A Unifying Framework for Dynamic Monitoring and a Taxonomy of Optimizations

Marie-Christine Jakobs and Heiko Mantel
{jakobs,mantel}@cs.tu-darmstadt.de

Department of Computer Science, TU Darmstadt, Germany

Abstract. Reducing the performance overhead of run-time monitoring is crucial for making it affordable to enforce more complex requirements than simple security or safety properties. Optimizations for reducing the overhead are becoming increasingly sophisticated themselves, which makes it mandatory to verify that they preserve what shall be enforced.

In this article, we propose a taxonomy for such optimizations and use it to develop a classification of existing optimization techniques. Moreover, we propose a semantic framework for modeling run-time monitors that provides a suitable basis both, for verifying that optimizations preserve reliable enforcement and for analytically assessing the performance gain.

1 Introduction

Run-time verification is a popular technique for ensuring that a program satisfies given requirements. Conceptually, a monitor observes runs of a target program and interferes, when a violation of some requirement is about to occur. In contrast to static analysis and formal verification, run-time verification does not analyze all possible runs of the target program, but only the ones that actually occur. This simplifies the analysis. In addition, a dynamic analysis has knowledge of actual values in a run and can thereby achieve better precision.

The down-side of run-time monitoring (short: RTM) is that the monitor induces a performance overhead each time the target program is run. This overhead can become unacceptably high, and it is a common reason for abstaining from using run-time monitoring in practice [7]. When complex requirements are enforced by RTM and complex target programs are monitored, then the performance overhead becomes an even bigger problem. This is, for instance, relevant when using RTM in software reengineering for adding properties to existing software.

The goal of optimizing RTM is to reduce this performance overhead. To achieve acceptable performance, complex optimizations for RTM [34, 18] are suggested. However, they are lacking full formal correctness proofs. When applying such optimizations, one currently gives up formal guarantees of RTM [33, 29]. Thus, we need better foundations to prove complex optimizations for RTM.

As a first step, we provide the foundations to prove RTM optimizations from two simple classes of optimizations. Along the way, we developed a semantic framework for characterizing the behavior of run-time monitors. Our framework

Published in In: Margaria T., Steffen B. (eds): ISoLA 2020, LNCS volume 12477, pp. 72–92, 2020.
© Springer Verlag
The final authenticated version is available online at https://doi.org/10.1007/978-3-030-61470-6_6.

allows us to carry the distinction of observable and controllable events [8] over to optimizations. Moreover, we distinguish the affirmative treatment of requirements from a preventive treatment, which both occur in prior work. We clarify where they differ technically and how they can be used interchangeably. We aim at using our semantic framework in the future to verify further classes of optimizations.

Several heterogeneous optimizations for RTM are proposed in the literature [11, 10, 34, 30, 18, 5, 35, 22, 3]. Since we aim at using run-time monitoring to enforce specifications, we focus on optimizations that aim for soundness, i.e., to not miss any requirement violations. To identify (interesting) classes of optimizations proposed in the literature, we compare the existing approaches on a conceptual level. Previously proposed taxonomies for RTM [14, 27, 32, 20] do not cover optimizations. We introduce a 3-dimensional taxonomy for RTM optimizations, which considers how requirements are specified, how the necessary information for the optimization is obtained, and how the optimization is performed. We use our taxonomy to classify prior work on optimization of RTM.

In summary, we conduct first steps in a larger research effort to formally prove optimizations for RTM. We show how to formally prove optimizations for RTM and what needs to be done. As an unexpected bonus, our semantic framework helped us to come up with new, more sophisticated optimizations.

2 Preliminaries

Basic Notions and Notation. We use $\langle\rangle$ to denote the empty sequence, $\langle a\rangle$ to denote the sequence consisting of a single symbol a, and $a.w$ to denote the sequence starting with the symbol a, followed by the sequence w.

We define the concatenation of two sequences w_1 and w_2 (denoted by $w_1 \cdot w_2$) recursively by $\langle\rangle \cdot w_2 = w_2$ and $(a.w_1) \cdot w_2 = a.(w_1 \cdot w_2)$. Overloading notation, we use $w.a$ as an abbreviation for $w \cdot \langle a\rangle$. A sequence w_1 is a prefix (w_2 is a suffix) of a sequence w if there exists a sequence w_2 (a sequence w_1) such that $w = w_1 \cdot w_2$ holds. A set of sequences L is prefix-closed (is suffix-closed) if $w \in L$ implies that $w_1 \in L$ holds for each prefix (for each suffix) w_1 of w. The suffix-closure of L (denoted $\mathcal{SUF}(L)$) is the smallest suffix-closed super-set of L.

We define the projection of a sequence w to a set A (denoted by $w|_A$) recursively by $\langle\rangle|_A = \langle\rangle$, $(a.w)|_A = w|_A$ if $a \notin A$, and $(a.w)|_A = a.(w|_A)$ if $a \in A$.

We lift concatenation and projection from sequences to sets of sequences by $L_1 \cdot L_2 = \bigcup_{w_1 \in L_1, w_2 \in L_2} \{w_1 \cdot w_2\}$ and by $L|_A = \bigcup_{w \in L} \{w|_A\}$. We define the *-operator as the smallest fixed-point of $L^* = \{\langle\rangle\} \cup \{a.w \mid a \in L \wedge w \in L^*\}$.

Projection preserves the $\in$- and the $\subseteq$-relationship, i.e., $w \in L$ implies $w|_A \in L|_A$, and $L_1 \subseteq L_2$ implies $L_1|_A \subseteq L|_{A_2}$. By contraposition, $w|_A \notin L|_A$ implies $w \notin L$, and $L_1|_A \not\subseteq L_2|_A$ implies $L_1 \not\subseteq L_2$. We will exploit these facts in proofs.

We denote the space of total functions and the space of partial functions from A to B by $A \to B$ and $A \hookrightarrow B$, respectively. For $f : A \hookrightarrow B$, we write $f(a){\downarrow}$ if f is defined at $a \in A$, i.e. $\exists b \in B.\, f(a) = b$, and $f(a){\uparrow}$ if f is undefined at a.

2.1 Labeled Transition Systems and Properties

We use labeled transition systems to formally characterize the behavior of programs in a uniform fashion. Given a program in some programming, byte-code or machine language, it is straightforward to construct from the language's small-step operational semantics a corresponding labeled transition system where events correspond to executing individual instructions. Our use of labeled transition systems is also compatible with coarser-grained or finer-grained events.

Definition 1. *A* labeled transition system *(brief:* LTS*) is a tuple* (S, S_0, E, Δ)*, where* S *is a* set of states*,* $S_0 \subseteq S$ *is a* set of initial states*,* E *is a* set of events*, and* $\Delta \subseteq (S \times E \times S)$ *is a* transition relation*.*

The relation $\Delta^* \subseteq (S \times E^* \times S)$ *is defined inductively by* $(s, \langle\rangle, s) \in \Delta^*$ *and by* $(s, a.w, s'') \in \Delta^*$ *if* $\exists s' \in S. ((s, a, s') \in \Delta \wedge (s', w, s'') \in \Delta^*)$*.*

As usual, we use traces to model program runs. The *set of traces induced by an LTS* $lts = (S, S_0, E, \Delta)$ *starting in* $s \in S$ is the smallest set $Traces(lts, s) \subseteq E^*$ such that $Traces(lts, s) = \{w \in E^* \mid \exists s' \in S. (s, w, s') \in \Delta^*\}$. The *set of traces induced by lts* is $Traces(lts) = \bigcup_{s_0 \in S_0} Traces(lts, s_0)$. Note that both sets, $Traces(lts)$ and $Traces(lts, s)$ are prefix-closed.

We focus on properties in the sense of Alpern and Schneider [4].

