Runtime Verification of Timed Regular Expressions in Larva

Miguel Dingli, supervised by Prof. Gordon J. Pace, and Dr. Christian Colombo
Department of Computer Science, Faculty of ICT, University of Malta
Email: { miguel.dingli.15 | gordon.pace | christian.colombo } @ um.edu.mt

ABSTRACT

In runtime verification, a computer program's expected behaviour is specified using some logic, the choice of which has an effect on the degree of overhead produced by the verification framework at runtime. To minimizing this overhead, multiple logics have been put to the test throughout the years. This project investigates the potential of timed regular expressions as a specification language and, using the Larva RV tool, presents and compares, in terms of overhead, two approaches by which this is realised: (i) timed derivatives, and (ii) state-exploration using timed automata.

1. INTRODUCTION

With the ever-increasing complexity of software systems, there is a similar demand for efficient techniques that facilitate the process of verifying that a system is operating according to its specification while maintaining a high coverage of the system's behaviour. Three traditional techniques are theorem proving, model checking, and testing [15].

Runtime verification (RV) is a more recent technique based on the monitoring of a system at runtime. Monitors compare a system's behaviour to a specification of its expected behaviour. An effective RV solution has a low resource footprint, given that resources are shared with the monitored system, and provides a highly expressive specification language to easily describe expected behaviour. Linear temporal logic [6], regular expressions [17], rule-based solutions [5], and automata [14] are examples of such languages.

1.1 Objectives

Timed regular expressions (TREs) [2] extend the well-known regular expressions with the concept of time so that they can be used to specify timing properties, such as "The server must respond to each request it receives within 3 seconds". The objectives of this project are to explore TREs as a specification language and to implement and compare two approaches by which this is possible using an RV tool.

The proposed approaches are (i) the use of timed derivatives [7, 18] of TREs, and (ii) the translation of TREs into timed automata [1], which shall be evaluated in terms of the CPU and memory overheads that they add to a system.

Two ways of expressing correctness properties are explored: (i) good-behaviour TREs, which indicate what the system must satisfy, and (ii) bad-behaviour TREs, which define what it must not satisfy. These dictate whether a matched trace or failing to match is a positive or negative outcome.

1.2 Solution

As an RV foundation, Larva [10] was used. Larva is built around a rich automaton-based logic called dynamic automata with timers and events (DATEs). The native timers in Larva were crucial to implement the solution, given the theme of time in this project. Larva was used to develop a tool that encompasses the derivatives and timed automata approaches by implementing two translation chains:

1. Timed REs $\longrightarrow$ DATEs
2. Timed REs $\longrightarrow$ Timed Automata $\longrightarrow$ DATEs

For the first approach, a form of operational semantics – timed derivatives – were used. The type of timed automata chosen for the second approach is *non-deterministic* TAs *with silent transitions*, the main reason being their high expressive power [16]. However, Larva does not natively support non-determinism. A major contribution of this project is that it shows how certain non-determinism can be handled by deterministic RV tools using a replication technique.

The implementations of the above translation chains were applied to a mock financial transaction system called FiTS[1] for testing purposes. They were also applied to an open-source FTP server called MinimalFTP[2]. Five FTP properties were specified and the server was accordingly monitored using both approaches. To evaluate the two approaches, the CPU and memory overheads that the monitors due to these properties add to the server were measured and compared.

1.3 Results

Overall, the derivatives approach was found to be superior to the automata approach, both in terms of performance and simplicity of the implementation. We discover that the timed derivatives approach produces less CPU and memory overhead. Memory leaks due to a limitation in the Larva framework led to significant memory overheads for both approaches which increased with the number of FTP clients.

2. RUNTIME VERIFICATION

Runtime verification (RV) is a dynamic software verification technique based on the monitoring of a system during runtime to determine whether it satisfies correctness properties [15] that describe its expected behaviour. Given such properties, an RV tool *instruments* checks into the system [9] and generates the monitor that will observe system events. The checks act as an interface between the monitor and the event trace generated by the running system. The monitor analyses this trace and outputs a verdict corresponding to whether the trace satisfies or violates the required behaviour.

Scalability, But Less Coverage.

As a system grows, the increase in possible execution paths makes exhaustive verification an intractable task [9]. RV monitors only verify the actual path of execution. However, this reduces the overall coverage of the program.

Ongoing Fault Detection, But With Overheads.

RV can even be used in release versions of a program to serve as a runtime safety net that can actively react to faults [8]. However, a challenge in RV is to limit the monitoring overhead, since resources are shared with the monitored system. This is typically achieved by reducing the expressive power of the property language [9]. Balancing these presents an extra challenge. Also, the satisfaction of *real-time* properties is sensitive to changes in resource availability. Adding or removing monitors may affect property satisfaction [11].

[1]FiTS documentation: https://goo.gl/c5v53S
[2]https://github.com/Guichaguri/MinimalFTP

2.1 Methods for Runtime Verification

This section will identify two approaches to RV which differ in (i) the formalism used and (ii) the detection of violations, and also in (a) the compactness of the specification language and (b) the processing required for each event that occurs. The below discussions provide a foundation for the two *timed* approaches that will be presented further on.

2.1.1 Explicit State Exploration

The explicit state exploration approach involves the pre-runtime construction of the full structure that will be traversed by the monitor in response to events. This typically implies the use of automata, with edges labelled with events that dictate what behaviour is allowed from each state.

