *ForSpec*[1], which is Intel's formal specification language. The key features of FTL are as follows:

- FTL is a *linear* temporal logic, based on Pnueli's LTL, [Eme90],
- it is based on a rich set of logical and arithmetical operations on bit vectors to describe state properties (since this feature is orthogonal to the temporal features of the logic, which are the main focus of this paper, it will not be discussed here),
- it enables the user to define temporal connectives over time windows,
- it enables the user to define regular events, which are regular sequences of Boolean events, and then relate such events via special connectives,
- it enables the user to refer to the history of the computation using past connectives, and
- it includes constructs that enable the users to model multiple clocks and reset signals, which is useful in the verification of hardware design.
- it has a rigorous formal semantics for model verification and simulation, and the complexity of model checking and reasoning is well understood.

The design of FTL was started in 1997. FTL 1.0, with its associated tools, was released to Intel users in 2000. FTL 2.0 is currently under design, in collaboration with

[1] ForSpec includes also a modeling language. It also includes various linguistic features, such as *parameterized templates*, which faciliates the construction of standard property libraries.

language design teams from Co-Design Automation, Synopsys, and Verisity Design, and is expected to be realesed in 2002[2]. The language described in this paper is close to FTL 2.0, though some syntactical and semantical details may change by the final release. The goal of this paper is not to provide a full documentation of FTL, but rather to describe its major features, as well as explain the rationale for the various design choices. Our hope is that the publication of this paper, concomitant with the release of the language to users, would result in a dialog on the subject of property-specification logic between the research community, language developers, and language users.

2 Expressiveness and Usability

2.1 The Nature of Time

Two possible views regarding the nature of time induce two types of temporal logics. In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal formulae are interpreted over linear sequences, and we regard them as describing the behavior of a single computation of a system. In *branching* temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal formulae are interpreted can be viewed as infinite *computation trees*, each describing the behavior of the possible computations of a nondeterministic system.

In the linear temporal logic LTL [Eme90], formulas are composed from the set of propositional constants using the usual Boolean connectives as well as the temporal operators G ("always"), F ("eventually"), X ("next"), and U ("until"). In the branching temporal logic CTL [Eme90], every temporal operator is preceded by the *universal* path quantifier A or the *existential* path quantifier E. Most incarnations of SMV, (see [BBL98] and [CCGR00]), as well as VIS, use CTL, or extensions of it, as their property specification logic. In contrast, Cadence SMV uses LTL, while FormalCheck uses a set of property templates that collectively has the expressive power of ω-automata (also a linear-time formalism) as its property specification language [Kur98]. Temporal ***e***, a property-specification language used in simulation, is also a linear-time formalism [Mor99]. Our first decision in designing a property-specification logic for the next generation Intel model-checking tool was the choice between the linear- and branching-time approaches.

We chose the linear-time approach after concluding that the branching-time framework suffers from several inherent deficiencies as a framework for property specification logics (for a more thorough discussion of these issues see [Var01]):

- Verification engineers find branching time unintuitive. IBM's experience with RuleBase has been that "nontrivial CTL equations are hard to understand and prone to error" [SBF+97].
- The branching-time framework was designed for reasoning about *closed* systems. Bran-ching-time modular reasoning is exceedingly hard [Var01].

[2] See press release at `http://biz.yahoo.com/bw/011105/50371_1.html`.

– Combining formal and dynamic validation (i.e., simulation) techniques for branching-time model checking is possible only in a very limited fashion, as dynamic validation is inherently linear.

LTL includes several connectives that handle unbounded time. For example, the formula eventually p asserts that p will hold at some point in the future. There's, however, only one connective in LTL to handle bounded time. To assert that p will hold at time 5, one has to write next next next next next p. Asserting that p will hold between time 10 and time 15, is even more cumbersome. By using *time windows* FTL provides users with bounded-time temporal connectives. (Bounded temporal connectives are typically studied in the context of real-time logics, cf. [AH92]). For example, to express in FTL that p will hold between time 10 and time 15, one simply writes eventually $[10, 15]$ p.

2.2 ω-Regularity

Since the proposal by Pnueli in 1977 to apply LTL to the specification and verification of concurrent programs, the adequacy of LTL has been widely studied. One of the conclusions of this research is that LTL is not expressive enough for the task. The first to complain about the expressive power of LTL was Wolper [Wol83] who observed that LTL cannot express certain ω-regular events (in fact, LTL expresses precisely the star-free ω-regular events [Eme90]).

The weakness in expressiveness is not just a theoretical issue. Even when attempting to verify simple properties, users often need to express relevant properties, which can be rather complex, of the environment of the unit under verification. These properties are the *assumptions* of the Assume-Guarantee paradigm (cf. [Pnu85]). Assume-guarantee reasoning is a key requirement at Intel, due to the high complexity of the designs under verification. Thus, the logic has to be able to express assumptions that are strong enough to imply the assertion under verification. In other words, given models M and E and a property ψ that $E|M \models \psi$ (we use $|$ to denote *parallel composition*), the logic should be able to express a property ψ such that $E \models \varphi$ and $M \models \varphi \rightarrow \psi$. As was shown later [LPZ85], this makes LTL inadequate for *modular* verification, since LTL is not expressive enough to express strong enough assumptions about the environment. It is now recognized that a linear temporal property logic has to be expressive enough to specify all ω-regular properties [Pnu85]. Several extensions to LTL have been proposed with the goal of obtaining full ω-regularity: (a) Vardi and Wolper proposed ETL, the extension of LTL with temporal connectives that correspond to ω-automata [Wol83, VW94]), (b) Banieqbal and Barringer proposed extending LTL with fixpoint operators [BB87], yielding a linear μ-calculus, (cf. [Koz83]), and (c) Sistla, Vardi, and Wolper proposed QPTL, the extension of LTL with quantification over propositional variables [SVW87].