Definition 2. *A* property *over a set* E *is a function* $prop : E^* \to Bool$*.*

A labeled transition system $lts = (S, S_0, E, \Delta)$ satisfies a property $prop : E^* \to Bool$ *iff* $\forall tr \in Traces(lts). prop(tr) = \top$ *holds.*

2.2 Finite Automata and Formal Languages

We use automata to formally characterize the behavior of run-time monitors.

Definition 3. *A* finite automaton *(brief:* FA*) is a tuple* (S, S_0, S_F, A, δ)*, where* S *and* A *are finite sets,* $S_0 \subseteq S$ *and* $S_F \subseteq S$ *are nonempty, and* $\delta : (S \times A) \hookrightarrow S$*.*

The function $\delta^* : (S \times A^*) \hookrightarrow S$ *is defined inductively by* $\delta^*(s, \langle\rangle) = s$ *and by* $\delta^*(s, a.w) = s''$ *if* $\exists s' \in S. (\delta(s, a) = s' \wedge \delta^*(s', w) = s'')$*.*

Given a finite automaton $fa = (S, S_0, S_F, A, \delta)$, S is *the set of states*, S_0 is *the set of initial states*, S_F is *the set of final states*, A is *the alphabet*, and δ is *the transition function* of fa. An FA $fa = (S, S_0, S_F, A, \delta)$, is *total in* $a \in A$ iff $\forall s \in S. \exists s' \in S. \delta(s, a) = s'$, and fa is *total in* $A' \subseteq A$ iff fa is total in each $a \in A'$.

A *word over an alphabet* A is a finite sequence over A. A *language over an alphabet* A is a set of words over A. We call a language L over A *simple* iff $L = \{\}$, $L = \{\langle\rangle\}$, or $L = \{\langle a\rangle\}$ holds for some $a \in A$. A language L is *regular* iff L can be composed from simple languages using the operators $\cup$, $\cdot$, and *.

Definition 4. *A* word $w \in A^*$ is accepted in $s \in S$ *by a finite automaton* $fa = (S, S_0, S_F, A, \delta)$ *iff* $\delta^*(s, w) \in S_F$*. The language accepted by fa in* $s \in S$ *is the set of all such words, i.e.,* $Lang(fa, s) = \{w \in A^* \mid \delta^*(s, w) \in S_F\}$*.*

A word $w \in A^*$ is accepted by fa *iff fa accepts w in some initial state* $s_0 \in S_0$*. The* language accepted by fa *is* $Lang(fa) = \bigcup_{s_0 \in S_0} Lang(fa, s_0)$*.*

A word $w \in A^*$ is rejected by fa *iff w is not accepted by fa.*

The expressiveness of finite automata is given by Kleene's theorem [26]. The language $Lang(fa)$ is regular for each finite automaton fa, and, for each regular language L, there exists a finite automaton fa with $Lang(fa) = L$.

Finite automata where every state is final, have a prominent role in this article. The languages accepted by this class of finite automata coincide with the languages that are regular and prefix closed [25].

We introduce three operations for adapting transition functions: The lifting of a transition function $\delta\colon (S\times A)\hookrightarrow S$ to an alphabet B augments the domain of δ by adding stuttering steps for all pairs in $(S\times(B\setminus A))$. The restriction of δ to B restricts the domain of δ to events in $(A\cap B)$. The completion of δ to B adds a stuttering step for each $b\in B$ and $s\in S$, wherever $\delta(s,b)$ is undefined.

Definition 5. *Let $\delta : (S\times A) \hookrightarrow S$ be a transition function and B be an alphabet. The* lifting *of δ to B is $\delta{\uparrow}^B\colon (S\times(A\cup B)) \hookrightarrow S$, the* restriction *of δ to B is $\delta|_B\colon (S\times(A\cap B)) \hookrightarrow S$, and the* completion *of δ to B is $\delta\circlearrowleft^B\colon (S\times A) \hookrightarrow S$ with*

$$\begin{array}{rll} \delta{\uparrow}^B(s,a) = \delta(s,a) & \text{if } a\in A & \delta{\uparrow}^B(s,a) = s \quad \text{if } a\in(B\setminus A) \\ \delta|_B(s,a) = \delta(s,a) & \text{if } a\in(A\cap B) & \\ \delta\circlearrowleft^B(s,a) = \delta(s,a) & \text{if } \delta(s,a)\downarrow & \delta\circlearrowleft^B(s,a) = s \quad \text{if } a\in(A\cap B)\wedge\delta(s,a)\uparrow \end{array}$$

Definition 6. *The* lifting of a finite automaton *$fa = (S,S_0,S_F,A,\delta)$ to an alphabet B is the finite automaton $fa^{\uparrow B} = (S,S_0,S_F,(A\cup B),\delta{\uparrow}^B)$.*

3 A Framework for Monitoring and Enforcement

A run-time monitor checks whether actions of a target program are permissible before they occur. If the monitor classifies an action as affirmative, then the action may occur, and, otherwise, the monitor takes appropriate countermeasures.

Historically, the field of run-time monitoring has close ties to automata theory. This connection is beneficial since constructions and insights from automata theory can be exploited in the development and analysis of RTM solutions.

The use of automata for run-time monitoring, however, is not uniform. For using finite automata, e.g., one can observe two approaches:

1. Each word accepted by an FA corresponds to an *affirmative* behavior.
2. Each word accepted by an FA corresponds to a *preventive* behavior.

At the level of formal languages, the approaches are duals of each other, and switching from the one to the other is straightforward: Given a regular language L specifying the affirmative behaviors, one can use an FA that recognizes L for monitoring (following the first approach). Alternatively, one can use an FA that recognizes the complement of L (following the second approach to monitoring).

Due to this duality at the level of formal languages, the two approaches are sometimes treated as if they were fully interchangeable. However, when implementing run-time monitoring, one needs to commit to one of these approaches. This choice might impact performance overhead, and this is not merely an implementation-level issue, as we will clarify in Section 4.

In the development of our semantic framework, we carefully distinguish

- between finite automata and the requirements that shall be enforced and
- between two approaches of using finite automata for RTM.

As usual, we use formal languages to specify the requirements to be enforced. However, to clearly distinguish between the two approaches to run-time monitoring, we refer to such languages as *policies*, if they specify affirmative runs, and as *anti-policies*, if they specify preventive runs. We deliberately do not identify policies/anti-policies with accepting automata, because this would limit the use of our framework for verifying soundness of optimizations.

We formally introduce a notion of monitors on top of finite automata. This allows us to explicitly distinguish between events of the target that can be fully controlled from events that can be observed but not controlled. This distinction is relevant for the enforceability of properties [8], as well as for the performance overhead (see Section 4). We use the term *monitor* when following the first approach to RTM and the term *watch-dog* when following the second approach.

The interplay between target programs, monitors/watch-dogs, properties, and policies/anti-policies is visualized on the left-hand side of Fig. 1.

3.1 Policies and Anti-Policies

We use policies and anti-policies to specify restrictions that shall be enforced when a target program is running. A policy specifies which behaviors are affirmed, while an anti-policy specifies which behaviors are prevented.

Definition 7. *A* policy *is a pair $pol = (A, Tr)$, where A is an alphabet, and $Tr \subseteq A^*$ is a non-empty and prefix-closed language over A, the* affirmative traces*.*

An anti-policy *is a pair $apol = (A', Tr')$, where A' is an alphabet and $Tr' \subseteq A'^*$ is a language over A' with $\langle\rangle \notin Tr'$, the* preventive traces*.*

We define the meaning of policies/anti-policies for labeled transition systems:

Definition 8. *The properties specified by $pol = (A, Tr)$ and $apol = (A', Tr')$, respectively, for a set E are $prop^E_{pol}, \overline{prop}^E_{apol} : E^* \to Bool$ defined by:*

$$prop^E_{pol}(w) = \begin{cases} \top & \text{if } w|_A \in Tr \\ \bot & \text{otherwise.} \end{cases}$$

$$\overline{prop}^E_{apol}(w) = \begin{cases} \bot & \text{if } \exists w_1, w_2 \in E^*. (w = w_1 \cdot w_2 \wedge w_1|_{A'} \in Tr') \\ \top & \text{otherwise.} \end{cases}$$

Intuitively, policies and anti-policies are dual concepts. The following theorem substantiates this conceptual duality more precisely based on our definitions.

