The Early Bird Catches the Worm: first Verify, then Monitor!

Angelo Ferrando

University of Genova, Via Dodecaneso 35, Genova, Italy
Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS)

Abstract

Trace expressions are a compact and expressive formalism, initially devised for runtime verification of agent interactions in multiagent systems, which has been successfully employed to model real-world protocols, and to generate monitors for mainstream multiagent system platforms, and generalized to support runtime verification of different kinds of properties and systems. In this paper, we propose an algorithm to check Linear Temporal Logic (LTL) properties satisfiability on trace expressions. To do this, we show how to translate a trace expression into a Büchi Automaton in order to realize an *Automata-Based Model Checking*. We show that this translation generates an *over-approximation* of our trace expression leading us to obtain a sound procedure to verify LTL properties. Once we have statically checked a set of LTL properties, we can conclude that: (1) our trace expression is formally correct (2) since we use this trace expression to generate monitors checking the runtime behavior of the system, the LTL properties verified by this trace expression are also verified by the monitored system.

Key words: Runtime Verification of Object-Oriented Programming; Trace expressions; Automata-Based Model Checking; Runtime Monitoring; Combining Static and Runtime Verification

1. Introduction

Runtime Verification (RV) is a software verification technique that complements formal static verification (like Model Checking [1, 2, 3, 4]) and testing [5, 6, 7].

When the system we want to verify becomes larger, model checking it (as well as the environment where it is immersed) becomes quickly intractable. In these scenarios, a valid alternative is RV. The main difference with respect to

Email addresses: angelo.ferrando@dibris.unige.it (Angelo Ferrando)

standard static verification is the stage when it is applied, which is at execution time. In fact, in RV we do not need to simulate all possible paths that the system may generate during its execution, but we limit the analysis directly to the paths exposed and generated by the system during its real execution. As a consequence, RV could be more suitable and applicable than static verification in black-box scenarios, where there is no access to the source code of the system we want to verify.

One possible way to achieve the RV of a system is using one (or more) monitor(s) generated from a formal specification (that is the property we want to verify, for instance an LTL property [8]).

Monitoring can be classified with respect to three main aspects:

1. When the monitoring is executed [9].
   - *online*, the monitor checks the executions incrementally at runtime;
   - *offline*, the monitor checks recorded executions (for instance log files).
2. How the monitoring is implemented [10].
   - *inline*, the monitor is inserted within the monitored program;
   - *outline*, the monitor runs in a thread or process different from the monitored program.
3. Which kind of errors the monitor arises [11].
   - *precise*, the monitor has observed an error in the execution trace analyzed;
   - *predictive*, the monitor indicates errors that have not occurred in the observed execution trace but could possibly occur in other executions of the program.

A possible way to specify the monitor behavior is through the set of all correct traces (finite or infinite sequences of events) which can be generated during the system execution[1]. This set of traces can be defined using different formalisms. In this paper we adopt trace expressions [12, 13], a formalism inspired by session types [14, 15]. As it will be clearer in the rest of the work, with respect to the three main aspects reported above, our RV approach can be classified as: *online*/*offline*[2], *outline*, *precise*.

Since we generate a monitor starting from a formal specification, we represent statically what we will check dynamically. If the static representation of the allowed event traces were error-free, we would be sure that monitoring a system according to that representation, would allow us to intercept all and only the possible violations due to unexpected or unwanted sequences of events. Unfortunately, our static representation might contain design and formalization

[1] For instance, the system might be a multiagent system and the events of interest might be messages exchanged among agents which must respect some interaction protocol property, or an object oriented systems, where events subject to monitoring might be method calls.

[2] We can analyze both the execution traces at runtime and the recorded traces (log files).

errors too, making unproductive to monitor the system behavior. If we were able to verify properties of the static representation before using it for the dynamic monitoring, the runtime verification would lead to more controlled and meaningful results. In the case of trace expressions, a possible way for obtaining this static check before the use at runtime is verifying whether all the traces satisfy an LTL property [8]. For instance, a common useful property to be checked could be $a \Rightarrow \Diamond b$ which says that if a takes place *sooner or later*, b will take place as well. Considering that the monitors used to verify the system are generated starting from a trace expression[3], we can conclude that all properties satisfied by trace expressions are either satisfied by the monitored system, or their violation is recognized by the monitor. This represents an important aspect because we can fuse static and dynamic verification approaches obtaining the best of both: the formal verification at the *static* level and the runtime monitoring at the *dynamic* level. For instance, the combination of static and runtime verification can simplify – reduce the size of – our monitors. If we were able to check statically a part of our specification, we could verify dynamically at runtime only what we are not able to check at static time [16]. Or also, if we want to reduce the state space analyzed by a static verifier, we can make assumptions and relax the model that is being validated. But, in this way, we can not be sure the real system is compliant with our assumptions, and if it is not, it would make the static verification so obtained useless[4]. Combining the use of a monitor to the static verifier, we can simplify the model verified statically, and we can add a monitor at runtime in order to recognize assumption violations with respect to our model [17]. These are just possible advantages in combining static and runtime verification. In this work, we focus on another possible combination of static and runtime verification, showing how we can **first verify** statically our monitors, and **then**, how to use them to **monitor** our systems. Our research question can be summarized in:

How can we trust our specification before using it to monitor our system?

One possible way of achieving the static verification of our specification (thus, our monitor) is translating it into a model more suitable for static verification purposes. In this work, we will show how to translate our specifications into Büchi Automatons, that are the standard automata version of LTL formulas used in model checking. We chose Büchi Automatons because they are a standard representation supported by most of the existing model checker [18]. In particular, we will use the SPIN model checker [19, 20, 21] to verify LTL formulas directly on our specifications.

Naturally, if we used LTL for generating our monitor, we would not need to verify anything statically, because the monitor would already denote the properties we want to check. But, LTL could be too limiting when used for RV

[3]To be more precise, in our implementation the monitors interpret the trace expression by implementing the transition rules which define the trace expression semantics.

[4]Our assumptions might be too strong and we need to relax them.

purposes, and we might need more expressive formalisms that allow defining more complex monitors. As we will see in Section 3.2, the formalism we chose to use for defining our monitors is more expressive than LTL[5]. Thus, if we want to be sure that our "complex" monitor still satisfies a set of LTL properties, we need a way to verify them on it. Finally, we can conclude that, even though we have a complex specification that is used to verify at runtime complex properties on our system, we are still able to guarantee that this specification satisfies a given set of LTL properties (without being limited to only those).

The contents of the paper are organized in the following way: Section 2 presents the context of the work and the motivations, Section 3 introduces all the preliminary concepts such as LTL, Büchi Automaton and the corresponding LTL Model Checking, Section 3.1 introduces trace expressions, Section 4 describes the algorithm used to verify LTL properties on trace expressions through an *Automata-Based Model Checking* approach, Section 5 shows two experiments developed using the SWI-Prolog [22] implementation and the SPIN Model Checker [19, 20, 21], while Section 6 provides a brief survey of related works. Conclusions are drawn in Section 7.

2. Motivations

As anticipated previously, RV can be seen as a middle approach between model checking and testing.

More in detail, the main differences between RV and model checking can be summarized as follows [9]:

- *time*, model checking is applied statically on the system and not during its execution;
- *exhaustiveness*, model checking generates all possible executions of the system while RV checks only the executions generated by the system at runtime (language inclusion problem vs word problem);
- *invasiveness*, model checking (generally) is applied to white-box scenarios, when the source code of the system is available, while RV can be used both in white-box and black-box scenarios, when the source code could be unavailable;
- *traces length*, model checking can consider arbitrary positions of a infinite trace while RV considers finite executions of increasing size.

In Section 1, we presented RV as a counterpart of testing. In particular, RV is extremely close to a specific form of testing, which is sometimes termed as oracle-based testing. In [9] the authors compare RV with oracle-based testing. In the following, we propose our revised version of the main differences between RV and oracle-based testing [9]:

[5]Or better, it is more expressive than LTL when the latter is used for RV purposes.

- *time*, testing is applied during the development of the system while RV is applied before and after the deployment[6];
- *granularity*, testing can test only the output of the system while RV can go into the details of the system behavior checking not only the observable outputs but also how these outputs have been generated (for instance, RV can tackle the nondeterminism, so even if the observable events are correct, they could have been generated in a nondeterministic way).
- *formalization*, in testing, an oracle is typically defined directly, rather than generated from some high-level specification as happen in RV.

After having compared RV with model checking and test, we are ready to answer at the question: *When RV should be used?* [9]:

- when the complexity of the system makes it intractable for an exhaustive analysis;
- when some information is available only at runtime;
- when a precise description of the environment does not exist;
- when there are security issues in the case of safety-critical systems, where it is useful to monitor properties that have been statically proved or tested, mainly to have a double check.

Now that the main differences among RV, model checking and testing have been presented, we can focus on the motivations of the work.

Supposing we have a system to check, in order to do the RV of it we have to start writing the specification we want to verify. In the next section we will present a possible formalism that can be used to obtain this, but for now, we focus only on the problem. Once we have defined our specification, we can use it to generate a monitor to check the system behavior. If the system does something inconsistent with our specification, the monitor will notice it and will act accordingly (implementation dependant). But, what happens if our specification contains design errors? We believe that we have defined a correct specification that satisfies our intentions, but unfortunately we have introduced errors in it. Even though we can not really check if the specification is actually representing our intentions – too abstract concept inside our minds – we can still check if our specification satisfies all the properties we think should. If we were able to check our specification against these properties statically, we could resolve this trust problem.

Since RV does not need to exhaustively check the system behavior, it is less inclined to fall in state space explosion problems. This results from the fact

[6]RV applied before the deployment is used to check if the system respects our model. RV applied after the deployment is used to monitor the system behavior in order to prevent malfunctions (or at least signalling the user in time to reduce the damage).

that, in its mathematical essence, RV answers the "word problem", *i.e.* the problem whether a given word is included in some language [9]. Model checking, which tries to solve the "language inclusion" problem, is traditionally used to verify finite-state systems against regular specifications. But, requirements which involve counting cannot be formalized by regular specifications. To the best of our knowledge, the must common approach to solve this issue is using pushdown specifications [23], *i.e.*, context-free properties represented by pushdown automata. Anyway, research on model-checking techniques for pushdown specifications is rare [24]. In RV instead, it is not uncommon to be interested in verifying complex properties and many formalisms exist [25]. Unfortunately, the complexity can lead to introduce mistakes also in the development of our specifications compromising the entire RV process (design errors). One possible way to solve this problem is by checking directly the specification statically before using it. With such preprocessing we can achieve the two following advantages:

- *trust*, the RV process is more reliable since we are sure that our complex property satisfies a set of constraints (verifiable statically);
- *propagation*, as long as the system is consistent with the monitor generated by the specification, it is also consistent with the constraints checked statically (by construction).

In this paper we have chosen to use the trace expression formalism [13] to represent our specifications.

3. Background

LTL [8, 26, 27, 28] is a modal logic which has been introduced for specifying temporal properties of systems; despite its original main application in static verification through model checking, more recently it has been adopted as a specification formalism for RV, and some RV tools support it [29, 30].

