Implementing Monitor Controllability

Jasmine Xuereb

Faculty of ICT

University of Malta

May 2019

Contents

List of Figures

List of Tables

Implementing Monitor Controllability

May 2019

1 Introduction

Monitors are computational entities responsible for observing the execution of other programs to infer properties about them, such as acceptance or rejection flags (i.e. verdicts), violation of a safety property[1], or whether the program's behaviour conforms to the specified requirement. In a typical monitoring setup, monitors are generally considered to be part of the *Trusted Computing Base*, without neither interfering nor in any way altering the behaviour of the system under scrutiny [1], [2]. Subsequently, they are expected to guarantee the manifestation of a correct behaviour themselves. One of the properties a monitor under scutiny must exhibit in order to ascertain this correctness requirement is *deterministic behaviour*. However, this correctness criterion is often left ambiguous. Indeed, there are various candidate definitions for deterministic monitor behaviour, one of which is the contextual definition of *consistent-detection* as presented by Francalanza in [3].

▶ **Definition 1.** A monitor m is said to be *consistently detecting* for some system s iff for all traces (i.e. sequences of events) t generated by the system s, it either *always* gives the same verdict (i.e. it always accepts or it always rejects it) or else it *never* provides a verdict. ◀

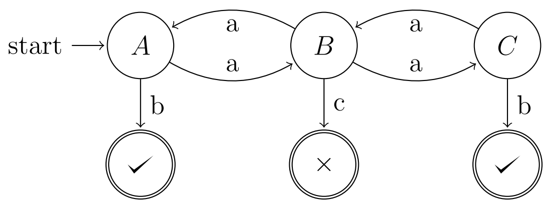


Figure 1: Labelled Transition System (*LTS*) description of a monitor

[1]A safety property guarantees that something bad will *never* happen.

▶ **Example 1**. Consider the monitor depicted in Figure 1. This monitor consistently *accepts* all strings having an odd number of a's followed by a single c, consistently *rejects* all strings having an even number of a's followed by a single b, and fails to provide a verdict for all other patterns, regardless of the path chosen[2]. Thus, this monitor is consistently-detecting, regardless of its *internal non-determinism*, meaning that it exhibits *deterministic behaviour*. ◀

Even though it is immediately discernible that the monitor description presented in the previous example behaves deterministically, it is not always straightforward. In fact, as already alluded, internal non-determinism does not force the monitor under scrutiny to exhibit non-deterministic behaviour. This seriously complicates the analysis, and thus gives rise for the need of some systematic analysis technique.

▶ **Example 2**. Recall Fig. 1 and suppose that the given monitor description accepts traces from a thermostat, where the transition with label a represents a **read** event, the transition with label b represents a **set** event, and the transition with label c represents an **abort** event. This thermostat works by either setting the temperature after reading it an even number of times, or by aborting all operations after reading it an odd number of times.

Notice that all the events **read***(v)*, **set***(v)*, and **abort***(v)* quantify over an arbitrary value v, where v is some temperature reading. This value is dynamically learnt—there is an *infinite* number of possible values that can be read. Consequently, this monitor is able to observe an *infinite* number of **read** events. Hence, upon observing a **read** event from state A, the monitor can transition to *infinitely many states* and could potentially branch depending on the value read as illustrated in Fig. 4. The same argument also applies to the other two events. As a result, when the event value domain is infinite, then it becomes impossible to explore all possible execution paths. ◀

In order to be able to automate the analysis of monitors capable of transitioning to infinitely-many states, it is becomes obligatory to argue about events in a *symbolic* manner and aim for a more *expressive* definition. This can be done by adding boolean constraints to both transitions and states as shown in Fig. 4, where for a monitor to be in a particular state, it must satisfy the underlying boolean condition b, whereas in order to transition from the current state to some other state, then it must also satisfy the boolean condition accompanying the symbolic transition b' i.e. $b \wedge b'$ must be *satisfiable*. Otherwise, that state can never be reached.

[2] *Different* paths may be taken as long as they lead to the *same* verdict.

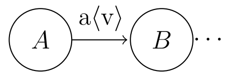


Figure 2: Monitor with an event carrying a value

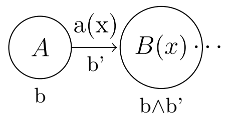


Figure 3: Monitor with a symbolic event

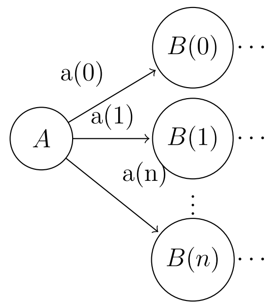


Figure 4: Monitor with an event carrying a variable

However, even though this approach is more expressive—the infinite number of states *breath-wise* in Fig. 3 have been reduced to only one—problems still arise when put to practise, especially when the monitor under scrutiny contains loops. Since after each transition new boolean conditions are added, the monitor is prohibited from transitioning to some previous state, possibly leading to an infinite number of states *depth-wise* as depicted in Fig. 5.