EXAMPLE 2.1. *Figure 1 indicates that after an input, process, and output, the system is allowed to stop (S4) or repeat.*

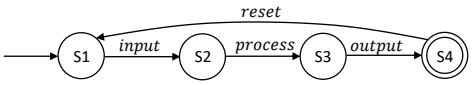


Figure 1: Simple automaton

A violation occurs when there is available transition. For a *liveness* property, satisfaction is when an accepting states is reached. Otherwise, a *safety* property can be considered satisfied only if it is not violated until the end of execution.

2.1.2 Derivative-Based Approach

The symbolic on-the-fly computation of derivatives of expressions will be presented in terms of, but is not limited to, regular expressions (REs) [7]. The language of an RE is the set of all sequences of symbols that it matches. A derivative determines the remaining sequences that the language contains after considering a prefix of symbols. At runtime, derivatives *with respect to* events are computed cumulatively.

EXAMPLE 2.2. *The language of the RE-equivalent of Figure 1, $re_1 = (i.p.o.(r.i.p.o)^*)$, is the set of words $\mathscr{L}(re_1) = \{ipo, iporipo, iporiporipo, ...\}$. The derivative re_1' of re_1 w.r.t 'ipo' has the language $\mathscr{L}(re_1') = \{\varepsilon, ripo, riporipo, ...\}$, where the empty word ε is the equivalent of reaching a final state.*

The property is violated when the cumulative derivative's language becomes empty. The satisfaction of a liveness property represented by an RE is when the derivative's language contains the empty word ε. Once again, a *safety* property is only satisfied if it is not violated until the end of execution.

2.2 Larva

We now present Larva [10], the RV tool used in this project.

2.2.1 Specification Language

Larva's specification language is a formalism based on timed automata (TAs) called dynamic automata with timers and events (DATEs). TAs consist of a set of states joined by transitions, each having a triggering event, an optional set of clock conditions (*guard*), and an optional set of clock resets. DATE transitions generalise this concept by using an '*event**guard**actions*' template, where the guard and actions are respectively any Java condition and statements. Given a textual DATE, the Larva compiler generates the monitoring code and event-extraction code. The following are features of DATEs that are important to this project.

Clocks. Besides checking a clock's time and resetting it, Larva clocks can be used as stopwatches to generate timeout events upon reaching a specific value. They can be paused or stopped to temporarily or permanently disable their events.

Channels. These are a means for multiple DATEs to communicate. This communication triggers channel events which a DATE can use to reactively move between states.

Variables. Besides clocks and channels, DATEs can declare and use any Java variables. The main attraction of variables is that they can be used to simplify an automaton.

Foreach. This allows DATEs to be parametrised in terms of a placeholder object of a Java class so that a state-machine is created for each object of the class, rather than only one.

2.2.2 Event Extraction and Monitoring

For event extraction, Larva uses AspectJ[3], which is based on *aspects* that are *weaved* into the system. This creates an interface for the monitor to analyse the system events and to send feedback. AspectJ aspects attach to method calls, and so Larva monitors deal with a sequence of such calls.

2.2.3 Determinism

Larva DATEs are deterministic by virtue of how a transition is chosen when an event occurs. The selected transition is the first eligible one in an ordered collection. Eligibility depends on the event that occurs and the transition guards.

3. TIMED FORMALISMS

The syntax and semantics of timed regular expressions (TREs) and timed automata (TAs) will now be defined.

3.1 Timed Regular Expressions

Timed regular expressions (TREs) [2] extend classical REs with *timed words*, i.e. sequences of events with duration information, and an interval operator, which places time restrictions on the timed words. The following definition is a modified version of the syntax presented by Asarin et al. [2].

DEFINITION 3.1 (SYNTAX OF TREs). *Timed regular expressions $\mathbb{E}$ over an alphabet Σ are defined using BNF as:*

$$\text{Base} ::= 0 \mid 1 \mid ? \mid a \mid {\sim}a$$
$$\mathbb{E} ::= \text{Base} \mid \mathbb{E}^* \mid \mathbb{E}.\mathbb{E} \mid \mathbb{E} \,\&\, \mathbb{E} \mid \mathbb{E} + \mathbb{E} \mid \langle \mathbb{E} \rangle_I$$

where 0 matches nothing, 1 matches ε, ? matches any $a \in \Sigma$, a matches symbol $a \in \Sigma$, $\sim a$ matches any $b \in \Sigma \setminus \{a\}$, $\mathbb{E}^*$ is Kleene closure, $\mathbb{E}.\mathbb{E}$ is concatenation, $\mathbb{E} \,\&\, \mathbb{E}$ is intersection, $\mathbb{E} + \mathbb{E}$ is union, and $\langle \mathbb{E} \rangle_I$ is the time restriction operator, where I is a time interval $[t, t')$, where $t, t' \in (\mathbb{R}_{\geq 0} \cup \{\infty\})$.

EXAMPLE 3.1. *The property "Every three bad logins must be followed by a good login within 5 seconds" can be expressed using the good-behaviour TRE $((\sim b)^*.b.b.b.\langle g \rangle_{[0,5)})^*$, while a bad-behaviour TRE to represent "There can never be three bad logins within 10 seconds" is $(?)^*.b.\langle b.b \rangle_{[0,10)}$.*

TREs are based on events $a \in \Sigma$ having an implicit arbitrary time value $t \in \mathbb{R}_{\geq 0}$, collectively ${}_t a$, where t is the time elapsed from the previous event, or the start of the trace, and the occurrence of a. If t is unrestricted, the event associated with symbol a can occur after any duration of time. The interval operator $\langle e \rangle_{[t,t')}$ requires the timed words satisfying 'e' to have a duration (the sum of the time values t'' of all constituent events ${}_{t''} a$) within the interval $[t, t')$. The below semantics are based on those of Asarin et al. [2].