Theorem 1. *Let $pol = (A, Tr)$ be a policy and $apol = (A, Tr')$ be an anti-policy. If $Tr' = A^* \setminus Tr$ then $prop^E_{pol}(w) = \overline{prop}^E_{apol}(w)$ holds for all $w \in E^*$.*

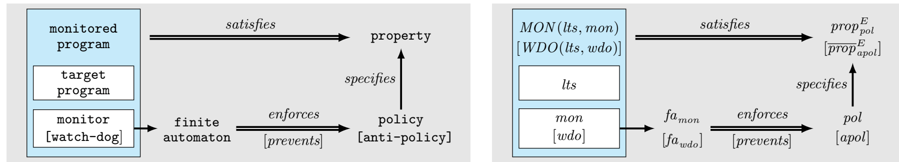

Fig. 1. Run-time monitoring and enforcement conceptually (lhs) and formally (rhs)

Theorem 1 can be used to construct from a policy an anti-policy that specifies the same property. This construction is universal in the set of events. In the other direction, constructing a policy from an anti-policy $apol=(A, Tr')$ is only possible if Tr' is suffix-closed, as, otherwise, $(A,(A^*\setminus Tr'))$ is not a policy.

3.2 Monitors and Enforcement of Policies

For monitors, we use transition functions as follows to specify which actions are affirmed in which states: If the transition function is defined for a given state and action, then this action is affirmed in this state and, otherwise, not. This reflects, e.g., the behavior of security automata [33] or truncation automata [29].

A monitor might only supervise a subset of a target program's actions. Among those actions, we distinguish between actions whose occurrences the monitor can prevent and actions whose occurrences the monitor cannot prevent. This allows us to faithfully capture actions whose occurrences simply cannot be prevented (e.g., the progress of time). This distinction will also be relevant for our performance model (see Section 4) and optimizations (see Section 5).

The distinction of observable and controllable events has been exploited before in other contexts. For instance, Basin et al. [8] use it to analyze enforceability.

As a notational convention, we use Γ_o to denote the set of events whose occurrences a monitor can observe but not control, and we use Γ_c to denote the set of events whose occurrences a monitor can observe and also control.

Definition 9. *A* monitor *is a tuple* $mon = (MS, ms_0, \Gamma_o, \Gamma_c, \delta)$*, where*
- MS *is a finite* set of monitor states *with* initial state $ms_0 \in MS$*,*
- Γ_o *is a finite* set of observed events*,*
- Γ_c *is a finite* set of controlled events *with* $\Gamma_o \cap \Gamma_c = \emptyset$*, and*
- $\delta : (MS \times (\Gamma_o \cup \Gamma_c)) \hookrightarrow MS)$ *is a* transition function *that is total in* $(MS \times \Gamma_o)$*.*

For the rest of this subsection, let $mon = (MS, ms_0, \Gamma_o, \Gamma_c, \delta)$ be a monitor, let $\Gamma = \Gamma_o \cup \Gamma_c$, and let $lts = (S, S_0, E, \Delta)$ be a labeled transition system.

We model the effects of supervising a target program by a monitor.

Definition 10. Monitoring *the labeled transition system lts with monitor mon results in the labeled transition system* $MON(lts, mon) = (S', S'_0, E, \Delta')$*, where*

$$\begin{array}{rcl} S' &=& S \times MS \ , \\ S'_0 &=& S_0 \times \{ms_0\} \ , \ \textit{and} \\ ((s,ms), e, (s',ms')) \in \Delta' & \textit{iff} & \begin{pmatrix} (s, e, s') \in \Delta \\ \wedge\ (e \in \Gamma \Rightarrow ms' = \delta(ms, e)) \\ \wedge\ (e \notin \Gamma \Rightarrow ms' = ms) \ . \end{pmatrix} . \end{array}$$

Definition 10 captures the intended operational behavior of monitors: A monitor updates its state whenever events occur that the monitor can observe (i.e., events in $\Gamma_o \cup \Gamma_c$). When non-observable events occur, the monitor does not modify its state. Moreover, a monitored program can perform an event in Γ_c only if the target program and the monitor can make a transition for this event. If the event is in $E \setminus \Gamma_c$, then it suffices that the target can make a transition for this event.[1]

[1] Recall from Definition 9 that a monitor is total in Γ_o.

Before characterizing the effects of monitoring denotationally, let us make the conceptual similarity between monitors and finite automata precise.

Definition 11. $fa_{mon} = (MS, \{ms_0\}, MS, \Gamma, \delta)$ *is the FA induced by mon.*

Note that all monitor states are final states of the finite automaton fa_{mon}. This reflects that the acceptance of an event by a monitor solely depends on whether the transition relation is defined for the current monitor state and this event.

Recall from Section 2 that the expressiveness of finite automata where all states are final is the class of regular languages that are prefix-closed [25].

The denotational effects of monitoring a target by a monitor now can be captured concisely by using the lifting of fa_{mon} to the events of the target:

Theorem 2. $Traces(MON(lts, mon)) = Traces(lts) \cap Lang(fa_{mon}^{\uparrow E})$ *holds.*

Given $pol = (A, Tr)$, one can construct a monitor that soundly enforces this policy via the construction of an FA that accepts a sub-language of Tr.

Theorem 3. *Let* $fa = (MS, \{ms_0\}, MS, \Gamma, \delta)$ *and* $mon = (MS, ms_0, \emptyset, \Gamma, \delta)$. *Then, for every policy* $pol = (A, Tr)$*, the following implication holds:*

$$Lang(fa) \subseteq Tr \text{ implies that } MON(lts, mon) \text{ satisfies } prop_{pol}^{E}.$$

Therefore, we say that *fa enforces pol* if $Lang(fa) \subseteq Tr$ holds. In the theorem, the set Γ_o of the monitor is empty. We will clarify in Section 4 what the benefits are of moving events from Γ_c to Γ_o and in Section 5 when it is safe to do so.

The right-hand side of Fig. 1 visualizes the interplay between the formal representations of target programs, monitors, properties, and policies.

3.3 Watch-dogs and Prevention of Anti-Policies

As explained, we use the terms watch-dog and anti-policy instead of monitor and policy, respectively, when the specification focuses on preventive behavior. We observed that in this case, finite automata are usually used in a slightly different way than described in Section 3.2, including prior work on optimizations of run-time monitoring. As such technical details matter when verifying optimizations, we extend our semantic framework to watch-dogs and clarify the technical differences to monitors. We also present a construction suitable for moving from watch-dogs to monitors while preserving the behavior of a monitored target.

If a watch-dog reaches a final state, then a preventive behavior has occurred. Therefore, a watch-dog must take countermeasures *before* it reaches a final state.

The main differences to the definition of monitors (see Definition 9) are the existence of a set of final states and that the transition function is total.

Definition 12. *A* watch-dog *is a tuple* $wdo = (WS, ws_0, WS_F, \Gamma_o, \Gamma_c, \delta)$ *where*
- WS *is a finite* set of watch-dog states *with* initial state $ws_0 \in WS$,
- $WS_F \subseteq WS$ *is a* set of final watch-dog states *with* $ws_0 \notin WS_F$,
- Γ_o *is a finite* set of observed events,
- Γ_c *is a finite* set of controlled events *with* $\Gamma_o \cap \Gamma_c = \emptyset$, *and*
- $\delta : (WS \times (\Gamma_o \cup \Gamma_c)) \rightarrow WS)$ *is a* transition function *that is total and for which* $\delta(ws, \gamma) \in (WS \setminus WS_F)$ *holds for all* $\gamma \in \Gamma_o$ *and* $ws \in WS$.