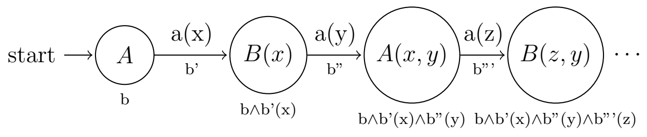


Figure 5: Labelled Transition System (*LTS*) description of a monitor

Francalanza claims that an analysis technique that can be used to check whether a monitor is consistently detecting is through the notion of *Controllability*. However, this definition does not scale well for symbolic events. Therefore, Francalanza presents a second, more expressive analysis technique, namely *Symbolic Controllability* [3]. He further claims that both notions are implementable and that the latter is more scalable than the former.

▶ **Definition 2.** A relation $\mathcal{S} \subseteq (\text{BE}\textsc{xp} \times \mathcal{P}(\text{M}\textsc{on}))$ is said to be *symbolically controllable* iff for all constrained monitor-sets in that relation, $\langle b, M\rangle \in \mathcal{S}$, the following two conditions are satisfied:

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

▶ **Definition 3.** For a monitor m to be *symbolically controllable*, there must exist some symbolically controllable finite relation $\mathcal{C}_{sym}$ s.t $\langle true, \{m\}\rangle \in \mathcal{C}_{sym}$. ◀

The notion *Symbolic Controllability* is of a *co-inductive* nature. Even though it is more amenable to mechanisation, this concept assumes the existance of a witness relation and does *not* provide an algorithm. However, since co-inductive definitions can shape structures from how they can be deconstructed [4], in order to show that some monitor m is Symbolically Controllable, then it suffices to provide a symbolically controllable relation $\mathcal{S}$ that contains the constrained monitor-set $\langle true, \{m\}\rangle$.

In order to verify the implementability claim, a tool that automates this analysis shall be implemented. However, despite the merits afforded by the notion of Symbolic Controllability, a direct implementation of this mathematical definition still does not suffice for a full automation, especially when the monitor under scutiny is of a recursive nature. Thus, an obstacle in creating this automation tool is managing increasing aggregating boolean conditions, which in turn leads to an infinite number of states depth-wise as alluded to in Fig. 5.

This may be catered for through an added *garbage collection functionality* wherby all *redundant* boolean constraints are removed. For example, in Fig. 5, the boolean condition b does not impose any constraint on state $B(x)$, and thus it can be removed. Similarly, the boolean sub-condition $b \wedge b'(x)$ is redundant at state $B(z, y)$ since it plays no *effective role* in binding the free variables of this state, namely z and y. Hence, this subcondition may be discarded. Through this added functionality, variables which were once bound will eventually become free, resulting in states with underlying boolean conditions identical to ones that have already been explored, which in turn addresses the problem of possibly infinite number of states depth-wise.

Moreveover, as already mentioned, in order for a monitor to transition, then the boolean condition of the current state and that along which the event occurs must be satisfiable. Thus, another struggle is engineering a method of checking the *satifiability* of boolean conditions. Using an external satisfiability solver component would also entail its seamless integration with the automation tool. Moreover, if implemented naively, this external component must be invoked repeatedly, resulting in a lot of inefficieny, which in turn affects the total computation time. To try and minimise this overhead, alternative optimisation techniques must be investigated.

2 Identified Bottlenecks

A preliminary automation tool was implemented, verifying the implementability claim.

An additional but crucial property the tool must observe for its long term-success is *scalability*. Ideally, for any tool to be usable in practise, it must neither be under- nor over-engineered and it must be able to handle complex monitor descriptions without resulting in an excessive spike in the cost [5], in this scenario on the fronts of responsiveness.

However, there is no existing suite of examples on which the evaluation of the artifact developed may be carried out. Taking into consideration the monitor's grammar, there are infinitely many possible monitor descriptions that can be anayslsed. In the light of this observation, a suite of examples covering as many possible diverse monitor descriptions as possible were designed to carry out evaluation in a systematic manner. These examples, shown in Table 1, were generated in an automated fashion, whereby their complexity is reflected by either the number of choices afforded or the number of nested conditional monitors.

The following section attempts to evaluate the tool by pushing it to its breaking point so as to find its threshold, thus identifing the existent bottlenecks. Subsequently, multiple alternative engineering methods shall be explored with the aim of eliminating or circumventing them.