DEFINITION 3.2 (SEMANTICS OF TREs). *The semantics $[\![\ ]\!] : \mathbb{E} \to Seq$ of timed regular expressions, where the function* len$: Seq \to \mathbb{R}_{\geq 0}$ *denotes the time duration of a timed word, is given by the following set equalities:*

[3]https://www.eclipse.org/aspectj/

$$[\![0]\!] = \emptyset \qquad [\![e^*]\!] = \bigcup_{i=0}^{\infty} [\![\underbrace{e \cdot ... \cdot e}_{i \text{ times}}]\!]$$
$$[\![1]\!] = \{\varepsilon\}$$
$$[\![?]\!] = \bigcup_{a \in \Sigma} [\![a]\!] \qquad [\![e_1.e_2]\!] = [\![e_1]\!] \cdot [\![e_2]\!]$$
$$[\![e_1 \ \&\ e_2]\!] = [\![e_1]\!] \cap [\![e_2]\!]$$
$$[\![a]\!] = \{{}_t a \mid t \in \mathbb{R}_{\geq 0}\} \quad [\![e_1 + e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$$
$$[\![\sim a]\!] = [\![?]\!] \setminus [\![a]\!] \qquad [\![\langle e \rangle_{[t,t')}]\!] = \{s \in [\![e]\!] \mid \text{len}(s) \in [t, t')\}$$

EXAMPLE 3.2. *The semantics of a TRE for "Once 'a' occurs, 'b' and 'c' must occur after 5 seconds, but up to 10" is:* $[\![a.\langle b.c\rangle_{[5,10)}]\!] = \{{}_{t_1} a \mid t_1 \in \mathbb{R}_{\geq 0}\} \cdot \{s \in [\![b.c]\!] \mid len(s) \in [5, 10)\}$.

3.1.1 Parametrised Events

TREs can be extended with the concept of parametrised events, using which properties can apply to each element of a particular group in the system rather than globally.

EXAMPLE 3.3. *Consider that any session 's' starts with a login, consists of multiple reads and writes, and ends with a logout. We narrow our scope to individual sessions as follows:* $\mathbf{login(s)}.(\mathbf{read(s)} + \mathbf{write(s)})^*.\mathbf{logout(s)}$, *such that the alphabet is* $\Sigma = \{login(s), ...\}$ *and not* $\Sigma = \{login, ...\}$.

Properties expressed using parametrised events are considered violated if at least one instance violates the TRE, whereas satisfaction is if no instance violates the property.

3.2 Timed Automata

Timed automata (TAs) [1] extend finite state automata with the notion of time by adding a finite set of real-valued clocks to measure the time that elapses between the traversal of two or more transitions, which is possible by a clock-resetting mechanism and clock constraints in transitions. The below syntax definition is that of Asarin et al. [2].

DEFINITION 3.3 (TIMED AUTOMATA). *A timed automaton is a tuple* $\mathscr{A} = (Q, C, \Delta, \Sigma, s, F)$ *where* Q *is a set of states,* C *is a set of clocks,* Σ *is the alphabet,* Δ *is a transition relation (below),* $s \in Q$ *is the initial state, and* $F \subset Q$ *is a set of accepting states.* Δ *consists of tuples* (q, ϕ, ρ, a, q') *where* $q \in Q$, $q' \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $\rho \subseteq C$ *is the set of clocks to reset, and* ϕ *(the guard) is a boolean combination of formulae* $(x \in I)$ *for some clock* x *and some integer-bounded interval* I *that must be satisfied for the transition to be traversable.*

Asarin et al. also defined a *clock valuation* as $\mathbf{v}: C \to \mathbb{R}_+$, which returns the time of a clock, and the *configuration* of a TA as a pair $(q, \mathbf{v}) \in Q \times \mathscr{H}$, consisting of the current state and current clock valuation, where $\mathscr{H} = \mathbb{P}(C \to \mathbb{R}_+)$. Additionally, every subset $\rho \subseteq C$ induces a *reset function* $Reset_\rho: \mathscr{H} \to \mathscr{H}$ which resets the clocks in ρ to zero.

EXAMPLE 3.4. *Consider* $\mathscr{A} = (\{S1, S2, S3, S4\}, \{c\}, \Delta, \Sigma, S1, \{S4\})$ *where* $\Sigma = \{in, pr, out, rst\}$ *and* $\Delta = \{(S1, true, \{c\}, in, S2), (S2, true, \emptyset, pr, S3), (S3, c < 10, \emptyset, out, S4), (S4, \phi, \rho, rst, S1)\}$. *This extends Figure 1 with the added constraint that the output has to be produced within 10 seconds.*

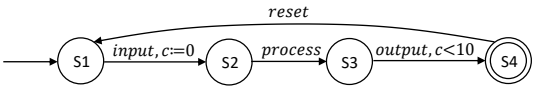


Figure 3: Simple timed automaton

The notions of a *step*, *run*, *acceptance*, and *violation*, based on the definitions of Asarin et al. [2], will now be presented.