LTL syntax and semantics. Given a finite set of atomic propositions AP, the set of LTL formulas over AP is inductively defined as follows:

- *true* is an LTL formula;
- if $p \in AP$ then p is an LTL formula;
- if φ and ψ are LTL formulas then $\neg\psi$, $\varphi \vee \psi$, $X\psi$, and $\varphi\, U\, \psi$ are LTL formulas.

Additional operators can be derived in the standard way: $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$, $F\varphi$ (or $\Diamond\varphi$) $= true\, U\, \varphi$, and $G\varphi$ (or $\Box\varphi$) $= \neg(true\, U \neg\varphi)$.

Let $\Sigma = 2^{AP}$ be the set of all possible subsets of AP; if $p \in AP$ and $a \in \Sigma$, then p holds in a iff $p \in a$. An LTL model is an infinite trace $w \in \Sigma^\omega$; $w(i)$ denotes the element $a \in \Sigma$ at position i in trace w; more formally, if $w = aw'$, then $w(0) = a$, and $w(i) = w'(i-1)$ if $i > 0$.

The semantics of a formula φ depends on the satisfaction relation $w, i \vDash \varphi$ (w satisfies φ in i) defined as follows:

- $w, i \vDash p$ iff $p \in w(i)$;
- $w, i \vDash \neg\phi$ iff $w, i \nvDash \phi$;
- $w, i \vDash \varphi \vee \psi$ iff $w, i \vDash \varphi$ or $w, i \vDash \psi$;
- $w, i \vDash X\varphi$ iff $w, i+1 \vDash \varphi$ (next operator);
- $w, i \vDash \varphi U \psi$ iff $\exists j \geq 0 \;\; w, j \vDash \psi$ and $\forall 0 \leq k < j \;\; w, k \vDash \varphi$ (until operator).

Finally, $w \vDash \varphi$ (w satisfies φ) holds iff $w, 0 \vDash \varphi$ holds.

We recall that the set of all models of LTL formulas is the language of star-free ω-regular languages over Σ [31].

In order to encode an LTL formula into an equivalent trace expression we exploit the result stating that an LTL formula can be translated into an equivalent non deterministic Büchi automaton [32].

LTL Model Checking [33, 34]. Given a model M and an LTL formula φ: (1) all traces of M must satisfy φ, (2) if a trace of M does not satisfy φ we have found a *Counterexample*.

We call Σ_M the set of traces of M and Σ_φ the set of traces that satisfy φ. We check if $\Sigma_M \cap \Sigma_{\neg\varphi} = \emptyset$.

Büchi Automaton. A Büchi Automaton [35] is an automaton which accepts infinite traces.

Definition 1. *A Büchi Automaton is a 4-tuple $\langle S, I, \delta, F \rangle$ where, S is a finite set of states, $I \subseteq S$ is a set of initial states, $\delta \subseteq S \times S$ is a transition relation and $F \subseteq S$ is a set of accepting states.*

An infinite sequence of states is accepted iff it contains (at least) one of the accepting states infinitely often.

Automata-Based Model Checking. Given a model M and an LTL formula φ: (1) build the Büchi Automaton $\mathrm{B}_{\neg\varphi}$; (2) compute product of M and $\mathrm{B}_{\neg\varphi}$; (3) the product accepts the traces of M that are also traces of $\mathrm{B}_{\neg\varphi}(\Sigma_M \cap \Sigma_{\neg\varphi})$; (4) if at least one sequence is accepted by the product, then we have found one counterexample.

3.1. Trace expressions

Trace expressions [13, 36] are a specification formalism expressly designed for RV and inspired by initial work on monitoring of agent interactions in multiagent systems [12, 37].

Trace expressions are based on the basic notions of *event* and *event type*.

Events. In the following, we denote by $\mathcal{E}$ a fixed universe of events. A trace of events over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$. In the rest of the paper the meta-variables e, w, σ and u will range over the sets $\mathcal{E}$, $\mathcal{E}^\omega$, $\mathcal{E}^*$, and $\mathcal{E}^\omega \cup \mathcal{E}^*$, respectively. A trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$. As a possible example, if we consider an object-oriented scenario where our events correspond to the invocation of methods, we might have

$$\mathcal{E} = \{o.m \mid o \text{ object identifier}, m \text{ method name}\}$$

Event types. Defining trace expressions on top of the concept of events could not be general enough. A way to make more flexible and general our trace expressions is constructing them on top of event types (chosen from a set $\mathcal{ET}$). An event type can be represented as a predicate of arity $k \geq 1$, where the first implicit argument corresponds to the event e under consideration. Considering again the object-oriented scenario where the events are method invocations, we may introduce the type $safe(o)$ of all safe method invocations for a given object o, defined by the predicate $safe$ of arity 2 s.t. $safe(e, o)$ holds iff $e = o.isEmpty$.

To make the definition of event types more compact we leave the first argument of the predicate implicit, and we write $e \in safe(o)$ to mean that $safe(e, o)$ holds.

In a similar way, we can specify the set of events denoted by the event type ϑ as $[\![\vartheta]\!]$; for instance, $[\![safe(o)]\!] = \{e \mid e \in safe(o)\}$.

To be more general, we do not specify the formalism used for defining event types; in practice, we do not expect that much expressive power is required. From an implementation point of view, a word recognizer for trace expressions exists and has been implemented in SWI-Prolog. The choice of SWI-Prolog has brought to implement the concept of event types as very simple SWI-Prolog predicates; in fact, they can be defined by plain facts (clauses without bodies).

Trace expressions. Now that we have introduced the concept of events and event types, we can present the formalism we are going to use in the rest of the work: the trace expression formalism.

A trace expression τ denotes a set of possibly infinite event traces, and is defined on top of the following operators:[7]

- ϵ (empty trace), denoting the singleton set $\{\epsilon\}$ containing the empty event trace ϵ.
- $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace of τ..
- $\tau_1{\cdot}\tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2.
- $\tau_1{\wedge}\tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2.

[7] Binary operators associate from left, and are listed in decreasing order of precedence, that is, the first operator has the highest precedence.

- $\tau_1 \vee \tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2.
- $\tau_1 | \tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces of τ_1 with the traces of τ_2.
- $\vartheta \gg \tau$ (*filter*), it is a derived operator denoting the set of all traces contained in τ, when deprived of all events that do not match ϑ.

Trace expressions are cyclic terms, thus they can support recursion without introducing an explicit construct. These cyclic terms corresponds in particular to trees where nodes are either the leaf ϵ, or the node (corresponding to the prefix operator) ϑ with one child, or the nodes $\cdot$, $\wedge$, $\vee$, and $|$ all having two children. According to the standard definition of rational trees [38, 39], their depth is allowed to be infinite, but the number of their subtrees must be finite.

As will be clear in the rest of the work, it is really important to understand how and when a term is a subterm of another one. In particular, we give the intuition of subterm through the the concept of subtree. The notion of subtree on infinite trees is intuitive but it is different from that on finite trees; in fact, it is no longer an order relation (it is not antisymmetric). This concept will be used in the following to indicate the presence of cycles inside our trace expressions.

Even though our trace expressions contain cycles, in the rest of the paper, we limit our investigation to *contractive* (a.k.a. *guarded*) trace expressions.

Definition 2. *A trace expression τ is* contractive *if all its infinite paths contain the prefix operator.*

A contractive trace expression is a trace expression where all recursive subexpressions are always guarded by the prefix operator; for instance, the trace expression defined by $T_1 = (\epsilon \vee (\vartheta{:}T_1))$ is contractive: its infinite path contains infinite occurrences of $\vee$, but also of the : operator; conversely, the trace expression $T_2 = (\vartheta{:}T_2)|T_2$ is not contractive.

Trivially, every trace expression corresponding to a finite tree (that is, a non cyclic term) is contractive. Trace expressions support recursion through cyclic terms expressed by finite sets of recursive syntactic equations.

For all contractive trace expressions, any path from their root must always reach either a ϵ or a : node in a finite number of steps. Since in this paper all definitions over trace expressions consider $\vartheta{:}\tau$ as a base case, if we restrict our trace expressions to be contractive, all our definitions and proofs can be done inductively (rather than coinductively). Also at the implementation level, trace expressions becomes considerably simpler. For this reason, in the rest of the paper we will only consider contractive trace expressions.

We can specify the semantics of trace expressions by the transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ and $\mathcal{E}$ denote the set of trace expressions and of events, respectively. We write $\tau_1 \xrightarrow{e} \tau_2$ to mean $(\tau_1, e, \tau_2) \in \delta$. If we have the trace expression τ_1 that specifies the current valid state of the system, we say that an event e is valid iff there exists a transition $\tau_1 \xrightarrow{e} \tau_2$. In such a case, τ_2 denotes the next valid state of the system. Otherwise, the event e is not considered a

$$\text{(prefix)}\frac{}{\vartheta{:}\tau \xrightarrow{e} \tau}\ e\in\vartheta \qquad \text{(or-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1\vee\tau_2 \xrightarrow{e} \tau_1'} \qquad \text{(or-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1\vee\tau_2 \xrightarrow{e} \tau_2'}$$

$$\text{(and)}\frac{\tau_1 \xrightarrow{e} \tau_1' \quad \tau_2 \xrightarrow{e} \tau_2'}{\tau_1\wedge\tau_2 \xrightarrow{e} \tau_1'\wedge\tau_2'} \qquad \text{(shuffle-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1|\tau_2 \xrightarrow{e} \tau_1'|\tau_2} \qquad \text{(shuffle-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1|\tau_2 \xrightarrow{e} \tau_1|\tau_2'}$$

$$\text{(cat-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1\cdot\tau_2 \xrightarrow{e} \tau_1'\cdot\tau_2} \qquad \text{(cat-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1\cdot\tau_2 \xrightarrow{e} \tau_2'}\ \epsilon(\tau_1)$$

$$\text{(cond-t)}\frac{\tau \xrightarrow{e} \tau'}{\vartheta\gg\tau \xrightarrow{e} \vartheta\gg\tau'}\ e\in\vartheta \qquad \text{(cond-f)}\frac{}{\vartheta\gg\tau \xrightarrow{e} \vartheta\gg\tau}\ e\notin\vartheta$$

Figure 1: Operational semantics of trace expressions

$$\text{(ϵ-empty)}\frac{}{\epsilon(\epsilon)} \qquad \text{(ϵ-or-l)}\frac{\epsilon(\tau_1)}{\epsilon(\tau_1\vee\tau_2)} \qquad \text{(ϵ-or-r)}\frac{\epsilon(\tau_2)}{\epsilon(\tau_1\vee\tau_2)} \qquad \text{(ϵ-shuffle)}\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1|\tau_2)}$$

$$\text{(ϵ-cat)}\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1\cdot\tau_2)} \qquad \text{(ϵ-and)}\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1\wedge\tau_2)} \qquad \text{(ϵ-cond)}\frac{\epsilon(\tau)}{\epsilon(\vartheta\gg\tau)}$$

Figure 2: Empty trace containment

valid event in the current state denoted by τ_1. Figure 1 defines the inductive rules for the transition function.

In Figure 1 we reported the rules for the transition relation δ that define the non empty traces of a trace expression. While in Figure 2, we inductively define the $\epsilon(\_)$ predicate by its rules; through this predicate we specify trace expressions that can contain the empty trace ϵ. If $\epsilon(\tau)$ holds, then the empty trace is a valid trace for τ (in this way we can also recognize finite sequences of events, definitely important in RV).

