Safe Runtime Verification of Real-Time Properties*

Christian Colombo[1], Gordon J. Pace[1], and Gerardo Schneider[2]

[1] Department of Computer Science, University of Malta, Msida, Malta.
[2] Department of Informatics – University of Oslo, Oslo, Norway
{christian.colombo, gordon.pace}@um.edu.mt, gerardo@ifi.uio.no

Abstract. Introducing a monitor on a system typically changes the system's behaviour by slowing the system down and increasing memory consumption. This may possibly result in creating new bugs, or possibly even 'fixing' bugs, only to reappear as the monitor is removed. Properties written in a real-time logic, such as duration calculus, can be particularly sensitive to such changes induced through monitoring. The same problem occurs in other scenarios such as when a system is ported to a faster machine. In this paper, we identify a class of real-time properties, in duration calculus, which are monotonic under the slowing down (speeding up) of the underlying system. We apply this approach to the real-time runtime monitoring tool LARVA, where we use duration calculus as a monitoring property specification language, so we automatically identify properties which can be shown to be monotonic with respect to system re-timing.

1 Introduction

Runtime verification has been steadily gaining popularity, but scepticism still exists regarding its applicability in real-time systems. The introduction of a monitor overseeing a system, normally slows down the system, which may prove to be too detrimental in performance intensive or real-time systems. However, the introduction of monitors also modifies the behaviour of the system, changes which may lead to the creation of new bugs, or the eradication of others.[1] Such situations are typically difficult to identify and fix.

One important consideration in such situations is the underlying system, and what access it has to the underlying machine. For instance, a program which times its running behaviour and branches accordingly, may exhibit aberrant behaviour under monitoring as time values are affected by the monitor. However, the expressivity of the underlying programming model is not the only issue —

*The research work disclosed in this publication is partially funded by Malta Government Scholarship Scheme grant number ME 367/07/29 and by the Malta National Research and Innovation (R&I) Programme 2008 project number 052.

[1] These bugs are known as *Heisenbugs*, after Heisenberg's uncertainty principle, related to the observer effect: the act of observing modifies the system.

another factor is the expressivity of the logic used to express properties from which the monitors are synthesised.

Not all logics (property languages) are sensitive to such problems. For instance, in single-threaded systems with no references to memory and temporal properties, the order in which methods are invoked are typically invariant under monitoring since such order remains unchanged. On the other hand, other logics (such as real-time logics) are particularly prone to this phenomenon, and thus one must be careful when verifying properties expressed in such formalisms at runtime.

There are two sides to monitor-sensitivity: (i) the behaviour of the original system being monitored may depend on the existence of a monitor (e.g., by checking memory usage); and (ii) the properties being monitored may change their truth value depending on the presence of a monitor (e.g., it can slow down the system to such an extent that a reactivity property may be broken). In this paper, we investigate system re-timing insensitivity of the second type — addressing the issue independent of a particular real-time logic, and instantiating the results in duration calculus. In the rest of the paper, we will thus assume that the system is itself monitor-insensitive — the order of the events generated by the system itself is invariant under monitoring. Single-threaded systems, with no branching based on real-time, memory allocation and other machine-centric notions, are instances of such systems.

In order to guarantee such monitor-insensitiveness, we need to characterise which real-time properties are monitor-sensitive. Consider the property 'No more than three resets of a user account may occur within any one hour period'. Clearly, slowing down a system which satisfies this constraint will not break the property. On the other hand, speeding up such a system may result in breaking it. We identify two classes of real-time properties, ones which cannot be broken (if true, they remain true) under slowing down, and ones which cannot be broken under speeding up. One application of this approach is to enable reasoning about the effect of adding, or removing a monitor on the system. The approach can be extended to other applications, such as automatic derivation of temporal correctness of functionally correct optimisations.

In this paper, we focus on the application of these techniques for runtime-verification. We integrate the analysis into Larva, a real-time runtime verification framework, which can monitor properties expressed using, amongst other logics, counterexample traces, a subset of duration calculus. The main contribution of the paper is twofold: (i) we define a formal mathematical framework to reason about real-time system retiming; and (ii) study the effect of such retiming for a duration calculus as a real-time logic.

The paper is organised into two main parts: a generic theoretical framework till section 3.3, and an application of the theory to a subset of duration calculus from thereon. In section 2 we present background material on duration calculus, followed by section 3, in which we present the principal results on speedup and slowdown invariance. In section 4, we outline how counterexample traces can be

used to synthesize runtime monitors using LARVA, and show an application of the techniques on a small case study in section 5.

2 Background

2.1 Duration Calculus

Duration calculus (DC) [23] is a real-time logic, describing properties which hold over time intervals, and that speaks about boolean states which change over time. The calculus is based on two main concepts: *integration* (measuring how long a boolean state holds over an interval) and the *chop* operator (which splits an interval into two parts). Various other operators can be defined in terms of these basic ones together with boolean connectives, to enable the expression of properties such as: (i) $\Box(\lceil Leak \rceil \Rightarrow \ell < 1)$, meaning that any subinterval in which *Leak* is continuously true, should not last longer than 1 time unit — leaks should not last longer than 1 time unit; and (ii) $\Box(\lceil Leak \rceil; \lceil \neg Leak \rceil; \lceil Leak \rceil \Rightarrow \ell \geq 30)$, meaning that any interval in which there is a falling edge followed by a rising edge on *Leak*, has to last longer than 30 time units — there are at least 30 time units between consecutive leaks.