2.1 Free Variables Bottleneck

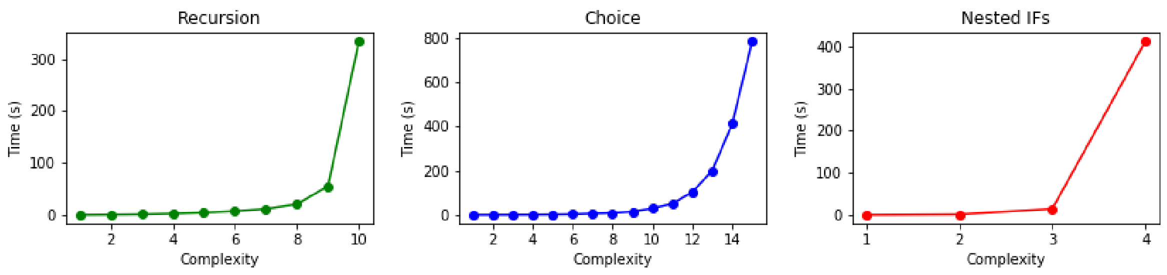


Figure 6: Mean Running Time for Different Monitors

The preliminary automation tool was evaluated based on the suite of examples laid out in Table 1. Plots illustrating how the mean execution time for each monitor description varies according to its complexity are shown in Fig. 6. It is immediately discernible that the preliminary tool does *not* scale well because it severly undermines its load scalability—even though it has a low response time for light loads, its performace degrades substantially as the complexity of the monitor increases slightly.

The inner workings of the tool was better investigated and one bottleneck was identified. Directly implementing mathematical definitions afforded by Definition X entails the computation of the set of free variables for each monitor at each step. Even though in the majority of the cases the time taken for this is negligibly small, overheads arise in case of recursion monitors, especially as their complexity start to increase. This increase in running time is especially visible in Fig. 6, where if the complexity of recursion monitors is increased from 9 to 10, the time taken increases from $56s$ to $334s$.

This problem was adderessed through the use of a map which maps recursion variables to its set of free variables. Subsequently, instead of having to compute this set each time, the map is used to retrieve the corresponding result.

After this optimisation was taken into effect, evaluation was carried out again, resulting in improved running times as shown in the figure below, particularly for monitors of a recursive nature.

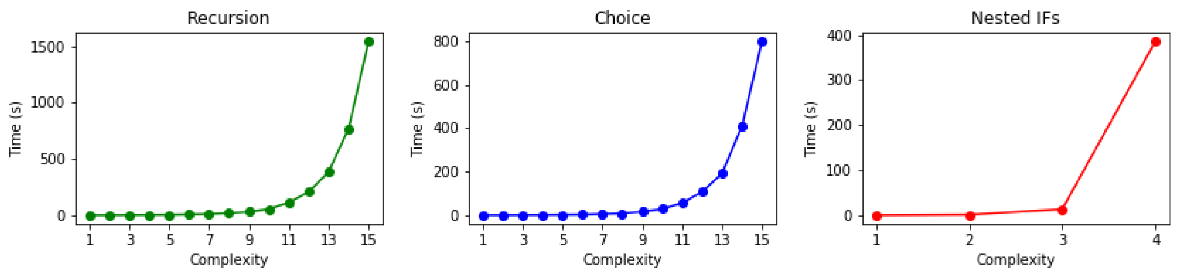


Figure 7: Mean Running Time for Different Monitors

2.2 Z3 Integration

For the scope of checking whether a given logical formula is satisfiable or not, Microsoft's theorem prover, Z3, was chosen because due to its numerous APIs, it allows a seamless integration with our tool.

The aim of this section is to integrate the external satisfiability component seamlessly with the artefact developed, trying to keep the overhead emanating from this component to a minimum. This is done by trying to exploit Z3's functionality.

2.2.1 Preliminary Integration

This low-level external component solves logical formulas by first pushing all the formulas into its internal stack. If there exists some satisfiability assignment (i.e. an interpretation), then all the asserted formulas are true, meaning that they are satisfiable. Otherwise, if there is none, then the set of logical formulas are unsatisfiable.

The preliminary automation tool was evaluated based on the suite of examples laid out in Table 1.

After a thorough investigation of the artifact developed, it resulted that a lot of the inefficiency was being caused by the external satisfiability solver component. Even though Z3's theorem solver usually tends to perform well for many logical formulas, it may still perform poorly, especially when the class of problems presented is novel or intricate, as is the case in this scenario.

2.2.2 Using Tactics

As already alluded to, if the Z3 library is used naively for complex boolean conditions, it ceases from performing well.

It might make more sense if a large set of boolean conditions was divided into smaller subconditions, or rather smaller building blocks which once conjucted together are representative intial logical formula.

In the light of this, an alternative strategy for expressing the set of assertions was explored. Instead of converting all the boolean expresions into Z3 expressions, pushing them onto Z3's internal stack and invoking the SAT solver to check whether there exists some satisfiability assignment, a different approach was opted for. In this alternative technique, boolean expressions are first converted into a set of *goals*, which may be regarded as the basic building blocks. The set of *goals* are then combined together using what are known as *tacticals*, forming *tactics*, which fundamentally represent functions. These tacticals are responsible for processing the corresponding set of goals and returning whether they are satisfiable or not.