The set of traces $[\![\tau]\!]$ denoted by a trace expression τ is totally defined on top of the transition relation δ, and the predicate $\epsilon(\_)$. Since $[\![\tau]\!]$ may contain infinite traces, the definition of $[\![\tau]\!]$ is coinductive.

Definition 3. *For all possibly infinite event traces u and trace expressions τ, $u \in [\![\tau]\!]$ is coinductively defined as follows:*

- *either $u = \epsilon$ and $\epsilon(\tau)$ holds,*
- *or $u = e\,u'$, and there exists τ' s.t. $\tau \xrightarrow{e} \tau'$ and $u' \in [\![\tau']\!]$ hold.*

Deterministic trace expressions. There are trace expressions τ for which the problem of word recognition is less efficient because of non determinism. We can have non determinism inside our trace expressions caused by the use of the union, shuffle, and concatenation operators. These are the only operators that can introduce non determinism because for each of them two possibly overlapping transition rules are defined.

In the rest of the paper we focus only on deterministic trace expressions: in fact, the problem of word recognition is simpler and more efficient in the deterministic case.

Deterministic trace expressions are defined as follows.

Definition 4. *Let τ be a trace expression; τ is* deterministic *if for all finite event traces σ, if $\tau \xrightarrow{\sigma} \tau'$ and $\tau \xrightarrow{\sigma} \tau''$ are valid, then $[\![\tau']\!] = [\![\tau'']\!]$.*

Stack example. Always in the object-oriented scenario, besides the already introduced event type $safe(o)$ s.t. $e \in safe(o)$ iff $e = o.isEmpty$, we define the following other event types:

$$[\![pop(o)]\!] = \{o.pop\},\ [\![top(o)]\!] = \{o.top\},\ [\![push(o)]\!] = \{o.push\},$$
$$[\![stack(o)]\!] = \{o.pop, o.top, o.push, o.isEmpty\},$$
$$[\![unsafe(o)]\!] = \{o.pop, o.top, o.push\}.$$

Our purpose now is to specify through the *Stack* trace expression all and only the safe traces of method invocations on a stack object o (initially empty). Safety requires that methods *top* and *pop* can never be invoked on o when o represents the empty stack.

More in details, a trace of method invocations on a given object having identity o is correct iff any finite prefix does not contain more $pop(o)$ event types than $push(o)$, and the event type $top(o)$ can appear only if the number of $pop(o)$ event types is strictly less than the number of $push(o)$ event types occurring before $top(o)$.

The *Stack* trace expression is deterministic and is defined as follows:

$$\begin{aligned} Stack &= Any \wedge unsafe(o) \gg Unsafe \\ Any &= \epsilon \vee stack(o) : Any \\ Unsafe &= \epsilon \vee (push(o) : (Unsafe | (Tops \cdot (pop(o) : \epsilon \vee \epsilon)))) \\ Tops &= \epsilon \vee top(o) : Tops \end{aligned}$$

A correct stack trace can be specified by *Stack* where we have the intersection of *Any* and $unsafe(o) \gg Unsafe$; *Any* specifies any possible trace of method invocations on stack objects, whereas if an event has type $unsafe(o)$, then it has to verify the trace expression *Unsafe*, which requires that a *push* event must precede a possibly empty trace of *top* events, which, in turn, must precede an optional event *pop*; the expression is shuffled with itself, since any *push* event can be safely shuffled with a *top* or a *pop* event.

3.2. Trace expressions vs LTL$_3$

Before going into the details of the original contents of the paper, we need to briefly summarize the relation between LTL$_3$ and the trace expression formalism.

LTL$_3$ is the three-valued semantics of LTL that has been proposed in [40] to be used for RV purposes. With LTL$_3$ we can focus on finite traces because a third truth value "?" is introduced to specify that after a finite trace of events has occurred, the outcome of a monitor can be inconclusive. That is a big

difference with respect to LTL, where all the traces of events are supposed to be always infinite.

Trace expressions semantics implicitly models the RV limitation to finite traces of events. Thus, when we are interested in comparing the trace expression formalism with LTL, we have to consider the three-valued semantics (so they are both applied to a RV scenario).

In [13] the authors demonstrated that when the three-valued semantics is considered, then trace expressions are strictly more expressive than LTL. In the following we report the summary of the demonstration.

The intuition is based on the fact that all LTL formulas φ, can be represented as a Finite State Machine (FSM), which is a Deterministic Finite Automaton (DFA). Meaning that, the behavior of the resulting FSM respects the LTL_3 semantics of φ.

The sequence of steps required to generate from an LTL formula φ an FSM that respects the LTL_3 semantics of φ [40] is summarized in Figure 3.

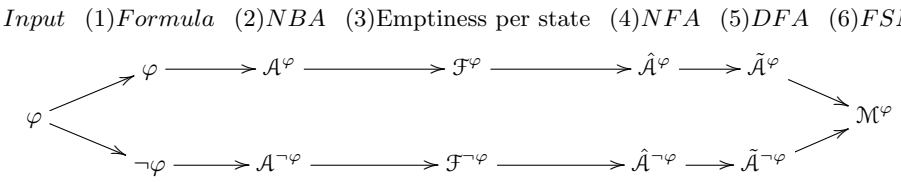

Figure 3: Steps required to generate a FSM from an LTL formula φ, where NBA is the acronym for Nondeterministic Büchi Automaton and NFA for Nondeterministic Finite Automaton.

To translate an LTL formula φ into a trace expression τ s.t. the three-valued semantics is preserved, we exploit the result presented in [40]. First, φ is translated into an equivalent FSM $\mathcal{M}^{\varphi}$, then $\mathcal{M}^{\varphi}$ is translated into an equivalent contractive and deterministic trace expression τ^{φ}.

The latter translation is defined as follows:

- if the initial state returns $\top$, then φ is a tautology, and the corresponding trace expression is the constant 1;
- if the initial state returns $\bot$, then φ is a unsatisfiable, and the corresponding trace expression is the constant 0;
- if the initial state returns ?, then the corresponding trace expression is defined by a finite set of equations $X_1 = \tau_1, \ldots, X_n = \tau_n$, where n is the number of states in $\mathcal{M}^{\varphi}$ that return ?. The expressions τ_i are defined as follows: let k be the number of states $q_1, \ldots, q_k$ that do not return $\bot$ for which there exists an incoming edge, labeled with the element $a_i \in 2^{AP}$, from the node associated with X_i; we know that $k > 0$, because the node associated with X_i returns ?. Then $\tau_i = a_1{:}f(q_1) \vee \ldots \vee a_k{:}f(q_k)$, where $f(q)$ is defined as follows: if q returns $\top$, then $f(q) = 1$, otherwise (that

is, q returns ?), $f(q) = X_q$ (that is, the variable uniquely associated with q is returned).

In [13] it is also presented the theorem supporting the correctness of this translation.

4. Trace expressions Model Checking

In this section, we present the model checking process used to check if an LTL property is satisfied by a trace expression.

Given a trace expression τ, we want to verify if an LTL property φ is satisfied by all the traces recognized by τ. To achieve this, we can follow this 3-steps algorithm:

1. Rewrite τ obtaining its abstraction τ' (*over-approximation*). Following the definition of $[\![\tau]\!]$ (Definition 3), given as the set of traces denoted by the trace expression τ, we have that $[\![\tau']\!] \supseteq [\![\tau]\!]$.
2. Translate τ' into an equivalent Büchi Automaton $\mathrm{B}_{\tau'}$.
3. Compute the product of $\mathrm{B}_{\tau'}$ and $\mathrm{B}_{\neg\varphi}$. If the product accepts some traces, τ does not satisfy φ and a trace representing the counterexample is raised.

4.1. 1st step: Rewriting

The first step is the most important one, in fact, it consists in translating a given trace expression to a trace expression representing its *over-approximation*. This phase is necessary because, in general, it is not always possible to translate a trace expression into an equivalent Büchi Automaton; this is due to expressiveness differences. We assume the reader to be familiar with the theory of formal languages and of ω-regular languages, see for example [41, 42, 43]. A Büchi Automaton recognizes ω-*regular* languages while trace expressions can represent more expressive languages ([13] for more details) that can be also ω-*context-free* and ω-*context-sensitive*. Consequently, we have to manage all trace expressions which are too expressive (through abstraction).

Non-expansive trace expressions. As analyzed by Ancona *et al.* in [13], trace expressions are a formalism more expressive than LTL when used for RV purposes. In particular, trace expressions are able to recognize also *context-free* and *context-sensitive* languages (and the corresponding ω version). This expressivity is due to the presence of *expansive* terms which allow trace expressions, for instance, to count events (necessary for recognizing *context-free* languages, such as $a^n b^n$).

Definition 5. *A concatenation $\tau = \tau_1 \cdot \tau_2$ is* expansive *if τ is a subtree of τ_1.*

Example 1. *An example of an* expansive *concatenation term is:*
$[\![\vartheta_1]\!] = \{e_1\} \quad [\![\vartheta_2]\!] = \{e_2\}$
$\tau = \tau_1 \cdot \tau_2 \quad \tau_1 = (\vartheta_1 : \tau) \vee \epsilon \quad \tau_2 = \vartheta_2 : \epsilon$
$\tau \xrightarrow{e_1} (\tau_1 \cdot \tau_2) \cdot \tau_2 \xrightarrow{e_1} ((\tau_1 \cdot \tau_2) \cdot \tau_2) \cdot \tau_2 \xrightarrow{e_1} (((\tau_1 \cdot \tau_2) \cdot \tau_2) \cdot \tau_2) \cdot \tau_2 \xrightarrow{e_1} \ldots$

Definition 6. *A shuffle $\tau = \tau_1|\tau_2$ is* expansive *if τ is a subtree of τ_1 or τ_2.*

Example 2. *An example of an* expansive *shuffle term is:*
$[\![\vartheta_1]\!] = \{e_1\} \quad [\![\vartheta_2]\!] = \{e_2\}$
$\tau = \tau_1|\tau_2 \quad \tau_1 = \vartheta_1{:}\epsilon \quad \tau_2 = \vartheta_2{:}\tau$
$\tau \xrightarrow{e_2} \tau_1|(\tau_1|\tau_2) \xrightarrow{e_2} \tau_1|(\tau_1|(\tau_1|\tau_2)) \xrightarrow{e_2} \tau_1|(\tau_1|(\tau_1|(\tau_1|\tau_2))) \xrightarrow{e_2} ...$

Non-expansive trace expressions are defined as follows.

Definition 7. *Let τ be a trace expression; τ is* non-expansive *if it does not contain neither expansive concatenations nor expansive shuffles terms.*

Below we report the pseudocode of a *recursive* implementation of the **rewrite** function (Algorithm 1). This algorithm takes three arguments, that are: the trace expression we want to *over-approximate*, a map of dangerous trace expressions (*dangerous*), a set of already seen trace expressions (*safe* $\cup$ *dangerous*), and it returns the *over-approximation* of the trace expression.