For the rest of this subsection, let $wdo = (WS, ws_0, WS_F, \Gamma_o, \Gamma_c, \delta)$ be a watch-dog, let $\Gamma = \Gamma_o \cup \Gamma_c$, and let $lts = (S, S_0, E, \Delta)$ be an LTS.

We model the effects of supervising a target program by a watch-dog.

Definition 13. Monitoring *lts with the watch-dog wdo results in the labeled transition system* $WDO(lts, wdo) = (S', S'_0, E, \Delta')$, *where*

$$S' = S \times WS \ ,$$
$$S'_0 = S_0 \times \{ws_0\} \ , \text{ and}$$
$$((s,ws), e, (s',ws')) \in \Delta' \quad \textit{iff} \quad \begin{pmatrix} (s, e, s') \in \Delta \\ \wedge \ (e \in \Gamma \Rightarrow (ws' = \delta(ws, e) \wedge ws' \notin WS_F)) \\ \wedge \ (e \notin \Gamma \Rightarrow ws' = ws)\} \end{pmatrix} .$$

Note that Definition 13 faithfully captures the intended operational behavior of watch-dogs: A monitored target can perform an event in Γ_c only if the non-monitored target is able to make a transition for this event and if the occurrence of this event would not result in a final watch-dog state. If the event is in $E \backslash \Gamma_c$, then it suffices that the target is able to make a transition for this event.[2]

We identify which finite automaton corresponds to a given watch-dog:

Definition 14. $fa_{wdo} = (WS, \{ws_0\}, WS_F, \Gamma, \delta)$ *is the FA induced by wdo.*

Analogously to Section 3.2, we capture the denotational effects of monitoring a target by a watch-dog using the liftings of fa_{wdo} to the events of the target:

Theorem 4. $Traces(WDO(lts, wdo)) = Traces(lts) \setminus \mathcal{SUF}(Lang(fa_{wdo}^{\uparrow E}))$ *holds.*

Note that, in Theorem 4, the suffix closure of the automaton's language occurs on the right-hand side of the equation. This is a difference to the analogous theorem for monitors (i.e., Theorem 2). Operationally, this is due to the fact that an action of a target program is prevented by a watch-dog if the action would result in the watch-dog reaching a final state. That is, a watch-dog not only prevents all words that it accepts but also all suffixes of such words.

Remark 1. Note that for the finite automaton corresponding to a monitor, every state is final and the transition function may be partial. In contrast, for a finite automaton corresponding to a watch-dog, at least the initial state must not be final, and the transition function must be total. Despite these differences in the underlying classes of automata, one can construct a monitor from a watch-dog while preserving the behavior of all monitored target systems. To this end, one changes the transition function of fa_{wdo} to be undefined for all arguments that would lead to a final state, afterwards one removes all final states from the set of states, and finally, one applies the usual complement constructions on finite automata by making all final states non-final and vice versa.

4 A Performance Model for Monitors

Run-time monitoring and enforcement inherently comes at the cost of some performance overhead. A monitor needs to learn about the action that the target

[2] Recall that a watch-dog cannot reach a final state when events in Γ_o occur.

program is about to perform. The monitor then needs to decide whether to affirm or prevent an action. If an action is affirmed, the target needs to be enabled to perform the action and, otherwise, countermeasures must be taken.

Usually the individual actions of a monitor are rather fast, but such delays accumulate during a run. This is why it has become good practice to accompany the development of tools for run-time monitoring and enforcement with experimental performance evaluations, e.g., at a benchmark like DaCapo [9].

The overhead caused by monitoring depends on how the monitor itself is implemented and also on how the combination with the target program is technically realized. The *inlining-technique* [19] is a popular technique that incorporates the monitor code sequentially into the target's code. *Outlining* places the monitor into a separate process [27] that runs in parallel to the target. The *crosslining technique* [23] combines the two by sequentially inlining parts of the monitor code while outlining other parts to run in parallel.

While experimental performance evaluations of tools for run-time monitoring and enforcement have become common practice, there is little work on analytical performance evaluations. A notable exception is an article by Drábik et al. [15]. They propose a framework for analyzing the costs of enforcement.

In this section, we propose a novel performance model for run-time monitoring and enforcement. Our model is similar in spirit to the one in [15], but we take a broader set of events and context-dependence of costs into account.

We introduce performance vectors to model the time needed by a monitor to observe an action, to check whether it is permissible, to permit an action to happen, and to terminate a run. In addition, a performance vector models any base overhead that is induced by the mere existence of a monitor.

Definition 15. *A* performance vector *for a set of events E is a tuple $\mu = (\mu_o, \mu_c, \mu_p, \mu_t, \mu_\emptyset)$, where $\mu_o, \mu_c, \mu_p, \mu_t, \mu_\emptyset : ((E \times E^*) \to Time)$.*
A performance vector $\mu = (\mu_o, \mu_c, \mu_p, \mu_t, \mu_\emptyset)$ is context independent *iff $\mu_\alpha(e, w) = \mu_\alpha(e, w')$ holds for all $\alpha \in \{o, c, p, t, \emptyset\}$, $e \in E$, and $w, w' \in E^*$.*

Intuitively, $\mu_o(e, tr)$ models the time a monitor needs to learn that the target is about to perform the event $e \in E$. The second parameter, i.e. tr, models the context in which e occurs by the trace of events that have happened before.

Similarly, $\mu_c(e, tr)$ models the time needed to check whether e is permissible in context tr. The functions μ_p and μ_t model the time needed for enabling the target to perform e and for preventing e by terminating the run, respectively. The values of μ_p and μ_t shall include time needed to update the monitor's internal state. The function $\mu_\emptyset$ models the base overhead of monitoring for events that the monitor can neither observe nor control.

We parametrized the functions in a performance vector by both, an event and a context. This design choice allows one to specify the performance overhead very precisely. If determining such a precise performance vector, in practice, is infeasible or too expensive, one can coarsen the model, e.g., by defining a context-independent performance vector that approximates the actual performance costs.

We are now ready to introduce a novel performance model for monitors.

Definition 16. *The* overhead *caused by a monitor* $mon = (MS, ms_0, \Gamma_o, \Gamma_c, \delta)$ *under a performance vector* $\mu = (\mu_o, \mu_c, \mu_p, \mu_t, \mu_\emptyset)$ *for* $e \in E$ *and* $tr \in E^*$ *is*

$$\mu_{mon}(e, tr) = \begin{cases} \mu_\emptyset(e, tr) & , \text{ if } e \in (E \setminus (\Gamma_o \cup \Gamma_c)) \\ \mu_o(e, tr) & , \text{ if } e \in \Gamma_o \\ \mu_o(e, tr) + \mu_c(e, tr) + \mu_p(e, tr) & , \text{ if } e \in \Gamma_c \text{ and } \delta(ms_{tr}, e)\downarrow \\ \mu_o(e, tr) + \mu_c(e, tr) + \mu_t(e, tr) & , \text{ if } e \in \Gamma_c \text{ and } \delta(ms_{tr}, e)\uparrow \end{cases}$$

where $ms_{tr} = \delta^*(ms_0, tr|_{\Gamma_o \cup \Gamma_c})$. *The* overhead *of mon for a trace is defined recursively by* $\mu^*_{mon}(\langle\rangle) = 0$ *and* $\mu^*_{mon}(tr.e) = \mu^*_{mon}(tr) + \mu_{mon}(e, tr)$.

In Section 5, we use this model to characterize the performance gain by selected optimizations, while using the terms in Definition 16 purely symbolically.

