On Implementing Symbolic Controllability *

Adrian Francalanza[1] and Jasmine Xuereb[1]

University of Malta, Malta
{adrian.francalanza,jasmine.xuereb.15}@um.edu.mt

Abstract. Runtime Monitors observe the execution of a system with the aim of reaching a verdict about it. One property that is expected of monitors is consistent verdict detections; this property was characterised in prior work via a symbolic analysis called symbolic controllability. This paper explores whether the proposed symbolic analysis lends itself well to the construction of a tool that checks monitors for this deterministic behaviour. We implement a prototype that automates this symbolic analysis, and establish complexity upper bounds for the algorithm used. We also consider a number of optimisations for the implemented prototype, and assess the potential gains against benchmark monitors.

Keywords: Deterministic Monitors · Symbolic Analysis · Runtime Verification

1 Introduction

Monitors are computational entities that are *instrumented* to execute alongside a *program* of interest. This paper focusses on a specific class of monitors called execution monitors [34], also termed sequence recognisers [25] or partial-identity monitors [21]. Execution monitors observe a *sequence of events* exhibited by the running program with the aim of reaching an *irrevocable verdict*. Conceptually, these monitors may be described as suffix-closed sets of traces of events that lead to the respective verdicts [36,11,5]. Operationally, however, they are best conceived as a branching structure whereby a sequence of events may lead a monitor to reach *a number of possible states* [22,20,16,2]. This better captures the potential monitor behaviour in concurrent/distributed settings [26,15,7,9,23,8,19], or the behaviour encountered in practical implementations that may occasionally (and unexpectedly) operate erratically [12,32,13]. Put differently, monitors themselves may, either by necessity or inadvertently, behave *non-deterministically*.

In spite of this potential behaviour, deterministic monitor operation for the verdicts reached is still a desirable quality and is often a prerequisite for monitor correctness [6]. In prior work [17, Def. 6], we proposed an observational definition for a *consistently-detecting* monitor. Intuitively, fixed a trace exhibited by the program it is instrumented with, such a monitor is required to always

* This research was supported by project BehAPI, funded by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant (No:778233).

reach the *same* verdict for *that* trace. Crucially, consistent detection allows such a monitor to pass through *different intermediate states* during the course of its verdict-reaching trace analysis (since these states are not observable from a consistently-detecting sense). An alternative characterisation called *monitor controllability* [17, Def. 11] is also proposed in this work, with the aim of providing a more tractable method for assessing deterministic monitor behaviour. This characterisation improves on reasoning about monitor consistent detection in two ways: (*i*) it *avoids universal quantifications* over the programs that a monitor can be instrumented with (*i.e.,* contexts); (*ii*) it is *coinductive*, permitting reasoning about an infinite number of traces in a finite manner (for certain monitor cases). Monitor controllability is also shown to be both sound and complete w.r.t. consistent detection.

There is one further complication when reasoning about monitor behaviour. In most practical settings, events carry a *payload* from some infinite data domain. A refinement to the coinductive definition, called *symbolic (monitor) controllability*, is thus developed in [17] to assist with abstracting over universal quantifications on payload data and data-dependent monitor states. This work also claims that the resulting *symbolic analysis* mandated by the new definition lends itself well to the construction of a tool that analyses monitors for their capacity to perform deterministic detections. The goal of our paper is to verify this claim. The contributions are twofold:

1. In Sec. 3, we build a prototype that automates the analysis for symbolic controllability, demonstrating the *implementability* of the approach proposed in [17]; we also provide *complexity bounds* for the algorithm implemented.
2. In Sec. 4, we identify implementation bottlenecks that limit the *scalability* of the tool in practice. Subsequently, in Sec. 5, we empirically evaluate a number of proposed solutions using a series of pathological monitor descriptions devised in Sec. 4.

2 Preliminaries

We assume the existence of an expression language, $e, d \in \textsc{Exp}$ and a boolean expression language $b, c \in \textsc{BExp}$. Expressions are defined in terms of a denumerable set of *expression variables*, $x, y \in \textsc{Vars}$, and a value domain, $v, u \in \textsc{Val}$; for expository purposes, we assume the value domain to be infinite. Boolean expressions are defined over the expression language Exp, and include the standard constructs for the basic values true and false, conjunctions $b \wedge c$, expression equality $e{=}d$, and negation $\neg b$. The meta-function $\mathbf{fv}(e)$ and $\mathbf{fv}(b)$ computes the *free variables* in the respective expressions e and b. (Boolean) expressions *without* any free variables are said to be *closed*, and *open* otherwise. *Substitutions*, denoted by $[\vec{e}/\vec{x}]$, are partial maps from Vars to Exp, with the term $d[\vec{e}/\vec{x}]$ signifying that every *free* occurrence of $x_i \in \vec{x}$ in d is substituted by the corresponding expression $e_i \in \vec{e}$. As is standard, open terms are interpreted over *valuations*, $\rho \in \textsc{Vars} \to \textsc{Val}$, *i.e.,* complete maps instantiating free variables to concrete

Monitors

$$
\begin{array}{rcll}
w, o \in \textsc{Verd} & ::= & \top \quad \text{(accept)} & \mid \bot \quad \text{(reject)} \\
& \mid & \mathbf{0} \quad \text{(inconclusive)} & \\
m, n \in \textsc{Mon} & ::= & w \quad \text{(verdict)} & \mid \mathsf{let}\ x = e\ \mathsf{in}\ m \quad \text{(evaluate)} \\
& \mid & l\langle e \rangle.m \quad \text{(expression guard)} & \mid l(x).m \quad \text{(quantified guard)} \\
& \mid & m + n \quad \text{(choice)} & \mid \mathsf{if}\ b\ \mathsf{then}\ m\ \mathsf{else}\ n \quad \text{(conditional)} \\
& \mid & \mathsf{rec}\, X.m \quad \text{(recursion)} & \mid X \quad \text{(monitor variable)}
\end{array}
$$

Symbolic Transitions

$$\textsc{sVer}\ \frac{}{w \xrightarrow[\mathsf{true}]{\theta} w} \qquad \textsc{sIfT}\ \frac{}{\mathsf{if}\ b\ \mathsf{then}\ m\ \mathsf{else}\ n \xrightarrow[b]{\tau} m} \qquad \textsc{sIfF}\ \frac{}{\mathsf{if}\ b\ \mathsf{then}\ m\ \mathsf{else}\ n \xrightarrow[\neg b]{\tau} n} \qquad \textsc{sRec}\ \frac{}{\mathsf{rec}\, X.m \xrightarrow[\mathsf{true}]{\tau} m[\mathsf{rec}\, X.m/X]}$$

$$\textsc{sChL}\ \frac{m \xrightarrow[b]{\mu} m'}{m + n \xrightarrow[b]{\mu} m'} \qquad \textsc{sGrE}\ \frac{}{l\langle e \rangle.m \xrightarrow[e=x]{l\langle x \rangle} m} \qquad \textsc{sGrQ}\ \frac{}{l(y).m \xrightarrow[\mathsf{true}]{l\langle x \rangle} m[x/y]} \qquad \textsc{sLet}\ \frac{}{\mathsf{let}\ x = e\ \mathsf{in}\ m \xrightarrow[\mathsf{true}]{\tau} m[e/x]}$$

Weak Symbolic Transitions and Reductions

$$\textsc{sWTr1}\ \frac{m \xrightarrow[b]{\theta} m'}{m \overset{\theta}{\underset{b}{\Longrightarrow}} m'} \qquad \textsc{sWTr2}\ \frac{m \xrightarrow[b]{\tau} m' \qquad m' \overset{\theta}{\underset{c}{\Longrightarrow}} m''}{m \overset{\theta}{\underset{b \wedge c}{\Longrightarrow}} m''} \qquad \textsc{sWRd1}\ \frac{}{m \underset{\mathsf{true}}{\Longrightarrow} m} \qquad \textsc{sWTr2}\ \frac{m \xrightarrow[b]{\tau} m' \qquad m' \underset{c}{\Longrightarrow} m''}{m \underset{b \wedge c}{\Longrightarrow} m''}$$

Fig. 1: A Symbolic Semantics for Monitors

values. (Open) expressions and boolean expressions come equipped with partial evaluation functions taking a valuation and returning the respective values, $[\![e\rho]\!] = v$ and $[\![b\rho]\!] \in \{\mathsf{true}, \mathsf{false}\}$; the terms $e\rho$ and $b\rho$ denote the instantiation of the free variables in e and b respectively by the corresponding values mapped to in ρ. A boolean expression b is *satisfiable* if there exists some valuation ρ that maps b to true, *i.e.*, $\mathbf{sat}(b) = \exists \rho.[\![b\rho]\!] = \mathsf{true}$.