As fixpoint calculi are notoriously difficult for users, we decided against the fixpoint approach. Keeping the goal of implementability in mind, we also decided against a full implementation of ETL and QPTL, as full QPTL has a nonelementary time complexity [SVW87], while implementing full ETL, which explicily incorporates Büchi automata, requires a complementation construction for Büchi automata, still a topic under active

research [Fin01]. Instead, FTL borrows from ETL, as well as PDL [FL79], by extending LTL with regular events[3]. As we show later, this extension provides us with full ω-regularity.

2.3 The Past

There are two reasons to add past connectives to temporal logic. The first argument is that while past temporal connectives do not add any expressive power, the price for eliminating them can be high. Many natural statements in program specification are much easier to express using past connectives. In fact, the best known procedure to eliminate past connectives may cause an exponential blow-up of the considered formulas.

A more important motivation for the restoration of the past is again the use of temporal logic in modular verification. In global verification one uses temporal formulas that refer to locations in the program text. This is absolutely *verboten* in modular verification, since in specifying a module one can refer only to its external behavior. Since we cannot now refer to program location we have instead to refer to the history of the computation, and we can do that very easily with past connectives This lead to the study of extensions of temporal logic with past connectives (cf. [KPV01]). These extensions allow arbitrary mixing of future and past connectives, enabling one to say, for example, "eventually, sometimes in the past, p holds until q holds".

For the same motivation, FTL also includes past connectives. Unlike the aforementioned extensions of LTL with past connectives, the usage of past connectives is circumscribed. We found little practical motivation to allow arbitrary mixing of future and past connectives, and such mixing has a nonnegligible implementation cost[4]. In fact, since the motivation of adding the past is to enable referring to the history of the computation, FTL's past connectives allow only references to such history. Thus, FTL allows references to past values of Boolean expressions and regular sequences of Boolean expressions. FTL does not allow, however, references to past values of temporal formulas. Thus, past connectives in FTL are viewed as Boolean rather than temporal connectives.

2.4 Hardware Features

While the limited mechanisms for automata connectives and quantification over propositional variables are sufficient to guarantee full ω-regularity, we decided to also offer direct support to two specification modes often used by verification engineers at Intel: *clocks* and *resets*. Both clocks and resets are features that are needed to address the fact that modern semiconductor designs consists of interacting parallel modules. As we shall see, while clocks and resets have a simple underlying intuition, explained below, defining their semantics formally is rather nontrivial.

Today's semiconductor design technology is still dominated by synchronous design methodology. In synchronous circuits, clocks signals synchronize the sequential

[3] For other extensions of LTL with regular events see [ET97,HT99]. These works propose adding regular events to the until operator. We prefer to have explicit connectives for regular events, in the style of the "box" and "diamond" modalities of PDL.

[4] It requires using two-way alternating automata, rather then one-way alternating automata [KPV01].

logic, providing the designer with a simple operational model. While the asynchronous approach holds the promise of greater speed, designing asynchronous circuits is significantly harder than designing synchronous circuits. Current design methodology attempt to strike a compromise between the two approaches by using multiple clocks. This methodology results in architectures that are globally asynchronous but locally synchronous. The temporal-logic literature mostly ignores the issue of explicitly supporting clocks (clocks, however, are typically studied in the context of modelling languages, see [CLM98]). Liu and Orgun [LO99] proposed a temporal framework with multiple clocks. Their framework, however, supports clocks via a clock calculus, which is separate from the temporal logic. Emerson and Trefler [ET97] proposed an extension of LTL in which temporal connectives can be indexed with multiple independent clocks.

In contrast, the way clocks are being used in FTL is via the *current clock*. Specifically, FTL has the construct change_on $c\ \varphi$, which states that the temporal formula φ is to be evaluated with respect to the clock c; that is, the formula φ is to be evaluated in the trace defined by the high phases of the clock c. The key feature of clocks in FTL is that each subformula may advance according to a different clock.

Another aspect of the fact that modern designs consist of interacting parallel modules is the fact that a process running on one module can be reset by a signal coming from another module. Reset control has long been a critical aspect of embedded control design. FTL directly supports reset signals. The formula accept_on $a\ \varphi$ states that the property φ should be checked only until the arrival of the reset signal a, at which point the check is considered to have *succeeded*. In contrast, reject_on $r\ \varphi$ states that the property φ should be checked only until the arrival of the reset signal r, at which point the check is considered to have *failed*. The key feature of resets in FTL is that each subformula may be reset (positively or negatively) by a different reset signal.