In DC, time ($\mathbb{T}$) is modelled as the non-negative real numbers. Although the logic talks about time intervals, the underlying behaviour is modelled using boolean states, functions from the time domain to booleans ($BState = \mathbb{T} \to \mathbb{B}$). Validity of duration formulae is dependant on an underlying interpretation I, fixing the behaviour of the boolean states: $I \in Bstate \to (\mathbb{T} \to \mathbb{B})$. It is assumed that over any finite interval, state variables have a finite number of discontinuous points.

Boolean states can be combined into boolean expressions using standard boolean operators such as $X \wedge \neg Y$. An interpretation can be lifted over boolean expressions by applying the interpretation to the constituent boolean states, e.g. $I(X \wedge \neg Y)(t) = I(X)(t) \wedge \neg(I(Y)(t))$.

Duration formulae[2] act over time intervals. The basic duration formulae are $\int P = n$ (P holds for a total of n time units over a particular time interval), and the chop operator $D; E$ (the time interval can be split into two such that D holds over the first part, E on the second). For a given interpretation I, $I \vDash_{[b,e]} D$ means that the formula D holds for the given interval $[b, e]$, as defined below:

$$I \vDash_{[b,e]} \int P = n \stackrel{\text{def}}{=} \int_b^e I(P)(t)dt = n$$

$$I \vDash_{[b,e]} D; E \stackrel{\text{def}}{=} \text{for some } m \in [b, e],\ I \vDash_{[b,m]} D \text{ and } I \vDash_{[m,e]} E$$

As with boolean expressions, boolean operators are also lifted over duration formulae, e.g. $I \vDash_{[b,e]} D \wedge \neg E$ holds exactly when $I \vDash_{[b,e]} D$ and $I \nvDash_{[b,e]} E$.

Based on these operators, other operators are defined syntactically. Comparison operators on the duration of boolean expressions are defined, e.g. $\int P \geq n$

[2] By convention we will use X and Y to refer to state variables, P and Q to refer to state expressions, and D and E to refer to duration formulae.

is defined as $\int P = n; \text{true}$. The other comparators can be similarly defined. The length of an interval (written ℓ) is defined as $\int 1$ (where 1 is the constantly true boolean state). The duration formula $\lceil P \rceil$, meaning that P holds almost everywhere[3] throughout the given interval, is defined as: $\lceil P \rceil \stackrel{\text{def}}{=} \int P = \ell \wedge \ell > 0$. State expression invariance over an interval is written as $\boxminus P$, defined as: $\ell = 0 \vee \lceil P \rceil \vee \lceil \neg P \rceil)$. Based on the chop operator one can define the standard $\Diamond$ and $\Box$ modalities, which in DC read as "there exists a subinterval" and "for any subinterval" respectively: $\Diamond D \stackrel{\text{def}}{=} \text{true}; D; \text{true}$, and $\Box D \stackrel{\text{def}}{=} \neg\Diamond\neg D$.

Various other operators have been defined to simplify the expression of real-time properties. For instance, the *leads-to* operator $P \xrightarrow{n} Q$, which states that if P holds for at least n time units then Q must hold immediately afterwards, can be expressed as: $\Box((\lceil P \rceil \wedge \ell = n); \ell > 0 \Rightarrow \ell = n; \lceil Q \rceil; \text{true})$. This can be shown to be equivalent to $\neg\Diamond((\lceil P \rceil \wedge \ell = n); \lceil \neg Q \rceil)$ or even $\neg\Diamond((\lceil P \rceil \wedge \ell \geq n); \lceil \neg Q \rceil)$.

A duration formula is *valid under an interpretation* if it holds for all time prefixes: $I \vDash D \stackrel{\text{def}}{=} \forall t : \mathbb{T} \cdot I \vDash_{[0,t]} D$. Finally, a duration formula is said to be a *tautology* if it holds under all interpretations: $\vDash D \stackrel{\text{def}}{=} \forall I \cdot I \vDash D$.

2.2 Counterexample Traces

In general, DC is known to be too expressive to be monitored with a bounded number of clocks [1]. However, a class of implementable DC formulae (known as the class of *implementables*) has been identified [20, 22]. New operators are added on to DC in counterexample traces [20]. The two operators $\nwarrow P$ and $\nearrow P$ enable knowing whether a state expression was satisfied just before (or after) a point in time: $I \vDash_{[b,e]} \nwarrow P \stackrel{\text{def}}{=} b = e \wedge \exists m : \mathbb{T} \cdot m < b \wedge I \vDash_{[m,b]} \lceil P \rceil$, and $I \vDash_{[b,e]} \nearrow P \stackrel{\text{def}}{=} b = e \wedge \exists m : \mathbb{T} \cdot m > e \wedge I \vDash_{[e,m]} \lceil P \rceil$, respectively. Using these operators, one can syntactically define others. For instance, $\updownarrow P$ identifies whether there is a discontinuity at a point in time: $(\nwarrow \neg P \wedge \nearrow P) \vee (\nwarrow P \wedge \nearrow \neg P)$. Conversely, $\not\updownarrow P$ indicates that P does not change at that point in time: $\neg \updownarrow P \wedge \ell = 0$.