Programs are seen as entities that generate *events* of the form $l\langle v \rangle$ where $l, k \in \textsc{Lab}$ is the *event label* and v is the *payload* from the value domain. A sequence of events, *i.e.*, a *trace*, thus represents a program execution that is analysed by the instrumented monitor.[1] For our study, monitors are modelled as Labelled Transition Systems (LTSs), described by the syntax in Fig. 1. They consist of two *conclusive verdicts* (namely *acceptance*, $\top$, and *rejection*, $\bot$) and an *inconclusive verdict*, $\mathbf{0}$, to describe the state a monitor transitions to when it

[1] Operationally, a program p is instrumented with a monitor m as $m \triangleleft p$ where p drives the execution and m passively reacts by analysing observable events generated by p [16,18,4]. In the case of controllability, the results in [17] show how this instrumentation can be abstracted as a monitor reacting to an event trace.

is asked to analyse an event it is not expecting. The syntax defines two guards describing event analysis. Expression guards, $l\langle e\rangle.m$, require the monitor to first analyse an event $l\langle v\rangle$ where the payload v is equal to (the evaluation of) the expression e, and then to proceed as the continuation m. Quantified guards, $l(x).m$, require the monitor to dynamically learn the payload v from an analysed event $l\langle v\rangle$ with a matching label l, and then bind the learnt payload value v to the variable x in the continuation m; we use the suggestive notation $l(\_).m$ when the binding variable is not used in m. The remaining constructs are standard.

Example 1. A program operating a thermostat is initialised to a starting temperature i via the event $\mathsf{init}\langle i\rangle$ ($i \in \mathbb{N}$). After this, it can either terminate reporting $\mathsf{end}\langle j\rangle$ with the error code $j \in \mathbb{N}$, or repeatedly read the current temperature value i, $\mathsf{get}\langle i\rangle$, and adjust the temperature i, $\mathsf{set}\langle i\rangle$, for some value i.

$$
\begin{aligned}
m_1 = {} & \mathsf{init}\langle 0\rangle.\mathsf{end}(\_).\bot \\
m_2 = {} & \mathsf{init}\langle 50\rangle.\mathsf{rec}\, X.\mathsf{get}(y).\mathsf{if}\ y{>}50\ \mathsf{then}\ \mathsf{set}(\_).\bot\ \mathsf{else}\ \mathsf{set}\langle y+1\rangle.X \\
m_3 = {} & \mathsf{init}(x).\mathsf{let}\ \mathit{lim} = e_{\mathit{calc}}\ \mathsf{in}\ \big(\mathsf{if}\ x{<}\mathit{lim}\,\mathsf{then}\,\mathsf{end}(\_).\bot\ \mathsf{else} \\
& \qquad\qquad \mathsf{rec}\, X.\mathsf{get}(y).\mathsf{if}\ y \geq \mathit{lim}\,\mathsf{then}\,\mathsf{set}\langle y+1\rangle.\top\ \mathsf{else}\ \mathsf{set}(\_).X\big)
\end{aligned}
$$

Monitors m_1, m_2 and m_3 check for three different specifications. Monitor m_1 rejects executions that terminate after the thermostat is initialised to 0. When the thermostat is initialised to 50, monitor m_2 repeatedly checks that it is *not* set if the temperature read is greater than the initialisation value. Monitor m_3 checks whether the initialised value (learnt at runtime) is less than some predetermined value calculated via some complex calculation e_{calc}: if so, it rejects terminations and accepts executions where the thermostat is set to the temperature just read increased by one, where the former is higher than the predetermined value. The monitor instrumentation assumed (used extensively in other settings [16,18,1,4]) preempts the monitor execution to the inconclusive state, $\mathbf{0}$, whenever the monitor is presented with an event this not specified by its description. For instance, if the monitor m_1 is presented with the event $\mathsf{set}\langle 42\rangle$ (or event $\mathsf{init}\langle 42\rangle$ for that matter), the instrumentation aborts the runtime analysis by reducing the monitor to $\mathbf{0}$. ■

Following [17], the monitor semantics is given in Fig. 1 in terms of a *symbolic* LTS $\langle \textsc{Mon}, \textsc{BExp}, \textsc{Act}, \longrightarrow\rangle$ where $\textsc{Act}$ is a set containing symbol actions, $\theta \in \textsc{SEvt}$, and the silent action $\tau \notin \textsc{SEvt}$. Symbolic actions, $l\langle x\rangle$, *abstract* over concrete trace events by carrying variables x instead of values; we let $\mu \in \textsc{SEvt} \cup \{\tau\}$. The transition relation $\longrightarrow \subseteq (\textsc{Mon} \times \textsc{Act} \times \textsc{BExp} \times \textsc{Mon})$ is denoted as $m \xrightarrow[b]{\mu} n$; it models the transition from a monitor state m to a new monitor state n via the symbolic action μ where the predicate b constrains the free variables in the action μ and the monitor states m and n. It is defined as the least relation satisfying the rules in Fig. 1 (we elide the symmetric rule $\textsc{sChR}$). Rule $\textsc{sVer}$ states that a verdict w can analyse any symbolic action θ under any circumstance, *i.e.*, true, and transition to itself, modelling *verdict irrevocability*.

A conditional monitor $\mathsf{if}\ b\ \mathsf{then}\ m\ \mathsf{else}\ n$ can (silently) τ-transition to either m under the pretext that b holds (rule sIfT), or to n if the converse, $\neg b$, holds (rule sIfF). The other key rules are sGrE and sGrQ for expression and quantified guards respectively: whereas the latter rule transitions with the symbolic event $l\langle x\rangle$ without constraining x, the former rule requires that x is equivalent to the guard expression e, *i.e.*, $e = x$. The remaining rules are fairly straightforward; see [17] for details. Fig. 1 also defines derivation rules for *weak symbolic transitions* (without trailing silent actions), $m \overset{\theta}{\underset{b}{\Longrightarrow\!\!\!\!\!\!\rightarrow}}\ n$, and *reductions*, $m \underset{b}{\Longrightarrow} n$. The predicate $m \overset{\theta}{\underset{b}{\Longrightarrow\!\!\!\!\!\!\rightarrow}}$ is used as a shorthand notation for the requirement $\exists n \cdot m \overset{\theta}{\underset{b}{\Longrightarrow\!\!\!\!\!\!\rightarrow}}\ n$.

The symbolic transitions in Fig. 1 are defined over general terms that are potentially open. They are used to abstract over concrete transitions—defined over closed monitor terms—in our symbolic analysis. The pair $\langle b, m\rangle$ is used to represent the set of concrete terms $\{\, m\rho \mid [\![b\rho]\!] = \mathsf{true}\,\}$. Typically, a symbolic analysis starts off from a concrete term m, denoted by the pair $\langle \mathsf{true}, m\rangle$ where m is closed. General constraining conditions b in a pair $\langle b, m\rangle$ are accrued from prior transitions as follows. The symbolic transition relation $m \overset{\mu}{\underset{c}{\longrightarrow}}\ n$ is used to abstractly calculate the set of concrete transitions $m\rho \xrightarrow{\mu\rho} n\rho$ from the pair $\langle b, m\rangle$ for any ρ satisfying b, *i.e.*, $[\![b\rho]\!] = \mathsf{true}$, *whenever* ρ *also* satisfies c. In order to record this fact, the resulting set of monitor states is encoded as $\langle b \wedge c, n\rangle$.

Our symbolic analysis rests on another important technical machinery. Since it is concerned with abstracting over internal non-determinism (as long as it does not manifest itself in terms of the verdicts reached) we need to (symbolically) reason with respect to *sets* of (open) monitor terms, $M \subseteq \textsc{Mon}$, denoting the set of possible monitor states that we could have reached thus far. Concretely, the symbolic analysis works on *constrained monitor-sets*, $\langle b, M\rangle$, where the boolean condition b constrains the free variables present in *every* monitor $m \in M$, *i.e.*, $[\![\langle b, M\rangle]\!] \overset{\mathbf{def}}{=} \{\, m\rho \mid m \in M \text{ and } [\![b\rho]\!] = \mathsf{true}\,\}$. The meta-function $\mathbf{fv}(-)$ is lifted to constrained monitor-sets in the obvious manner *i.e.*, $\mathbf{fv}(\langle b, \{m_1, \ldots, m_n\}\rangle) = \mathbf{fv}(b) \cup \mathbf{fv}(m_1) \cup \ldots \cup \mathbf{fv}(m_n)$. In the sequel, we also use the notation $\wedge B$ for some set of boolean conditions $B = \{c_1, \ldots, c_n\}$ to denote the syntactic conjunction of all the conditions in B. The helper function $\mathbf{frsh}(V)$ is also used to generate the next fresh variable x which is *not* in the variable set $V \subseteq \textsc{Vars}$.

Symbolic controllability employs two predicates on constrained monitor-sets. The predicate $\mathbf{spr}\big(\langle b, M\rangle, w\big)$ holds if some monitor $m \in M$ that can symbolically reach a verdict after a finite sequence of silent actions along some condition c where $b \wedge c$ is satisfiable. The predicate $\mathbf{spa}\big(\langle b, M\rangle, \theta, c\big)$ holds if some monitor $m \in M$ can weakly analyse the event θ with condition c with a satisfiable $b \wedge c$.