DEFINITION 3.4 (STEPS, RUNS, ACCEPTANCE, VIOLATION). *A* ***step*** *is either (i) discrete* $(q, \mathbf{v}) \xrightarrow{a} (q', \mathbf{v}')$ *where* $a \in \Sigma \cup \{\varepsilon\}$ *and* $\exists (q, \phi, \rho, a, q') \in \Delta$ *s.t.* $\mathbf{v}$ *satisfies* ϕ *and* $\mathbf{v}' = Reset_\rho(\mathbf{v})$, *or (ii) timed:* $(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1})$ *where* $t \in \mathbb{R}_+$, $\mathbf{v}$ *is treated as a* $|C|$*-dimensional vector and* $\mathbf{1}$ *is the vector* $(1, ..., 1)$. *A* ***finite run*** *is a finite sequence of steps:* $(q_0, \mathbf{v}_0) \xrightarrow{z_1} (q_1, \mathbf{v}_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, \mathbf{v}_n)$, *the* ***trace*** *of which is the sequence* $z_1.z_2.\cdots.z_n$. *A* ***trivial run*** *is just a configuration* $(q, \mathbf{v})$ *with an empty trace* (ε). *An* ***accepting run*** *starts from* $(s, \mathbf{0})$, *where* $\mathbf{0}$ *is the vector* $(0, ..., 0)$, *and terminates by a discrete step to a final state; i.e.* $q_n \in F$ *and* z_n *is a discrete step. The* ***language*** *of a TA consists of all the traces of its accepting runs. A* ***violating run*** *starts from* $(s, \mathbf{0})$ *and terminates by a discrete step* $(q_n, \mathbf{v}) \xrightarrow{a} (q_{bad}, \mathbf{v}')$ *or time step* $(q_n, \mathbf{v}) \xrightarrow{t} (q_n, \mathbf{v} + t\mathbf{1})$, *where* $q_n \notin F$ *and* $q_{bad} \in \Sigma \uplus \{q_{bad}\}$, *s.t. no further discrete steps are possible.*

EXAMPLE 3.5. *An accepting run example for the automaton in Figure 3 is* $(S1, \mathbf{0}) \xrightarrow{3} (S1, \mathbf{3}) \xrightarrow{in} (S2, \mathbf{0}) \xrightarrow{pr} (S3, \mathbf{0}) \xrightarrow{5} (S3, \mathbf{5}) \xrightarrow{out} (S4, \mathbf{5})$ *whereas a violating run is* $(S1, \mathbf{0}) \xrightarrow{in} (S2, \mathbf{0}) \xrightarrow{5} (S2, \mathbf{5}) \xrightarrow{rst} (S_{bad}, \mathbf{5})$, *for some implicit bad state* S_{bad}. *A timeout-induced violaton is* $(S1, \mathbf{0}) \xrightarrow{9} (S1, \mathbf{9}) \xrightarrow{in} (S2, \mathbf{0}) \xrightarrow{pr} (S3, \mathbf{0}) \xrightarrow{15} (S3, \mathbf{15})$.

4. MONITORING OF TIMED LOGICS

We now present the approaches used to convert the two timed logics to Larva DATEs. Since the starting point is TREs, the translation from TREs to TAs is also presented.

4.1 Timed Regular Expressions

We build on Section 2.1.2 by formally defining *timed* derivatives that cater for the passage of time. The following intermediary operators will be assumed throughout this section: **(i)** $\forall e: \mathbb{E} \cdot \lambda(e) \Leftrightarrow \varepsilon \in [\![e]\!]$; **(ii)** $\forall e: \mathbb{E} \cdot \omega(e) \Leftrightarrow [\![e]\!] = \emptyset$; and **(iii)** $\forall e: \mathbb{E} \cdot \nu(e) = e': \mathbb{E}$ s.t. $[\![e']\!] = [\![e]\!] \setminus \{\varepsilon\}$.

4.1.1 Timed Derivatives

To extend the derivatives of Brzozowski [7] to *timed* regular expressions, we define two types of timed derivatives.

DEFINITION 4.1 (TYPE 1 TIMED DERIVATIVES). *Given the time elapsed* $\delta t \in \mathbb{R}_{\geq 0}$, *the derivative* $e': \mathbb{E}$ *of* $e: \mathbb{E}$ *w.r.t* δt *is denoted by* $e \overset{\delta t}{\rightsquigarrow} e'$, *where* $\rightsquigarrow \subseteq \mathbb{E} \times \mathbb{R}_{\geq 0} \times \mathbb{E}$ *and* $\rightsquigarrow$ *is functional on* $\mathbb{E} \times \mathbb{R}_{\geq 0}$, *such that* $[\![e']\!] = \{\mathbf{s} \mid {}_{\delta t}\varepsilon.\mathbf{s} \in [\![e]\!]\}$.

DEFINITION 4.2 (TYPE 2 TIMED DERIVATIVES). *Given (i) an event* $a \in \Sigma$, *and (ii) the time elapsed* $\delta t \in \mathbb{R}_{\geq 0}$, *the derivative* $e': \mathbb{E}$ *of* $e: \mathbb{E}$ *w.r.t* a *and* δt *is denoted by* $e \xrightarrow{a}_{\delta t} e'$, *where* $\longrightarrow \subseteq \mathbb{E} \times \Sigma \times \mathbb{R}_{\geq 0} \times \mathbb{E}$ *and* $\longrightarrow$ *is functional on* $\mathbb{E} \times \Sigma \times \mathbb{R}_{\geq 0}$, *such that* $[\![e']\!] = \{\mathbf{s} \mid {}_{\delta t}a.\mathbf{s} \in [\![e]\!]\}$.