3 Basic Features

Core FTL: Formulae of FTL are built from a set Prop of propositional constants. (Actually, FTL provides a rich set of logical and arithmetical operations on bit vectors to describe state properties. All vector expressions, however, are compiled into Boolean expressions over Prop, so in this paper we suppress that aspect of the language). FTL is closed under the application of Boolean connectives (we use !, ||, and &&, implies , iff for negation, disjunction, and conjunction, implication, and equivalence, respectively). FTL also enables us to refer to future and past values of Boolean expressions. For a Boolean expression b, we refer to the value of b in the next phase[5] by future (b); this is essentially the "primed" version of b in Lamport's Temporal Logic of Actions. For a Boolean expression b, FTL uses past (b, n), where n is a positive integer, to refer to the value of b, n phases in the past. Note that past and future are Boolean connectives, so, for example, "ack implies past $(send, 10)$" is also a Boolean expression. Note also that the future connective is more limited than the past connective; one cannot write future $(b, 2)$. The rationale for that is that future is a somewhat nonstandard Boolean connective, since the value of future (p) is indeterminate at a given point in time. The

[5] Note that the notion of next phase is independent from the notion of the next clock tick, which is discussed later in the paper.

role of future is mostly to define *transitions*, e.g., $(!b)\&\&$ future (b) holds at points at which b is about to rise. FTL is also closed under the temporal connectives next , wnext , eventually , globally , until and wuntil .

FTL is interpreted over *computations*. A *computation* is a function $\pi : N \rightarrow 2^{\mathsf{Prop}}$, which assigns truth values to the elements of Prop at each time instant (natural number). The semantics of FTL's temporal connectives is well known. For example, for a computation π and a point $i \in \omega$, we have that:

- $\pi, i \models p$ for $p \in$ Prop iff $p \in \pi(i)$.
- $\pi, i \models$ future (b) iff $\pi, i+1 \models b$.
- $\pi, i \models$ past $(b, 1)$ iff $i > 0$ and $\pi, i-1 \models b$.
- $\pi, i \models$ past (b, n) iff $i > 0$ and $\pi, i-1 \models$ past $(b, n-1)$.
  Note that the past operator cannot go "before 0". Thus, $\pi, 0 \not\models$ past $(b, 1)$. In other words, the default past value is 0.
- $\pi, i \models \varphi$ until ψ iff for some $j \geq i$, we have $\pi, j \models \psi$ and for all k, $i \leq k < j$, we have $\pi, k \models \varphi$.

We say that a computation π *satisfies* a formula φ, denoted $\pi \models \varphi$, if $\pi, 0 \models \varphi$. We use $models(\varphi)$ to denote the set of computations satisfying φ.

Time windows modify the temporal connectives by specifying intervals in which certain events are expected to occur. The aim is to enable the expression of assertions such as globally $(req \rightarrow$ next $[10]$ $ack)$. The syntax of temporal connectives with intervals is as follows:

1. next $[m]$ φ, where $1 \leq m < \infty$
2. eventually $[m, n]\varphi$, globally $[m, n]\varphi$, φ until $[m, n]\psi$, and φ wuntil $[m, n]\psi$ where $0 \leq m \leq n \leq \infty$ (but $m = n = \infty$ is not allowed)

The semantics of temporal connectives with time windows is defined by induction on the window lower bound. For example,

- $\pi, i \models$ next $[1]$ φ if $\pi, i+1 \models \varphi$
- $\pi, i \models$ next $[m]$ φ, for $m > 1$, if $\pi, i+1 \models$ next $[m-1]$ φ
- $\pi, i \models \varphi$ until $[0, n]$ ψ if for some k, $n \geq k \geq 0$, we have that $\pi, i+k \models \psi$ and for all j, $0 \leq j < k$, we have that $\pi, i+j \models \varphi$.
- $\pi, i \models \varphi$ until $[m, n]$ ψ if $\pi, i+1 \models \varphi$ until $[m-1, n-1]$ ψ

Clearly, time windows do not add to the expressive power of FTL, since formulae with time windows can be translated to formulae without windows. Since the lower and upper bounds of these windows are specified using decimal numbers, eliminating time windows may involve an exponential blow-up. (See discussion below of computational complexity.)
The practical implication of this result is that users should be careful with time windows. The formula next $[m]$ p could force the compiler to introduce m BDD variables, so large windows could result in "state explosion".

Regular events: *Regular events* describe regular sequences of *Boolean events*, where Boolean events are simply Boolean expressions in terms of propositional constants. For example, the regular expression $(send, (!ack)^*, send)$ (comma denotes concatenation) describes a sequence in which there is no ack event between two $send$ events.

Thus, the formula globally $!(send, (!ack)^*, send)$ forbids the occurrence of such sequences. Regular events are formed from Boolean events (denoted here with a_i, b_i, etc.) using the follwing constructs[6]: (a) ,: concatenation, (b) $\backslash$: glue ($a_1 \ldots a_m \backslash b_1 \ldots b_n$ is $a_1 \ldots a_m \&\& b_1 \ldots b_n$), (c) $||$: union, (d) $a*$: zero or more, (e) $a+$: one or more, and (f) $e\{m, n\}$: between m and n occurrences of e. With each regular event e we associate, in the standard way, a language $L(e)$ of finite words over the Boolean events. (It is important to note that our semantics refers only to $L(e)$ and not to the syntax of e. Thus, adding further regular operators can be accomplished without any need to adapt the semantics. (The ForSpec compiler only requires a translation of regular expression to finite-state automata.) In this respect FTL differs from other property-specification languages, such as Sugar [BBDE$^+$01] or Temporal ***e*** [Mor99], where the syntax of regular events is intertwined with the semantics of the logic.) We do not allow regular events whose language contains the empty word (this is checked by the compiler).