Definition 1 (Symbolic Predicates [17]). *A constrained monitor-set* $\langle b, M\rangle$

1. symbolically potentially reaches *a verdict* w*, denoted as* $\boldsymbol{spr}\big(\langle b, M\rangle, w\big)$*, whenever* $\exists c \in \textsc{BExp}, \exists m \in M$ *such that* $m \underset{c}{\Longrightarrow} w$ *and* $\boldsymbol{sat}(b \wedge c)$.
2. symbolically potentially analyses *an event* θ *along condition* c*, denoted as* $\boldsymbol{spa}\big(\langle b, M\rangle, \theta, c\big)$*, whenever* $\exists m \in M$ *where* $m \overset{\theta}{\underset{c}{\Longrightarrow\!\!\!\!\!\!\rightarrow}}$ *and* $\boldsymbol{sat}(b \wedge c)$. ■

Example 2. Recall monitor m_2 from E.g. 1. Consider the constrained monitor-set $\langle b, M\rangle$ where b is $y{<}20$, $M = \{\mathsf{if}\ y{>}50\ \mathsf{then}\ \mathsf{set}(\_).\bot\ \mathsf{else}\ \mathsf{set}\langle y{+}1\rangle.m_2'\}$, and

$$m_2' = \mathsf{rec}\, X.\mathsf{get}(y).\mathsf{if}\ y{>}50\ \mathsf{then}\ \mathsf{set}(\_).\bot\ \mathsf{else}\ \mathsf{set}\langle y+1\rangle.X.$$

This constrained monitor-set *cannot* potentially reach a verdict, $\neg\mathbf{spr}\big(\langle b, M\rangle, w\big)$. In fact, via (symbolic) τ-transitions it can only reach the monitor states $\mathsf{set}(\_).\bot$ and $\mathsf{set}\langle y{+}1\rangle.m_2'$. When observing the (symbolic) event $\mathsf{set}\langle z\rangle$, the monitor-set M can weakly transition to two potential monitor states: one, $\bot$, along condition $c_1 = (y > 50)$ and the other, m_2', along $c_2 = (y \leq 50) \wedge (y' = y + 1)$. However, since the condition $b \wedge c_1$ is *not* satisfiable, only the second branch corresponds to an actual transition in the concrete semantics (*i.e.*, there is a valuation ρ that satisfies $(y < 20) \wedge (y \leq 50) \wedge (z = y + 1)$). In fact, we can say that the constrained monitor-set can potentially analyse the event $\mathsf{set}\langle z\rangle$ along c_2, *i.e.*, predicate $\mathbf{spa}\big(\langle b, M\rangle, \mathsf{set}\langle z\rangle, c_2\big)$ from Def. 1. ■

From a specific set of potential states in a monitor computation, say $\langle b, M\rangle$, the symbolic analysis needs to calculate the possible next set of (symbolic) events the potential states can analyse. This does not only depend on the ability to symbolically transition with an event θ, but also the *conditions* required for this transition to be performed. The function $\mathbf{rc}(M, \theta)$ defined below computes the *set of all possible conditions* along which event θ may occur; it also accounts for the computation sequences that lead a monitor to deadlock and not be able to (weak-) symbolically transition with event θ. Once this set of *relevant* conditions for event θ is calculated, $\{c_1, \ldots, c_n\}$, the analysis needs to calculate which of these are realisable when paired with b from $\langle b, M\rangle$. Since each of these conditions can either be satisfied or violated at runtime, $\mathbf{sc}(b, \{c_1, \ldots, c_n\})$ returns the set of all the *possible ways* $\{b, c_1', \ldots, c_n'\}$ can be combined together where c_i' is either equal to c_i or its negation; the resulting combinations partition the valuations satisfying b, with some of the them being possibly *empty*. This then allows the symbolic analysis to calculate $\mathbf{saft}(\langle b, M\rangle, \theta, c)$, the reachable (symbolic) monitor states from $\langle b, M\rangle$ after analysing event θ with condition c. Note also how $\mathbf{saft}(\langle b, M\rangle, \theta, c)$ accounts for the possibility that an execution branch of $\langle b, M\rangle$ is unable to analyse a symbolic event θ along c: when this is the case, it introduces the inconclusive verdict, $\mathbf{0}$, in the set of reachable monitors to model monitor analysis preemption.

Definition 2 (Symbolic Reachability Analysis [17]). *The* relevant conditions *for a monitor-set M w.r.t. the symbolic event θ is given by:*

$$\boldsymbol{rc}(M, \theta) \stackrel{def}{=} \{\, c \mid \exists m \in M \cdot (m \underset{c}{\stackrel{\theta}{\Longrightarrow}} \ \text{or}\ \exists n \cdot (m \underset{c}{\Longrightarrow} n \ \text{and}\ n \stackrel{\tau}{\not\Longrightarrow} \ \text{and}\ n \stackrel{\theta}{\not\Longrightarrow})) \,\}$$

The satisfiability combinations *w.r.t. b for set $\{c_1, \ldots, c_n\}$ is given by:*

$$\boldsymbol{sc}(b, \{c_1, \ldots, c_n\}) \stackrel{def}{=} \{\, \{b, c_1', \ldots, c_n'\} \mid \forall i \in 1..n \cdot (c_i' {=} c_i \ \text{or}\ c_i' {=} \neg c_i) \,\}$$

The reachable constrained monitor-sets *from $\langle b, M\rangle$ after θ along c are:*

$$\boldsymbol{saft}(\langle b, M\rangle, \theta, c) \stackrel{def}{=} \{\, \langle \wedge B, \boldsymbol{saft}(M, B, \theta)\rangle \mid B \in \boldsymbol{sc}(b \wedge c, \boldsymbol{rc}(M, \theta)) \ \text{and}\ \boldsymbol{sat}(\wedge B) \,\}$$

$$\boldsymbol{saft}(M,B,\theta) \stackrel{def}{=} \left\{ n \;\middle|\; \begin{array}{l} \exists m \in M, c \cdot \boldsymbol{sat}((\wedge B) \wedge c) \text{ and } \left(m \stackrel{\theta}{\underset{c}{\Longrightarrow}} n \right. \\ \left. \text{or } (\exists n' \cdot m \underset{c}{\Longrightarrow} n' \not\xrightarrow{\tau} \text{ and } n' \not\xrightarrow{\theta} \text{ and } n = \mathbf{0})\right) \end{array} \right\} \quad \blacksquare$$

Equipped with this set of machinery, we can define Symbolic Monitor Controllability. It requires that:

1. whenever a set of potential states can (autonomously) reach a conclusive verdict, they must all do so and must do it immediately (without requiring further τ-transitions, since this can be interfered with when the instrumented process diverges to create a form of spinlock);
2. whenever a set of potential states can analyse an event, the reachable set of monitor states after carrying out that event is also included in the relation (*i.e.,* the relation is closed).

The interested reader is invited to consult [17] for further details.

Definition 3 (Symbolic Monitor Controllability [17]). *A relation over constrained monitor-sets* $\mathcal{S} \subseteq \left(\text{BExp} \times \mathcal{P}(\text{Mon})\right)$ *is said to be* symbolically controllable *iff for all* $\langle b, M\rangle \in \mathcal{S}$*, the following two conditions are satisfied:*

1. $\boldsymbol{spr}(\langle b,M\rangle, w)$ *and* $w \in \{\top, \bot\}$ *implies* $M = \{w\}$*;*
2. $\boldsymbol{spa}(\langle b,M\rangle, l\langle x\rangle, c)$ *where* $\boldsymbol{frsh}(\boldsymbol{fv}(\langle b, M\rangle) = x$ *implies* $\boldsymbol{saft}(\langle b, M\rangle, l\langle x\rangle, c) \subseteq \mathcal{S}$*.*

For a monitor m *to be symbolically controllable, there must exist some symbolically controllable relation* $\mathcal{S}$ *s.t.* $\langle true, \{m\}\rangle \in \mathcal{S}$*.* $\blacksquare$

Since symbolic controllability is both *sound* and *complete* w.r.t. consistent monitor detection, it can also be used to determine violations to the latter definition (recall that consistent detection is defined in terms of concrete events).

Example 3. It is tempting to monitor for the consolidated specifications denoted by m_2 and m_3 from E.g. 1, via the monitor $m_4 = m_2 + m_3$. Upon observing the (concrete) event $\mathsf{init}\langle 50\rangle$, m_4 may reach either of two monitor states, m_2' (from E.g. 2) and m_3' (described below); this is permitted by symbolic controllability (and by consistent detection), as long as both states reach the same verdict.

$$m_3' = \ \mathsf{let}\ lim = e_{calc}\ \mathsf{in}\ \big(\mathsf{if}\ x{<}lim\ \mathsf{then}\ \mathsf{end}(\_).\bot\ \mathsf{else}\ \mathsf{rec}\,X.\mathsf{get}(y).\mathsf{if}\ y \geq lim\ \mathsf{then}\ \mathsf{set}\langle y+1\rangle.\top\ \mathsf{else}\ \mathsf{set}(\_).X\big)$$

But consider a trace of events such as $\mathsf{init}\langle 50\rangle \cdot \mathsf{get}\langle 60\rangle \cdot \mathsf{set}\langle 61\rangle$. If the monitor transitions to the first monitor state, m_2', the execution will *always* be rejected, whereas if the monitor transitions to the second monitor state, $m_3'[^{50}/_x]$, two further cases must be considered. If the predetermined value *lim* is larger than 50, a conclusive verdict will *never* be reached. Otherwise, the execution is accepted. The aforementioned trace thus proves that m_4 is *not* consistently detecting.