The most interesting part of the algorithm concerns the *expansive* terms recognition. As we have already seen before, an expansive term is a term containing or a concatenation ($\tau_1 \cdot \tau_2$) having a cycle in the head, or a shuffle ($\tau_1 \mid \tau_2$) containing a cycle in its left or right branch (expansive concatenations/shuffles). In order to recognize them, we have to remember the trace expressions we have already visited during the exploration of the trace expression's subtrees differentiating between *safe* and *dangerous* cycles. To achieve this, we maintain in memory the history of all visited states {*safe* $\cup$ *dangerous*} and the history of the *dangerous* states. In this way, we can detect if a cycle is expansive, or not, simply searching inside the *dangerous* map, which must be updated each time we find a new concatenation (or shuffle) operator.

More precisely, the *dangerous* map can be represented as a set of tuples (TExp, Danger) that the algorithm uses to keep track of possibly expansive cycles; where, Danger $\in$ {**maybe**, **true**}, meaning that, if (t, **true**) $\in$ *dangerous*, t is an expansive term and we have to *rewrite* it, while, if (t, **maybe**) $\in$ *dangerous*, we have already seen t in the analysis of TExp and it could be an expansive term because it contains a concatenation or a shuffle subtree. Concluding, if both (t, **true**) and (t, **maybe**) $\notin$ *dangerous*, it means that t is not considered, for now, a dangerous term, because we do not have encounter any concatenation or shuffle subtree inside it.

Before going into the details of the algorithm implementation, it could be useful showing what we obtain when we apply it to a simplified version of the stack example.

Example 3. *Considering a stack object, we define the set of correct traces, having only the methods push and pop.*

$[\![\vartheta_{push}]\!] = \{o.push\} \quad [\![\vartheta_{pop}]\!] = \{o.pop\}$

$\tau = \tau_{push} \cdot \tau_{pop}$

$\tau_{push} = \vartheta_{push} : (\tau \vee \epsilon)$
$\tau_{pop} = \vartheta_{pop} : \epsilon$

$rewrite(\tau, \{\}, \{\}) \rightarrow \tau'$
$\tau' = \tau'_{push} \cdot \tau'_{pop}$
$\tau'_{push} = (\vartheta_{push} : \epsilon) \cdot (\tau'_{push} \vee \epsilon)$
$\tau'_{pop} = (\vartheta_{pop} : \epsilon) \cdot (\tau'_{pop} \vee \epsilon)$

Before going on with the presentation of the rewrite algorithm's pseudocode, we can spend some words on this example. Starting from our trace expression τ, we want to obtain a trace expression τ', such that $[\![\tau]\!] \subseteq [\![\tau']\!]$ (τ' is an *over-approximation* of τ). We can easily note that simply deriving the two languages defined by τ and τ'

$$[\![\tau]\!] = \{o.push^n o.pop^n | n \in \mathcal{N}^+\} \cup \{o.push^\omega\}$$

$$[\![\tau']\!] = \{o.push^n o.pop^m | n \in \mathcal{N}^+, m \in \mathcal{N}\} \cup \{o.push^\omega\} \cup \{o.push^n o.pop^\omega | n \in \mathcal{N}^+\}$$

and, since $\{o.push^n o.pop^n | n \in \mathcal{N}^+\} \subseteq \{o.push^n o.pop^m | n \in \mathcal{N}^+, m \in \mathcal{N}\}$, we conclude that $[\![\tau]\!] \subseteq [\![\tau']\!]$. Intuitively, we have just passed from a context-free language to a regular language. With τ we defined that a trace of events containing *o.push* and *o.pop* was a correct trace iff the number of *o.push* was equal to the number of *o.pop* (with also the possibility of having an infinite number of *o.push*[8]). With τ' we want to reduce instead the expressivity relaxing our constraints. In this specific case, since we are counting and constraining the number of events (making the recognized language at least context-free), when we relax the constraints reducing the expressivity we can simply stop counting the events. Practically speaking, we can just stop forcing *o.push* and *o.pop* to have the same cardinality. Removing this constraint it is easy to note that the language we are recognizing now is not context-free anymore, but it is regular. We will show in the the rest of the paper that these kinds of *over-approximations* allow us to have a more suitable representation of our specifications, in particular, when we are interested in verifying them statically.

In Algorithm 1 we reported the pseudocode of the *rewrite* function. In the following we describe step by step all the cases handled by the algorithm.

The first case (line 1) tackled is when the trace expression *texp* is the empty trace ϵ. This is the simplest case because we do not have to rewrite anything and we just return the empty trace ϵ.

The second case (line 3) handles the presence of dangerous cycles. If *texp* belongs already to the *dangerous* set[9], it means that *texp* is a cyclic term (since it is in *dangerous* we have already seen it), and it is a subtree of the head of a concatenation term or of a shuffle term. Consequently, *texp* is an expansive

[8] Until we see the first *o.pop* a monitor can not say if the trace is good o not and consequently we accept also the infinite trace of *o.push*.

[9] We do not care if it is **maybe** or **true**, because now force it to be **true**.

Algorithm 1 Rewrite function's pseudocode

```
function TExp rewrite(texp, dangerous, already_seen){
 1: if texp = ϵ then
 2:     return ϵ
 3: else if (texp, _) ∈ dangerous then
 4:     remove (texp, maybe) from dangerous
 5:     /*texp already visited, we are in a dangerous term*/
        add (texp, true) to dangerous
 6:     return ϵ
 7: else if texp ∈ already_seen then
 8:     return rewritten(texp, already_seen) /*texp visited, we are not in dangerous term*/
 9: else if texp matches head · tail then
10:     /*update the set of texps already visited*/
        already_seen1 = (already_seen ∪ {texp})
11:     /*example: {(s1, true), (s2, maybe)} ∪ {s1, s2, s3} =
        {(s1, true), (s2, maybe), (s3, maybe)}*/
        dangerous_head = dangerous ∪ already_seen1
12:     /*rewrite the head subterm*/
        new_head = rewrite(head, dangerous_head, already_seen1)
13:     if {t | t ∈ already_seen1 and (t, true) ∈ dangerous_head} ≠ ∅ then
14:         /*the concatenation is expansive, thus rewrite the tail subterm*/
            dangerous_tail = dangerous ∪ already_seen1
15:         new_tail = rewrite(tail, dangerous_tail, already_seen1)
16:         /*create new head and new tail*/
            T1 = new_head · (T1 ∨ ϵ)
17:         T2 = new_tail · (T2 ∨ ϵ)
18:         return T1 · T2
19:     else
20:         remove already_seen1 from dangerous
21:         new_tail = rewrite(tail, dangerous, already_seen1)
22:         return new_head · new_tail /*there are no cycles*/
23: else if texp matches left | right then
24:     already_seen1 = (already_seen ∪ {texp})
25:     dangerous_left = (dangerous ∪ already_seen1)
26:     dangerous_right = (dangerous ∪ already_seen1)
27:     /*dangerous variables refer to different objects*/
        new_left = rewrite(left, dangerous_left, already_seen1)
28:     new_right = rewrite(right, dangerous_right, already_seen1)
29:     dangerous = dangerous_left ∪ dangerous_right
30:     t = {t' | t' ∈ already_seen1 and (t', true) ∈ dangerous}
31:     if t ≠ ∅ then
32:         T1 = new_left · T1
33:         T2 = new_right · T2
34:         return T1 | T2
35:     else
36:         return new_left | new_right
37: else if texp matches left ∨ right then
38:     /*(the same for ∧)*/
        already_seen1 = copy(already_seen ∪ {texp})
39:     new_left = rewrite(left, dangerous, already_seen1)
40:     new_right = rewrite(right, dangerous, already_seen1)
41:     return new_left ∨ new_right /*(the same for ∧)*/
42: else if texp matches prefix : body then
43:     /*(the same for ≫)*/
        already_seen1 = copy(already_seen ∪ {texp})
44:     new_body = rewrite(body, dangerous, already_seen1)
45:     return prefix : new_body /*(the same for ≫)*/
}
```

term and we have to update the *dangerous* set (now we know it is **true**). After that, we return ϵ (why ϵ will be clear in the fourth and fifth cases).

The third case (line 7) considers the safe cycles. We are not inside the head of a concatenation nor a shuffle, otherwise *texp* would have been inside the dangerous set (second case). Consequently, if *texp* belongs to the set of the *already_seen* trace expressions, it means that we have found a cycle that is not dangerous. We can not return directly *texp*, because *texp* is the old version that can be expansive. Instead, we want to return the new rewritten version of *texp*. In all the cases of the algorithm, each time we update the set of *already_seen* trace expressions, we also implicitly add the rewritten version of the trace expression[10]. In this way, when we encounter a safe cycle we can just return the term that will contain the non-expansive rewritten version of *texp*. We can achieve this using a function called *rewritten* that, given a trace expression and a set of *already_seen* trace expressions, returns the rewritten version.

The fourth case (line 9) is one of the complex cases. Here we handle the concatenation terms. The first thing we have to do is to update the set of *already_seen* trace expressions adding *texp* (the current one). After that, differently from the other simple cases, we have to update also the *dangerous* set, merging it with the set of *already_seen* states. Since now we are approaching to analyze the head of the concatenation, all the terms we have already encounter can cause an expansion of our term, because *texp* is a subtree of each one of them. First of all, we need to analyze the head of the concatenation, if we find a cycle, we handle it as a bad one (second case) and not as a good one (first case). This is totally derived from the definition of expansive concatenation (Definition 5). If we find a dangerous cycle inside the head (the if statement at line 13), we have to rewrite first the tail, and after that, we can construct the new trace expression corresponding to the *over-approximation* of our expansive concatenation. We can obtain that concatenating the new head with the new tail (line 18). Naturally, we have also the good scenario where our concatenation is not expansive, and this can be derived from the absence of dangerous cycles inside the head of the concatenation. In this scenario, we do not change the structure of the concatenation and we limit to concatenate the rewritten head with the rewritten tail. Even though the concatenation is not expansive, there might be expansive terms inside the head or the tail that have been rewritten from the rewrite function (expansive terms not influencing our concatenation because we have checked it at line 13). Before going on with the fifth case, it is time to motivate why at the second step we returned ϵ. In order to understand this, it is enough to see what we do with the rewritten head returned by the function and saved into the variable *new_head*. We create a new term (line 16) concatenating *new_head* with itself. If we think about the original head of the

[10]Even though it is not ready, we can add a reference to the term that will be unified with it (in SWI-Prolog, on which the algorithm is actually implemented, we can easily obtain this using free variables).

concatenation, where now we have an ϵ, before we had a cycle that made the concatenation expansive. This cycle allowed the concatenation to accumulate a new tail and restart consuming a new head (and so on). If we remove this cycle, we have to simulate the same behavior without the accumulation of the tail. To do so, we can substitute the cycle with a ϵ and combine the new head so generated with itself. In this way we obtain a cycle on the content of the head, as we were doing before but without the accumulation of the tail. To simulate this accumulation we can just do the same thing for the tail, we concatenate the *new_tail* with itself and we conclude concatenating the two terms so generated (line 18). In this way, before we could consume n time the head concatenating with the expansion of the term n times the tail, now, we simply consume a certain number of times n the head without accumulating and, after that, we consume a certain number of times m the tail (where n and m might be different).