We define the semantics of regular events in terms of *tight satisfaction*: $\pi, i \models e$ if for some $j \geq i$ we have $\pi, i, j \models\!\!\!\models e$. The intuition of tight satisfaction is that the event e holds *between* the points i and j. Formally, $\pi, i, j \models\!\!\!\models e$ if there is a word $w = b_0 b_1 \ldots b_n \in L(e)$, $n = j - i$ such that $\pi, i + k \models b_k$ for all k, $0 \leq k \leq n$.

We can also use regular events to refer to the past. If e is a regular expression, then ended (e) is a Boolean expression saying that the regular event e has just "ended". Formally $\pi, i \models$ ended (e) if there is some $j < i$ such that $\pi, j, i \models\!\!\!\models e$. Note that j need not to unique. For example, ended $(\mathsf{true}, (\mathsf{true}, \mathsf{true})*)$ holds at all even $(0, 2, \ldots)$ time points.

Since we have defined $\pi, i \models e$, we can now combine regular events with all other connectives. Thus, globally $!(req, (!ack)^*, ack)$ says that the computation cannot contain a sequence of states that match $(req, (!ack)^*, ack)$ and

$$\text{globally } (p \text{ iff ended } (req, (!ack)^*, ack)$$

says that p holds precisely at the end of sequences of states that match $(req, (!ack)^*, ack)$. FTL has two additional temporal connectives to faciliate assertions about regular events. The formula e follows_by φ, where e is a regular event and φ is a formula, asserts that some e sequence is followed by φ. The formula e triggers φ, where e is a regular event and φ is a formula, asserts that all e sequence are followed by φ. The follows_by and triggers connectives are essentially the "diamond" and "box" modalities of PDL [FL79]. Formally:

- $\pi, i \models e$ follows_by φ if for some $j \geq i$ $\pi, i, j \models\!\!\!\models e$ and $\pi, j \models \varphi$.
- $\pi, i \models e$ triggers φ if for all $j \geq i$ such that $\pi, i, j \models\!\!\!\models e$ we have $\pi, j \models \varphi$.

For example, the formula

$$\text{globally } ((req, (!ack)^*, ack) \text{ triggers } (\mathsf{true}^+, grant, (!rel)^*, rel))$$

asserts that a request followed by an acknowledgement must be followed by a grant followed by a release. Note that follows_by and triggers are dual to each other.

Lemma 1. $\pi, i \models !(e \text{ follows\_by } \varphi)$ *iff* $\pi, i \models e$ triggers $(!\varphi)$.

[6] FTL 2.0 will include some additional regular constructs.

4 Hardware Features

FTL offers direct support to two specification modes often used by verification engineers at Intel: *clocks* and *resets*. While these features do not add to the expressive power of FTL, expressing them in terms of the other features of the language would be too cumbersome to be useful. While these features have a clear intuitive semantics, capturing it rigorously is quite nontrivial, as we show in the rest of this section. Our semantics, however, is defined in a modular fashion, so users who do not use these advanced features can continue to use the semantics of Section 3. Defining the semantics in a modular fashion was necessary to achieve a proper balance between expressiveness and usabilty, ensuring that users do not need to understand complex semantic aspects in order to use the basic features of the language. (The ForSpec User Guide describes the language at an intuitive level, while the ForSpec Reference Manual provides the formal semantics in a modular fashion.)

Clocks: FTL allows formulae of the form change_on $c\ \varphi$, which asserts that the temporal formula φ is to be evaluated with respect to the clock c; that is, the formula φ is to be evaluated in the trace defined by the high phases of the clock c. Every Boolean expression can serve as a clock. Note that the computation is sampled at the high phases of c rather than at the points where c changes. Focusing on the high phases is simpler and more general. For example, as we saw earlier the high phases of the Boolean expression $(!b)\&\&$ future (b) are the points at which b is about to rise.

As soon as we attempt to formalize further this intuition, we realize that there is a difficulty. How does one guarantee that the clock c actually "ticks", i.e., has high phases? We could have required that all clocks are guaranteed to tick infinitely often, but then this would have to be checked. Since we wanted to give users the ability to use arbitrary Boolean events as clocks, we decided not to require that a clock be guaranteed to tick. Instead, we introduced two operators. The formula change_on $c\ \varphi$ asserts that c *does* tick and φ holds with respect to c, while change_if $c\ \varphi$ asserts that *if* c ticks *then* φ holds with respect to c. Since the concept of "next tick" is now not always defined, we need to introduce a weak dual wnext to the next connective; next requires the existence of a next tick, while wnext does not (see formal semantics below).