According to our symbolic analysis of Def. 3, for m_4 to be symbolically controllable, there *must* exists some relation $\mathcal{S}$ that contains $\langle \mathsf{true}, \{m_4\}\rangle$. By Def. 3.2, $\mathcal{S}$ must also contain $\langle \mathsf{true} \wedge x = 50, \{m_2', m_3'\}\rangle$. If we assume that *lim*

greater than 50 (the converse case is similar), $\mathcal{S}$ must also contain $\langle \mathsf{true} \wedge (x{=}50) \wedge \mathsf{true}, \{\mathsf{if}\ y{>}50\ \mathsf{then}\ \mathsf{set}(\_).\bot\ \mathsf{else}\ \mathsf{set}\langle y+1\rangle.m_2', \mathsf{end}(\_).\bot\}\rangle$ and, in turn (after considering the symbolic event $\mathsf{set}\langle z\rangle$ with condition $y{>}50$), it must also contain $\langle \mathsf{true} \wedge (x{=}50) \wedge \mathsf{true} \wedge (y{>}50), \{\bot, \mathbf{0}\}\rangle$. But, clearly, the latter constrained monitor-set violates Def. 3.1. Thus, no such symbolically controllable relation exists. ■

The reachability closure requirement of Def. 3.2, defined using the (symbolic after) $\mathbf{saft}(\langle b, M\rangle, \theta, c)$ function of Def. 2, keeps on aggregating the conditions of the transitions to the constraining condition b in a constrained set $\langle b, M\rangle$. This complicates the formulation of a finite symbolic relation (whenever this exists). To overcome this, the work in [17] defines a sound method for consolidating constraining boolean conditions, thus garbage collecting redundant constraints that bear no effect on the meaning of the (open) monitor-set M.

Definition 4 (Optimised Symb. Controllability [17]). *The consolidation of a boolean expression b w.r.t. variable-set V, denoted $\boldsymbol{cns}(b, V)$, is defined as:*

$$\boldsymbol{cns}(b, V) \stackrel{def}{=} b_1 \ \textit{whenever}\ \boldsymbol{prt}(b, V) = \langle b_1, b_2\rangle \ \textit{for some}\ b_2$$

where the boolean expression partitioning operation $\boldsymbol{prt}(b, V)$ is defined as:

$$\boldsymbol{prt}(b, V) \stackrel{def}{=} \begin{cases} \langle b_1, b_2\rangle & \textit{if}\ \boldsymbol{sat}(b)\ \textit{and}\ b{=}b_1 \wedge b_2\ \textit{and}\ \boldsymbol{fv}(b_1) \subseteq V\ \textit{and}\ V \cap \boldsymbol{fv}(b_2) = \emptyset \\ \langle b, \mathit{true}\rangle & \textit{otherwise} \end{cases}$$

Let optimised symbolic reachability *from $\langle b, M\rangle$ for θ and c, $\boldsymbol{osaft}(\langle b, M\rangle, \theta, c)$, be defined as:*

$$\boldsymbol{osaft}(\langle b, M\rangle, \theta, c) \stackrel{def}{=} \left\{ \langle \boldsymbol{cns}(\wedge B, V), \boldsymbol{saft}(M, B, \theta)\rangle \;\middle|\; \begin{array}{c} B \in \boldsymbol{sc}(b \wedge c, \boldsymbol{rc}(M, \theta)) \\ \textit{and}\ \boldsymbol{sat}(\wedge B)\ \textit{and} \\ V = \boldsymbol{fv}(\boldsymbol{saft}(M, B, \theta)) \end{array} \right\}$$

A relation $\mathcal{S} \subseteq \big(\textsc{BExp} \times \mathcal{P}(\textsc{Mon})\big)$ is called optimised symbolically-controllable *iff for all $\langle b, M\rangle \in \mathcal{S}$:*

1. $\boldsymbol{spr}(\langle b, M\rangle, w)$ *and* $w \in \{\top, \bot\}$ *implies* $M = \{w\}$;
2. $\boldsymbol{spa}(\langle b, M\rangle, l\langle x\rangle, c)$ *s.t.* $\boldsymbol{frsh}(\boldsymbol{fv}(\langle b, M\rangle)) = x$ *implies* $\boldsymbol{osaft}(\langle b, M\rangle, l\langle x\rangle, c) \subseteq \mathcal{S}$.

The largest *optimised symbolically-controllable relation is denoted by $\mathcal{C}^{opt}$, and contains all optimised symbolically-controllable relations. We say that a monitor m is optimised symbolically-controllable iff there exists an optimised symbolically-controllable relation $\mathcal{S}$ such that $\langle \mathit{true}, \{m\}\rangle \in \mathcal{S}$.* ■

Example 4. Although monitoring for a different combined specification involving m_1 and m_3 from E.g. 1, *i.e.*, monitor $m_1 + m_3$, may reach different internal states, it can be shown to be symbolically controllable via the relation $\mathcal{S}$ defined below:

$$\mathcal{S} = \left\{ \begin{array}{l} \langle \mathsf{true}, \{m_1 + m_3\}\rangle,\ \langle x{=}0, \{\mathsf{end}(\_).\bot, m_3''\}\rangle,\ \langle \mathsf{true}, \{\bot\}\rangle,\ \langle x{\neq}0, \{m_3''\}\rangle, \\ \langle \mathsf{true}, \{m_3'''\}\rangle,\ \langle \mathsf{true}, \{\mathsf{get}(y).\mathsf{if}\ y{\geq}e_{calc}\ \mathsf{then}\ \mathsf{set}\langle y{+}1\rangle.\bot\ \mathsf{else}\ \mathsf{set}(\_).m_3'''\}\rangle \end{array} \right\}$$

where

$$m_3'' = \begin{cases} \mathsf{let}\ lim{=}e_{calc}\ \mathsf{in}\ \mathsf{if}\ x{<}lim\,\mathsf{then}\,\mathsf{end}(\_).\bot\ \mathsf{else} \\ \qquad\qquad \mathsf{rec}\,X.\mathsf{get}(y).\mathsf{if}\ y{\geq}lim\,\mathsf{then}\,\mathsf{set}\langle y{+}1\rangle.\bot\ \mathsf{else}\,\mathsf{set}(\_).X \end{cases}$$

$$m_3''' = \mathsf{rec}\,X.\mathsf{get}(y).\mathsf{if}\ y{\geq}e_{calc}\,\mathsf{then}\,\mathsf{set}\langle y{+}1\rangle.\bot\ \mathsf{else}\,\mathsf{set}(\_).X$$

Def. 4 allows us to discard redundant boolean conditions in the constrained monitor-sets of $\mathcal{S}$, collapsing semantically equivalent entries into the same syntactic representation. For instance, the entry $\langle \mathsf{true}, \{m_1 + m_3\}\rangle$ can potentially analyse the event $\mathsf{init}\langle x\rangle$ with the relevant conditions $\{\mathsf{true}, x{=}0\}$. The satisfiability combinations, $\mathbf{sc}(\mathsf{true}, \{\mathsf{true}, x{=}0\})$, are given by $\{\mathsf{true}{\wedge}x{=}0, \mathsf{true}{\wedge}\neg(x{=}0)\}$. The reachable monitor-set obtained by $\mathbf{saft}(\langle \mathsf{true}, \{m_1{+}m_3\}\rangle, \mathsf{init}\langle x\rangle, \mathsf{true}{\wedge}x{=}0)$ is $\langle \mathsf{true}{\wedge}x{=}0, \{\mathsf{end}(\_).\bot, m_3''\}\rangle$, and that obtained by the symbolic calculation $\mathbf{saft}(\langle \mathsf{true}, \{m_1 + m_3\}\rangle, \mathsf{init}\langle x\rangle, \mathsf{true}{\wedge}x \neq 0)$ is $\langle \mathsf{true}{\wedge}x{\neq}0, \{m_3''\}\rangle$. The respective conditions are consolidated as $x{=}0$ and $x{\neq}0$.

Similarly, the entry $\langle x{=}0, \{\mathsf{end}(\_).\bot, m_3''\}\rangle$ can potentially analyse the event $\mathsf{end}\langle x'\rangle$ with the relevant conditions $\{\mathsf{true}, x{<}e_{calc}{\wedge}\mathsf{true}\}$. The monitor-set obtained by $\mathbf{saft}(\langle \mathsf{true}, \{\mathsf{end}(\_).\bot, m_3''\}\rangle, \mathsf{end}\langle x'\rangle, (x{=}0){\wedge}(x{<}e_{calc}){\wedge}\mathsf{true})$ is given by $\langle (x{=}0){\wedge}(x{<}e_{calc}){\wedge}\mathsf{true}, \{\bot\}\rangle$; importantly, the conditions are consolidated as true since none of them impose any constraint on monitor-set $\{\bot\}$. ■

3 Preliminary Implementation

Symbolic Controllability, Defs. 3 and 4, is declarative in nature: to show that a monitor m is symbolically controllable, it suffices to provide a symbolically controllable relation $\mathcal{S}$ containing the constrained monitor-set $\langle \mathsf{true}, \{m\}\rangle$. However, this does *not* provide any indication on how this relation can be obtained.