The timed words in $[\![e']\!]$ are identical to those of $[\![e]\!]$ starting with ${}_{\delta t}\varepsilon$ (or ${}_{\delta t}a$), but with the ${}_{\delta t}\varepsilon$ (or ${}_{\delta t}a$) prefix removed. In def^{n} 4.1, the ε in ${}_{\delta t}\varepsilon$ indicates that no event is consumed. These definitions are better understood through rules such as those below, but these were excluded due to limited space.

$$\frac{u: Base}{u \overset{\delta t}{\rightsquigarrow} u} \qquad \frac{e \overset{\delta t}{\rightsquigarrow} e' \qquad f \overset{\delta t}{\rightsquigarrow} f' \qquad \lambda(e)}{e.f \overset{\delta t}{\rightsquigarrow} e'.f + f'}$$

$$\frac{a \neq b}{a \xrightarrow{b}_{\delta t} 0} \qquad \frac{e \xrightarrow{a}_{\delta t} e' \qquad \delta t \in [t, t')}{\langle e \rangle_{[t,t')} \xrightarrow{a}_{\delta t} \langle e' \rangle_{[t \ominus \delta t, t' - \delta t)}}$$

EXAMPLE 4.1. *The following are three derivative chains:*

$a.\langle b.c\rangle_{[5,10)} \overset{11}{\rightsquigarrow} a.\langle b.c\rangle_{[5,10)} \xrightarrow{a}_1 \langle b.c\rangle_{[5,10)} \xrightarrow{b}_1 \langle c\rangle_{[4,9)} \xrightarrow{c}_1 0$

$a.\langle b.c\rangle_{[5,10)} \xrightarrow{a}_7 \langle b.c\rangle_{[5,10)} \overset{2}{\rightsquigarrow} \langle b.c\rangle_{[3,8)} \xrightarrow{b}_5 \langle c\rangle_{[0,3)} \xrightarrow{c}_4 0$

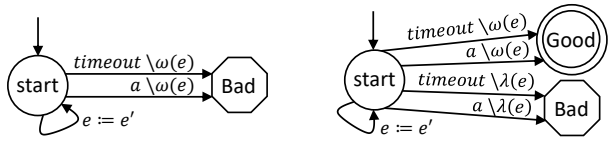
$a.\langle b.c\rangle_{[5,10)} \xrightarrow{a}_1 \langle b.c\rangle_{[5,10)} \xrightarrow{b}_3 \langle c\rangle_{[2,7)} \overset{1}{\rightsquigarrow} \langle c\rangle_{[1,6)} \xrightarrow{c}_5 1$

4.1.2 Timed Derivatives in DATEs

Derivatives are not associated with state machines. However, since these will be used in the automaton-based Larva DATEs, a minimum of three states is required in the general case: (i) the start state, from which derivatives are computed, (ii) the bad state, reached upon violation of the property, and (iii) the good state, reached upon its satisfaction.

The notions of good and bad behaviour must be taken into consideration, since they determine which state, good or bad, the DATE transitions to when the TRE is violated or satisfied. The violation of a good (bad) behaviour TRE is a negative (positive) outcome, and thus the property that it represents is violated (satisfied). Moreover, satisfying a bad-behaviour TRE is a negative outcome, but for a good-behaviour TRE, assumed to represent a safety property, we require the continuous non-violation of the TRE. In the context of DATEs, the good state is never reached in this case.

Figure 4 summarises this section through two DATEs originating from a good and bad-behaviour TRE, respectively. The a-transitions represent the transitions for any $a \in \Sigma$. Timeout transitions are triggered implicitly whenever any time window expires. The transition $e := e'$ computes the cumulative derivative of the original TRE and is implicitly traversed for any event or timeout before considering the other transitions. In the case of timeouts, $e \overset{\delta t}{\rightsquigarrow} e'$, while for an event a, $e \xrightarrow{a}_{\delta t} e'$, where δt is the elapsed time.



(a) Good behaviour DATE **(b) Bad behaviour DATE**

Figure 4: The two main types of DATEs for derivatives

4.2 Timed Automata

The translation from TAs to DATEs is simplified by the fact that DATEs are a superset of TAs. However, TAs will be derived from TREs. Also, since non-deterministic timed automata (NTAs) cannot be directly represented using the deterministic DATEs, some changes are necessary. In practice, non-determinism is typically unavoidable. Unlike their untimed counterparts, which have equal expressive power, deterministic timed automata (DTAs) are strictly less expressive than NTAs, which are themselves less expressive than NTAs with non-observable silent (ε) transitions [16].

The extra expressive power of NTAs with ε transitions makes them more desirable. However, methods to convert NTAs to DTAs apply only to a subclass of NTAs, such as with length-bound traces [16], or cause an exponential blow-up in the size of the resultant DTA [4]. To use DATEs, we found a way to encode ε transitions and a limited degree of non-determinism using an automaton replication approach, implemented using channels and Foreach constructs. Thus, the following chain of translations will be presented:

$$\text{(TREs)} \longrightarrow \text{(NTAs with } \varepsilon \text{ transitions)} \longrightarrow \text{(DATEs)}$$

4.2.1 From TREs to TAs

An approach to translate from TREs to timed automata was defined by Asarin et al. [2]. Given its size and the space limitations, only some representative parts of the definition, which was slightly modified, will be presented.