In counterexample traces, *events* correspond to a change (or lack of change) of the value of state expressions: $\mathit{event} ::= \updownarrow P \mid \not\updownarrow P \mid \mathit{event} \vee \mathit{event} \mid \mathit{event} \wedge \mathit{event}$.

Similarly, *phases* consist of a conjunction of three constraints: (i) a state expression which holds uniformly over the interval $\lceil P \rceil$, or simply true; (ii) a constraint on the length of the interval of the form $\ell \leq n$, $\ell \geq n$, $\ell < n$ or $\ell > n$; and (iii) a number of state expressions which are invariant over the interval $\boxminus P_1 \wedge \boxminus P_2 \wedge \ldots \boxminus P_n$.

A *counterexample trace* is the negation of a chop separated sequence of phases and events: $\neg(\mathit{phase}; (\mathit{phase} \mid \mathit{event})^*; \text{true})$. We say that a counterexample trace is a *lower bound trace* if none of the phases refer to $\ell > n$ or $\ell \geq n$. Similarly it is said to be an *upper bound trace* if none of the phases refer to $\ell < n$ or $\ell \leq n$[4].

[3] It is standard in DC to use the "almost everywhere" since finite (negligible) variability does not affect the duration formulae, as durations are defined as the integral of certain variable over time.

[4] Note that counterexample traces are negated.

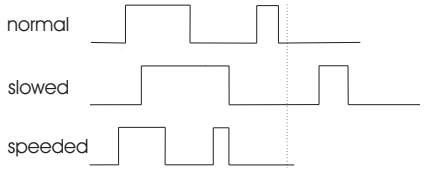

Fig. 1. A stretch and compressed interpretation of a state variable.

Length independent counterexample trace formulae are ones which include no reference to interval length in the phases.

Consider the leads-to operator given in the previous section. The duration formula $P \xrightarrow{n} Q$, is equivalent to $\neg\Diamond((\lceil P \rceil \wedge \ell \geq n); \lceil \neg Q \rceil)$, or the counterexample trace $\neg(\text{true}; (\lceil P \rceil \wedge \ell \geq n); \lceil \neg Q \rceil; \text{true})$, which is an upper-bound trace.

3 Stretch and Compress Truth Preservation

In DC, the underlying real-time behaviour of a system is encapsulated in the interpretation of the state variables. A slowed down (or sped up) variant of the behaviour of a system corresponds to a stretching (or compression) transformation on the underlying state variable interpretations. Fig. 1 illustrates a state variable interpretation and its slowed down and sped up variants.

In this section, we build a mathematical framework to model time stretches and compressions, independent of the real-time logic being considered. Using this framework, we proceed by characterising the fragment of DC satisfying these (stretch and compress) preservation properties.

3.1 Time Transforms

We start by characterising time transforms, corresponding to monotonic homeomorphisms over the positive real number line. Time transforms can be used to retime real-time variable interpretations in such a way that events are not lost, created, or reordered.

Definition 1. *A total continuous function $s \in \mathbb{T} \to \mathbb{T}$ is said to be a* time transform *($s \in \mathbb{T}$) if (i) $s(0) = 0$; (ii) $\lim_{t\to\infty} s(t) = \infty$; (iii) s is monotonic ($t_1 < t_2 \Rightarrow s(t_1) < s(t_2)$).*

The simplest time transform is the identity function *id* which, given time t, returns t as output: $id(t) = t$. Also note that the functional composition of two time transforms is also a time transform. Moreover:

Lemma 1. *Time transforms are bijective functions.*

A naïve way of defining a time transform to be a time-stretch, is to insist that $s(t) \geq t$. However, this would only guarantee that all event timings of

an interpretation I occur earlier than those of the I_s — this is not what we require, since it does not guarantee that the intervals between events are always longer in the slow interpretation than the fast counterpart. Consider the case when the first two changes of a boolean variable X occur at times 5 and 10 under an interpretation I, but at times 9 and 11 under I_s. All other events occur at the same time in the two interpretations. Although all events of I_s occur later than the events in I, if one looks at the time between the first and second event, it is actually smaller in the case of I_s. If causality of events is seen to start building up from the previous event, we need to look at lengthening intervals between events, not at delays in the events on an absolute time line — this requires interval-monotonicity. To formally define concepts related to the notions of slowing down and speeding up a system we need the formalisation of what *time-stretching* and *time-compression* mean.

Definition 2. *A time transform $s \in \mathbb{T}$ is said to be a* time-stretch *($s \in \overleftrightarrow{\mathbb{T}}$) if it is monotonic on intervals: $s(t_2) - s(t_1) \geq t_2 - t_1$ (for $t_1 < t_2$). Similarly, it is said to be a* time-compression *($s \in \overbar{\overleftrightarrow{\mathbb{T}}}$) if it is anti-monotonic on intervals: $s(t_2) - s(t_1) \leq t_2 - t_1$ (for $t_1 < t_2$).*

It can be proved that time-stretches satisfy that $s(t) \geq t$. Since time transforms are bijective, one can talk about their inverse, which relates time-stretches and time-compressions.