Remark 2. The definition of an analogous performance model for watch-dogs is straightforward based on our semantic framework. When instantiating the resulting performance models, however, be aware that differences between monitors and watch-dogs should be taken into account when defining performance vectors. In particular, supervising events in Γ_c might have higher performance costs for watch-dogs than for monitors: While a watch-dog needs to compute the resulting watch-dog state and check that it is not final before allowing the target to continue, a monitor only needs to check whether a transition for the event exists and may update its state in parallel to the target's execution.

5 Towards a more Formal Treatment of Optimizations

Optimizations are crucial for lowering the performance overhead of run-time monitoring and enforcement. However, such optimizations need to be applied with care, because optimizing a monitor could endanger its effectiveness.

In our study of prior work on optimizing run-time monitoring and enforcement, we observed that the arguments for the preservation of properties mostly remain at an informal level. Given the growing importance of optimizations and their increasing level of sophistication, we think the time is ready for more scrutiny. After all, what is the value of formal verifications of run-time monitors, if afterward optimizations are applied that have only been informally analyzed?

The advantages of a more formal treatment of optimizations are twofold:

- precise, formal definitions of preconditions clarify better what one needs to check for before applying a given optimization and
- formally verified preservation results provide reliable guarantees for the preservation of properties under an optimization if the preconditions are met.

One possibility for decreasing performance overhead, is to limit the events tracked in run-time monitoring and enforcement. This optimization technique is popular, and it also appeared as an integral part of more complex optimizations (see Section 6). This is the optimization on which we focus in this section.

Based on our semantic framework, we clarify which preconditions guarantee the preservation of which properties under this optimization. Formally, the optimization corresponds to removing events from the alphabet of the automaton underlying a monitor or watch-dog while restricting the transition function accordingly. However, our framework provides a more fine-grained distinction,

namely between events under the control of a monitor/watch-dog (i.e., Γ_c) and events whose occurrences the monitor/watch-dog can observe but not control (i.e., Γ_o). This allows us to split the optimization into two more primitive ones:

- removal of events from the control while keeping track of them and
- removal of events from the set of events that are tracked.

In our formal model, *reduction of control* (brief: *ROC*) corresponds to moving events from Γ_c to Γ_o, and *reduction of tracking* (brief: *ROT*) corresponds to removing events from Γ_o. Each of these transformations reduces the performance overhead already if applied in isolation. If applied in combination, *ROC* and *ROT* result in the *removal of events from the supervision* (brief: *ROS*). Our split is increasing the application spectrum of such optimizations as there are cases, where *ROC* may be applied, while applying *ROS* would be problematic.

Like in the previous section, we limit our technical exposition to monitors. For the rest of this section, let $mon = (MS, ms_0, \Gamma_o, \Gamma_c, \delta)$ be a monitor.

5.1 Formal Definitions of Optimizations and Performance Gain

We define reduction of control, reduction of tracking, and reduction of supervision as operators that transform monitors. Each of these operators takes an event as second argument. This is the event whose supervision is altered.

Definition 17. *Let* $\gamma_c \in \Gamma_c$, $\gamma_o \in \Gamma_o$, *and* $\gamma \in \Gamma_o \cup \Gamma_c$.

$$ROC(mon, \gamma_c) = (MS, ms_0, (\Gamma_o \cup \{\gamma_c\}), (\Gamma_c \setminus \{\gamma_c\}), \delta\circlearrowleft^{\{\gamma_c\}})$$
$$ROT(mon, \gamma_o) = (MS, ms_0, (\Gamma_o \setminus \{\gamma_o\}), \Gamma_c, \delta|_{(\Gamma_o \cup \Gamma_c) \setminus \{\gamma_o\}})$$
$$ROS(mon, \gamma) = ROT(ROC(mon, \gamma), \gamma)$$

Note that, if $mon = (MS, ms_0, \Gamma_o, \Gamma_c, \delta)$ is a monitor then $\delta\circlearrowleft^{\{\gamma_c\}}$ is total in $(\Gamma_o \cup \{\gamma_c\})$ and $\delta|_{(\Gamma_o \cup \Gamma_c) \setminus \{\gamma_o\}}$ is total in $(\Gamma_o \setminus \{\gamma_o\})$. Therefore, if mon is a monitor then ROC(mon, γ_c) and ROT(mon, γ_o), indeed, are monitors.

In the definition of *ROC*, $\delta\circlearrowleft^{\{\gamma_c\}}$ is a transition function that is total in $\{\gamma_c\}$. The addition of stuttering steps to δ by this completion makes a monitor more affirmative. The removal of Γ_o from the alphabet of a monitor in the definition of *ROT* also makes monitoring more affirmative (cf. Definition 10).

We characterize the effects of the transformations on the monitoring overhead based on our performance model. For simplicity, we assume the monitoring overhead for any given action of the target to depend only on this action.

Theorem 5. *Let* $\mu = (\mu_o, \mu_c, \mu_p, \mu_t, \mu_\emptyset)$ *be a context-independent performance vector, and let* E *be a set of events. The following conditions hold for all* $tr \in E^*$, $\gamma_c \in \Gamma_c$, $roc = ROC(mon, \gamma_c)$, $\gamma_o \in \Gamma_o$, *and* $rot = ROT(mon, \gamma_o)$:

$$\mu_o(\gamma_c) \leq \mu_c(\gamma_c) \implies \mu^*_{roc}(tr) \leq \mu^*_{mon}(tr)$$
$$\mu_\emptyset(\gamma_o) \leq \mu_o(\gamma_o) \implies \mu^*_{rot}(tr) \leq \mu^*_{mon}(tr)$$

On the implementation level, moving γ_c from Γ_c to Γ_o corresponds to reducing the monitor code that runs at each program point where γ_c might occur. The monitor still needs to be informed about such occurrences, but no run-time check is needed, as it is clear a priori that the decision will be positive. Applying ROT corresponds to reducing the number of program points at which monitor code runs. That is, $\mu_o(\gamma_c) \leq \mu_c(\gamma_c)$ and $\mu_\emptyset(\gamma_o) \leq \mu_o(\gamma_o)$ should hold for most monitor implementations, and, hence, ROC and ROT, indeed, are optimizations.

5.2 Application Scenarios for the Optimizations

We point out and informally discuss multiple possibilities for applying the optimizations ROC, ROT, and ROS. The informal presentation in this subsection, will be substantiated by formalizations of conditions and theorems in Section 5.3.

Assume a policy $pol = (A, Tr)$ that specifies the requirements to be enforced, and a monitor constructed by firstly, determining a sublanguage of Tr that is regular and prefix-closed, then synthesizing $fa = (MS, \{ms_0\}, MS, \Gamma, \delta)$ that accepts this sublanguage, and defining the monitor to be $mon = (MS, ms_0, \emptyset, \Gamma, \delta)$. According to Theorem 3, the monitor mon soundly enforces the property induced by pol (i.e., $prop^E_{pol}$) for every labeled transition system $lts = (S, S_0, E, \Delta)$.

At the level of program code, one could check whether the policy's alphabet contains events that cannot be generated by the target program. This check can be realized, e.g., by a syntactic search in the program code for patterns that correspond to these events, or, more sophisticated, by a reachability analysis.

At the level of labeled transition systems, the syntactic check corresponds to checking whether the set $A \setminus E$ is non-empty, and the reachability analysis corresponds to checking, based on Δ, whether any events in A are never enabled. Intuitively, monitoring such events is unnecessary. Hence, one could exempt them from the monitor's supervision by applying ROS for all such events. [A]

A reachability analysis using the monitor's transition function could be used to detect events that are always permitted. For such events the monitor's check is unnecessary, and, hence, one can optimize the monitor by ROC.[3] [B1]

A more sophisticated variant is to detect events the monitor permits in all states that the monitor reaches by observing possible traces of the target. [B2]

Note that optimizations might make other optimizations applicable. For instance, removing an event from a monitor's control by ROS [A] might make, for some other event, all monitor states unreachable in which this event is prevented by the monitor and, hence, ROC could become applicable due to [B1] or [B2].