What should be the formal meaning of "holds with respect to c"? Suppose for simplicity that c has infinitely many high phases. It seems intuitive to take the *projection* $\pi[c]$ of a computation π to be the sequence of values $\pi(i_0), \pi(i_1), \pi(i_2), \ldots$ of π at the points $i_0, i_1, i_2, \ldots$ at which c holds. We then can say that π satisfies change_on $c\ \varphi$ if $\pi[c]$ satisfies φ. The implication of such a definition however, would be to make clocks *cumulative*. That is, if change_on $d\ \psi$ is a subformula of φ, then ψ needs to be evaluated with respect to a projection $\pi[c] \ldots [d]$. This went against the intuition of our users, who want to make assertions about clocks *without* losing access to faster clocks. For example, the subformula change_on $d\ \psi$ could be a library property, describing an environment that is not governed by the clock c. (Recall that multiple clocks are needed to capture local synchrony in a globally asynchronous design.) This led to the decision to define the semantics of clocks in a non-cumulative fashion. This is done by defining the semantics as a four-way relation, between computations, points, clocks, and formulae, instead of a three-way relation as before. That is, on the left-hand-side of $\models$ we have a triple π, i, c, where c is a *clock expression* (see below), which is the *context* in which the

formula is evaluated.[7] Before we issue any change_on c or change_if c, the default clock expression is true. That is, $\pi, i \models \varphi$ if $\pi, i, \mathsf{true} \models \varphi$.

The semantics of propositional constants is insensitive to clocks:

– $\pi, i, c \models p$ if $\pi, i \models p$.

FTL , however, has a special propositional constant clock that enables users to refer explicitly to clock ticks. To avoid circularity, however, clock expressions are Boolean expressions that do not refer to clock . Given a clock expression c and a Boolean expression d, we denote by $[c \mapsto d]$ the result of substituting c for clock in d. For example if c is $!p\&\&$ future (p) and d is clock $\&\&q$, then $[c \mapsto d]$ is $(!p\&\&$ future $(p))\&\&q$. Note that $[c \mapsto d]$ is a clock expression, since it does not refer to clock . The semantics of clock is defined as follow:

– $\pi, i, c \models$ clock if $\pi, i \models c$.

In this case we say that *c ticks* at point i of π, or, equivalently, that i is a *tick point* of c on π. When we refer to tick points $i_1 < \ldots < i_l$ at or after i, we assume that no other point between i and i_l is a tick point.

We start defining the four-way sematics by defining satisfaction of the future and past connectives. Recall that the role of future is to define transition constraints. Consequently, its semantics is insensitive to clocks.

– $\pi, i, c \models$ future (b) if $\pi, i + 1 \models b$.

In contrast, the past operator is extended to allow reference to clocks. For a Boolean expression b, FTL uses past (b, n, d), where n is a positive integer and d is a Boolean expression, to refer to the value of b in the phase of the clock d, n ticks in the past. Formally,

– $\pi, i, c \models$ past $(b, 1, d)$ if $\pi, j, [c \mapsto d] \models b$, where j is the the largest integer less than i such that $\pi, j, [c \mapsto d] \models d$.
– $\pi, i, c \models$ past (b, n, c) if $\pi, j, [c \mapsto d] \models$ past $(b, n - 1, d)$ where j is the the largest integer less than i such that $\pi, j, [c \mapsto d] \models d$.

Note that if there is no $j < i$ such that $\pi, j, [c \mapsto d] \models d$, then $\pi, i, c \not\models$ past (b, n, c).

We continue defining the four-way sematics by defining satisfaction of regular events. Recall that satisfaction of regular events is defined in terms of tight satisfaction: $\pi, i, c \models e$ if for some $j \geq i$ we have $\pi, i, j, c \Subset e$. Tight satisfaction, which is a five-way relation, is defined as follows: $\pi, i, j, c \Subset e$ if

– there are $n \geq 0$ tick points $i_1 < \ldots < i_n$ of c after i such that $i = i_0 < i_1$ and $i_n = j$,
– there is a word $B = b_0 b_1 \ldots b_n \in L(e)$ such that $\pi, i_m, c \models b_m$ for $0 \leq m \leq n$.

[7] It is quite common in logic to define satisfaction of a formula with respect to a structuture and a context. For example, for 1st-order formulas with free variables the context provides an assignment of domain elements to the free variables.

Note that eveluation of regular events always starts at the present point, just as a Boolean event is aways evaluated at the present point. Once we have defined tight satisfaction with clocks, the semantics of ended is as before: $\pi, i \models$ ended (e) if there is some $j < i$ such that $\pi, j, i \not\models e$.

FTL has a special regular event tick , which is an abbreviation for the regular event $((!$ clock $)^*,$ clock $)$. Thus, $\pi, i, j \not\models$ tick if j is the first tick point of c at or after i.

We can now define the semantics of the temporal connectives. For example:

- $\pi, i, c \models$ next φ if for some $j \geq i + 1$ we have that $\pi, i + 1, j, c \not\models$ tick and $\pi, j, c \models \varphi$.
- $\pi, i, c \models$ wnext φ if for some all $j \geq i + 1$ we have that if $\pi, i + 1, j, c \not\models$ tick , then $\pi, j, c \models \varphi$.
- $\pi, i, c \models \varphi$ until ψ if for some $j \geq i$ we have that $\pi, j \models c$ and $\pi, j, c \models \psi$, and for all k, $i \leq k < j$ such that $\pi, k \models c$, we have $\pi, k, c \models \varphi$. (Note that only tick points of c are taken into consideration here.)
- $\pi, i, c \models e$ triggers φ if for all $l \geq k \geq j \geq i$ such that $\pi, i, j, c \not\models$ tick , $\pi, j, k, c \not\models e$ and $\pi, k, l, c \not\models$ tick , we have that $\pi, l, c \models \varphi$. (Note that the evaluation of e starts at the first tick point at or after i, and the evaluation of f starts at the first tick point at or after the end of e.)