Our preliminary attempt devising this algorithm is described in Alg. 1. Intuitively, the procedure starts from the initial constrained monitor-set $\langle \mathsf{true}, \{m\}\rangle$, checks for clause Def. 3.1 and then generates new monitor-sets to analyse using clause Def. 3.2. Constrained monitor-sets are represented as a pair containing a list of conditions (*i.e.,* conjuncted constraining conditions) and a list of monitors (*i.e.,* the monitor-set); the base condition true is represented by the empty list. The algorithm uses a list of pairs, $\mathcal{S}$, and a queue, $\mathcal{Q}$. The list of pairs (initialised to empty) stores the constrained monitor-sets that will make up the relation we are trying to construct; the queue, initialised to the singleton element $\langle \mathsf{true}, \{m\}\rangle$, is used to store the constrained monitor-sets that have not been analysed yet. Lists are convenient for reading and adding data; however, queues perform better when data needs to be removed since they have a time complexity of $O(n)$ and $O(1)$ respectively. The list $\mathcal{S}$ observes two key invariants, namely that (i) all the pairs in $\mathcal{S}$ satisfy Def. 3.1 and (ii) all reachable constrained monitor-sets from these pairs, obtained via $\mathbf{saft}$(-), are either in $\mathcal{S}$ itself or in $\mathcal{Q}$, waiting to be analysed. When $\mathcal{Q}$ becomes empty, a fixpoint is reached: all reachable constrained monitor-sets from $\mathcal{S}$ must be in $\mathcal{S}$ itself, satisfying Def. 3.2, and the resulting $\mathcal{S}$ is closed.

```
1  def CompSymRel(S, Q)
2   if Q.empty then
3    return true
4   else
5    # unseen constrained monitor-set
6    ⟨b, M⟩ ← Q.remove
7    S ← ⟨b, M⟩
8    # condition (1) true
9    if spr ⟨b, M⟩ then
10    # generate a fresh variable
11    x  ← frsh(fv⟨b, M⟩)
12    sevts ← GenSymEvents(M, x)
13    # generate the reachable cms
14    Q ← CompReach(sevts,⟨b, M⟩,Q,S)
15    CompSymRel (S, Q)
16   else
17    # condition (1) false
18    return false

19 def CompReach(sevts,⟨b, M⟩,Q,S)
20  for s in sevts do
21   c ← rc(M, s)
22   satComb ← sc(b, c)
23   for scomb in satCombs do
24    if spa ⟨b, M⟩ s scomb then
25     cms ← saft(⟨b, M⟩, s, scomb)
26     for cm in cms do
27      if cm ∉ S then
28       S ← ⟨b, M⟩
29       Q.append cm # add to queue
30  return Q

31 def IsSymControllable(M)
32  b ← [ ] # [ ] represents true
33  cm ← ⟨b, M⟩
34  Q← cm # init a queue
35  CompSymRel([ ], Q)
```

Alg. 1: Pseudocode for the Algorithm automating Symbolic Controllability

Function ComSymRel( ) in Alg. 1 is the main function. If $\mathcal{Q}$ is empty, there are no further constrained monitor-sets to analyse and *true* is returned (line 3). Otherwise, a constrained monitor-set is removed from $\mathcal{Q}$. Condition Def. 3.1 is checked (line 9) and the analysis terminates with *false* if violated. Line 12 obtains all symbolic events that can be observed by the current constrained monitor-set using function GenSymEvents( ), which is then used to get the reachable constrained monitor-sets using function CompReach( ). This function follows closely Def. 1 and Def. 2, but function sc( ) on line 22 returns only the combinations that are satisfiable; this removes the need to compute $\mathbf{sc}(b \wedge c)$ in $\mathbf{spa}(\text{-})$ and $\mathbf{sc}(\wedge B)$ in **saft**(-). The reachable constrained monitor-sets are generated on line 25, and those that have not been analysed yet are pushed to $\mathcal{Q}$ on line 29. Alg. 1 is implemented in straightforward fashion using OCaml [27].

Interfacing with the SAT Solver. Generating the set of satisfiability combinations w.r.t. a set of relevant conditions (line 22) requires the invocation of an external satisfiability solver to determine reachable paths. We used the *Z3* [28] theorem prover for this; its numerous APIs allow a seamless integration with our tool. Z3 relies on hand-crafted heuristics [30] to determine whether a set of formulas, also known as *assertions*, is satisfiable. Instead of opting to use the default solver, we used a custom strategy based on the built-in tactics **ctx-solver-simplify**( ) and **propagate-ineqs**( ), performing simplification and inequality propagation respectively. We used another important feature of Z3: instead of returning a boolean verdict, the function invoking the SAT solver, sat( ), returns the simplified formula together with the verdict. This increases the number of discarded conditions during boolean consolidation and makes future satisfiability checks that refine this condition *less* expensive.

Complexity Bounds. The complexity of Alg. 1 depends on two parameters:

1. The terms reachable from the initial monitor m via the symbolic semantics of Fig. 1, denoted here as the set $\mathbf{reach}(m)$. Since our monitors are expressed using a regular grammar, we can show that this set is finite for any monitor $m \in \textsc{Mon}$, *i.e.,* $\mathbf{size}\big(\mathbf{reach}(m)\big)=i$ for some $i \in \mathbb{N}$; see [4] for a similar proof of this fact. As our controllability analysis relies on *sets* of reachable monitors, the standard complexity for the power set construction is $O(2^i)$.
2. The satisfiability checks of the boolean constraints b generated by the symbolic analysis. In general, Alg. 1 needs to check the satisfiability of the boolean condition of *every* monitor set from the previous point. Satisfiability is usually a function of the number of free variables, $j \in \mathbb{N}$, in the boolean condition b. Although the standard boolean satisfiability would be $O(2^j)$, the boolean conditions in Alg. 1 involve variables for integers with operators, *i.e.,* integer programming. Since we are agnostic of the expression language used, this is not decidable for general integer expressions [29] (*e.g.,* expressions with both addition and multiplication). Limiting expressions to Presburger arithmetic would recover decidability [10], yielding a complexity that can be safely approximated to $2^{O(j)}$.

When decidable, the complexity of Alg. 1 can be safely approximated to $2^{O(i+j)}$.

4 Evaluating Efficiency

Although Sec. 3 demonstrates that controllability analysis can be implemented, albeit with high worst-case complexity bound, it is unclear whether the implementation scales well in practice. In this section we devise an evaluation strategy for our tool that attempts to capture typical use-cases; whenever performance bottlenecks are detected, alternative implementation methods are studied in Sec. 5 and compared to our baseline implementation.

A Benchmark for Assessing Deterministic Monitor Analysis. A major obstacle for assessing the scalability of Alg. 1 is the absence of a proper benchmark. To this end, we use the monitor modelling syntax of Fig. 1 to design a suite of pathological monitor template descriptions that are parametrisable in size and complexity, allowing us to carry out our evaluation in a systematic manner; see Tab. 1. Each template targets a specific feature of a symbolic analysis for non-deterministic behaviour. Concretely, $\mathrm{M}_{\mathrm{rec}}(n)$ is a monitor template that generates monitor instances that can transition to multiple sub-monitors, some of which lead to a verdict while others recurse to induce further iterations in the monitor analysis loop; this pathological behaviour induces large relation sizes $\mathcal{S}$ in the analysis of Alg. 1. In $\mathrm{M}_{\mathrm{cnd}}(n)$, (symbolic) events may be observed along various boolean conditions with the intention of increasing the number of constraints b in the corresponding constrained monitor-set $\langle b, M\rangle$ analysed in Alg. 1. The monitor instances generated by $\mathrm{M}_{\mathrm{cnd}}(n)$ also have a high branching

$$\mathrm{M}_{\mathrm{rec}}(n) \;=\; \mathsf{rec}\, X.\, \textstyle\sum_{i=1}^{n+1} \big(k\langle i\rangle.(l\langle i\rangle.X) + (q\langle i\rangle.\top)\big)$$

$$\mathrm{M}_{\mathrm{cnd}}(n) = \begin{array}{l} l(x).(\mathsf{if}\ x{=}4\ \mathsf{then}\ k\langle x\rangle.\bot\ \mathsf{else}\ k\langle x\rangle.\top) \\ \quad + \textstyle\sum_{i=2}^{n} \big(\mathsf{if}\ x \bmod 2{=}0\ \mathsf{then}\ \overbrace{\mathsf{if}\, x{<}2(n{-}i{+}3)\ \mathsf{then}\ldots\mathsf{if}\, x{<}2(n{-}i{+}3)\ \mathsf{then}}^{i=2..n} \\ \qquad\qquad \mathsf{if}\ x{>}2\ \mathsf{then}\ k\langle x\rangle.\bot \underbrace{\mathsf{else}\ k\langle x\rangle.\top \ldots \mathsf{else}\ k\langle x\rangle.\top}_{i=2..n+1}\big) \end{array}$$

$$\mathrm{M}_{\mathrm{brc}}(n) = \begin{array}{l} l(x).\big(\mathsf{if}\ x{=}4\ \mathsf{then}\ k\langle x\rangle.\bot\ \mathsf{else}\ k\langle x\rangle.\textstyle\sum_{j=1}^{3n}(k\langle j\rangle.\top)\big) \\ \quad + \textstyle\sum_{i=2}^{n} \big(\mathsf{if}\ x \bmod 2 = 0\ \mathsf{then}\ \overbrace{\mathsf{if}\ x{<}2(n{-}i{+}3)\ \mathsf{then}\ldots\mathsf{if}\ x{<}2(n{-}i{+}3)\ \mathsf{then}}^{i=2..n} \\ \qquad\qquad \mathsf{if}\ x{>}2\ \mathsf{then}\ k\langle x\rangle.\textstyle\sum_{j=1}^{3n}(k\langle j\rangle.\bot) \\ \qquad\qquad \underbrace{\mathsf{else}\ k\langle x\rangle.\textstyle\sum_{j=1}^{3n}(k\langle j\rangle.\top)\ldots\mathsf{else}\ k\langle x\rangle.\textstyle\sum_{j=1}^{3n}(k\langle j\rangle.\top)}_{i=2..n+1}\big) \end{array}$$