Proposition 1. *The inverse of every time-stretch transformation is a time-compress transformation, and vice-versa.*

Stretching a time interval increases the integral under a (finitely-variable) state expression, and conversely, compressing a time interval decreases it.

Proposition 2. *Given a time-stretch s and interpretation of state expression α, $\int_{s(b)}^{s(e)} \alpha(t)dt \geq \int_b^e \alpha(s(t))dt$. Similarly, given a time-compression f, $\int_{f(b)}^{f(e)} \alpha(t)dt \leq \int_b^e \alpha(f(t))dt$.*

3.2 Duration Calculus and Time Transforms

Through the use of time transforms, we can now define the effect of compression and stretching on an interpretation, and consequently on the validity of duration formulae. Applying a time transformation to an interpretation, yields an interpretation where each point is the output of the time transform function on the corresponding point.

Definition 3. *Given a time transformation s, and an interpretation I, the* transformed interpretation of I w.r.t. *s, written I_s, is: $I_s(P)(s(t)) = I(P)(t)$.*

Applying the identity function to an interpretation I gives I: $I_{id} = I$. Also, applying time transforms f and g in sequence on an interpretation, is equivalent to applying the functional composition $g \circ f$: $(I_f)_g = I_{g \circ f}$.

Definition 4. *A duration formula D is said to be* stretch truth-preserving, $\mathrm{str}_t(D)$, *if for any interpretation on which it is valid, it is also valid under any stretching of the interpretation. Similarly for* stretch falsity-preserving, $\mathrm{str}_f(D)$*:*

$$\mathrm{str}_t(D) \stackrel{def}{=} \forall s : \overleftrightarrow{\mathcal{T}},\ I \cdot I \vDash D \implies I_s \vDash D$$
$$\mathrm{str}_f(D) \stackrel{def}{=} \forall s : \overleftrightarrow{\mathcal{T}},\ I \cdot I \nvDash D \implies I_s \nvDash D$$

Similarly, we define the corresponding notions for compression $\mathrm{com}_t(D)$ *and* $\mathrm{com}_f(D)$*. A formula D is said to be* stretch invariant, $\mathrm{str}_i(D)$*, if it is both stretch truth and falsity truth-preserving. Similarly, we define the notion of* compression invariance, $\mathrm{com}_i(D)$*.*

The above notions consider the validity of a formula — the satisfaction of the formula on all time prefixes under an interpretation. In order to enable reasoning about individual stretched or compressed time intervals, we define the following:

Definition 5. *A duration formula D is said to be* interval-stretch (interval-compress) truth-preserving $\mathrm{istr}_t(D)$ *(*$\mathrm{icom}_t(D)$*) if, for all interpretations on which D is satisfied on all subintervals, D also holds for all subintervals under any stretching (compression) of the interpretation:*

$$\mathrm{istr}_t(D) \stackrel{def}{=} \forall s : \overleftrightarrow{\mathcal{T}},\ I,\ (b,e) \cdot I \vDash_{[b,e]} D \implies I_s \vDash_{[s(b),s(e)]} D$$
$$\mathrm{icom}_t(D) \stackrel{def}{=} \forall s : \overleftarrow{\mathcal{T}},\ I,\ (b,e) \cdot I \vDash_{[b,e]} D \implies I_s \vDash_{[s(b),s(e)]} D$$

Similarly, we define the notions of $\mathrm{istr}_f(D)$, $\mathrm{icom}_f(D)$, $\mathrm{istr}_i(D)$ *and* $\mathrm{icom}_i(D)$*.*

Using the surjectivity of time transforms, one can prove that interval truth preservation is a stronger notion than truth preservation.

Theorem 1. *An interval-stretch truth-preserving duration formula is also stretch truth-preserving:* $\mathrm{istr}_t(D) \Rightarrow \mathrm{str}_t(D)$*. Similar results follow for the other predicates defined.*

In the rest of the paper, we will be using the notion of interval truth (falsity) preservation, since we know that this guarantees truth (falsity) preservation.

Using the duality of time-stretches and time-compressions (see Proposition 1), and that of truth and falsity preservation, we can show that stretch truth preservation is equivalent to compress falsity preservation:

Proposition 3. *The class of interval-stretch truth-preserving formulae is equivalent to interval-compress falsity-preserving formulae:* $\mathrm{istr}_t(D) \Leftrightarrow \mathrm{icom}_f(D)$*. Similarly:* $\mathrm{icom}_t(D) \Leftrightarrow \mathrm{istr}_f(D)$*. It thus directly follows that interval-compress invariance is equivalent to interval-stretch invariance:* $\mathrm{icom}_i(D) \Leftrightarrow \mathrm{istr}_i(D)$*.*

3.3 Duration Formulae Under the Effect of Time Transforms

The proposition just proved can be used to relate truth preservation of a formula and its negation.