5.3 Preservation Theorems

The following theorem justifies our uses of ROC in Section 5.2. The three preconditions in the theorem, respectively, correspond to [A], [B1], and [B2].

Theorem 6. *Let $\gamma_c \in \Gamma_c$, $\Gamma = \Gamma_o \cup \Gamma_c$, and $roc = ROC(mon, \gamma_c)$. The equation*

$$Traces(lts) \cap Lang(fa^{\uparrow E}_{roc}) = Traces(lts) \cap Lang(fa^{\uparrow E}_{mon})$$

holds if at least one of the following conditions is satisfied:

1. $\Delta \cap (S \times \{\gamma_c\} \times S) = \emptyset$,
2. $\forall w \in \Gamma^*.\ (\delta^*(ms_0, w))\downarrow \implies (\delta^*(ms_0, w.\gamma_c))\downarrow$, *or*
3. $\forall (tr.\gamma_c) \in Traces(lts).\ (\delta^*(ms_0, tr|_\Gamma))\downarrow \implies (\delta^*(ms_0, (tr|_\Gamma).\gamma_c))\downarrow$.

The following theorem justifies our uses of ROT in Section 5.2. The precondition in the theorem corresponds to [A].

[3] In such a situation, one might be tempted to instead apply the more powerful optimization ROS, but this, in general, does not guarantee the preservation of $prop^E_{pol}$.

Theorem 7. *Let $\gamma_o \in \Gamma_o$, $\Gamma = \Gamma_o \cup \Gamma_c$, and $rot = ROT(mon, \gamma_o)$. The equation*

$$Traces(lts) \cap Lang(fa^{\uparrow E}_{rot}) = Traces(lts) \cap Lang(fa^{\uparrow E}_{mon})$$

holds if $\Delta \cap (S \times \{\gamma_o\} \times S) = \emptyset$.

If the respective preconditions are fulfilled, Theorems 6 and 7 guarantee that ROC and ROT do not alter the intersection of the sets of traces of the non-monitored target with the language of the lifted automaton. In combination with Theorem 2, this guarantees the set of traces of a monitored target to remain unchanged. Thus, *all* properties are preserved if the preconditions are met.

Remark 3. The application spectrum of optimizations could be broadened by taking the policy into account to relax the preconditions of optimizations. Here, we see substantial room for improving optimizations, as it suffices to preserve one property, namely $prop^{E}_{pol}$. This could be a valuable direction for future work.

6 Optimizations for Run-Time Monitoring

We focus on optimizations [11, 10, 34, 30, 18, 5, 35, 22, 3] that aim at sound enforcement. For such optimizations, we develop a taxonomy for RTM optimizations and then classify the existing approaches in our taxonomy.

Most publications on optimizations for run-time monitoring use a more technical and a less formal description than Sections 3 and 5. Although we encountered ambiguities in descriptions of optimizations or monitors and the different representations made it difficult to identify similarities, we follow the optimizing approaches and describe our taxonomy on an informal level, too.

Optimization approaches for RTM get as input a program and a specification. We use the general term specification for the technical input(s) that describe the objective of RTM, e.g., which execution traces are allowed and how to deal with forbidden traces. The shape of the specification differs among the RTM approaches. For example, one can use a finite state automaton to describe the forbidden execution traces and specify that forbidden traces are truncated.

The first step of all optimization approaches for RTM is to gather information about the program with respect to the specification. This information is then used for optimization, which tackles the program's structure, its instrumentation, or the specification. Although the general workflow is the same, approaches for optimizing RTM differ a lot, as we have seen for ROC and ROT.

6.1 A Taxonomy of Optimizations for RTM

Figure 2 shows the taxonomy we propose to classify optimizations for RTM. Elements shown in blue do not occur in any of the reviewed optimizations, but we think they are natural extensions. Our taxonomy uses three main dimensions to characterize optimizations: the specification, information gathering, and optimizing transformations. The latter two specify the two steps in the workflow. A specification in run-time monitoring defines which enforcement is applied to which execution trace. Next, we discuss the dimensions in more detail.

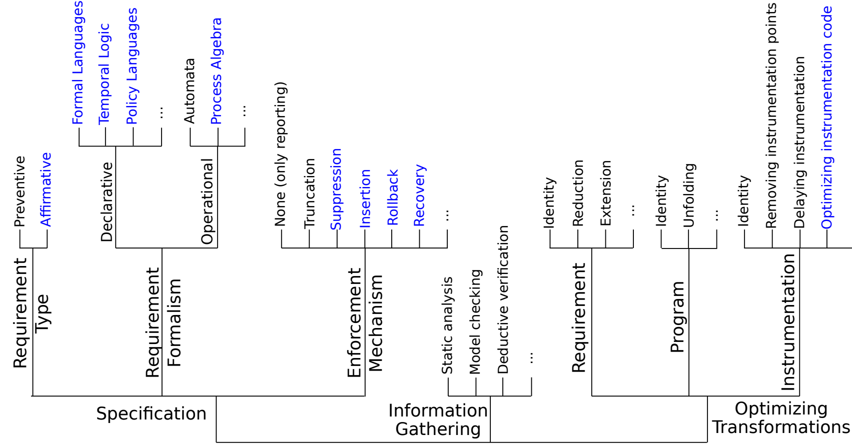

Fig. 2. A taxonomy for optimizing run-time monitoring

Specification The requirement type and formalism describe the requirement. The requirement type determines whether the requirement specifies the allowed behavior (affirmative) or the forbidden behavior (preventive). This corresponds to the two approaches distinguished in Section 3. The formalism fixes the requirement language. We distinguish between declarative and operational formalisms. Commonly, RTM uses both [33, 29]: a declarative requirement to describe the intended behavior and an operational requirement to implement the monitoring. Optimizations rarely consider both, but focus on operational requirements.

Formal languages [7], temporal logic [28] or dedicated policy languages like policy and anti-policy are possible declarative formalisms for requirements. Process algebras [37] and automata [33] are options for operational formalisms. All optimizations we know optimize RTM approaches that (1) are preventive and (2) express requirements in automata formalisms, which range from simple finite state automaton to more advanced automata in DATE [13] and ppDATE [3]. This motivated us to include watch-dogs into our semantic framework. The third element of a specification is the enforcement mechanism. Some optimizations [35, 22] specify that they optimize an RTM approach that uses truncation[4]. However, for most of the approaches it remains unclear what enforcement mechanism is used by the underling RTM approach. Countermeasures suggested by StaRVOOS [3] and Clara [11] are for example error reporting and error recovery. Further enforcement mechanisms [7] used in RTM are rollback and suppression or insertion of events. Since the information gathering phase of all reviewed optimizations does not consider enforcement, it is likely that enforcement beyond truncation is not fully compatible with those optimizations.

Information Gathering All approaches we reviewed statically inspect the program to gather information. StaRVOOS [3] uses deductive verification. Often, reachability analyses like model checking [35, 22] or more efficient dataflow analyses [10] are applied. Also, syntactic, flow-insensitive analyses are run [11].

[4] Recall that we also focused on truncation in Sections 3-5.