Table 1: Parametrisable Monitor Descriptions

factor, which induces larger monitor-sets M. The final monitor template $\mathrm{M}_{\mathrm{brc}}(n)$ generates monitors with nested branching that alternates with event analysis; this impacts the number of relevant conditions that need to be considered when calculating the reachable constrained monitor-sets in Alg. 1.

Preliminary Results. We evaluated the mean running time (over 3 repeated runs) of our preliminary (*Naive*) implementation for the three monitor templates of Tab. 1, instantiated by an ascending parameter n. All experiments were conducted on a Quad-Core Intel Core i5 64-bit machine with 16GB memory, running OCaml version 4.08.0 on OSX Catalina. They can be *reproduced* using the sources provided at `https://github.com/jasmine97xuereb/sym-cont`, whereby the *master* branch contains the preliminary implementation and the other branches the individual optimisations.

The plotted time results (in blue) are reported in Fig. 2 on a *logarithmic* scale; missing plot-points mean that the (controllability) analysis did not terminate within a stipulated time threshold (over 10 hours). The results confirm that the preliminary implementation does not scale well; although it has a low response time for low values of n, its performance degrades quickly as n increases (the worst behaviour measured was that for the pathological cases of $\mathrm{M}_{\mathrm{brc}}(n)$, where we immediately witnessed a sharp spike at $n = 3$). A closer inspection into the working of the algorithm reveals that the invocations to the Z3 solver are expensive operations, incurring a cost that is magnitudes higher than any other aspect of the analysis. In turn, the number of invocations is dependent on the number of relevant conditions: in the preliminary implementation of Alg. 1, the algorithm considers all 2^i possible combinations for a given number i of relevant conditions, each of which needs to be checked for satisfiability. This insight gave us a focus of attack for improving the tool's scalability.

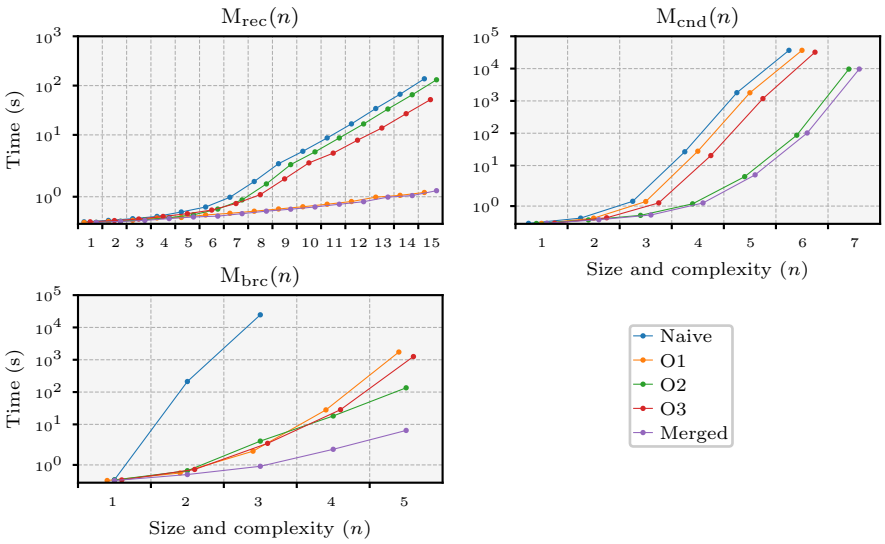

Fig. 2: Mean Running Time for different monitors

5 Optimisation Techniques

Upon closer inspection, we notice that a substantial number of conjuncted conditions generated by SC( ) of Alg. 1 are (trivially) unsatisfiable. Ideally, these cases should *not* result in invocations to the satisfiability solver.

Optimisation Technique O1. The first optimisation technique relies on the notion of (easily identifiable) *mutual exclusion*, whereby the satisfaction of one boolean condition necessarily violates that of the other.

Example 5. Recall the constrained monitor-set $\langle b, \{m_1 + m_2\}\rangle$ from E.g. 1. The relevant conditions w.r.t. event $\mathsf{init}\langle x\rangle$ are $\{b_1, b_2\}$, where b_1 is $x{=}0$ and b_2 is $x{=}50$. Accordingly, the satisfiability combinations generated for $\mathbf{sc}(\mathsf{true}, \{b_1, b_2\})$ are $b_1 \wedge b_2$, $\neg b_1 \wedge b_2$, $b_1 \wedge \neg b_2$, and $\neg b_1 \wedge \neg b_2$. Since x cannot be equal to values 0 and 50 simultaneously, conditions b_1 and b_2 are mutually exclusive. ■

A close inspection of the transition rules in Fig. 1 reveals that the constraints introduced tend to be of the form $x{=}e$; whenever $e{=}v$, it is easy to syntactically determine mutually exclusive conditions as in E.g. 5. The pseudocode in Alg. 2 first partitions the set of boolean conditions into two (line 3): the first partition, X, consists solely of variable assignments *i.e.,* expressions of the form $x{=}n$ for $n{\in}\mathbb{N}$, whereas the second partition, Y, contains the remaining conditions. For partition Y, all the possible combinations are generated as in Alg. 1 (line 7). As for partition X, we first cluster them according to the constrained variable (line 6); for each cluster, either one condition is *true*, or all of them are *false*

```
1  def SC(b,cs)
2    result ← [ ]
3    (X,Y) ← partition cs # X contains var assignments and Y all others
4    for x ∈ VAR(X) do # cluster expressions in X by their variable name
5      X_x ← {(y = v) ∈ X | x = y}
6      first_x ← X_x ++ [∧(negate all c in X_x)]
7    second ← all possible combinations for Y
8    combinations ← (⨯_{x∈VAR(X)} first_x) × second # cartesian product of first and second
9    for c in combinations do
10     t ← SAT ( [b, c] ) # t:(bool, exp list)
11     if FST(t) then
12       result ← result ++ SND(t)
```

Alg. 2: Pseudocode for first optimised function $\mathbf{sc}(b, \{c_1, ..., c_n\})$

(since they are necessarily mutually exclusive). The resulting combinations are merged by computing their *Cartesian Product* (line 8).

Example 6. Consider the *open* monitor term $m_5 = \mathsf{init}\langle y\rangle.\top$ and constrained monitor-set $\langle b, M\rangle$ where $M{=}\{m_1 + m_2 + m_5\}$ with m_1 and m_2 from E.g. 1. The relevant conditions for M along event $\mathsf{init}\langle x\rangle$, $\mathbf{rc}(M, \mathsf{init}\langle x\rangle)$ are $c{=}\{b_1, b_2, b_3\}$, where b_1 is $x{=}0$, b_2 is $x{=}50$, and b_3 is $x{=}y$. When calculating $\mathbf{sc}(\mathsf{true}, c)$ using Alg. 2, condition-set c is partitioned into $X{=}\{b_1, b_2\}$ and $Y{=}\{b_3\}$. All the conditions in the X are mutually exclusive. Thus, the possible ways the conditions in X can be combined are given by the condition-set $\{b_1, b_2, \neg b_1 \wedge \neg b_2\}$. The possible combinations of the conditions in Y are generated as before, and are given by $\{b_3, \neg b_3\}$ on line 7. These two resulting sets are merged as: $\{b_1, b_3\}$, $\{b_1, \neg b_3\}$, $\{b_2, b_3\}$, $\{b_2, \neg b_3\}$, $\{\neg b_1 \wedge \neg b_2, b_3\}$, and $\{\neg b_1 \wedge \neg b_2, \neg b_3\}$. Note that whereas Alg. 1 generates 8 combinations (and SAT solver invocations), this is now reduced to 6. Moreover, the latter combinations of logical formulas are less complex. ■

In general, given a set of relevant conditions of length k, a set of clusters $X_{x\in\text{VAR}(X)}$ and Y where $|X_{x\in\text{VAR}(X)}|{=}n_x$, $|Y|{=}m$, and $k{=}m{+}\sum_{x\in\text{VAR}(X)} n_x$, the number of times the SAT solver is invoked is reduced from $2^k{=}2^m \prod_{x\in\text{VAR}(X)} 2^{n_x}$ to $2^m \prod_{x\in\text{VAR}(X)}(n_x{+}1)$. Hence, the larger the first partition is, *i.e.*, $|X|$, the more effective the optimisation. When we evaluate the optimised implementation against the benchmark in Tbl. 1, depicted by the plot labelled *O1* in Fig. 2, we noticed that even though the running time for monitors $\mathrm{M}_{\mathrm{rec}}(n)$ and $\mathrm{M}_{\mathrm{brc}}(n)$ decreased substantially, that of monitors $\mathrm{M}_{\mathrm{cnd}}(n)$ was unaffected.