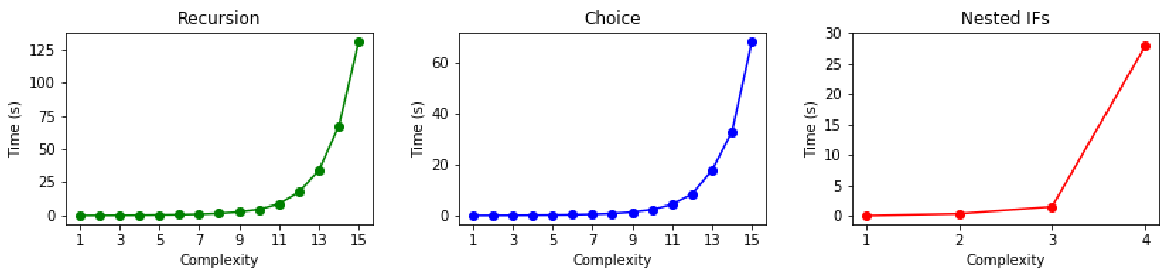


Figure 8: Mean Running Time for Different Monitors

2.2.3 Boolean Simplification

▶ **Example 3**. Consider the following monitor.

$$m_1 = l(x).l(y).(\texttt{if}\ \ x == 5 \wedge y > 5)\ \ \texttt{then}\ \ k\langle x\rangle.k\langle y\rangle.\bot\ \ \texttt{else}\ \ k\langle x\rangle.k\langle y\rangle.\top$$

After observing the symbolic trace of events $l(\$0).l(\$1)$, where both $\$0$ and $\$1$ are fresh variables, the intial constrained monitor-set $\langle \texttt{true}, \{m_1\}\rangle$ transitions to the following constrained monitor-set.

$$\langle \texttt{true}, \underbrace{\{\texttt{if}\ \ (\$0 == 5 \wedge \$1 > 5)\ \ \texttt{then}\ \ k\langle \$0\rangle.k\langle \$1\rangle.\top\ \ \texttt{else}\ \ k\langle \$0\rangle.k\langle \$1\rangle.\top\}}_{M'}\rangle.$$

The first condition of Definition X, $\texttt{spr}(\langle \texttt{true}, M'\rangle)$, holds trivially. As for the second condition, M' can potentially analyse the symbolic event $k\langle \$2\rangle$ along the relevant conditions b_1 and b_2.

$$\mathbf{rc}(M'), l\langle \$3\rangle) = \{\underbrace{\$0 = 5 \wedge \$1 > 5 \wedge \$2 = \$}_{b_1}, \underbrace{\neg(\$0 = 5 \wedge \$1 > 5) \wedge \$2 = \$0}_{b_2}\}$$

The set of satisfiability combinations, $\mathbf{sc}(\texttt{true}, \{b_1, b_2\})$ is then computed. From a mechanical point of view, this function works by first generating all the possible combinations of b_1 and b_2, irrespective of whether they are satisfiable or not, and then adding each combination possible with the boolean condition of the constrained monitor-set under analysis via a logical *and*. Note, however, that since in this case the boolean condition of the constrained monitor-set is $\texttt{true}$, it is not added because $\texttt{true} \wedge \texttt{anything} \implies \texttt{anything}$ by the equivalence laws of propositional logic.

$$\begin{aligned}
c_1 &= (\$0 = 5 \wedge \$1 > 5 \wedge \$2 = \$0) \wedge (\neg(\$0 = 5 \wedge \$1 > 5) \wedge \$2 = \$0)\\
c_2 &= (\$0 = 5 \wedge \$1 > 5 \wedge \$2 = \$0) \wedge \neg(\neg(\$0 = 5 \wedge \$1 > 5) \wedge \$2 = \$0)\\
c_3 &= \neg(\$0 == 5 \wedge \$1 > 5 \wedge \$2 = \$0) \wedge (\neg(\$0 = 5 \wedge \$1 > 5) \wedge \$2 = \$0)\\
c_4 &= \neg(\$0 = 5 \wedge \$1 > 5 \wedge \$2 = \$0) \wedge \neg(\neg(\$0 = 5 \wedge \$1 > 5) \wedge \$2 = \$0)
\end{aligned}$$

The satisfiability of each aggregated condition is then checked by invoking the function $\mathbf{sat}()$ which in turn invokes the external satisfiability component. If the aggregated condition is satisfiable, it is added to the set of satisfiable combinations. Otherwise, it is disregarded. In this case, the combinations c_2, c_3 and c_4 are satisfiable, whereas c_1 is not. Hence, the former combinations are returned.

For the first satisfiability combination c_2, $\mathbf{spa}(\langle \texttt{true}, M', k\langle \$2\rangle, c_2)$ is true, resulting in a new constrained monitor-set. The same is done for the other two satisfiability combinations.

$$\mathbf{spa}(\langle \texttt{true}, M', k\langle \$2\rangle, c_2) \implies \mathbf{saft}(\langle \texttt{true}, M', k\langle \$2\rangle, c_2) = \langle c_2, \{k\langle \$1\rangle.\bot\}\rangle \tag{1}$$
$$\mathbf{spa}(\langle \texttt{true}, M', k\langle \$2\rangle, c_3) \implies \mathbf{saft}(\langle \texttt{true}, M', k\langle \$2\rangle, c_2) = \langle c_3, \{k\langle \$1\rangle.\top\}\rangle \tag{2}$$
$$\neg\mathbf{spa}(\langle \texttt{true}, M', k\langle \$2\rangle, c_3) \tag{3}$$

Note however, that when carrying out Boolean consolidation of the underlying constraints in both (1) and (2), even though variable \$0 is not in the set of free variables of the constrained monitor-set, it cannot be removed because the condition imposed by **prt** is not satisfied, meaning that no Boolean sub-condition can be removed.