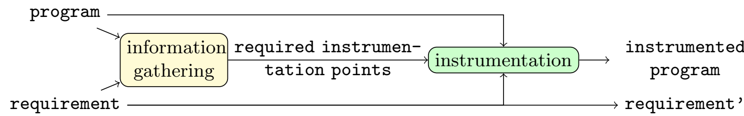

Fig. 3. Workflow of removal of instrumentation points

Optimizing Transformations Reviewing approaches that optimize RTM, we identified three basic concepts for optimizations: transformations of the operational requirement description, the program, or the instrumentation of the program. Since some optimizing approaches apply more than one concept, we describe an optimizing transformation by a triple of the three concepts and use the identity transformation for concepts that are not applied. Thus, unoptimized RTM is identical to only using identity transformations. The three transformations presented in Section 5.1 reduce the operational requirement, i.e., they (re)move events. Requirement reduction can remove transitions and conditions, too. In addition, operational requirements can be extended, e.g., by adding transitions and events. Both, reduction and extension, occur stand-alone [11, 3, 5] like in Section 5 or in combination with the removal of instrumentation points [6], a transformation of the instrumentation code. The program transformation unfolding is typically combined with the removal of instrumentation points [35, 30]. However, the removal of instrumentation points can also be used stand-alone [10, 22]. In contrast, delayed instrumentation [18] is currently only used in combination with requirement extension because it replaces sets of instrumentation points by a later instrumentation that aggregates their effect. We are not aware of any approach that optimizes the instrumentation code, e.g., specializes the instrumentation code at instrumentation points using partial evaluation [24], which we think is an interesting direction.

Our taxonomy excludes optimizations for multiple policies [31], online optimizations [17, 36], and unsound optimizations [12, 21, 16] that may miss violations.

6.2 Classifying Optimizations into Optimizing Transformations

In the following, we present six classes for the taxonomy dimension optimizing transformation. All classes occur in the literature and are future candidates for formalizing and proving optimizations for RTM. We start with three classes that only take one of the optimization concepts into account. Thereafter, we discuss three classes that combine different optimizing transformations.

Stand-alone Removal of Instrumentation Points This class of optimizations keeps the requirement and program as they are and decreases the number of instrumentation points. The left-hand side of Fig 3 shows the workflow. It first determines the required instrumentation points. The instrumentation includes the optimization and leaves out instrumentation points that are not required.

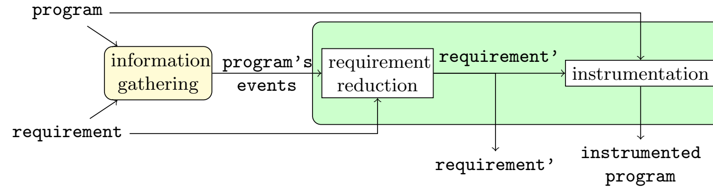

Fig. 4. Workflow of stand-alone requirement reduction

Nop-shadow analysis [10] and its optimization [34] employ forward and backward, flow-sensitive analyses to detect the instrumentation points (called shadows) that are irrelevant. JAM [22] uses model checking and counterexample-guided abstraction refinement with a limited number of refinement steps. The model checker outputs all traces that violate the requirement and are either real or could not be ruled out. The instrumentation step adds monitoring code for the reported traces and stops the program just before a violation would occur.

Stand-alone Requirement Reduction Stand-alone requirement reduction describes optimizations that neither change the program nor the instrumentation procedure, which is unaware of the optimization and uses the reduced requirement. Based on the gathered information, these optimizations (re)move elements from automata (the operational requirement formalism).

The most inexpensive reduction (see right-hand side of Fig. 4) in the literature applies the ROS transformation in scenario A (Section 5.2). It is used in Clara's QuickCheck [11] to reduce finite state automata and in absent event pruning [5] to reduce requirements written in DATE. Additionally, QuickCheck removes transitions that cannot reach one of the automaton's final states.

Clara's orphan-shadow analysis [11] and object-specific absent event pruning [5] extend this reduction idea with object sensitivity. Since type state requirements track the automaton state per object of interest, used events are detected per object and a requirement reduction (an ROS transformation) is performed for each object. Afterward, the reduced requirements are combined again.

Unusable transition pruning [5] integrates flow-sensitive information to the reduction. It takes the control-flow into account and removes automaton transitions that are not activated by any (syntactic) trace of the program.

In its first optimization step, StarVOORS [2] deletes all pre- and postcondition pairs from its ppDATE requirement that are proven by the verifier KeY [1].

Stand-alone Requirement Extension Stand-alone requirement extension is similar to stand-alone requirement reduction. The only difference is that the gathered information is used to adapt the automaton (operational requirement) by generalizing existing or adding new elements. The only approach in this class we are aware of is the second step of the StaRVOORS [3, 2] approach. To reduce the overhead caused by checking the pre- and postconditions in the ppDATE requirement, StaRVOORS tries to discharge the pre- and postcondition pairs with the deductive verifier KeY [1]. While complete proofs are used for reduction, open proof goals are used to split the precondition into a proven and non-proven

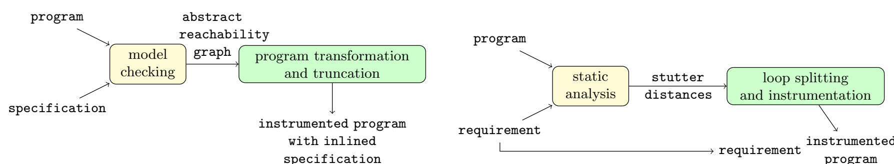

Fig. 5. Zero overhead RTM (lhs) and stutter-equivalent loop optimization (rhs)

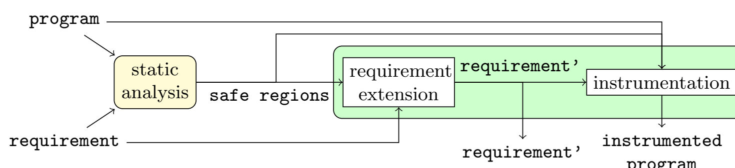

Fig. 6. Workflow of safe regions approach

part and the pre-/postcondition pair is replaced by a refined pair consisting of the non-proven precondition and the unmodified postcondition.

Combining Unfolding with Removal of Instrumentation Points Optimizations in this class do not modify the requirement, but unfold the program to enable the removal of instrumentation points.

Zero overhead run-time monitoring [35], shown on the left-hand side of Fig. 5, starts to model check the program with respect to the requirement. The model checking algorithm parallely executes the requirement and a predicate abstraction (with a limited number of refinements). The result of model checking is an abstract reachability graph (ARG) that represents the explored abstract state space. Due to the combination of the requirement and predicate abstraction, the ARG unfolds the program such that all paths leading to a final state of the requirement automaton are syntactic error traces. For optimization, the ARG is translated back into a program and statements leading to a final state in the ARG, i.e., causing a requirement violation, are replaced by HALT statements. The result is a transformed program with inlined enforcement, in which all instrumentation code except for the enforcement itself is delete.

Stutter-equivalent loop optimization [30], shown on the right-hand side of Fig. 5, splits loops into two loops. The first loop runs as long as RTM may change the requirement state. The second loop is not instrumented and executes the remaining loop iterations, for which the requirement state remains stable. A static analysis is used to determine for each loop the maximal number of loop iterations, the stutter distance, required to reach a stable requirement state. The stutter distance is used to restrict the execution of the first loop.

Delayed Instrumentation with Requirement Extension In this class, optimizations do not transform the program, but extend the operational requirement and replace groups of instrumentation points by a summary instrumentation point. The only approach in this category we are aware of is the safe regions approach [18]. Its workflow is shown in Fig. 6. The safe regions approach starts with a static analysis to detect safe regions, i.e., code blocks that cannot reach a

requirement violation and all paths through the code block that start in the same requirement state end in the same requirement state. Instrumentation does not insert instrumentation code into safe regions, but adds instrumentation code after a safe region, which triggers a special region event. The requirement is extended with transitions for the region events, which summarize the regions' behavior with respect to the requirement.

Combining Requirement Reduction with the Removal of Instrumentation Points Optimization approaches in this class reduce the operational requirement and remove unnecessary instrumentation points, but do not change the program. Clara [11], which combines QuickCheck, the orphan and nop-shadows analysis, and CLARVA [6], which combines the DATE requirement reductions [5] with the removal of instrumentation points, fall into this category.