DEFINITION 4.3 (TRE TO TA). *Let $\mathscr{A}_1 = (Q_1, C_1, \Delta_1, \Sigma, s_1, F_1)$ and $\mathscr{A}_2 = (Q_2, C_2, \Delta_2, \Sigma, s_2, F_2)$ be the timed automata accepting the languages $[\![e_1]\!]$ and $[\![e_2]\!]$ respectively, for $e : \mathbb{E}$ and $e' : \mathbb{E}$. It is assumed that Q_1 and Q_2 as well as C_1 and C_2 are disjoint and that $a \in \Sigma$.*

- *TA for $[\![a]\!]$ is $(\{s, f\}, \emptyset, \{(s, true, \emptyset, a, f)\}, \Sigma, s, \{f\})$.*
- *TA for $[\![e_1 + e_2]\!]$ is $(Q_1 \cup Q_2 \cup \{s\}, C_1 \cup C_2, \Delta, \Sigma, s, F_1 \cup F_2)$, where Δ is built by adding to $\Delta_1 \cup \Delta_2$ two new ε transitions $(s, true, C_i', \varepsilon, s_i)$, where $i \in \{1, 2\}$ and $C_i' = \{c \in C_i \mid \neg\exists\, (q_1', \phi', \rho', a', q_2') \in \Delta_i \cdot (q_2' \neq s_i \wedge c \in \rho')\}$.*
- *TA for $[\![e_1.e_2]\!]$ is $(Q_1 \cup Q_2, C_1 \cup C_2, \Delta, \Sigma, s_1, F_2)$, where Δ is built from $\Delta_1 \cup \Delta_2$ by adding for any $(q_1, \phi, \rho, a, f_1) \in \Delta_1$ with $f_1 \in F_1$ a new $(q_1, \phi, C_2', a, s_2)$ where $C_2' = \{c \in C_2 | \neg\exists\, (q_1', \phi', \rho', a', q_2') \in \Delta_2 \cdot (q_2' \neq s_2 \wedge c \in \rho')\}$.*
- *TA for $[\![\langle e_1 \rangle_I]\!]$ is $(Q_1 \cup \{f\}, C_1 \cup \{x\}, \Delta, \Sigma, s_1, \{f\})$, where Δ is obtained by adding for any $(q, \phi, \rho, a, f_1) \in \Delta_1$ with $f_1 \in F_1$ a new transition $(q, \phi \wedge (x \in I), \rho, a, f)$.*

Ignoring trivial changes to the definition of Asarin et al., the conditions of the form $C_i' = \{c \in C_i \mid \neg\exists(q_1', \phi', \rho', a', q_2') \in \Delta_i \cdot (q_2' \neq s_i \wedge c \in \rho')\}$, for some i, were added to the TAs for $[\![e_1^*]\!]$, $[\![e_1 + e_2]\!]$, and $[\![e_1.e_2]\!]$ to remove unnecessary clock resets, thus preventing timeouts from being set prematurely. Scenario (ii) in Section 4.2.2 will discuss this further.

4.2.2 From TAs to DATEs

The basic aspects of the TAs resulting from Definition 4.3, such as states, transitions, and clocks, are natively available in Larva DATEs and so they were represented directly. This section will focus on satisfaction and violation, and an approach to deal with non-determinism and silent transitions, both of which are not supported natively by DATEs.

Violation and Acceptance of the Specification.

Good and bad behaviour once again dictate which state the DATE transitions for accepting or violating runs. Acceptance is is similar to that for TREs, with the assumption made also applying to TAs. In this case, accepting traces are defined by the final states. On the other hand, violating traces can occur in two ways: (i) the occurrence an event for which no transition exists, or (ii) a clock timeout:

(i) For the first scenario, the DATE was made total with respect to each event, where the new transitions lead to the bad state (good state, in the case of bad behaviour) and are only considered if no other transition is eligible.

(ii) For the second scenario, for any clock c associated with a condition $c < t$, a timeout of t time units is set upon any reset of the clock. If the condition is reached before the timeout occurs, the timeout is cancelled. Otherwise, the timeout event occurs and the TA transitions to the bad (or good) state. The TA was also made total w.r.t timeout events, so that these are acknowledged from any state. Given that any reset of a Larva clock starts its associated timeout, if any, the elimination of extra clock resets in the translation from TREs to TAs (Section 4.2.1) means that timeouts are set only when necessary, thus preventing spurious timeouts.

Handling Silent Transitions and Non-Determinism.

Silent transitions (STs) and non-determinism (ND) are not natively supported in Larva DATEs. Thus, an approach to deal with both of these missing aspects was designed. Points of ND in an automaton are like alternate realities; one for each possible path. This was interpreted as creating multiple copies of an automata, one for each possible path.

Whenever a point of ND or STs is reached, an automaton copy is created for each ST or possible ND path for which the transition guard is satisfied. Copies are simulated using a Larva Foreach construct so that only the automaton's configuration has to be copied. Using Larva channels, copies are sent the original automaton's configuration, through which

they are instructed to skip to the original automaton's state and are given the values of its clocks. In the case of ND, a dummy state is created to consume the event. Due to replication, violation and satisfaction are updated as follows:

- For a property represented by a good (bad) behaviour TA to be *violated* (*satisfied*), the traces of *all* copies have to be violating, i.e. all copies reach the *bad* (*good*) state.
- For a property represented by a bad-behaviour TA to be *violated*, the trace of *at least one* copy has to be an accepting trace, i.e. at least one reaches the *bad* state.
- Properties represented by a good-behaviour TA are assumed to be safety properties and thus never satisfied.

5. ARCHITECTURE OF THE TOOL

The implemented tool[4] and the testing performed will be outlined in this section. The core of the implementation is the TRE library, comprised of the representations of TREs and TAs and a TRE parser. These are used by a Larva Generator to construct a Larva script based on the selected approach (derivatives or TAs). The generated script is compiled by the Larva Compiler. The AspectJ compiler is then executed on its output, the system to be monitored, and a set of implemented Larva Tools, which extend the runtime functionality of the DATEs. To implement the parametrised events extension of TREs, nested Larva Foreach constructs were used, one for each unique parameter in the TRE.