Again, for the constrained monitor-set in (1), the first condition of Definition X is trivially true. Moving on to the second condition, it can potentially analyse the symbolic event $k\langle \$4\rangle$ along the Boolean condition $b_3 = (\$4 = \$1)$. The set of satisfiability combinations, $\mathbf{sc}(c_2, \{b_3\})$ is then computed. At this point, it becomes necessary to invoke the satisfiability solver once more to check whether the aggregated boolean condition $c_2 \wedge b_3$ is satisfiable or not, even though it has already been verified that condition c_2 is satisfiable at some previous computation step. Also, it is key to acknowledge the fact the set of relevant conditons along which some symbolic event may occur is not always made up of one single clause—meaning that the satisfiability of the Boolean condition of the constrained monitor-set under scrutiny, in this case c_2, must be checked multiple times. Similarly, the same argument applies for the constrained monitor-set in (2).

◀

From the example discussed, it is evident that there is a notable amount of excessive computation resulting from the satisfiability solver component. However, upon closer inspection, it is immediately discernible that this can be circumvented. It would be more plausible if instead of carrying forth the whole Boolean conditions laid out in (1), (2) and (3), these aggregated conditions were orchestrated in such a way that facilitates the upcoming satisfiability checking by opting to inspect simpler and fewer logical formulas whenever possible.

For example, recall Boolean condition c_2. After performing the following algebraic manipulations, c_2 is transformed into an equivalent condition but with minimised terms.

$$
\begin{aligned}
c_2 &= (\$0 = 5 \wedge \$1 > 5 \wedge \$2 = \$0) \wedge \neg(\neg(\$0 = 5 \wedge \$1 > 5) \wedge \$2 = \$0) \\
&\equiv (\$0 = 5 \wedge \$1 > 5 \wedge \$2 = \$0) \wedge \neg(\neg(\$0 = 5 \wedge \$1 > 5) | \neg \$2 = \$0) \\
&\equiv (\$0 = 5 \wedge \$1 > 5 \wedge \$2 = \$0) \wedge ((\$0 = 5 \wedge \$1 > 5) | \wedge \$2 = \$0) \\
&\equiv \$0 = 5 \wedge \$1 > 5 \wedge \$2 = 5 \\
&= {c_2}'
\end{aligned}
$$

Similarly, Boolean condition c_3 can be simplified into $(\neg(\$0 = 5 \wedge \$1 > 5) \wedge \$2 = \$0)$, whereas Boolean condition c_4 can be simplified into $(\neg \$2 = \$0)$.

The functionality offered by Z3, or more specifically by Z3's tactics, allows this to be carried out when checking the satisfiability of the given set of goals, where the sub-goals returned are in their most simplified form. This not only permits the external satisfiability solver to perform better in subsequent computations but may also result in an increased number of discarded sub-conditions during Boolean consolidation.

For example, recall the resulting constrained monitor-set in (1). As already mentioned, none of the Boolean sub-conditions can be removed. However, if the Boolean constraint c_2 was replaced by the simplified Boolean condition ${c_2}'$, we end up with a simpler and more condensed constrained monitor-set. Clearly, \$1 is the only free variable in the monitor-set. Hence, this Boolean condition can now be partitioned into p and q, where p and q are equal to $(\$1 > 5)$ and $(\$0 = 5 \wedge \$2 = 5)$ respectively since $\mathbf{fv}(p) \subseteq \mathbf{fv}(M')$ and $\mathbf{fv}(q) \cap \mathbf{fv}(M') = \emptyset$. Subsequently, the latter can be discarded, resulting in the constrained monitor-set $\langle \$1 > 5, \{M'\} \rangle$, where the number of Boolean constraints is reduced even further.

After this optimisation was taken into effect, evaluation was carried out again, resulting in improved running times as shown in the figure below.

2.3 Satisfiability Combinations Optimisation

The current automation tool relies on the external satisfiability component when computing the set of satisfiability combinations. Nonetheless, since as already seen in the previous section, the satisfiability solver is an expensive process, its invokation should be kept to a minimum.