On Correctness of Optimizations Full correctness of an optimization is rarely shown. Exceptions are zero overhead RTM [35] and the DATE reductions [5]. For Clara's nop shadow [10, 11], the stutter-equivalent loop optimization [30], and the safe regions approach [18] only the underlying idea for the optimization is proven. The correctness of Clara's QuickCheck and orphan-shadow analysis [11] is discussed rather informally. The StarVOORS approach [2] provides the foundations and proves the soundness of the ppDate translation, but lacks to prove the correctness of the ppDate optimization. At worst, for the JAM approach [22] and the improvement of the nop shadow approach [34] correctness of the optimization is hardly discussed at all.

7 Conclusion

We presented a semantic framework for formalizing different approaches to RTM in a uniform fashion and a novel performance model. Their intended application domain is the analysis of optimizations of RTM. We showed at selected optimizations that both are suitable for this purpose. In fact, the formalization of optimizations alone already inspired ideas for broadening the application spectrum of known optimizations and for the development of novel optimizations.

Our taxonomy and classification provide a broader conceptual clarification about previously proposed optimizations. Since the taxonomy also covers possibilities not yet explored, it could serve as a road map for future directions.

Naturally, this article can only be one step towards making a more rigorous treatment of optimizations common practice in the field of RTM. Broadening the application spectrum of optimizations, improving their effects, and clarifying which optimizations can be safely applied under which conditions is an exciting research area that deserves and will require substantial future research.

Acknowledgments. We thank Barbara Sprick for helpful discussions. This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS 10001 (2016)
2. Ahrendt, W., Chimento, J.M., Pace, G.J., Schneider, G.: Verifying Data- and Control-Oriented Properties Combining Static and Runtime Verification: Theory and Tools. Formal Methods in System Design **51**(1), 200–265 (2017)
3. Ahrendt, W., Pace, G.J., Schneider, G.: A Unified Approach for Static and Runtime Verification: Framework and Applications. In: Leveraging Applications of Formal Methods, Verification and Validation. pp. 312–326. LNCS 7609 (2012)
4. Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters **21**, 181–185 (1985), North-Holland
5. Azzopardi, S., Colombo, C., Pace, G.J.: Control-Flow Residual Analysis for Symbolic Automata. In: Pre- and Post-Deployment Verification Techniques. EPTCS, vol. 254, pp. 29–43 (2017)
6. Azzopardi, S., Colombo, C., Pace, G.J.: CLARVA: Model-based Residual Verification of Java Programs. In: Model-Driven Engineering and Software Development. pp. 352–359 (2020)
7. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics. LNCS 10457, Springer (2018)
8. Basin, D.A., Jugé, V., Klaedtke, F., Zalinescu, E.: Enforceable Security Policies Revisited. Transactions on Information and System Security **16**(1), 3:1–3:26 (2013)
9. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: Object-Oriented Programming, Systems, Languages, and Applications. pp. 169–190 (2006)
10. Bodden, E.: Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent states. In: International Conference on Software Engineering. pp. 5–14 (2010)
11. Bodden, E., Hendren, L.J.: The Clara Framework for Hybrid Typestate Analysis. Journal on Software Tools for Technology Transfer **14**(3), 307–326 (2012)
12. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative Runtime Verification with Tracematches. In: Runtime Verification. pp. 22–37. LNCS 4839 (2007)
13. Colombo, C., Pace, G.J., Schneider, G.: Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In: Formal Methods for Industrial Critical Systems. pp. 135–149. LNCS 5596 (2008)
14. Delgado, N., Gates, A.Q., Roach, S.: A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. Transactions on Software Engineering **30**(12), 859–872 (2004)
15. Drábik, P., Martinelli, F., Morisset, C.: Cost-Aware Runtime Enforcement of Security Policies. In: Security and Trust Management. pp. 1–16. LNCS 7783 (2012)
16. Dwyer, M.B., Diep, M., Elbaum, S.G.: Reducing the Cost of Path Property Monitoring Through Sampling. In: Automated Software Engineering. pp. 228–237 (2008)
17. Dwyer, M.B., Kinneer, A., Elbaum, S.G.: Adaptive Online Program Analysis. In: International Conference on Software Engineering. pp. 220–229 (2007)

18. Dwyer, M.B., Purandare, R.: Residual Dynamic Typestate Analysis Exploiting Static Analysis: Results to Reformulate and Reduce the Cost of Dynamic Analysis. In: Automated Software Engineering. pp. 124–133 (2007)
19. Erlingsson, U., Schneider, F.B.: SASI Enforcement of Security Policies: A Retrospective. In: New Security Paradigms. pp. 87–95 (1999)
20. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A Taxonomy for Classifying Runtime Verification Tools. In: Runtime Verification. pp. 241–262. LNCS 11237 (2018)
21. Fei, L., Midkiff, S.P.: Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies. In: Programming Language Design and Implementation. pp. 84–95 (2006)
22. Fredrikson, M., Joiner, R., Jha, S., Reps, T.W., Porras, P.A., Saïdi, H., Yegneswaran, V.: Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement. In: Computer Aided Verification. pp. 548–563. LNCS 7358 (2012)
23. Gay, R., Hu, J., Mantel, H.: CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement. In: Information Systems Security. pp. 378–398. LNCS 8880 (2014)
24. Jones, N.D.: An Introduction to Partial Evaluation. ACM Computing Surveys **28**(3), 480–503 (1996)
25. Kao, J., Rampersad, N., Shallit, J.O.: On NFAs Where All States are Final, Initial, or Both. Theoretical Computer Science **410**(47–49), 5010–5021 (2009)
26. Kleene, S.C.: Representation of Events in Nerve Nets and Finite Automata. In: Automata Studies, pp. 3–41 (1956)
27. Leucker, M.: Teaching Runtime Verification. In: Runtime Verification. pp. 34–48. LNCS 7186 (2011)
28. Leucker, M., Schallhart, C.: A Brief Account of Runtime Verification. Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009)
29. Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. Journal of Information Security **4**(1-2), 2–16 (2005)
30. Purandare, R., Dwyer, M.B., Elbaum, S.G.: Monitor Optimization via Stutter-Equivalent Loop Transformation. In: Object-Oriented Programming, Systems, Languages, and Applications. pp. 270–285 (2010)
31. Purandare, R., Dwyer, M.B., Elbaum, S.G.: Optimizing Monitoring of Finite State Properties through Monitor Compaction. In: Software Testing and Analysis. pp. 280–290 (2013)
32. Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P.: A Comparison Framework for Runtime Monitoring Approaches. Journal of Systems and Software **125**, 309–321 (2017)
33. Schneider, F.B.: Enforceable Security Policies. Transactions on Information and System Security **3**(1), 30–50 (2000)
34. Wang, C., Chen, Z., Mao, X.: Optimizing Nop-shadows Typestate Analysis by Filtering Interferential Configurations. In: Runtime Verification. pp. 269–284. LNCS 8174 (2013)
35. Wonisch, D., Schremmer, A., Wehrheim, H.: Zero Overhead Runtime Monitoring. In: Software Engineering and Formal Methods. pp. 244–258. LNCS+8137 (2013)
36. Wu, C.W.W., Kumar, D., Bonakdarpour, B., Fischmeister, S.: Reducing Monitoring Overhead by Integrating Event- and Time-Triggered Techniques. In: Runtime Verification. pp. 304–321. LNCS 8174 (2013)
37. Yamagata, Y., Artho, C., Hagiya, M., Inoue, J., Ma, L., Tanabe, Y., Yamamoto, M.: Runtime Monitoring for Concurrent Systems. In: Runtime Verification. pp. 386–403. LNCS 10012 (2016)