The fifth case (line 23) is the other complex case and it is very similar to the fourth one. Here we handle the shuffle terms. Also in this case, we first update the set of *already_seen* terms and then we update the *dangerous* set considering all the super terms of the shuffle (the terms that have *texp* as subtree). After that, we check if we have found a bad cycle inside the left or the right operand, and if so, we rewrite the shuffle using the two new rewritten versions obtained from the two function calls. As it was for the concatenation term, also here we might have found a good shuffle term; consequently, we just construct the new trace expression combining the two rewritten operand terms (left and right) with the shuffle operator. This is totally derived from the definition of expansive shuffle (Definition 6).

In both the fourth and fifth cases, we expect that, if the subtrees of *texp* are not dangerous and do not contain any other expansive terms, *texp* is rewritten into itself.

The sixth case (line 37) handles the union terms (equivalently also the intersection terms). If *texp* is an union term, we simply apply the rewrite function to its operands and we combine the results so obtained using the union operator. Since the union operator does not introduce expansivity *per se*, we do not have to update the *dangerous* set (only concatenations and shuffles introduce expansivity, Definition 5 and 6).

The seventh case (line 42) handles the prefix terms (equivalently also the filter terms). If *texp* is a filter term, we simply apply the rewrite function to the remaining of the trace expression. The result will be combined again with the prefix operator, and then returned.

Removing the expansive subtrees (subterms) from our trace expression, we obtain a spurious solution, namely a solution representing a bigger set of traces; this is entirely due to the fact that only using expansive terms, as cyclic concatenations and shuffles, trace expressions are able to recognize languages more expressive than regular. But, having a less expressive trace expression (the *over-approximation*) we can translate it to an equivalent Büchi Automaton (second step of the algorithm).

Before going on with the presentation of the second step that presents how

to translate the rewritten trace expression to its equivalent Büchi Automaton, we have to spend more words on the correctness of the rewrite algorithm. In particular, we have to show that the trace expression returned by it is actually an *over-approximation*. In order to show this, we have to think about what we are trying to do with the rewrite function, that is to remove all the expansive subtrees from our trace expression. Consequently, we can show the correctness of our approach through two steps:

1. First of all, we have to show that given a trace expression τ, the rewrite algorithm removes all expansive subtrees from it returning the corresponding trace expression τ', after that,
2. we have to show that such trace expression τ' is an *over-approximation* of τ, meaning that $[\![\tau]\!] \subseteq [\![\tau']\!]$.

In order to prove the rewrite algorithm removes all the expansive subtrees we have to recall briefly which kind of expansive subtrees we can have, that are *expansive concatenations* (Definition 5) and *expansive shuffles* (Definition 6). But, what actually makes a concatenation expansive? As we have already seen previously, a concatenation is expansive if the entire concatenation is a subtree of its head. For instance, $\tau = (\vartheta_{push}{:}(\tau \vee \epsilon)) \cdot (\vartheta_{pop}{:}\epsilon)$ is an expansive concatenation because τ is inside the head. In the same way, we can recall when a shuffle is expansive, namely when the entire shuffle is inside the left or the right operand. For instance, $\tau = (\vartheta_{push}{:}\tau) | (\vartheta_{pop}{:}\epsilon)$ is an expansive shuffle because τ is inside the left operand (left subtree of the shuffle).

Now we show that, starting from an expansive concatenation, the rewrite algorithm removes all the expansive subtrees (the reasoning about the shuffle is almost the same). Recalling the algorithm's pseudocode presented in Algorithm 1, when τ is a concatenation we fall in the fourth case (line 9). The first thing that the algorithm does is updating the set of *already_seen* terms, since now we are analyzing τ, we will add τ to it. After that, since we are in a concatenation, we also update the *dangerous* set adding τ with value **maybe** (we do not know if it is really dangerous or not yet). Now we can call recursively the rewrite function on the head of the concatenation, passing the new updated sets. If the concatenation is expansive, that means we will encounter it again during the evaluation of the head. Since before going into the head we updated the *dangerous* set, if we encounter the concatenation during the evaluation of the head, we fall into the second step (line 3), because τ belongs to the *dangerous* set (with value **maybe**). At this point we know that we are analyzing an expansive subtree for sure. Thus, we can stop analysing it and we can just return back ϵ. The new head is obtained simply concatenating the head returned by the recursive call of the rewrite function, with itself (line 16). The new tail is also obtained in the same way (line 17). Consequently, the new concatenation is just the concatenation of the new head with the new tail (line 18). As already anticipated when we commented the pseudocode presented in Algorithm 1, combining the new head with the new tail we are simulating the previous behavior without counting the events. The rest of the term structure is unchanged. Considering again the example sketched before, we do not count

o.push and *o.pop* anymore, but we preserve their order. For instance, it will never happen that *o.push* is observed after *o.pop* (and so on).

For the expansive shuffle terms is almost the same. We update the *dangerous* set, we evaluate the left and right operands and once the rewritten versions are returned, we simply concatenate the left operand with itself (the same for the right operand) and we construct the result as the shuffle of the two terms so constructed.

Now that we have shown informally that given a trace expression τ the rewrite function remove the expansive cycles returning the rewritten version τ'; we have to show that τ' is an *over-approximation* of τ.

If τ is an expansive trace expression, it means that contains expansive concatenations or expansive shuffles inside. When the rewrite function finds an expansive concatenation, it rewrites it simulating its behavior without the accumulation of the tail. As we have shown before, we obtain in this way a new non-expansive concatenation where the number of events generated consuming the head is independent from the number of events generated consuming the tail. But if it is so, it means that the new concatenation generates a bigger set of events, namely $[\![\tau]\!] \subseteq [\![\tau']\!]$. Thus, we can conclude that τ' is an *over-approximation* of τ (a similar reasoning can be done also for expansive shuffle terms).

Now that we have finished to present the first step of the algorithm and we have obtained the *over-approximation* of our trace expression, we can show how we can translate such trace expression into an equivalent Büchi Automaton.

4.2. 2nd step: Translation

The second step consists in the translation of the trace expression, which was previously rewritten, in an equivalent Büchi Automaton recognizing the same language (set of traces).

Since this trace expression does not contain expansive terms (we removed them in the first step), we can translate it directly simulating the δ transition relation.

Given a non-expansive trace expression τ:

1. create a Büchi Automaton B_τ with an initial state T_0 associated to τ.
2. create a queue of couples Q containing the couple (τ, T_0).
3. extract the first couple (τ_i, T_i) from Q, knowing that the set of events which belong to an event types is finite, for each event e s.t $\tau_i \xrightarrow{e} \tau_i'$:
   - if $\tau_i' = \epsilon$, considering ψ a special event which does not belong to our set of possible events, we create a new state T_ψ and we add $\mathrm{T}_i \xrightarrow{\psi} \mathrm{T}_\psi$, $\mathrm{T}_\psi \xrightarrow{\psi} \mathrm{T}_\psi$ and we make T_i final (if T_ψ already exists, we add only the edge from T_i to it).
   - otherwise if τ_i' has already been seen before, we retrieve the corresponding automaton's state T_r and we add $\mathrm{T}_i \xrightarrow{e} \mathrm{T}_r$ and we make T_r final.

- otherwise, we create a new state T'$_i$ associated to τ_i' and we add $\mathrm{T}_i \xrightarrow{e} \mathrm{T}'_i$.

4. if Q is not empty, we restart from the point 3.

At the end of this process we obtain the Büchi Automaton B_τ equivalent to τ (see Theorem 1).

The ψ special event is used to make acceptable by the Büchi Automaton also the finite traces. In this way, if τ recognizes the finite trace $\sigma = abcd$, the corresponding Büchi Automaton will recognize the trace $u = abcd\,\psi^\omega$. It is important that the ψ event must not belong to the set of handled events in order to avoid *False Negative* results, where the ψ could make an LTL property erroneously satisfied by u.

Example 4. *Considering the same τ' obtained at the end of Example 3, we create the following Büchi Automaton $B_{\tau'}$:*

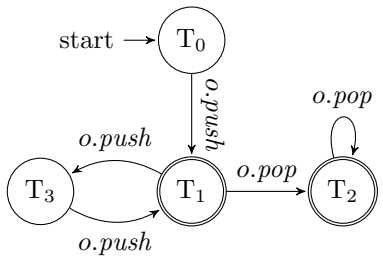

Figure 4: Büchi Automaton $\mathrm{B}_{\tau'}$

In Example 4, the trace expression τ' and the corresponding Büchi Automaton represent a superset of the traces recognized by the initial trace expression τ. In particular, we lose the ability to bind the number of *o.push* with the number of *o.pop* events. In fact, we can only represent traces where we have a number n_{push} of *o.push* before a number n_{pop} of *o.pop*, with $n_{push} \neq n_{pop}$ possibly. Thus, we have conserved only the causality between *o.push* and *o.pop*.

Lemma 1. *Given a trace expression τ, if τ is non-expansive then the set of trace expressions that can be obtained starting from τ in an arbitrary number of steps is finite.*

Proof. If τ is *non-expansive* we do not fall back in the cases like that showed in Example 1 and Example 2 where we have a trace expression generating an infinite set of new states (through term expansion). Removing all the expansive concatenations and shuffles, we have trace expressions that in an arbitrary number of steps become ϵ (we can terminate because we have no more steps to do) or become a trace expression already seen (we can terminate because we have already visited this trace expression). This is easy to note considering the operational semantics of trace expressions (see Section 3.1).

Theorem 1. *Let τ be a non-expansive trace expression, the Büchi Automaton B_τ obtained simulating the δ transition relation is equivalent to τ and it is always computable (Lemma 1), thus*

$$\forall_{u\ infinite\ trace}.u \in [\![\tau']\!] \iff B_{\tau'}\ accepts\ u$$
$$\forall_{\sigma\ finite\ trace}.\sigma \in [\![\tau']\!] \iff B_{\tau'}\ accepts\ \sigma{\cdot}\psi^\omega.$$

Proof. It follows directly from the Büchi Automaton $\mathrm{B}_{\tau'}$ construction.

4.3. 3rd step: Product

The last step consists in the real model checking phase.

In the previous steps, starting from a trace expression τ representing our model we generated its *over-approximation* τ' to have a model expressive as a Büchi Automaton (*1st step*), after that we translated τ' to its corresponding Büchi Automaton representation $\mathrm{B}_{\tau'}$ (*2nd step*). The only step left to complete is check if an LTL property φ is satisfied by the model $\mathrm{B}_{\tau'}$, or not.

As we presented in Section 3, we can follow an *Automata-Based Model Checking* approach computing the product $\mathrm{B}_{\tau'\times\neg\varphi}$ between $\mathrm{B}_{\tau'}$ and $\mathrm{B}_{\neg\varphi}$. Once $\mathrm{B}_{\tau'\times\neg\varphi}$ is created, we can test its emptiness, and if it is not we have found a counterexample and we can conclude that τ' does not satisfy φ.

To fill the gap we have to remind that τ and τ' are related.

Observation 1. $[\![\tau]\!] \subseteq [\![\tau']\!]$, τ' over-approximation *of* τ.