Theorem 2. *A formula is interval-stretch truth-preserving if and only if its negation is interval-stretch false preserving:* $\mathrm{istr}_t(D) \Leftrightarrow \mathrm{istr}_f(\neg D)$. *Similarly, the negation of a formula which is interval-compress truth-preserving, is interval-compress false preserving:* $\mathrm{icom}_t(D) \Leftrightarrow \mathrm{icom}_f(\neg D)$.

Combining the results of Proposition 3 and Theorem 2, we obtain:

Corollary 1. *A formula is interval-stretch truth-preserving if and only if its negation is interval-compress truth-preserving:* $\mathrm{istr}_t(D) \Leftrightarrow \mathrm{icom}_t(\neg D)$.

Hence, the truth of interval-compress and interval-stretch invariance does not change under negation: $\mathrm{icom}_i(D) \Leftrightarrow \mathrm{icom}_i(\neg D)$. *Similarly,* $\mathrm{istr}_i(D) \Leftrightarrow \mathrm{istr}_i(\neg D)$.

Negation switches the parity of the duration formula. On the other hand, a number of other operators preserve it, as shown in what follows.

Theorem 3. *The duration formulae* $\int P > c$ *and* $\int P \geq c$ *are interval-stretch truth-preserving duration formulae, while* $\int P < c$ *and* $\int P \leq c$ *are interval-compress truth-preserving.*

Proof. Consider the proof of $\mathrm{istr}_t(\int P > c)$*:*

$$
\begin{aligned}
& I \vDash_{[b,e]} \int P > c \\
\Longrightarrow\ & \textit{definition of } \int \\
& \textstyle\int_b^e I(P)(t)dt > c \\
\Longrightarrow\ & \textit{definition of } I_s \\
& \textstyle\int_b^e I_s(P)(s(t))dt > c \\
\Longrightarrow\ & \textit{proposition 2} \\
& \textstyle\int_{s(b)}^{s(e)} I_s(P)(t)dt \geq \int_b^e I_s(P)(s(t))dt \\
\Longrightarrow\ & \textit{transitivity of } > \\
& \textstyle\int_{s(b)}^{s(e)} I_s(P)(t)dt > c \\
\Longrightarrow\ & \textit{definition of } \int \\
& I_s \vDash_{[s(b),s(e)]} \textstyle\int P > c
\end{aligned}
$$

The other proofs follow similarly.

Theorem 4. *If* D *and* E *are interval-stretch (interval-compress) truth-preserving duration formulae, then so are the following: (i) true; (ii) false; (iii)* $\ell = 0$*; (iv)* $\lceil P \rceil$*; (v)* $D; E$*; (vi)* $D \wedge E$.

Proof. The proofs of (i), (ii) and (iii) follow directly from the definitions. Consider the proof of (iv) $\mathrm{istr}_t(\lceil P \rceil)$*:*

$$
\begin{array}{ll}
 & I \vDash_{[b,e]} \lceil P \rceil \\
\Longrightarrow & \textit{definition of } \lceil - \rceil \\
 & \int_b^e I(P)(t)dt = e - b \wedge e > b \\
\Longrightarrow & \textit{definition of } I_s \textit{ and monotonicity of time transforms} \\
 & \int_b^e I_s(P)(s(t))dt = e - b \wedge s(e) > s(b) \\
\Longrightarrow & \textit{basic calculus} \\
 & \int_{s(b)}^{s(e)} I_s(P)(t)dt = s(e) - s(b) \wedge s(e) > s(b) \\
\Longrightarrow & \textit{definition of } \lceil - \rceil \\
 & I_s \vDash_{[s(b),s(e)]} \lceil P \rceil
\end{array}
$$

The proof of (v) ($\mathrm{istr}_t(D; E)$*) is as follows:*

$$
\begin{array}{ll}
 & I \vDash_{[b,e]} D; E \\
\Longrightarrow & \textit{definition of } ; \\
 & \exists m : [b, e] \ \cdot \ I \vDash_{[b,m]} D \wedge I \vDash_{[m,e]} E \\
\Longrightarrow & \mathrm{istr}_t(D) \textit{ and } \mathrm{istr}_t(E) \\
 & \exists m : [b, e] \ \cdot \ I_s \vDash_{[s(b),s(m)]} D \wedge I \vDash_{[s(m),s(e)]} E \\
\Longrightarrow & \textit{monotonicity of } f \\
 & \exists m' : [s(b), s(e)] \ \cdot \ I_s \vDash_{[s(b),m']} D \wedge I \vDash_{[m',s(e)]} E \\
\Longrightarrow & \textit{definition of } ; \\
 & I_s \vDash_{[s(b),s(e)]} D; E
\end{array}
$$

The proof of (vi) follows similarly.

Theorem 5. *If D and E are interval-stretch (interval-compress) truth-preserving duration formulae, then so are the following formulae: (i) $\Diamond D$; (ii) $\Box D$; (iii) $D \vee E$; (iv) $\boxminus D$.*

Proof. Using the definition of $\Diamond$, and Theorem 4, it follows directly that if $\mathrm{istr}_t(D)$*, then* $\mathrm{istr}_t(\Diamond D)$*, and similarly if* $\mathrm{icom}_t(D)$*, then* $\mathrm{icom}_t(\Diamond D)$*.*

Recall that $\Box D$ is defined to be $\neg\Diamond\neg D$. Since $\mathrm{istr}_t(D)$*, it follows from Corollary 1 that* $\mathrm{icom}_t(\neg D)$*, and thus* $\mathrm{icom}_t(\Diamond\neg D)$*. Using Corollary 1 again, we get* $\mathrm{istr}_t(\neg\Diamond\neg D)$*. A similar proof can be used for interval-compresses.*

Expressing disjunction in terms of negation and conjunction enables a similar proof. The proof of $\boxminus D$ follows directly from its definition and the other proofs in this section.