We now define the semantics of change_on and change_on :

- $\pi, i, c \models$ change_on $d\ f$ if there exists some $j \geq i$, such that $\pi, i, j, [c \mapsto d] \not\models$ tick and $\pi, j, [c \mapsto d] \models f$.
- $\pi, i, c \models$ change_if $d\ f$ if whenever $j \geq i$ is such that $\pi, i, j, [c \mapsto d] \not\models$ tick , then $\pi, j, [c \mapsto d], a, r \models f$.

Both change_on and change_if force the evaluation of the formula at the nearest tick point, but only change_on requires such a tick point to exist. As one expects, we get duality between next and wnext and between change_on and change_if .

Lemma 2.

- $\pi, i, c \models !($ next $\varphi)$ *iff* $\pi, i, c \models$ wnext $!\varphi$.
- $\pi, i, c \models !($ change_on $c\ \varphi)$ *iff* $\pi, i, c \models$ change_if $c\ (!\varphi)$.

As an example, the formula change_on $c1$ (globally (change_if $c2\ p$)) asserts that the clock $c1$ ticks and p also holds at the first tick, if any, of $c2$ at or after a tick of $c1$.

Resets: FTL allows formulae of the form accept_on $a\ \varphi$, which asserts that the property φ should be checked only until the arrival of a reset signal a, at which point the check is considered to have *succeeded*, and reject_on $r\ \varphi$, which asserts that the property φ should be checked only until the arrival of a reset signal r, at which point the check is considered to have *failed*. In reset control-design methodology, a local reset signal does not replace a global reset signal. Thus, while our semantics for multiple clocks was non-cumulative, our semantics for resets is cumulative. Another important feature of our semantics is that reset signals are *asynchronous*, that is, they are not required to occur at tick points; rather, they are allowed to occur at any point. To capture *synchronous resets*, the users can write accept_on $(a \&\&$ clock $)\ \varphi$ or reject_on $(r \&\&$ clock $)\ \varphi$. As we shall see, we pay a price for the added expressiveness of asynchronous resets with

additional complexity of the semantics, as we have to account for resets between tick points.

As we saw earlier, we capture the semantics of clocks by adding a clock to the context in which formulae are evaluated, resulting in a four-way semantical relation. To capture the semantics of resets, we have to add both the accept signal and the reject signal to the context, resulting in a six-way semantical relation. That is, on the left-hand-side of $\models$ we have a quintuple π, i, c, a, r, where π, i, c are as before, and a and r are *disjoint* Boolean events defining the accept and reject signals, respectively. Before we issue any accept_on a or reject_on r, the default reset signals are both false, respectively. That is, $\pi, i \models \varphi$ if $\pi, i, \mathsf{true}, \mathsf{false}, \mathsf{false} \models \varphi$.

The semantics of accept_on and reject_on is defined by:

- $\pi, i, c, a, r \models$ accept_on $b\, \varphi$ if $\pi, i, c \models a||(b\&\&!r)$ or $\pi, i, c, a||(b\&\&!r), r \models \varphi$
- $\pi, i, c, a, r \models$ reject_on $b\, \varphi$ if $\pi, i, c \not\models r||(b\&\&!a)$ and $\pi, i, c, a, r||(b\&\&!a) \models \varphi$

Note how the cumulative semantics ensures that the accept and reject signals are always disjoint. Also, outer resets have preference over inner ones.

The presence of reset signals requires us to redefine the very basic semantics of propositions and Boolean connectives. An important issue now is the meaning of negation in the reset-control design methodology. Since negation switches success and failure, it should also switch accept and reject signals. We therefore define

- $\pi, i, c, a, r \models p$, for $p \in \mathsf{Prop}$ if either $\pi, i, c \models a$ or $\pi, i, c \models p$ and $\pi, i, c \models !r$ (That is, the accept "signal" always makes a proposition true, while the reject "signal" always makes it false. It may seem that we are giving the accept signal a preference over the reject signal, but as the accept and reject signals can never be true simultaneously, their roles are actually symmetric.)
- $\pi, i, c, a, r \models !\varphi$ if $\pi, i, c, r, a \not\models \varphi$.

We now define satisfaction of regular events. Recall again that satisfaction of regular events is defined in terms of tight satisfaction: $\pi, i, c, a, r \models e$ if for some $j \geq i$ we have $\pi, i, j, c, a, r \models\!\!\!\models e$. Tight satisfaction, which is a seven-way relation, is defined as follows: $\pi, i, j, c, a, r \models\!\!\!\models e$ if

- $\pi, k, c \models !a\&\&!r$ for $i \leq k < j$,
- there are precisely $l \geq 0$ tick points $i_1 < \ldots < i_l$ of c such that $i = i_0 < i_1$ and $i_l \leq j$,
- there is a word $B = b_0 b_1 \ldots b_n \in L(e)$, $n \geq l$ such that $\pi, i_m, c \models b_m$ for $0 \leq m < l$, and either
  - $\pi, j, c \models a$ and $i_l = j$, or
  - $\pi, j, c \models a$ and $\pi, i_l, c \models b_l$, or
  - $l = n$, and $i_n = j$, and $\pi, j, c \models b_n$, and $\pi, j, c \models !r$.