Since τ' is an *over-approximation* of τ and $\mathrm{B}_{\tau'}$ is its automata representation, if we do not find any traces (counterexample) which belongs to $\mathrm{B}_{\tau'\times\neg\varphi}$, we can deduce:

$$\nexists_{u\in \mathrm{B}_{\tau'}}.u \text{ satisfies } \neg\varphi \iff \nexists_{u\in[\![\tau']\!]}.u \text{ satisfies } \neg\varphi$$
(from *Theorem 1*)
$$\nexists_{u\in[\![\tau']\!]}.u \text{ satisfies } \neg\varphi \implies \nexists_{u\in[\![\tau]\!]}.u \text{ satisfies } \neg\varphi$$
(from *Observation 1*)

In this way, we can conclude that, if the *over-approximation* τ' satisfies φ (no counterexample has been found) then also τ satisfies φ. From Observation 1 we can deduce the implication for only one direction ($\implies$) because, since we are over-approximating our model, there could exist a counterexample trace u which does not satisfy φ s.t $u \in [\![\tau']\!]$ but $u \notin [\![\tau]\!]$ (*False Positive*).

One of the main advantages of following a standardized approach as the *Automata-Based Model Checking* is that, once we have obtained the Büchi Automaton $\mathrm{B}_{\tau'\times\neg\varphi}$, we can represent it directly using an existent and well-know programming language as PROMELA inside the popular open-source software verification tool SPIN[11].

[11] `http://spinroot.com/spin/`

5. Experiments

In the following we continue to use the stack example used in the rest of the paper. We show an experiment of use of our SWI-Prolog implementation of the 3-steps algorithm presented in Section 4.

Inside the SWI-Prolog environment, starting from our trace expression `T` representing our very simple stack, we first rewrite `T` into the corresponding non-expansive abstraction `Tr` (*1st Step*) using the corresponding `rewrite` predicate, then we translate `Tr` into the equivalent Büchi Automaton `Buchi` (2nd Step) using the corresponding `translate` predicate, finally we create a promela file `stack.pml` containing the `Buchi` representation and the LTL property that we want to check (for instance, `eventually pop` (`<>(pop)`)).

```
?- T = ((push:(T\/epsilon))*(pop:epsilon)),
rewrite(T, Tr), translate(Tr, Buchi),
write_promela_file(stack, Buchi, '(<>(pop))').
```

The previous SWI-Prolog implementation corresponds to the first two steps of our 3-steps algorithm (Section 4.1 and 4.2). This execution brings to the creation of the `stack.pml` file.

In the following we report the `stack.pml` content so obtained.

```
bool epsilon = 0;
bool pop = 0;
bool push = 0;

active proctype stack() {
 S0_init:
  if
   :: skip -> d_step { push=1; epsilon=0; pop=0 } goto accept_S1
  fi
 accept_S1:
  if
   :: skip -> d_step { push=1; epsilon=0; pop=0 } goto S2
  fi
 S2:
  if
   :: skip -> d_step { pop=1; epsilon=0; push=0 } goto accept_S3
   :: skip -> d_step { push=1; epsilon=0; pop=0 } goto accept_S1
  fi
 accept_S3:
  if
   :: skip -> d_step { epsilon=1; pop=0; push=0 } goto accept_Seps
   :: skip -> d_step { pop=1; epsilon=0; push=0 } goto accept_S3
  fi
```

```
 accept_Seps:
  if
   :: skip -> d_step { epsilon=1; pop=0; push=0 } goto accept_Seps
  fi
}

ltl { (<>(pop)) }
```

This file is the promela code corresponding to the Büchi Automaton presented in Figure 4.

The states in promela correspond to the Büchi Automaton states; for instance, `S0_init` corresponds to T_0 in Figure 4 (and similarly for all the other states involved). Also the Büchi Automaton transitions are translated into the promela code, in particular using the `d_step` construct. For instance, the *o.push* transition from T_0 to T_1 is translated into `d_step{push=1;epsilon=0;pop=0}` from `S0_init` to `accept_S1`. In Figure 4, T_1 was an accepting state and so it is in promela. The `accept_*` pattern is the one used in promela to represent accepting states.

Finally, having the Büchi Automaton representation of our trace expression resulting from the *2nd Step*, we can do the product between `Buchi` and the LTL property (Section 4.3) directly inside the SPIN model checker (*3rd Step*).

First of all, using the `stack.pml` file, we generate a verifier (model checker) for the specification.

In the shell we write:

```
-$ spin -a stack.pml
```

The output generated is a C file, named `pan.c`, that can be compiled to produce an executable verifier.

In the shell we compile the `pan.c` file obtaining the executable corresponding to our verifier.

```
-$ gcc pan.c -o stack_verifier
```

Then, we can run the verifier. Since the LTL property that we want to check is a liveness property, we have to add the `-a` flag in order to find acceptance cycles (violations are infinite executions).

```
-$ ./stack_verifier -a
 pan:1: acceptance cycle (at depth 4)
 ...
 State-vector 28 byte, depth reached 12, errors: 1
```

```
        8 states, stored (10 visited)
        0 states, matched
       10 transitions (= visited+matched)
        0 atomic steps
 ...
```

As we can see, the verifier finds an acceptance cycle (that is a violation of the LTL property). Using the `-r` flag we can read and execute the trail file (generated in the previous step) containing the counterexample trace.

```
-$ ./stack_verifier -r
MSC: ~G 4
  1: proc  0 (ltl_0) stack.pml:4 (state 3) [(!((pop)))]
  2: proc  1 (stack) stack.pml:8 (state 7) [(1)]
  3: proc  0 (ltl_0) stack.pml:4 (state 3) [(!((pop)))]
  4: proc  1 (stack) stack.pml:8 (state 5) [D_STEP8]
<<<<<START OF CYCLE>>>>>
  5: proc  0 (ltl_0) stack.pml:4 (state 3) [(!((pop)))]
  6: proc  1 (stack) stack.pml:12 (state 15) [(1)]
  7: proc  0 (ltl_0) stack.pml:4 (state 3) [(!((pop)))]
  8: proc  1 (stack) stack.pml:12 (state 13) [D_STEP12]
  9: proc  0 (ltl_0) stack.pml:4 (state 3) [(!((pop)))]
 10: proc  1 (stack) stack.pml:16 (state 29) [(1)]
 11: proc  0 (ltl_0) stack.pml:4 (state 3) [(!((pop)))]
 12: proc  1 (stack) stack.pml:17 (state 27) [D_STEP17]
spin: trail ends after 12 steps
#processes 2:
 12: proc 0 (ltl_0)  stack.pml:4 (state  3) (invalid end state)
(!((pop)))
 12: proc 1 (stack)  stack.pml:12 (state 15) (invalid end state)
(1)
global vars:
bit    pop: 0
local vars proc 1 (stack):
```

This cycle corresponds to the infinite trace containing only the `push` event. Consequently, it is not true that our specification recognizes only traces where `<>(pop)` is satisfied. In fact, $o.push^{\omega} \in [\![\tau]\!]$ (that is exactly the trace causing the generation of the trail file reported above).

We can try to check another LTL property. This time a safety property, for instance, `globally push` (`[](push)`). As before, we execute the predicates corresponding to the two first steps of the algorithm directly inside SWI-Prolog (passing the new LTL property as argument). After that we can compile the

new `pan.c` file generated with the `-DSAFETY` flag. Since the LTL formula is a safety property we need to search for assertion violations (violations are finite executions).

Finally we can run the model checker obtaining the following result.

```
-$ ./stack_verifier
pan:1: assertion violated  !( !((push))) (at depth 12)
...
State-vector 20 byte, depth reached 12, errors: 1
        7 states, stored
        0 states, matched
        7 transitions (= stored+matched)
        0 atomic steps
...
```

Also here, we find a counterexample that violates our LTL property. This time, since we are verifying a safety property, we search for a state violating an assertion.

As we did for the previous experiment, also here we can see the trail file generated by the model checker.

```
-$ ./stack_verifier -r
MSC: ~G 3
  1: proc  0 (ltl_0) stack.pml:3 (state 6) [(1)]
  2: proc  1 (stack) stack.pml:8 (state 7) [(1)]
  3: proc  0 (ltl_0) stack.pml:3 (state 6) [(1)]
  4: proc  1 (stack) stack.pml:8 (state 5) [D_STEP8]
  5: proc  0 (ltl_0) stack.pml:3 (state 6) [(1)]
  6: proc  1 (stack) stack.pml:12 (state 15) [(1)]
  7: proc  0 (ltl_0) stack.pml:3 (state 6) [(1)]
  8: proc  1 (stack) stack.pml:12 (state 13) [D_STEP12]
  9: proc  0 (ltl_0) stack.pml:3 (state 6) [(1)]
 10: proc  1 (stack) stack.pml:16 (state 29) [(1)]
 11: proc  0 (ltl_0) stack.pml:3 (state 6) [(1)]
 12: proc  1 (stack) stack.pml:16 (state 21) [D_STEP16]
pan:1: assertion violated  !( !((push))) (at depth 13)
spin: trail ends after 13 steps
#processes 2:
 13: proc 0 (ltl_0)  stack.pml:3 (state  6) (invalid end state)
(!((push)))
(1)
 13: proc 1 (stack)  stack.pml:21 (state 43) (invalid end state)
(1)
(1)
```

```
global vars:
bit    push: 0
local vars proc 1 (stack):
```

Differently from the previous experiment, we do not search for cycles, because we are verifying a safety property, but we search for assertion violations. As we can see in the trail file generated, we find a violation after 13 steps, when our property is violated since `push` is not satisfied anymore (for instance, $\{o.push\ o.push\ o.pop\ o.pop\} \in [\![\tau]\!]$ and $\{o.push\ o.push\ o.pop\ o.pop\}$ does not satisfy `[](push)`).

6. Related Work

In [44], Bodden *et al.* define a formalism which allows to build expressive formulae over temporal traces in an intuitive way as well as a complete implementation of that formalism, which instruments any given Java application in bytecode form with appropriate runtime checks. That approach is similar to ours, in fact, both pass through the automata theory to generate monitors for finite prefixes. In [44], the LTL over *pointcuts* version of the formalism is transformed into monitors (Deterministic Büchi Automata) that are rewritten as advice in in AspectJ[12], as in [13] where the trace expression formalism is transformed into the corresponding automata representing the LTL3[13] monitor.

Session types are used to verify object-oriented languages in [46], where the authors extend their work on session types for distributed object-oriented languages in three ways:

1. they attach a session type to a class definition, to specify the possible sequences of method calls;
2. they allow a session type (protocol) implementation to be modularized, *i.e.* partitioned into separately- callable methods;
3. they treat session-typed communication channels as objects, integrating their session types with the session types of classes.

Several papers by Dezani-Ciancaglini, Yoshida *et al.* [47, 48, 49, 50, 51] have combined session types, as specifications of protocols on communication channels, with the object-oriented paradigm. A characteristic of all of these works is that a channel is always created and used within a single method call.

LTL for Runtime Verification. In [52] the authors review LTL-derived logics for finite traces from a RV perspective. In particular, a number of two-valued semantics for LTL on finite traces have been proposed [53, 54, 55, 56, 57, 11].

[12] `https://eclipse.org/aspectj/`

[13] LTL_3 is a a three-valued semantics [45, 32] for LTL formulas, devised to adapt the standard semantics to RV, to correctly consider the limitation that at runtime only finite traces can be checked.