Example 1. Recall the example given in section 2.1 — a leak may not last longer than 1 time unit: $\Box(\lceil Leak \rceil \Rightarrow \ell < 1)$. This is equivalent to $\Box(\neg\lceil Leak \rceil \vee \int 1 < 1)$. Using theorem 3, it follows that $\int 1 < 1$ is compress truth-preserving. $\neg\lceil Leak \rceil$ is also compress truth-preserving by Theorem 2 and the fact that $\lceil Leak \rceil$ is stretch truth-preserving (Theorem 4). Finally, using Theorem 5 for disjunction and the

always operator, we can conclude that the original formula is compress truth-preserving. Similarly, one can show that the formula stating that there are at least 30 time units between consecutive leaks: $\Box(\lceil Leak \rceil; \lceil \neg Leak \rceil; \lceil Leak \rceil \Rightarrow \ell \geq 30)$ is stretch truth-preserving.

3.4 Counterexample Traces

Similar results as the ones given in the previous section can be proved about the two new counterexample trace operators.

Theorem 6. *$\nwarrow P$, $\nearrow P$ are interval-stretch and interval-compress invariant.*

From this theorem and Theorem 1, we extend invariance to the operator $\updownarrow P$.

Corollary 2. *$\updownarrow P$, $\not\updownarrow P$ are interval-stretch and interval-compress invariant.*

Theorem 7. *Upper bound counterexample trace formulae are compress truth-preserving, while lower bound counterexample traces are stretch truth-preserving. Length independent formulae are compress and stretch invariant.*

This follows using induction on the structure of the counterexample formulae using the theorems given in this section and the previous one.

Example 2. Recall the definition of the leads-to operator $P \xrightarrow{n} Q$, transformed into a counterexample trace in section 2.2 — $\neg(\text{true}; (\lceil P \rceil \wedge \ell \geq n); \lceil \neg Q \rceil; \text{true})$. Since the formula is an upper bound counterexample trace, one can conclude by Theorem 7 that all formulae of the form $P \xrightarrow{n} Q$ are compress truth-preserving.

All the above theorems enable syntactic analysis of duration formulae to calculate whether they are compress or stretch truth-preserving or invariant. Although, obviously, not a full decision procedure, many useful properties can be proved to fall into one of these categories.

3.5 Summary of Results

A summary of the theorems proved is provided in the table below:[5]

Property\ Fragment	$\int > c$	$\int < c$	$\neg$	;	$\wedge$	$\vee$	$\updownarrow$	$\not\updownarrow$
interval-stretch truth preserving	✓	×	⊠	✓	✓	✓	✓	✓
interval-stretch false preserving	×	✓	⊠	✓	✓	✓	✓	✓
interval-compress truth preserving	×	✓	⊠	✓	✓	✓	✓	✓
interval-compress false preserving	✓	×	⊠	✓	✓	✓	✓	✓

Note that the preserving fragments for interval-stretch truth preserving are equivalent to those for interval-compress false preserving. Similarly, the preserving fragments for interval-compress truth preserving are equivalent to those for interval-stretch false preserving. These results can be readily extended to other syntactic extensions to DC.

[5] ✓ denotes that the fragment preserves the property, × denotes the contrary, while ⊠ signifies that the property is inverted (e.g., if a property is interval-stretch truth preserving, its complement is interval-stretch false preserving).

4 Case Study

In this section, we present a specification of an intrusion detection system in terms of counterexample traces, to show how slowdown and speedup reasoning can be used on a real specification. Although the specification written in terms counterexample traces lacks the readability of one written in full DC, it has the distinct advantage of being automatically implementable as a runtime monitor. It is well known how to translate such formulae into phase event automata (PEAs) [9], which we use, in turn, to produce dynamic automata with timers and events (DATEs) which we have used for runtime verification of real-time properties [6], and are implemented in the tool Larva. The tool implements a transformation from PEAs to DATEs which has been proved to be sound and complete. Full details of the transformation and its proof of correctness can be found in [5].

A number of properties of the network intrusion detection system are expressed as counterexample traces, and used to detect possible malicious activities on a network connection. Each of these properties is stretch truth-preserving, i.e. if the property holds on a system, it will also hold on a slowed-down version of the system. This fact assures us that inserting monitors in the system, will not cause a violation of any of the monitored properties: we will not have false negatives signalled by the monitoring system. In what follows we will give a detailed account of each property of the case-study. Note that some of the variables which are external to the system being monitored are not being considered for the analysis of slowdown and speedup. The section is concluded by a brief account of the runtime verification process as carried out by the tool Larva.