Note that in the first case, only the prefix $b_0 b_1 \ldots b_{l-1}$ is checked, in the second case, only the prefix $b_0 b_1 \ldots b_l$ is checked, and in the third case the word B is checked in full.) As we remarked earlier, the complexity of the semantics is the result of the need to account for resets between tick points. In the first two cases, the checking of B is terminated due to an accept signal. We denote this by $\pi, i, j, c, a, r \models\!\!\!\models_a e$.

Let π be a computation, i a point, c a clock expression, and a and r reset expresions. Then $tick(\pi, i, c, a, r)$ is the least j such that $j \geq i$ and $\pi, i, j, c, a, r \models\!\!\!\models$ tick (note that

there is at most one such j). Note that $tick(\pi, i, c, a, r)$ need not be defined, since we may have that $\pi, i, j, c, a, r \not\models$ tick for all $j \geq i$. If it is defined, we write $tick(\pi, i, c, a, r) \downarrow$. Note that $\pi, i, j, c, a, r \models$ tick can hold in two (nonexclusive) ways:

- $\pi, j \models c$, in which case we write $tick(\pi, i, c, a, r) \downarrow_c$, or
- $\pi, j, c \models a$, in which case we write $tick(\pi, i, c, a, r) \downarrow_a$.

We write "$j = tick(\pi, i, c, a, r)$" as an abbreviation for "$tick(\pi, i, c, a, r) \downarrow$ and $j = tick(\pi, i, c, a, r)$."

We now illustrate the semantics of temporal connectives by considering the triggers connective:

- $\pi, i, c, a, r \models e$ triggers φ if $j = tick(\pi, i, c, r, a)$ entails that
  - $tick(\pi, i, c, r, a) \downarrow_r$ does not hold and
  - for all $k \geq j$ such that $\pi, j, k, c, r, a \models e$ we have that
    - $\pi, j, k, c, r, a \models_r e$ does not hold and
      - $l = tick(\pi, k, c, r, a)$ entails that $tick(\pi, k, c, r, a) \downarrow_r$ does not hold
      - $\pi, l, c, a, r \models f$.

Note that the definition is a direct extension of the earlier definition of triggers . The impact of resets is through the various conditions of the definition. The events tick , e, and again tick function as antecedents of an implication, and thus have negative polarity, which explains why the roles of a and r are reversed in the antecedents.

As a sanity check of the not-immediately-obvious semantics of resets we state another duality lemma.

Lemma 3.
$\pi, i, c, a, r \models !($ accept_on $d\ \varphi)$ *iff* $\pi, i, c, a, r \models$ reject_on $d\ (!\varphi)$.

As an example, the formula change_on c (accept_on $a\ (p$ until $q))$ declares c to be a strong clock and a to be an accept signal, relative to which p holds until q holds.

5 Expressiveness and Complexity

The addition of regular events and the new connectives (follows_by and triggers) has both a theoretical motivation and a pragmatic motivation. Regular events were shown to be useful in the context of hardware verification, cf. [BBL98,Mor99]. More fundamentally, as noted earlier, it was observed in [Wol83] that LTL cannot express certain ω-regular events, and it was shown in [LPZ85] that this makes LTL inadequate for modular verification. Let *regular LTL* be the extension of LTL with regular events and the connectives follows_by and triggers . Regular LTL, which is strictly more expressive than LTL, is expressive enough to capture ω-regularity.

Theorem 1. *Let L be an ω-regular set of computations. Then there is a formula φ_L of regular LTL such that $models(\varphi_L) = L$.*

Proof Sketch: Every ω-regular language can be expressed as a finite union of terms of the form $L_1 \cdot L_2$, where L_1 is a regular language (of finite words) and L_2 is a language of infinite words defined by a deterministic Büchi automaton. Thus, we can express L as a

disjunction of formulae of the form e follows_by φ, provided we can express languages of deterministic Büchi automata. Such languages can be expressed by disjunctions of formulae of the form $e_1 \&\& (e_1$ triggers $e_2)$, where both e_1 and e_2 are regular events. □

A consequence of this theorem is that FTL *fully supports* assume-guarantee reasoning, in the following sense. On one hand, we have that, for all models M and E and for all FTL formulas φ, ψ, if $E \models \varphi$ and $M \models \varphi \rightarrow \psi$ then $E|M \models \psi$. On the other hand, for all models M and E and for every FTL formula ψ such that $E|M \models \psi$, there is an FTL formula φ such that $E \models \varphi$ and $M \models \varphi \rightarrow \psi$. Furthermore, every FTL formula can serve both as an assumption and as an assertion (guarantee), and as assume-guarantee reasoning is performed via model checking, the complexity of such reasoning is the same as that of model checking. (Note, however, that full support of assume-guarantee reasoning is not guaranteed by the mere inclusion of regular events. Sugar adds regular events to CTL, resulting in a mixed linear-branching semantics [BBDE$^+$01], in the style of CTL* [Eme90], which makes it rather difficult to fully support assume-guarantee reasoning [Var01]. In Temporal ***e***, the main focus is on describing finite sequences using regular expressions. It is not clear whether the language has full ω-regularity [Mor99], which is required for full support of assume-guarantee reasoning.)