As remarked in Section 3, LTL was first proposed for the formal verification of computer programs (statically). In [13] the authors demonstrate that the trace expression formalism is strictly more expressive than LTL_3 [45], which is a 3-valued semantics (*true*, *false*, *inconclusive*) for LTL on finite traces (one possible exploitation of LTL for RV). Thanks to the greater expressivity, we can verify more complex properties at runtime, but, this greater expressivity brings also complexity in the static analysis of our properties. If we generate our monitor starting from an LTL_3 property (or other LTL semantics), we know by construction that the monitor satisfies the property (supposing the generation process to be correct). Instead, starting from a trace expression, we may have a complex property which can be more expressive than a combination of LTL properties. Consequently, we might be interested in checking if such kind of complex property still preserve a set of LTL properties (we do not have this by construction). Naturally, even though we lose the correctness by construction, using a more expressive formalism we can define more complex properties for more complex scenarios.

Combining Static and Runtime Verification. In [58, 59], Schneider *et al.* present a tool StaRVOOrs which combines static and runtime verification of Java programs using partial results extracted from static verification to optimize the runtime monitoring process. StaRVOOrs combines the deductive theorem prover KeY [60] and the runtime verification tool LARVA [61], and uses properties written using the ppDATE specification language which combines the control-flow property language DATE used in the runtime verification tool LARVA with Hoare triples assigned to states. In [62], Lin Gui *et al.* propose to combine testing (in particular, hypothesis testing) and model checking (in particular, probabilistic model checking) for non-deterministic systems. Their idea is to apply hypothesis testing to system components which are deterministic and use probabilistic model checking to lift the results through non-determinism. In [63], Artho *et al.* show how to retain information from static analysis for RV, or to compare the results of both techniques inside the NJuke [64] framework for static and dynamic analysis of Java programs. In this framework, a static analyzer looks for faults. Reports are then analyzed by a human, who writes test cases for each kind of fault reported. RV will then analyze the program possibly confirming the fault as a failure or counterexample.

Global types and multi-party sessions. Global types are formal specifications that describe communication protocols in terms of their global interactions. The multi-party sessions are the projection of global types on the participants of the protocol. In [65] the authors show how the projection can be obtained preserving soundness and completeness with respect to the set of traces of the originating global type.

Though trace expressions and global types [65, 66, 67] are rather similar (indeed, global types correspond to trace expressions without the concatenation and the intersection operators), the aim of trace expressions diverges from that of global types for many reasons:

- trace expressions are not intended to be used for annotating and statically checking programs, but rather, for specifying properties that have to be verified at runtime;
- while global types are explicitly designed for describing multiparty interactions between distributed components, trace expressions are meant as a more general formalism which can be used for runtime verification of different kinds of properties and systems;
- finally, trace expressions have a coinductive, rather than inductive, semantics, hence they can denote sets containing infinite traces; this is important for being able to verify systems that must not terminate.

Object-oriented languages. In the context of runtime verification of object-oriented languages, there exist several formalisms for specifying valid or invalid traces of method invocations. Program Query Language (PQL) [68] allows developers to express a large class of application specific code patterns. PQL is more expressive than context-free languages, since its class of languages is that of the closure of context-free languages combined with intersection, hence, the formalism seems to be as expressive as trace expressions. However, no formal semantics is defined for PQL, and it is not clear whether PQL queries can denote infinite traces.

The jassda [69] framework and tool enable runtime checking of Java programs against a CSP-like specification. Like in trace expressions, the trace semantics of a process is defined by collecting all event sequences that are possible with respect to the operational semantics. Processes are built with operators similar to those of trace expressions, except for concatenation and intersection, which are not supported by jassda. SAGA [70] is a tool for runtime verification of properties of Java programs specified with attribute grammars. The implementation is based on four different components: a state-based assertion checker, a parser generator, a debugger and a general tool for meta-programming. The tool is extremely powerful and has been successfully applied to an industrial case from the e-commerce with multi-threaded Java. The main difference w.r.t. our approach is that SAGA has been developed for runtime checking of a combination of protocol- and data-oriented properties of object-oriented programs, whereas, at the moment, trace expressions have been successfully employed for runtime verification of multiagent systems (trace expressions are totally independent from their use and the implementation in the MAS context is only a possible prototype application).

7. Conclusions and Future Work

In this paper, we showed how to use a standard static approach to verify a rich formalism used to generate monitors for runtime verification of object-oriented programs. By verifying LTL properties statically we obtain two main advantages: (1) we can check if the specification of our monitor is coherent with

our intentions, and (2) the system monitored by the verified monitor satisfies the same LTL properties, as long as it is consistent with the specification.

The first two steps of the algorithm presented in Section 4 are implemented in SWI-Prolog, while the last step (the Büchi Automaton product) is implemented using the SPIN model checker. The Büchi Automaton B_τ generated in the second step and the LTL property φ we want to verify are both compiled to PROMELA language.

The next steps will be to improve the search for cycles inside expansive terms, and to study in greater detail the expressivity of trace expressions in order to understand if an hybrid approach is possible, where the LTL properties verified statically can bring us to simplify the trace expression generating consequently a simpler version of the monitor.

In [71] the authors presented an parametric version of the trace expression formalism. Thanks to parameters inside event types, this extension is extremely more expressive than the standard one [13]. One possible future work will be achieving the static verification also for a parametric trace expression. The presence of parameters makes not usable the standard automata-based approach. One promising way to solve this problem is through the model checker Cubicle [72] (used for symbolic backward reachability analysis on infinite sets of states).

Acknowledgements

Thanks to prof. Davide Ancona, prof. Viviana Mascardi, and the anonymous reviewers for the helpful comments on the previous versions of the paper.

References

[1] W. Visser, K. Havelund, G. P. Brat, S. Park, F. Lerda, Model checking programs, Autom. Softw. Eng. 10 (2) (2003) 203–232. doi:10.1023/A:1022920129859.
URL https://doi.org/10.1023/A:1022920129859

[2] E. M. Clarke, O. Grumberg, D. A. Peled, Model checking, MIT Press, 2001.
URL http://books.google.de/books?id=Nmc4wEaLXFEC

[3] G. J. Holzmann, Software analysis and model checking, in: E. Brinksma, K. G. Larsen (Eds.), Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings, Vol. 2404 of Lecture Notes in Computer Science, Springer, 2002, pp. 1–16. doi:10.1007/3-540-45657-0_1.
URL https://doi.org/10.1007/3-540-45657-0_1

[4] S. Merz, Model checking: A tutorial overview, in: F. Cassez, C. Jard, B. Rozoy, M. D. Ryan (Eds.), Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000, Vol. 2067 of Lecture Notes in Computer Science, Springer, 2000, pp.

3–38. doi:10.1007/3-540-45510-8_1.
URL https://doi.org/10.1007/3-540-45510-8_1

[5] T. S. Chow, Testing software design modeled by finite-state machines, IEEE Trans. Software Eng. 4 (3) (1978) 178–187. doi:10.1109/TSE.1978.231496.
URL https://doi.org/10.1109/TSE.1978.231496

[6] M. Broy, B. Jonsson, J. Katoen, M. Leucker, A. Pretschner (Eds.), Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004], Vol. 3472 of Lecture Notes in Computer Science, Springer, 2005. doi:10.1007/b137241.
URL https://doi.org/10.1007/b137241

[7] G. J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, 3rd Edition, Wiley Publishing, 2011.

[8] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, IEEE Computer Society, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.
URL https://doi.org/10.1109/SFCS.1977.32

[9] M. Leucker, C. Schallhart, A brief account of runtime verification, The Journal of Logic and Algebraic Programming 78 (5) (2009) 293 – 303, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS?07). doi:https://doi.org/10.1016/j.jlap.2008.08.004.
URL http://www.sciencedirect.com/science/article/pii/S1567832608000775

[10] Y. Falcone, You should better enforce than verify, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, Vol. 6418 of Lecture Notes in Computer Science, Springer, 2010, pp. 89–105. doi:10.1007/978-3-642-16612-9_9.
URL https://doi.org/10.1007/978-3-642-16612-9_9

[11] M. d'Amorim, G. Rosu, Efficient monitoring of omega-languages, in: K. Etessami, S. K. Rajamani (Eds.), Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings, Vol. 3576 of Lecture Notes in Computer Science, Springer, 2005, pp. 364–378. doi:10.1007/11513988_36.
URL https://doi.org/10.1007/11513988_36

[12] D. Ancona, S. Drossopoulou, V. Mascardi, Automatic generation of self-monitoring mass from multiparty global session types in jason, in: M. Baldoni, L. A. Dennis, V. Mascardi, W. W. Vasconcelos (Eds.), Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers, Vol. 7784 of Lecture Notes in Computer Science, Springer, 2012, pp. 76–95. doi:10.1007/978-3-642-37890-4_5.
URL https://doi.org/10.1007/978-3-642-37890-4_5

[13] D. Ancona, A. Ferrando, V. Mascardi, Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, Springer International Publishing, Cham, 2016, Ch. Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification, pp. 47–64. doi:10.1007/978-3-319-30734-3_6.
URL http://dx.doi.org/10.1007/978-3-319-30734-3_6

[14] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems, ESOP '98, Springer-Verlag, London, UK, UK, 1998, pp. 122–138.
URL http://dl.acm.org/citation.cfm?id=645392.651876

[15] K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: C. Halatsis, D. G. Maritsas, G. Philokyprou, S. Theodoridis (Eds.), PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings, Vol. 817 of Lecture Notes in Computer Science, Springer, 1994, pp. 398–413. doi:10.1007/3-540-58184-7_118.
URL https://doi.org/10.1007/3-540-58184-7_118

[16] T. L. Hinrichs, A. P. Sistla, L. D. Zuck, Model check what you can, runtime verify the rest, in: HOWARD-60, Vol. 42 of EPiC Series in Computing, EasyChair, 2014, pp. 234–244.

[17] A. Ferrando, L. A. Dennis, D. Ancona, M. Fisher, V. Mascardi, Recognising assumption violations in autonomous systems verification, in: AAMAS, International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018, pp. 1933–1935.

[18] M. Y. Vardi, Automata-theoretic model checking revisited, in: VMCAI, Vol. 4349 of Lecture Notes in Computer Science, Springer, 2007, pp. 137–150.

[19] G. J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[20] G. J. Holzmann, The model checker SPIN, IEEE Trans. Software Eng. 23 (5) (1997) 279–295. doi:10.1109/32.588521.
URL https://doi.org/10.1109/32.588521

[21] G. J. Holzmann, The SPIN Model Checker - primer and reference manual, Addison-Wesley, 2004.

[22] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (1-2) (2012) 67–96.

[23] R. Alur, Model checking: From tools to theory, in: 25 Years of Model Checking, Vol. 5000 of Lecture Notes in Computer Science, Springer, 2008, pp. 89–106.

[24] C. Dubslaff, C. Baier, M. Berg, Model checking probabilistic systems against pushdown specifications, Inf. Process. Lett. 112 (8-9) (2012) 320–328.

[25] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in: Lectures on Runtime Verification, Vol. 10457 of Lecture Notes in Computer Science, Springer, 2018, pp. 1–33.