Initiating Connections For strict security concerns, one may wish to disable any incoming TCP packets which do not belong to connections initiated by the host machine being monitored. The initialisation of a TCP connection requires a complete three-way handshake: first a synchronization packet from the client, then a synchronization and acknowledgement packet from the server and another acknowledgement from the client. If the host machine receives a synchronization packet without having sent one beforehand, then an outsider is trying to open a connection. This property may be written as a counterexample trace forbidding and sending of synchronisation packets ($sendSYN$) before a synchronisation packet is received ($receiveSYN$):

$$\neg(\boxminus\ sendSYN; \updownarrow\ receiveSYN; true)$$

Since a TCP connection is a 4-tuple (an address and a port for both the server and the client), this property has to be monitored for each distinct tuple.

Redirect Messages In the case of a machine with a routing table, a lot of ICMP redirect messages can cause the system to slow down. Therefore, there is a need to control the number of such messages. The property, which disallows three redirect messages within less than two time units between subsequent messages, can be written as follows:

$$\neg(true\ ;\updownarrow redirectMsg; \lceil 1 \rceil \wedge \ell < 2; \\ \updownarrow redirectMsg; \lceil 1 \rceil \wedge \ell < 2; \updownarrow redirectMsg; true)$$

Note that $\lceil 1 \rceil$ between the events is not redundant because this ensures that the length of the interval is greater than zero (by definition of $\lceil \cdot \rceil$). Allowing an interval with zero length between two events would mean that the events are in fact the same. The property can be modified slightly to monitor against repeated ping messages and other unwanted (malicious) traffic.

Connection Failure Retries A denial-of-service attack can be carried out by initiating an excessive number of connection initialisations to a server and then leaving the handshake incomplete. The server will have to wait for each of these initialisations to timeout. A simple check would be to limit the number of subsequent failed connection retries originating from the same IP address. The property, which prohibits three consecutive connection failures with less than two time units between each subsequent pair of failures, may be written as follows:

$$\neg(true\ ;\updownarrow failedConn; \lceil 1 \rceil \wedge \ell < 2 \wedge \boxminus successConn; \\ \updownarrow failedConn; \lceil 1 \rceil \wedge \ell < 2 \wedge \boxminus successConn; \\ \updownarrow failedConn; true)$$

To monitor the above property, we can use two sub-properties which define what it means for a connection to be successful (*successConn*) or to fail (*failedConn*). A successful connection consists of a synchronisation packet from a client, the acknowledgement from the server and reciprocating acknowledgement from the client. Similarly, a TCP handshake which is not acknowledged within 5 seconds is considered as a failed connection. A TCP handshake must be monitored at a connection level with the usual four parameters (IP addresses and port numbers) while the monitoring of successive failed connection retries is performed for each individual IP address.

Note that these duration formulae inherently quantify over state variables, exploiting the inherent parametrisation over tuples of objects used in Larva — our duration formulae are thus seen as formulae quantified over state variables at the top-level of the formula. In summary, given a property as a counterexample trace, the following steps are required to monitor a Java program to detect any violations of the property: (i) analyse the counterexample trace with the tool to know whether the counterexample trace is slowdown and/or speedup truth preserving or none; (ii) use the tool to automatically convert the counterexample trace into a Larva script; (iii) relate the monitoring events to system events (such as method calls); (iv) if the property is to be monitored for each object of a particular class, modify the Larva script accordingly; (v) add any Java code to be invoked in case of a violation detection; (vi) compile the script to generate the monitoring system; and (vii) run the Java program with the generated monitoring files in place.

5 Related Work

Monitoring of real-time properties is far from new [4, 11, 10, 16, 3, 21], but to our knowledge, there is no existing work which identifies classes of properties to ensure invariant behaviour of the monitors under slowing down or speeding up. Our approach differs from other approaches [19, 8, 7, 2] in that our theory revolves around the observed behaviour rather than the semantic or syntactic definition of underlying logic, i.e. we analyse the effect on the observation rather than modifying the semantics of any logic. The main motivation behind the related work discussed here is that real-time logic in dense time allows the definition of properties with arbitrary precision. This is technically impossible to achieve from an engineering point of view. A possible solution to this problem is to use a discrete time model to avoid the problem of arbitrary precision. However, this might not always be desirable. For this reason, a robust interpretation of duration calculus has been suggested [7] where the satisfaction of a neighbourhood of formulae is considered rather than that of a single formula. A similar approach has been applied to timed automata [8] where the language accepted by robust timed automata is a set of *tubes* rather than individual trajectories where a tube is a set of *neighbouring* trajectories. A robust reachability analysis for timed automata [19] has been studied such that the analysis is robust with respect to a given maximum drift of the clocks. Rather than a semantic approach, Alur et al. [2], suggest MITL as a solution by allowing the specification of an interval of approximations rather than a constant with arbitrary precision.

Also related to our approach, but in a temporal setting without reference to real-time quantities, is the notion of stutter-invariant properties, corresponding to properties invariant under slowdown [13, 17, 12]. In a discrete time setting, such as LTL and CTL, the family of such properties corresponds to a syntactic restriction on the use of the *next* operator. As we have seen, the notion of stretching and compression invariance in a real-time setting introduces considerable more challenges.