Optimisation Technique 2. Storing the aggregated boolean conditions as a *flat structure* loses information regarding the monitor branching structure.

Example 7. Consider m_6, a slight modification of monitor m_2 from E.g. 1.

$m_6 = \mathsf{init}\langle 50\rangle.\mathsf{let}\ \mathit{lim} = e_{\mathit{calc}}\ \mathsf{in}\ {m_6}'$

${m_6}' = \mathsf{rec}\, X.\mathsf{get}(y).\mathsf{if}\ y{\geq}50\ \mathsf{then}\ \mathsf{set}(\_).\bot\ \mathsf{else if}\ y{<}\mathit{lim}\ \mathsf{then}\ \mathsf{set}\langle y{+}1\rangle.X\ \mathsf{else}\ \mathsf{set}(\_).\bot$

Upon observing event $\mathsf{init}\langle 50\rangle$, followed by event $\mathsf{get}\langle y\rangle$, both along the boolean condition true, the reachable monitor-set for $\langle \mathsf{true}, \{m_6\}\rangle$ is given by $\langle \mathsf{true}, \{m_6{''}\}\rangle$, where $m_6{''} = \mathsf{if}\ y{\geq}50\ \mathsf{then}\ \mathsf{set}(\_).\bot\ \mathsf{else}\ \mathsf{if}\ y{<}e_{calc}\ \mathsf{then}\ \mathsf{set}\langle y{+}1\rangle.m_6{'}\ \mathsf{else}\ \mathsf{set}(\_).\bot$.

Monitor-set $\{m_6{''}\}$ analyses event $\mathsf{set}\langle x\rangle$ with relevant conditions c_1, c_2, and c_3, where c_1 is $(y{>}50){\wedge}(y{<}e_{calc}){\wedge}(x{=}y{+}1)$, c_2 is $(y{>}50){\wedge}\neg(y{<}e_{calc})$, and c_3 is $\neg(y{>}50)$. Computing the set of satisfiable combinations, $\mathbf{sc}(\mathsf{true}, \{c_1, c_2, c_3\})$, in a naive manner entails the invocation of the SAT solver $2^3{=}8$ times. However, there are multiple combinations that cannot hold. For instance, $c_1 \wedge c_2$ is not satisfiable because condition c_2 occurs along an *if true* branch, whereas condition b_1 occurs along the *else* branch of the *same* monitor. ■

We consider a hierarchic representation of expressions, *i.e.,* expression trees represented as tuples $e{=}\langle e', [e{''}], [e{'''}]\rangle$ with e' as the root. For convenience, we use the suggestive dot notation (.) to access specific elements. The condition of expression tree, e', is accessed via the field e.cond. Expression trees have a list of left and a list of right expressions. The left expressions, $[e{''}]$, can only be reached if $\mathbf{sat}(e')$ and are accessed via the field e.true. Similarly, the right expressions, $[e{'''}]$, can only be reached if $\neg\mathbf{sat}(e')$ and are accessed via e.false. Since $[e{''}]$ and $[e{'''}]$ can be reached when e' is true or false respectively, the expressions along the left and the right paths are mutually exclusive. Condition true is still represented by an empty list; expressions $e{''}$ and $e{'''}$ may be expression trees themselves.

Example 8. Recall monitor-set $\{m_6{''}\}$ from E.g. 7. If we recompute the relevant conditions for this monitor-set w.r.t. event $\mathsf{set}\langle x\rangle$, $\mathbf{rc}(\{m_6{''}\}, \mathsf{set}\langle x\rangle)$, using the new representation we obtain $b = \langle y{\geq}50, [\langle y{<}e_{calc}, [x{=}y{+}1], [\,]\rangle], [\,]\rangle$. ■

```
1  def TRAV(e: exp)
2    def GETPATHS(e': exp list)
3      paths ← [ ]
4      if e' not empty then
5        for x in e' do
6          paths ← paths ++ TRAV(x)
7      # cartestian product of all p in paths
8      return X_{i=1}^{n} paths_i
9    if e is an expression tree then
10     branchT ← GETPATHS(e.true)
11     branchF ← GETPATHS(e.false)
12     # add e.cond to each p in branchT
13     x ← e.cond ∧ p_i · ∀p_i ∈ branchT
14     y ← ¬ e.cond ∧ p_i · ∀p_i ∈ branchF
15     return x ++ y
16   else
17     return e
```

```
18 def SC(b,cs)
19   paths ← [ ], result ← [ ]
20   # X contains only expression trees
21   (X, Y) ← partition cs
22   for x in X do
23     paths ← paths ++ TRAV(x)
24   # cartesian product of all p in paths
25   k ← X_{i=1}^{n} paths_i
26   first ← X ++ [∧(negate all x in k)]
27   second ← all combinations for Y
28   # cartesian prod of first and second
29   combinations ← first × second
30   # filter out unsatisfiable conditions
31   for c in combinations do
32     t ← SAT ( [b, c] )
33     if FST(t) then
34       result ← result ++ SND(t)
```

Alg. 3: Pseudocode for second optimised function $\mathbf{sc}(b, \{c_1, ..., c_n\})$

The pseudocode for the second optimisation in Alg. 3 relies on the function TRAV( ). It traverses expression tree e passed as parameter and returns a list of mutually exclusive conditions. This function recursively computes all the paths along the *true* and *false* branches on lines 10 and 11 respectively. Once all the possible combinations along the *true* branch of the initial condition e.cond are generated, each combination is conjuncted with the corresponding initial condition, e.cond, on line 13. Similarly, those along the *false* branch are conjuncted with its negation, $\neg e$.cond, on line 14.

The function computing the satisfiability combinations, SC( ) in Alg. 3, works by first partitioning the set of boolean conditions into two, X, and Y, such that X contains only expression trees. The set of possible combinations of the conditions in X is obtained via function TRAV( ), which returns a list of condition-sets, $\{c_1, \cdots, c_n\}$, where each condition-set consists of mutually exclusive conditions. The cartesian product of these condition-sets is then computed, $c_1 \times \cdots \times c_n$, denoted by the generalised cartesian product $\bigtimes_{i=1}^{n} c_i$ (line 8). The possible combinations relative to the conditions in Y are generated as before. These two lists of combinations are then joined through their cartesian product (line 29).

Example 9. Recall boolean condition $b = \langle b_1, [\langle b_2, [b_3], [\,]\rangle], [\,]\rangle$ from E.g. 8, where b_1 is $(y{\geq}50)$, b_2 is $(y{<}e_{calc})$, and b_3 is $(x{=}y{+}1)$. We illustrate how the set of combinations deducible from expression tree b are obtained. Calling TRAV( ) on expression b generates the set of all possible combinations by traversing its left and right sub-branches recursively (lines 10, 11) to produce two lists of mutually exclusive conditions, $[b_2 \wedge b_3, \neg b_2]$ and $[\,]$. The conditions in $[b_2 \wedge b_3, \neg b_2]$ are conjuncted with b_1 (line 13), resulting in $c_1 = b_1 \wedge b_2 \wedge b_3$ and $c_2 = b_1 \wedge \neg b_2$. Similarly, $[\,]$ is conjuncted with $\neg b_1$, resulting in $c_3 = \neg b_1$. TRAV( ) then returns $[c_1, c_2, c_3]$. Generating the satisfiability combinations, $\mathbf{sc}(\mathsf{true}, \{c_1, c_2, c_3\})$ in E.g. 7 decreases the number of possible combinations from 8 to 3. ■

It is worth noting that the effectiveness of this optimisation depends on both the depth and the number of expression trees, *i.e.*, size of partition X. Evaluating it against the benchmark in Tbl. 1, we obtain the plot labelled *O2* in Fig. 2. The resulting graph confirms that the tool performs better for $\mathrm{M}_{\mathrm{cnd}}(n)$.

Optimisation Technique 3. Despite the merits afforded by the preceding optimisations, multiple instances where the satisfiability solver must be invoked still prevail. We attempt to circumvent this overhead by batching the satisfiability checks. If all two-pairs are simultaneously satisfiable, the satisfiability of the entire list is checked, otherwise, if one pair is unsatisfiable, then it immediately follows that the list of conditions is unsatisfiable. This results in the mean running times shown by the plot labelled *O3* in Fig. 2 (recall that the values on the y-axis are in logarithmic form). This technique yields a substantial gain as well. For instance, comparing the mean running time against that of the preliminary version for monitor $\mathrm{M}_{\mathrm{cnd}}(5)$ from Tab. 1 there is a percentage decrease of 35%. Even better, for monitor $\mathrm{M}_{\mathrm{brc}}(3)$, there is a percentage decrease of 99.99%. However, the other two optimisation techniques depicted by the plots labelled *O1* and *O2* in Fig. 2 generally give improvements with better orders of magnitude.