[26] S. A. Kripke, Semantical analysis of modal logic i. normal propositional calculi, Zeitschrift fur mathematische Logik und Grundlagen der Mathematik 9 (56) (1963) 67–96.

[27] A. N. Prior, Time and Modality, Greenwood Press, 1955.

[28] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the temporal analysis of fairness, in: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '80, ACM, New York, NY, USA, 1980, pp. 163–173. doi:10.1145/567446.567462.
URL http://doi.acm.org/10.1145/567446.567462

[29] F. Chen, G. Rosu, Mop: an efficient and generic runtime verification framework, in: OOPSLA 2007, 2007, pp. 569–588.

[30] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, G. Rosu, Rv-monitor: Efficient parametric runtime verification with simultaneous properties, in: RV'14, Vol. 8734, Springer, 2014, pp. 285–300.

[31] J. Cohen, D. Perrin, J. Pin, On the expressive power of temporal logic, J. Comput. Syst. Sci. 46 (3) (1993) 271–294. doi:10.1016/0022-0000(93)90005-H.
URL https://doi.org/10.1016/0022-0000(93)90005-H

[32] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 14:1–14:64. doi:10.1145/2000799.2000800.
URL http://doi.acm.org/10.1145/2000799.2000800

[33] O. Lichtenstein, A. Pnueli, Checking that finite state concurrent programs satisfy their linear specification, in: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL

'85, ACM, New York, NY, USA, 1985, pp. 97–107. doi:10.1145/318593.318622.
URL http://doi.acm.org/10.1145/318593.318622

[34] M. Y. Vardi, An automata-theoretic approach to linear temporal logic, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 238–266. doi:10.1007/3-540-60915-6_6.
URL https://doi.org/10.1007/3-540-60915-6_6

[35] J. R. Büchi, On a Decision Method in Restricted Second Order Arithmetic, Springer New York, New York, NY, 1990, pp. 425–435. doi:10.1007/978-1-4613-8928-6_23.
URL https://doi.org/10.1007/978-1-4613-8928-6_23

[36] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. M. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. Vasconcelos, N. Yoshida, Behavioral types in programming languages, Foundations and Trends in Programming Languages 3 (2-3) (2016) 95–230.

[37] A. D., M. Barbieri, V. Mascardi, Constrained global types for dynamic checking of protocol conformance in multi-agent systems, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, 2013, pp. 1377–1379.

[38] D. Caucal, On infinite terms having a decidable monadic theory, in: K. Diks, W. Rytter (Eds.), Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, August 26-30, 2002, Proceedings, Vol. 2420 of Lecture Notes in Computer Science, Springer, 2002, pp. 165–176. doi:10.1007/3-540-45687-2_13.
URL https://doi.org/10.1007/3-540-45687-2_13

[39] W. Damm, Languages defined by higher type program schemes, in: A. Salomaa, M. Steinby (Eds.), Automata, Languages and Programming, Fourth Colloquium, University of Turku, Finland, July 18-22, 1977, Proceedings, Vol. 52 of Lecture Notes in Computer Science, Springer, 1977, pp. 164–179. doi:10.1007/3-540-08342-1_13.
URL https://doi.org/10.1007/3-540-08342-1_13

[40] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Transactions on Software Engineering and Methodology (TOSEM)In press.

[41] J. E. Hopcroft, J. D. Ullman, Formal languages and their relation to automata, Addison-Wesley series in computer science and information processing, Addison-Wesley, 1969.

[42] W. Thomas, Automata on infinite objects, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), 1990, pp. 133–192.

[43] L. Staiger, On omega-power languages, in: New Trends in Formal Languages, Vol. 1218 of Lecture Notes in Computer Science, Springer, 1997, pp. 377–394.

[44] E. Bodden, Efficient and Expressive Runtime Verification for Java, in: Grand Finals of the ACM Student Research Competition 2005, 2005.
URL http://www.bodden.de/pubs/bodden05efficient.pdf

[45] A. Bauer, M. Leucker, C. Schallhart, Monitoring of real-time properties, in: S. Arun-Kumar, N. Garg (Eds.), FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings, Vol. 4337 of Lecture Notes in Computer Science, Springer, 2006, pp. 260–272. doi:10.1007/11944836_25.
URL https://doi.org/10.1007/11944836_25

[46] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, A. Z. Caldeira, Modular session types for distributed object-oriented programming, SIGPLAN Not. 45 (1) (2010) 299–312. doi:10.1145/1707801.1706335.
URL http://doi.acm.org/10.1145/1707801.1706335

[47] S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, Amalgamating sessions and methods in object-oriented languages with generics, Theor. Comput. Sci. 410 (2-3) (2009) 142–167. doi:10.1016/j.tcs.2008.09.016.
URL http://dx.doi.org/10.1016/j.tcs.2008.09.016

[48] M. Dezani-Ciancaglini, E. Giachino, S. Drossopoulou, N. Yoshida, Formal Methods for Components and Objects: 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, Ch. Bounded Session Types for Object Oriented Languages, pp. 207–245. doi:10.1007/978-3-540-74792-5_10.
URL http://dx.doi.org/10.1007/978-3-540-74792-5_10

[49] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, S. Drossopoulou, Session types for object-oriented languages, in: ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings, 2006, pp. 328–352. doi:10.1007/11785477_20.
URL http://dx.doi.org/10.1007/11785477_20

[50] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, S. Drossopoulou, Trustworthy Global Computing: International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005. Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, Ch. A Distributed Object-Oriented Language with Session Types, pp. 299–318. doi:10.1007/11580850_16.
URL http://dx.doi.org/10.1007/11580850_16

[51] R. Hu, N. Yoshida, K. Honda, ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, Ch. Session-Based Distributed Programming in Java, pp. 516–541. doi:10.1007/978-3-540-70592-5_22.
URL http://dx.doi.org/10.1007/978-3-540-70592-5_22

[52] A. Bauer, M. Leucker, C. Schallhart, Comparing LTL semantics for runtime verification, J. Log. Comput. 20 (3) (2010) 651–674. doi:10.1093/logcom/exn075.
URL https://doi.org/10.1093/logcom/exn075

[53] D. Giannakopoulou, K. Havelund, Automata-based verification of temporal properties on running programs, in: 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA, IEEE Computer Society, 2001, pp. 412–416. doi:10.1109/ASE.2001.989841.
URL https://doi.org/10.1109/ASE.2001.989841

[54] D. Giannakopoulou, K. Havelund, Runtime analysis of linear temporal logic specifications, Tech. rep. (2001).

[55] K. Havelund, G. Rosu, Synthesizing monitors for safety properties, in: J. Katoen, P. Stevens (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, Vol. 2280 of Lecture Notes in Computer Science, Springer, 2002, pp. 342–356. doi:10.1007/3-540-46002-0_24.
URL https://doi.org/10.1007/3-540-46002-0_24

[56] K. Havelund, G. Rosu, Monitoring java programs with java pathexplorer, Electr. Notes Theor. Comput. Sci. 55 (2) (2001) 200–217. doi:10.1016/S1571-0661(04)00253-1.
URL https://doi.org/10.1016/S1571-0661(04)00253-1

[57] V. Stolz, E. Bodden, Temporal assertions using aspectj, Electr. Notes Theor. Comput. Sci. 144 (4) (2006) 109–124. doi:10.1016/j.entcs.2006.02.007.
URL https://doi.org/10.1016/j.entcs.2006.02.007

[58] J. M. Chimento, W. Ahrendt, G. J. Pace, G. Schneider, Starvoors: A tool for combined static and runtime verification of java, in: Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings, 2015, pp. 297–305. doi:10.1007/978-3-319-23820-3_21.
URL https://doi.org/10.1007/978-3-319-23820-3_21

[59] W. Ahrendt, G. J. Pace, G. Schneider, Starvoors - episode II - strengthen and distribute the force, in: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, 2016, pp. 402–415. doi:10.1007/978-3-319-47166-2_28.
URL https://doi.org/10.1007/978-3-319-47166-2_28

[60] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, M. Ulbrich (Eds.), Deductive Software Verification - The KeY Book - From Theory to Practice, Vol. 10001 of Lecture Notes in Computer Science, Springer, 2016. doi:10.1007/978-3-319-49812-6.
URL http://dx.doi.org/10.1007/978-3-319-49812-6

[61] C. Colombo, G. J. Pace, G. Schneider, LARVA — safer monitoring of real-time java programs (tool paper), in: Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009, 2009, pp. 33–37. doi:10.1109/SEFM.2009.13.
URL https://doi.org/10.1109/SEFM.2009.13

[62] L. Gui, J. Sun, Y. Liu, Y. Si, J. S. Dong, X. Wang, Combining model checking and testing with an application to reliability prediction and distribution, in: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013, 2013, pp. 101–111. doi:10.1145/2483760.2483779.
URL http://doi.acm.org/10.1145/2483760.2483779

[63] C. Artho, A. Biere, Combined static and dynamic analysis, Electr. Notes Theor. Comput. Sci. 131 (2005) 3–14. doi:10.1016/j.entcs.2005.01.018.
URL https://doi.org/10.1016/j.entcs.2005.01.018

[64] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, B. Zweimüller, Jnuke: Efficient dynamic analysis for java, in: Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings, 2004, pp. 462–465. doi:10.1007/978-3-540-27813-9_37.
URL https://doi.org/10.1007/978-3-540-27813-9_37

[65] G. Castagna, M. Dezani-Ciancaglini, L. Padovani, On global types and multi-party session, Logical Methods in Computer Science 8 (1).

[66] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, J. ACM 63 (1) (2016) 9:1–9:67. doi:10.1145/2827695.
URL http://doi.acm.org/10.1145/2827695

[67] L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions,

in: F. van Breugel, M. Chechik (Eds.), CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings, Vol. 5201 of Lecture Notes in Computer Science, Springer, 2008, pp. 418–433. doi:10.1007/978-3-540-85361-9_33.
URL https://doi.org/10.1007/978-3-540-85361-9_33

[68] M. C. Martin, V. B. Livshits, M. S. Lam, Finding application errors and security flaws using PQL: a program query language, in: OOPSLA 2005, 2005, pp. 365–383.

[69] M. Brörkens, M. Möller, Dynamic event generation for runtime checking using the JDI, Electr. Notes Theor. Comput. Sci. 70 (4) (2002) 21–35.

[70] F. S. de Boer, S. de Gouw, Combining monitoring with run-time assertion checking, in: M. Bernardo, F. Damiani, R. Hähnle, E. B. Johnsen, I. Schaefer (Eds.), Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures, Vol. 8483 of Lecture Notes in Computer Science, Springer, 2014, pp. 217–262. doi:10.1007/978-3-319-07317-0_6.
URL https://doi.org/10.1007/978-3-319-07317-0_6

[71] D. Ancona, A. Ferrando, L. Franceschini, V. Mascardi, Parametric trace expressions for runtime verification of java-like programs, in: Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017, ACM, 2017, pp. 10:1–10:6. doi:10.1145/3103111.3104037.
URL http://doi.acm.org/10.1145/3103111.3104037

[72] S. Conchon, A. Goel, S. Krstic, A. Mebsout, F. Zaïdi, Cubicle: A parallel smt-based model checker for parameterized systems - tool paper, in: CAV, 2012, pp. 718–724.