If implemented naively, the external satisfiability solver must be invoked for each possible combination, whereby the number of combinations depend on the number of relevant

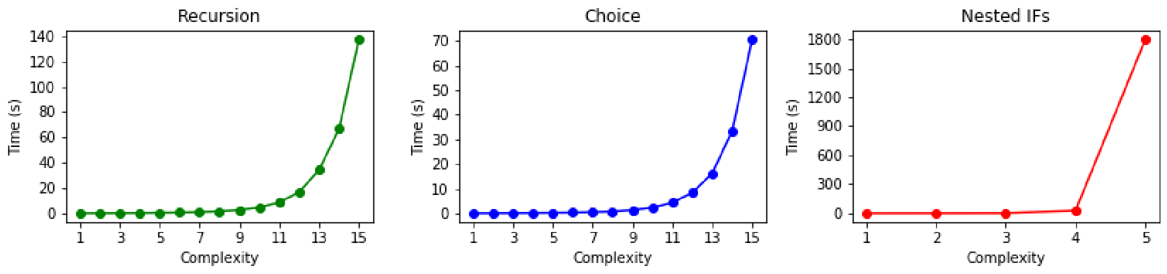


Figure 9: Mean Running Time for Different Monitors

conditions. For example, given 10 relevant conditions, there are $2^{10} = 1024$ possible combinations. Hence, the satisfiability solver must be invoked 1024 times to check the satisfiability of each. If the number of relevant conditions is increased to 15, then the number of possible combinations go up to $2^{15} = 32768$, meaning that now the number of times the satisfiability solver must be invoked has increased by 3100%.

Taking a closer look at the set of satisfiability combinations which are generated naively, it is clear that many of them cannot stand. However, in certain cases, determining whether a Boolean condition is satisfiable or not is not hard. In the light of this observation, if the satisfiability solver is limited only to these extreme cases, there will be a substantial gain in the artifact's responsiveness.

Thus, throughout this section, various optimisation techniques shall be explored with the aim of reducing the use of this external component.

2.3.1 Equality Partitioning

▶ **Example 4.** Consider the choice monitor m from Table 1, having complexity 2.

$$m = (\texttt{rec}X.k\langle 1\rangle.X) + (\texttt{rec}X.k\langle 2\rangle.X) + (\texttt{rec}X.k\langle 3\rangle.X)$$

The monitor-set composed of monitor m in the constrained monitor-set, $\langle \texttt{true}, \{m\}\rangle$, can potentially analyse the symbolic event $k\langle \$0\rangle$ along the Boolean conditions $b_1 = (\$1 = 1)$, $b_2 = (\$1 = 2)$, and $b_3 = (\$1 = 3)$. Naively computing the set of satisfiability combinations, $\textbf{sc}(\texttt{true}, \{b_1, b_2, b_3\})$, the following combinations are generated, ensued by the satisfiability checking of each.

$$c_1 = (\$1 = 1) \wedge (\$1 = 2) \wedge (\$1 = 3) \qquad c_2 = (\$1 = 1) \wedge (\$1 = 2) \wedge \neg(\$1 = 2)$$
$$c_3 = (\$1 = 1) \wedge \neg(\$1 = 2) \wedge (\$1 = 2) \qquad c_4 = \neg(\$1 = 1) \wedge (\$1 = 2) \wedge (\$1 = 2)$$
$$c_5 = (\$1 = 1) \wedge \neg(\$1 = 2) \wedge \neg(\$1 = 3) \qquad c_6 = \neg(\$1 = 1) \wedge \neg(\$1 = 2) \wedge (\$1 = 3)$$
$$c_7 = \neg(\$1 = 1) \wedge (\$1 = 2) \wedge \neg(\$1 = 3) \qquad c_8 = \neg(\$1 = 1) \wedge \neg(\$1 = 2) \wedge \neg(\$1 = 2)$$

Even though only c_5, c_6, c_7, and c_8 are returned by the satisfiability combination function, the external satisfiability component is invoked eight times regardlessly, one for each combination generated. ◀

Having a closer look at Boolean combinations $c_1 \dots c_8$ from the previous example, it is immediately obvious that multiple combinations can be ruled out right away. It is obvious that \$1 cannot be equal to more than one *different* value at once. In other words, b_1, b_2, and b_3 are *mutually exclusive*, meaning that for any one of the three to be true, all the others must be false. More specifically, for their combination to be satisfiable, only *one* or *none* of them must be true.

This immediately rules out all combinations composed of the aggregation of two or more mutually exclusive Boolean conditions—in the example above, combinations $c_1 \dots c_4$ can be dimissed at once, reducing the number of times the satisfiability solver is invoked by half.

```
procedure SC(b,cs)
    result = [ ]
    partition cs into (X,Y) s.t. X contains var assignments and Y all other expressions
    let $nCn$ = [$\wedge$(negate all x in X)]
    let $nCn\_minus\_1$ = X
    first = concat($nCn$, $nCn\_minus\_1$)
    let second = all possible combinations for Y
    for c in second do
        for c' in first do
            if sat ([b,c,c']) then add to result
            end if
        end for
    end for
end procedure
```

Algorithm 1: Pseudocode for the optimised function $\mathbf{sc}(b, \{c_1, ..., c_n\})$

Subsequently, the current implementation generating the satisfiability combinations was altered in such a way that combinations composed of mutually exclusive sub-conditions are not even generated in the first place. The pseudocode for this optimised function, given in detail in Alg. 1, works by first partioning the set of Boolean conditions into two, where the first partition consists solely of variable assignments i.e. expressions of the form $x = n$ for some $x \in \textsc{Var}, n \in \textsc{Nat}$, whereas the second partition contains all other expressions.