Suggesting a more practical approach, Pohlack et al. [18] deal with slowdown by compensating for the monitoring overheads. The overhead is taken into consideration so that the output of the monitors can be properly adjusted in the case of proliferation.

The most common approach to monitoring real-time properties [4, 11, 10, 16, 21] is based on simple time constraints where each time constraint compares the timestamps of two event occurrences. Such a constraint can be used to represent a delay or deadline constraint where an event should occur after a particular delay or before a certain deadline.

A recurrent problem with a simple treatment of time is that there is typically a lag from the moment an event occurs till the moment it is detected. A solution proposed in [15] is to treat each timestamp as an interval of possible time points at which the event might have occurred. Summapun et al. [21] have integrated these time constraints within the existing MaC framework. The idea of intervals is further investigated in [14] where each time constraint also includes a threshold which the satisfaction probability should exceed. Other approaches to

monitor real-time properties include the work of Bauer et al. [3] which provides a monitoring procedure for TLTL_3.

6 Conclusions

Real-time properties are notorious for their sensitivity to changes in a system. Inevitably, overheads are induced when introducing monitoring in a system. This unintended change, has to be addressed in order to make the verification of real-time properties more reliable. In our approach, we have identified a class of DC which guarantees that slowing down (or speeding up) the system would not break the validity of a formula. These checks have been incorporated in Larva, together with a sound and complete translation from counterexample traces into DATEs — the notation used in Larva. The case study presented shows how this approach can be useful is real scenarios. Though we have instantiated our definitions and results in DC, the approach could be extended to other real-time logics.

Our approach has other applications outside runtime verification. For instance, confidence gained through model-checking or testing of a compress truth-preserving property means that the system may be ported onto a faster machine with full confidence. Moreover, monitors for compress truth-preserving properties ensure that removing a system from its testing environment will not induce it to fail.

An underlying assumption in our work, is that all state variables are affected by the same time transform. This may not always be the case. For example, in a distributed real-time system the state variables on different machines may be differently re-timed. This is much more challenging than simply considering the affect of a single global time transform. Another limitation of this work is that the time transforms are allowed to stretch or compress the time-line in an unbounded manner — this is usually required since it is usually difficult (possibly impossible) to identify the maximum overhead induced by a runtime monitor. However, the approach we adopt enables a succinct characterisation of bounded slowdown, speedup, or even bounded retiming (allowing both speedup or slowdown to a limited degree), which would be interesting to explore in more detail.

Our approach addresses property invariance under re-timing. An interesting research direction is that of looking into the complementary issue of system analysis to ensure whether a program is also slowdown or speedup insensitive, so it will still produce the same traces (albeit with different timestamps). Together with the property analysis, this would enable full analysis of systems under monitor instrumentation.

References

1. A. Bouajjani, Y. Lakhnech, and R. Robbana. From duration calculus to linear hybrid automata. In *CAV'95*, volume 939 of *LNCS*, pages 196–210, 1995.

2. R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996.
3. A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS*, volume 4337 of *LNCS*, pages 260–272, 2006.
4. S. E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *RTSS'91*, pages 74–83. IEEE Computer Society, 1991.
5. C. Colombo. Practical runtime monitoring with impact guarantees of java programs with real-time constraints. Master's thesis, University of Malta, 2008.
6. C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS'08*. To appear in LNCS, 2008.
7. M. Fränzle and M. R. Hansen. A robust interpretation of duration calculus. In *Proceedings of NWPT*, pages 83–85, 2004.
8. V. Gupta, T. A. Henzinger, and R. Jagadeesan. Robust timed automata. In *HART'97*, pages 331–345, London, UK, 1997. Springer-Verlag.
9. J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.
10. F. Jahanian. Run-time monitoring of real-time systems. In *Advances in real-time systems*, pages 435–460. Prentice-Hall, Inc., 1995.
11. F. Jahanian, R. Rajkumar, and S. C. V. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, 1994.
12. Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 273–346, London, UK, 1994. Springer-Verlag.
13. L. Lamport. What good is temporal logic? In *Information Processing 83*, pages 657–668, 1983.
14. C.-G. Lee, P. Konana, and A. K. Mok. Monitoring of timing constraints with confidence threshold requirements. *IEEE Trans. Comput.*, 56(7):977–991, 2007.
15. A. K. Mok, C.-G. Lee, H. Woo, and P. Konana. The monitoring of timing constraints on time intervals. In *RTSS'02*, page 191. IEEE Computer Society, 2002.
16. A. K. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *RTTAS'97*, pages 252–262. IEEE, 1997.
17. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.
18. M. Pohlack, B. Döbel, and A. Lackorzyński. Towards runtime monitoring in real-time systems. In *Eighth Real-Time Linux Workshop*, Lanzhou, China, 2006.
19. A. Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.
20. A. P. Ravn. *Design of Embedded Real-Time Computing Systems*. PhD thesis, Technical University of Denmark, October 1995.
21. U. Sammapun, I. Lee, and O. Sokolsky. Checking correctness at runtime using real-time java. In *JTRES'05*, 2005.
22. M. Schenke and E.-R. Olderog. Transformational design of real-time systems part i: From requirements to program specifications. *Acta Informatica*, 36(1):1–65, 1999.
23. Z. ChaoChen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.