Reasoning and verification for FTL formulas is carried out via the automata-theoretic approach [Var96]: to check satisfiability of a formula φ, one constructs a nondeterministic Büchi automaton accepting $models(\varphi)$ and check its emptiness, and to model check a formula one constructs a nondeterministic Büchi automaton for the complementary formula, intersects it with the model and check emptiness. Because of the richness of FTL, the construction of the automaton is quite non-trivial, and will be described in full in a future paper. The ForSpec compiler actually generates a symbolic representation of the automaton; the size of this representation is linear in the length of the formula and in the size of the time windows. Since the lower and upper bounds of these windows are specified using decimal numbers, time windows may involve an exponential blow-up, as the formula next $[2^n]\, p$ is of length $O(n)$. (It was noted in [AH92] that such succinctness cause an exponential increase in the complexity of temporal reasoning.)

Theorem 2. *The satisfiability problem for* FTL *without time windows is PSPACE-complete. The satisfiability problem for* FTL *is EXPSPACE-complete.*

While reasoning in FTL is exponential in terms of the formula size and time windows' size in the worst case, aggressive optimization by the compiler ensures that the worst case almost never arises in practice; the computational bottleneck of model checking is due to the large nummber of design states. (In fact, in spite of the increased expressiveness of the language, the FTL model checker is much more efficient that Intel's 1st-generation model checker.)

The linear-time framework enable us to subject FTL properties to validation by both formal verification (model checking) and dynamic validation (simulation). The semantics of FTL is over *infinite* traces. This semantics induces a 3-valued semantics over *finite* traces, which are produced by simulation engines: "pass", "fail", and "ongoing". A formula φ *passes* (resp., *fails*) a finite trace τ if τ is an *informative prefix* for φ (resp., $\neg\varphi$) [KV01]. If it neither passes not fails, it is "ongoing". For example, eventually $!p$ passes on the finite trace $p, p, p, \overline{p}$ and is ongoing on the trace p, p, p, p. The formula

globally p fails on the trace $p, p, p, \overline{p}$. The fact that simulation semantics is induced by the standard semantics means that the language requires only a single compiler, ensuring consistency between formal and dynamic validation.

6 Discussion

This paper describes FTL, the temporal property-specification logic of ForSpec, Intel's formal specification language, which is being used by several formal-verification teams at Intel. FTL is an industrial property-specification language that supports hardware-oriented constructs as well as uniform semantics for formal and dynamic validation, while at the same time it has a well understood expressiveness and computational complexity, and it fully supports modular reasoning. The design effort strove to find an acceptable compromise, with trade-offs clarified by theory, between conflicting demands, such as expressiveness, usabilty, and implementability. Clocks and resets, both important to hardware designers, have a clear intuitive semantics, but formalizing this semantics is nontrivial.

References

[AH92] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 74–106. Springer-Verlag, 1992.

[BB87] B. Banieqbal and H. Barringer. Temporal logic with fixed points. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 62–74. Springer-Verlag, 1987.

[BBDE$^+$01] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Proc. Conf. on Computer-Aided Verification (CAV'00)*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer-Verlag, 2001.

[BBL98] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Computer Aided Verification, Proc. 10th International Conference*, volume 1427 of *Lecture Notes in Computer Science*, pages 184–194. Springer-Verlag, 1998.

[CCGR00] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *It'l J. on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CLM98] R. Cleaveland, G. Luttgen, and M. Mendler. An algebraic theory of multiple clocks. In *Proc. 8th Int'l Conf. on Concurrency Theory (CONCUR'97)*, volume 1243 of *Lecture Notes in Computer Science*, pages 166–180. Springer-Verlag, 1998.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, MIT press, 1990.

[ET97] E.A. Emerson and R.J. Trefler. Generalized quantitative temporal reasoning: An automata theoretic. In *Proc. Theory and Practice of Software Development (TAPSOFT)*, volume 1214 of *Lecture Notes in Computer Science*, pages 189–200. Springer-Verlag, 1997.

[Fin01] B. Finkbeiner. Symbolic refinement checking with nondeterministic BDDs. In *Tools and algorithms for the construction and analysis of systems*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.

[HT99] J.G. Henriksen and P.S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied Logic*, 96(1–3):187–207, 1999.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KPV01] O. Kupferman, N. Piterman, and M.Y. Vardi. Extended temporal logic revisited. In *Proc. 12th International Conference on Concurrency Theory*, volume 2154 of *Lecture Notes in Computer Science*, pages 519–535, August 2001.

[Kur97] R.P. Kurshan. Formal verification in a commercial setting. In *Proc. Conf. on Design Automation (DAC'97)*, volume 34, pages 258–262, 1997.

[Kur98] R.P. Kurshan. *FormalCheck User's Manual*. Cadence Design, Inc., 1998.

[KV01] O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal methods in System Design*, 19(3):291–314, November 2001.

[LO99] C. Liu and M.A. Orgun. Verification of reactive systems using temporal logics with clocks. *Theoretical Computer Science*, 220:377–408, 1999.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, Brooklyn, June 1985. Springer-Verlag.

[Mor99] M.J. Morley. Semantics of temporal e. In T. F. Melham and F.G. Moller, editors, Banff'99 *Higher Order Workshop (Formal Methods in Computation)*. University of Glasgow, Department of Computing Science Technic al Report, 1999.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 123–144. Springer-Verlag, 1985.

[SBF+97] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 41(4:5), 1997.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Var96] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, Berlin, 1996.

[Var01] M.Y. Vardi. Branching vs. linear time: Final showdown. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2001.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.