Extensive testing was performed. The components of the TRE Library were all unit-tested using JUnit[5] and Mockito[6], including some integration tests. A series of monitoring tests were also performed, using a simplified financial transaction system called FiTS as a dummy system, to test the whole process from start to finish, targeting various areas.

A known limitation of the tool is that the automaton replication may cause an infinite loop if the original TRE contains any e^* where $\lambda(e)$, i.e. $\varepsilon \in e$. This triggers a loop that does not expect any event. Since $\lambda(e^*)$, irrespective of e, a solution would be to apply the empty string eliminator (ν from Section 4.1) on e for any e^* throughout the TRE.

6. REAL-WORLD USE CASE

This section will present a real-world use case of the implemented tool along with a collection of five properties.

The selected real-world use case is an open-source Java FTP server called MinimalFTP[7], created by Guichaguri. The *real-world* nature of this use case shows that the solution is not only applicable to purpose-built test systems. MinimalFTP implements FTP commands as methods, and so a lot of observable events are generated, which properly push the RV solution. Lastly, this use case adds to existing work that also dealt with Larva and an FTP server[13].

The following are the properties defined for the use case:

1. For any command that the FTP server processes from the FTP client, a response should be sent within 1 second.

2. Connection commands, such as `syst`, `stat`, `mode`, `pasv`, `port`, requiring authentication must be preceded by it.

3. An anonymous client can only perform up to five file transfers and cannot start a transfer after one hour has elapsed from the start of the session.

4. Upon connection to the FTP, if a username or password is required, these must be supplied before the user is authenticated. Only then is the user allowed to transfer files.

5. As protection against DoS attacks, a user cannot initiate more than four file transfers in a timespan of 10 seconds.

MinimalFTP only violates 3 and 5. These were successfully detected by the monitoring code using any of the two implemented approaches, and no false alarms were raised.

7. EVALUATION

In this section, we turn to the original objective of this project; that of comparing the two implemented translation chains. This was done by monitoring the use case, MinimalFTP, based on the five properties. For each property and for each of the two approaches, i.e. derivatives and timed automata, the overhead added to the system was measured. On top of the previously discussed reasons, the suitability of MinimalFTP is also its minimality, which makes it easier to focus on measuring overheads and improves the reliability of results. This distances the results from the specific use case and makes them applicable at a more general level.

7.1 Evaluation Strategy

Scenario methods (Table 1) were implemented to interact with the FTP server programmatically using Apache Commons Net[8]. These are characterised by the number of users that they create and the empty-file transfers (alternating uploads and downloads) per user, depending on the property 'P' for which scenario 'P.X' was defined. Scenarios were paired up to observe the changes in overhead as the number of users or transfers doubles. Hereafter, 'scenario S' will refer to the pair 'scenario S.1' and 'scenario S.2'.

Table 1: The scenario methods

Scenario	1.1	1.2	2.1	2.2	3.1	3.2	4.1	4.2	5.1	5.2
No. Users	1	1	1000	2000	500	1000	500	1000	1	1
Trans/User	1000	2000	0	0	1	1	1	1	500	1000
Anonymous	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓

A set of experiments was formulated based on the properties and scenarios. The monitoring overhead was measured using Java VisualVM[9] by considering the following:

1. Peak Memory Usage (PMU): The peak memory usage was recorded with 10 samples per second. Garbage collection was performed using VisualVM before running a scenario. For any scenario $s \in \{1.1, ...5.2\}$ and approach $a \in \{Deriv, Autom\}$, the percentage increase relative to the same experiments run without monitors is:

$$\Delta\text{PMU}_{(s,\,a)} = \frac{\text{PMU}_{(s,\,a)} - \text{PMU}_{(s,\,\text{NoMon})}}{\text{PMU}_{(s,\,\text{NoMon})}} \times 100$$

2. Average Monitoring Time (AMT): For the CPU overhead, the percentage of time spent in any part of the monitoring code was found. No monitorless experiments were performed, given that this percentage is relative to the FTP server without monitors. For any scenario $s \in \{1.1, ...5.2\}$ and approach $a \in \{Deriv, Autom\}$, the percentage of monitoring relative to time spent on FTP tasks is:

$$\Delta\text{AMT}_{(s,a)} = \frac{\text{AMT}_{(s,a)}}{100\% - \text{AMT}_{(s,a)}} \times 100$$

7.2 Experiment Results

Experiments were performed three times and on the same Windows 10 system with an Intel Core i5-3210M 2.5GHz CPU and 6GB of RAM. The averaged readings are presented in Tables 2 and 3, where NoMon, Deriv, and Autom respectively refer to the monitorless experiments and those for the derivative-based and timed-automaton-based approaches.