Merged Optimisations. Merging all the optimisation techniques, an improvement in the mean running time is immediately noticeable, especially for monitors $\mathrm{M}_{\mathrm{brc}}(n)$ from Tbl. 1. For instance, comparing the mean running time for n=3 using the preliminary and the final optimised version, there is a percentage decrease of 99.996%. In fact, the plot labelled *Merged* in Fig. 2, acts as a lower bound for all other versions.

6 Conclusion

This paper investigates the implementability aspects of monitor controllability [17]. We discuss the realisability of a prototype that directed us towards the execution bottlenecks of the monitor analysis; we devised a number of solutions to these bottlenecks, implemented them, and studied which ones are the most effective. Our implementation remains closely faithful to the original definition of symbolic controllability, reassuring us of the correctness of our analysis.

Future Work. We plan to build translator tools that generate model descriptions of monitors in terms of the syntax discussed in Sec. 2, as is done in tools such as Soter [14]. This allows us to analyse a wider range of real-world monitor implementations using our tool. We also plan to investigate further optimisations to symbolic controllability that continue to improve the utility of our tool.

Related Work. An alternative approach to analysing for monitor deterministic behaviour is that of converting the monitor description itself into a deterministic one. This approach was investigated extensively in [2,3] for a variety of methods and concludes that any conversion typically incurs a triple exponential blow-up. The closest work to ours is [24], which uses SMT-based model checking to prove invariants about monitors. One illustrative invariant they consider is the analysis of a combined execution of two monitors (akin to our monitor sets) using k-induction (*i.e.,* bounded model checking); by contrast we consider the entire (possibly infinite) run through coinduction. Similar work on verifying dynamic programming monitors for LTL that uses the *Isabelle/HOL* proof assistant [33] is also limited to *finite* traces. *Isabelle/HOL* is used in [35] to extract certifiably-correct monitoring code from specifications expressed in Metric First-Order Temporal Logic (MFOTL). Although MFOTL uses quantifications over event data (similar to ours), the analysis in [35] is limited to formulas that are satisfied by *finitely-many* valuations; our techniques do not have this restriction. Further afield, the work in [31] uses symbolic analysis and SMT solvers to reason about the runtime monitoring of contracts. Their symbolic analysis is however concerned with shifting monitoring computation to the pre-deployment phase, which is different from our aim.

Acknowledgements. The authors thank Antonis Achilleos, Duncan Paul Attard, Stefania Damato, Clément Fauconnet and John Parnis for their help, and the anonymous reviewers for their comments and suggestions for improvement.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A.: A Framework for Parameterized Monitorability. In: FOSSACS. LNCS, vol. 10803 (2018). https://doi.org/10.1007/978-3-319-89366-2_11
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: On the Complexity of Determinizing Monitors. In: CIAA. LNCS, vol. 10329 (2017). https://doi.org/10.1007/978-3-319-60134-2_1
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: Determinizing monitors for HML with recursion. JLAMP **111** (2020). https://doi.org/10.1016/j.jlamp.2019.100515
4. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again. PACMPL **3**(POPL) (2019). https://doi.org/10.1145/3290365
5. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: An Operational Guide to Monitorability. In: SEFM. LNCS, vol. 11724 (2019). https://doi.org/10.1007/978-3-030-30446-1_23
6. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457 (2018). https://doi.org/10.1007/978-3-319-75632-5_1
7. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: Runtime verification with minimal intrusion through parallelism. FMSD **46**(3) (2015). https://doi.org/10.1007/s10703-015-0226-3
8. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. TCS **669** (2017). https://doi.org/10.1016/j.tcs.2017.02.009
9. Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Rosenblueth, D.A., Travers, C.: Decentralized Asynchronous Crash-Resilient Runtime Verification. In: CONCUR. LIPIcs, vol. 59 (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.16
10. Büchi, J.R.: Weak second-order arithmetic and finite automata. Mathematical Logic Quarterly **6**(1-6) (1960). https://doi.org/10.1002/malq.19600060105
11. d'Amorim, M., Roşu, G.: Efficient Monitoring of ω-Languages. In: CAV. LNCS, vol. 3576 (2005). https://doi.org/https://doi.org/10.1007/11513988_36
12. Debois, S., Hildebrandt, T., Slaats, T.: Safety, Liveness and Run-Time Refinement for Modular Process-Aware Systems with Dynamic Sub Processes. In: FM. LNCS, vol. 9109 (2015). https://doi.org/10.1007/978-3-319-19249-9_10
13. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. FMSD **46**(3) (2015). https://doi.org/10.1007/s10703-014-0218-8
14. D'Osualdo, E., Kochems, J., Ong, C.H.L.: Automatic Verification of Erlang-Style Concurrency. In: SAS. LNCS (2013). https://doi.org/10.1007/978-3-642-38856-9_24
15. Fraigniaud, P., Rajsbaum, S., Travers, C.: On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems. In: RV. LNCS (2014). https://doi.org/10.1007/978-3-319-11164-3_9
16. Francalanza, A.: A Theory of Monitors (Extended Abstract). In: FoSSaCS. LNCS, vol. 9634 (2016). https://doi.org/10.1007/978-3-662-49630-5_9
17. Francalanza, A.: Consistently-Detecting Monitors. In: CONCUR. LIPIcs, vol. 85 (2017). https://doi.org/10.4230/LIPIcs.CONCUR.2017.8

18. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the Hennessy-Milner logic with recursion. FMSD **51**(1) (2017). https://doi.org/10.1007/s10703-017-0273-z
19. Francalanza, A., Mezzina, C.A., Tuosto, E.: Reversible Choreographies via Monitoring in Erlang. In: DAIS. LNCS, vol. 10853 (2018). https://doi.org/10.1007/978-3-319-93767-0_6
20. Francalanza, A., Seychell, A.: Synthesising Correct concurrent Runtime Monitors. FMSD **46**(3) (2015). https://doi.org/10.1007/s10703-014-0217-9
21. Gommerstadt, H., Jia, L., Pfenning, F.: Session-Typed Concurrent Contracts. In: ESOP. LNCS, vol. 10801 (2018). https://doi.org/10.1007/978-3-319-89884-1_27
22. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime Verification Based on Register Automata. In: TACAS. LNCS, vol. 7795 (2013). https://doi.org/10.1007/978-3-642-36742-7_19
23. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and Blame Assignment for Higher-order Session Types. In: POPL (2016). https://doi.org/10.1145/2837614.2837662
24. Laurent, J., Goodloe, A., Pike, L.: Assuring the Guardians. In: RV. LNCS, vol. 9333 (2015). https://doi.org/10.1007/978-3-319-23820-3_6
25. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. IJIS **4**(1-2) (2005). https://doi.org/10.1007/s10207-004-0046-8
26. Luo, Q., Roşu, G.: EnforceMOP: A Runtime Property Enforcement System for Multithreaded Programs. In: ISSTA. ACM (2013). https://doi.org/10.1145/2483760.2483766
27. Minsky, Y., Madhavapeddy, A., Hickey, J.: Real World OCaml - Functional Programming for the Masses (2013)
28. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. LNCS (2008). https://doi.org/10.1007/978-3-540-78800-3_24
29. de Moura, L., Bjørner, N.: Satisfiability modulo Theories: Introduction and Applications. CACM **54**(9) (2011). https://doi.org/10.1145/1995376.1995394
30. de Moura, L., Passmore, G.O.: The Strategy Challenge in SMT Solving. In: Automated Reasoning and Mathematics - Essays in Memory of William W. McCune. LNCS, vol. 7788 (2013). https://doi.org/10.1007/978-3-642-36675-8_2
31. Nguyen, P.C., Tobin-Hochstadt, S., Horn, D.V.: Higher order symbolic execution for contract verification and refutation. JFP **27** (2017). https://doi.org/10.1017/S0956796816000216
32. Reger, G., Cruz, H.C., Rydeheard, D.E.: MarQ: Monitoring at Runtime with QEA. In: TACAS. LNCS, vol. 9035 (2015). https://doi.org/10.1007/978-3-662-46681-0_55
33. Rizaldi, A., Keinholz, J., Huber, M., Feldle, J., Immler, F., Althoff, M., Hilgendorf, E., Nipkow, T.: Formalising and Monitoring Traffic Rules for Autonomous Vehicles in Isabelle/HOL. In: IFM. LNCS, vol. 10510 (2017). https://doi.org/10.1007/978-3-319-66845-1_4
34. Schneider, F.B.: Enforceable Security Policies. ACM Trans. Inf. Syst. Secur. **3**(1) (2000). https://doi.org/10.1145/353323.353382
35. Schneider, J., Basin, D.A., Krstic, S., Traytel, D.: A Formally Verified Monitor for Metric First-Order Temporal Logic. In: RV. LNCS, vol. 11757 (2019). https://doi.org/10.1007/978-3-030-32079-9_18
36. Vardi, M.Y., Wolper, P.: Reasoning about Infinite Computations. Inf.& Comp. **115**(1) (1994). https://doi.org/http://dx.doi.org/10.1006/inco.1994.1092