For the second partition, all the possible combinations are generated in the exact same manner as in the naive implementation. However, a different approach is applied to the first partition. Since all the Boolean conditions in this set are mutually exclusive, either one of them is true, or all of them are false, giving the set of possible ways they can be combined together. Then, the resulting sets emerging from both are merged back together by computing their cross product (since all possible combinations are given by exploring all the different ways of choosing one from each set). The satisfiability of the merged combinations are checked and the final set of satisfiability combinations is returned. The inner workings of this optimised function are better demonstrated in Example X.

▶ **Example 5**. Consider the function $\texttt{sc}(\texttt{true}, c)$, where c is as following.

$$c = \{x = 1, x = 2, x = y, y < 5 \land x = 7\}$$

In the optimised technique, the set of Boolean conditions given by c is partitioned into X and Y, where $X = \{x = 1, x = 2\}$ and $Y = \{x = y, y < 5 \land x = 7\}$. The expressions in the the first partition are mutually exclusive. Hence, the resulting possible combinations of X are given by the set $c' = \{c_1, c_2, c_3\}$, whereas the possible combinations of Y are given by the set $c'' = \{c_4, c_5, c_6, c_7\}$, where $c_1 \ldots c_7$ are as below.

$$\begin{array}{ll} c_1 = (x = 1) \quad c_2 = (x = 2) & c_3 = \neg(x = 1) \land \neg(x = 2) \\ c_4 = (x = y) \land (y < 5 \land x = 7) & c_5 = \neg(x = y) \land (y < 5 \land x = 7) \\ c_6 = (x = y) \land \neg(y < 5 \land x = 7) & c_7 = \neg(x = y) \land \neg(y < 5 \land x = 7) \end{array}$$

The cross product of the Boolean combinations given by c' and c'' is then computed and the satisfiability of each is checked.

If this set of satisfiable combinations is generated naively, the satisfiability solver must be invoked $2^4 = 16$ times. However, using the optimisation technique discussed, the number of times this external component is invoked is reduced to 12. ◀

Since a fresh variable is introduced each time, it is guaranteed that all the Boolean conditions in the first partition are dependent on the same variable, meaning that all the literals are related to one another. There are two ways that the literals can be related to each other; either they are equal or they are distinct. However, Boolean expressions of the latter form are preempted when computing the set of relevant conditions along which a symbolic event may occur, leaving only the possibility that they are different from one another, which in turn forces all the expressions to be mutually exclusive.

If the first partition was to contain variable assignments with the value being some other variable, then this optimisation technique would no longer be correct. The reason for this is that the relationship between these two values would differ because the two expressions may possibly cease from being mutually exclusive, which is the foundation of this method. For instance, consider the conditions $x = y$ and $x = z$. These two expressions may or may not be mutually exclusive, depending on the values of y and z.

Quantitatively analysing this optimisation, given a set of conditions of length k, if they can be partitioned into X and Y where $|X| = n > 0$, $|Y| = m$, and $k = m + n$, the number of times the external satifiability component is invoked is reduced from $2^k = 2^m.2^n$ to $2^m.(n+1)$. The percentage decrease with respects to the size of the first partition, namely n, is better illustrated in the figure below.

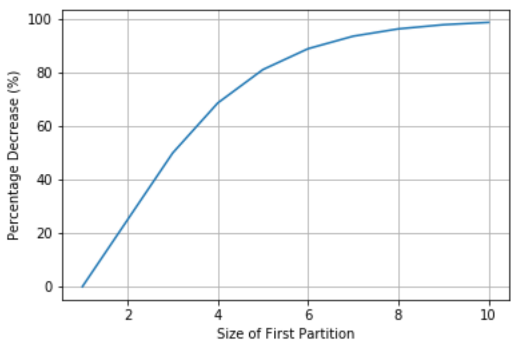


Figure 10: Percentage decrease with respects to the size of the first partition

The evaluation of the tool implementing the optimisation technique discussed was carried out again, resulting in a substantial improvement in the mean running times, especially for the first two monitor descriptions considered.

2.3.2 Partitioning Conditionals

Even though there was a signifcant improvement in the running times of both *choice* and *recursion* monitors when bringing into effect the optimisation technique, the performace

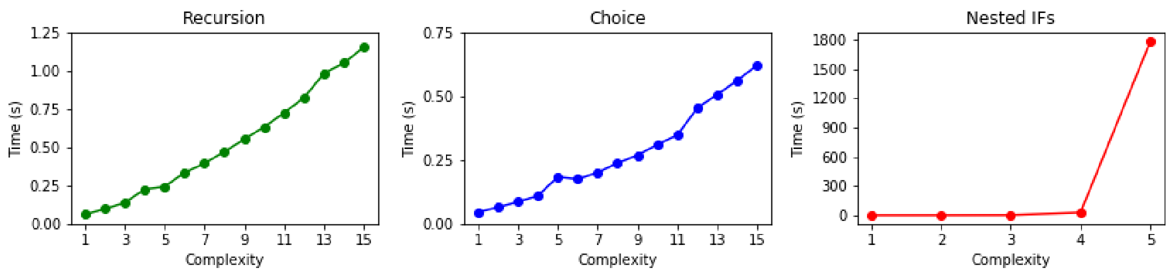


Figure 11: Mean Running Time for Different Monitors

of *conditions* monitors remained unaltered. Thus, an optimisation technique with the aim of adressing this problem will be explored.