[4]https://github.com/migueldingli1997/FYP
[5]https://junit.org/junit5/
[6]http://site.mockito.org/
[7]https://github.com/Guichaguri/MinimalFTP
[8]https://commons.apache.org/proper/commons-net/
[9]docs.oracle.com/javase/8/docs/technotes/guides/visualvm/

Table 2: Peak Memory Usage

Scenario		1.1	1.2	2.1	2.2	3.1	3.2	4.1	4.2	5.1	5.2
PMU (KB)	NoMon	8063	9104	11034	11024	10384	8974	12139	10379	8060	8583
	Deriv	25897	26267	86089	230517	46994	82189	45357	77793	26749	27346
	Autom	∞	∞	69624	150204	66376	109663	68014	277957	27398	27301
ΔPMU (%)	Deriv	221	189	680	1991	353	816	274	649	232	219
	Autom	∞	∞	531	1262	539	1122	460	2578	240	218

Table 3: Average Monitoring Time

Scenario		1.1	1.2	2.1	2.2	3.1	3.2	4.1	4.2	5.1	5.2
AMT (%)	Deriv	13.50	12.20	8.07	8.83	37.03	47.80	37.27	46.33	0.63	0.37
	Autom	∞	∞	19.90	22.90	78.83	91.43	77.47	92.53	1.23	0.73
ΔAMT (%)	Deriv	15.61	13.90	8.77	9.69	58.81	91.57	59.40	86.34	0.64	0.37
	Autom	∞	∞	24.84	29.70	372.44	1067.32	343.79	1239.29	1.25	0.74

PMU Analysis. Considering Table 2, an issue of very high memory usages can be noticed, with the ΔPMU ranging from 189% up to 2578%. These are mainly due to a limitation of Larva, which relies on hashmaps that store references to objects of the system being verified. These, and recursively any objects that they refer to, cannot be garbage collected. The increase in overhead between pairs of scenarios is also due to this issue. Overall, the ΔPMU values show that the timed automata approach typically reached higher peaks, except in scenario 2, but especially in scenario 4.

AMT Analysis. The results in Table 3 are not as worrying, but the ΔAMT is still very high for the automaton approach in scenarios 3 and 4. The AMT values indicate that the automata approach generally took a chunk of time around 2 to 3 times greater than the derivative-based approach, which translates to a 2 to 14 times increase in ΔAMT, the greatest increase once more being due to scenario 4 using the automata approach. This is most likely due to the complexity of property 4, which contains three possible paths for authentication, followed by a tight loop of file transfers, both of which involve non-determinism and ε transitions. The significant increase in ΔAMT between each pair $S.1$ and $S.2$ was not expected, but since this is only for scenarios in which the number of users increase (2 to 4), it is very likely due to resource leaks in the monitoring code.

8. CONCLUSIONS AND FUTURE WORK

This project explored the suitability of TREs as a formal specification language in runtime verification. This involved the implementation of two approaches; one based on timed derivatives, and the second based on non-deterministic timed automata with ε transitions. This was achieved by developing a tool to translate TREs into Larva's native language.

Due to time restrictions, only five properties were successfully applied to the FTP server use case. These still covered both good and bad-behaviour TREs, all operators, non-determinism, and parametrised events. The objective was ultimately to compare the two approaches. Through the evaluation, we conclude that the timed derivatives approach outperforms the timed automata approach on the basis of both the lower overhead of the resultant monitors and a simpler implementation. Further contributions of the project include the extension of the derivatives [7] to timed derivatives and an automaton replication technique that makes it possible to represent non-deterministic timed automata with silent transitions using a deterministic RV tool. The replication technique does not impose any limits on the automata and its traces and, in contrast to other techniques, it only introduces some dummy states and does not cause an exponential growth in the number of states. Lastly, some faults in Larva that were encountered throughout the lifetime of this FYP were fixed and can be viewed on GitHub[10].

As for future work, one could explore more Larva features, such as the ability to include multiple properties per script, which is expected to reduce the overall overhead since they can share some components of the monitoring code. Future work could also attempt to prevent (rather than eliminate) silent transitions and non-determinism, which is expected to yield automata that are more optimized. Lastly, an issue is that using the presented TREs, arbitrary silent periods with no events are not expressible without prefixing them to events. In search for a solution, future work can explore the renaming solution of Asarin et al. [2], and the solution of Asarin and Dima [3, 12], who avoided the use of renaming.

[10]https://github.com/migueldingli1997/larva-rv-tool

9. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[2] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[3] E. Asarin and C. Dima. Balanced timed regular expressions. *Electr. Notes Theor. Comput. Sci.*, 68(5):16–33, 2002.

[4] C. Baier, N. Bertrand, P. Bouyer, and T. Brihaye. When are timed automata determinizable? In *ICALP 2009, Proceedings, Part II*, pages 43–54.

[5] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI 2004, Proceedings*, pages 44–57.

[6] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[7] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[8] C. Colombo. Combining testing and runtime verification. In *CSAW 2012*, page 19.

[9] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS 2008*, pages 135–149.

[10] C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *SEFM 2009*, pages 33–37.

[11] C. Colombo, G. J. Pace, and G. Schneider. Safe runtime verification of real-time properties. In *FORMATS 2009, Proceedings*, pages 103–117, 2009.

[12] C. Dima. Regular expressions with timed dominoes. In *DMTCS 2003*, pages 141–154.

[13] A. Gauci, G. J. Pace, and C. Colombo. Statistics and runtime verification. *University Of Malta*, 2009.

[14] R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In *TACAS 2013. Proceedings*, pages 260–276.

[15] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[16] F. Lorber, A. Rosenmann, D. Nickovic, and B. Aichernig. Bounded determinization of timed automata with silent transitions. *Real-Time Systems*, 53(3):291–326, 2017.

[17] K. Sen and G. Rosu. Generating optimal monitors for extended regular expressions. *Electr. Notes Theor. Comput. Sci.*, 89(2):226–245, 2003.

[18] D. Ulus, T. Ferrère, E. Asarin, and O. Maler. Online timed pattern matching using derivatives. In *TACAS 2016*, pages 736–751.