▶ **Example 6.** Recall the *conditional* monitor given in Table X, having complexity 1, where $m = l(x).m'$ and $m' = \texttt{if}\ (x = 4)\ \texttt{then}\ k\langle x\rangle.\bot\ \texttt{else}\ k\langle x\rangle.\top$, and consider the induced monitor-set $M = \{m'\ [\$0/x]\}$. Monitor-set M can potentially analyse the symbolic event $k\langle\$1\rangle$. The set of relevant conditions of this monitor-set w.r.t. to this symbolic event, $\mathbf{rc}(M, k\langle\$1\rangle)$ is given by the set $\{(\$0 = 4) \wedge (\$1 = \$0), \neg(\$0 = 4) \wedge (\$1 = \$0)\}$. ◀

It is key for the following optimisation technique to observe that the set of relevant conditions is stored as a *flat structure*. A negative side effect introduced by the use of such a structure as a basis, is a form of *information loss* about the data stored within. The witstanding representation fails to provide any degree of relationship between the respective Boolean conditions, in the sense that it does not give any indication of whether two Boolean conditions can stand simultaneously or not.

For instance, in Example 6, it is obvious that the two relevant conditions afforded cannot be simultaneously true because one condition is given by the monitor's *if* branch, whereas the latter is given by the *else* branch of that *same* conditional monitor. However, by simply looking at the set of relevant conditions, there is no indication that these two Boolean conditions are mutually exclusive and that the occurence of one prohibits that of the other.

An alternative approach which supports a higher degree of information extraction about the data stored within, is through the use of tree representation. Hence, for the scope of this optimisation technique, the grammar rules defining expressions were modified slightly by adding the rule *expression tree* as shown in Fig. 12.

Expression trees are based on the concept of binary trees. The root of the tree is an expression itself and it has a list of children on both the left and right hand side (instead

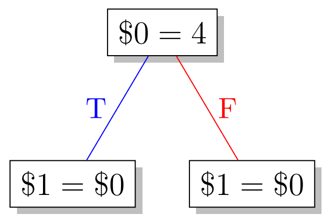
Expressions

$$\begin{array}{llll} e, e', e'' \in \textsc{Exp} ::= x & \text{(identifier)} & \mid n & \text{(literal)} \\ \quad\mid e' + e'' & \text{(binary expression)} & \mid \neg e' & \text{(unary expression)} \\ \quad\mid \langle e, [e'], [e''] \rangle & \text{(expression tree)} & & \end{array}$$

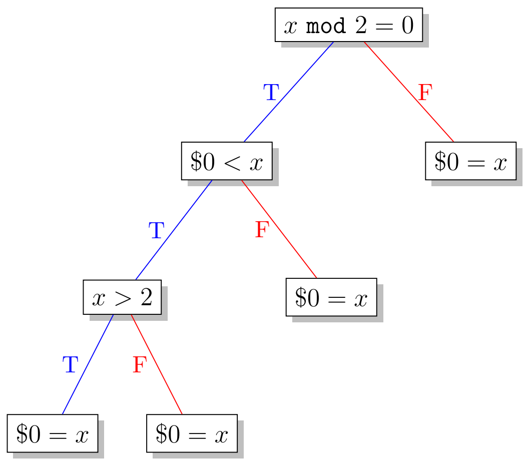
Figure 12: Extended syntax for Expressions

of just one). However, the left children can only be reached if the parent condition is true. Similarly, the right children can only be reached if the parent condition is false. This forces the left and the right children to be mutually exclusive. Subsequently, when generating the set of relevant conditons along which a symbolic event may occur, if the monitor under scrutiny is a conditional monitor, the respective Boolean condition is stored in an expression tree instead.

▶ **Example 7.** To better illustrate the inner mechanisms of the modified function RC, recall the monitor decription in Example 6 and consider monitor-set M. The set of relevant conditions along which the symbolic event $k\langle \$1 \rangle$ may occur w.r.t. to monitor-set M is now given by the set $\{b\}$ where $b = \langle \$0 = 4, [\$1 = \$0, [\,]], [\$1 = \$0, [\,]] \rangle$, as depicted in Fig. 13a. ◀



(a) Relevant Condition



(b) Relevant Condition

Figure 13: Conditions stored in a Binary Tree Structure

The optimised function computing the set of satisfiability combinations SC() relies on another key function, TRAVERSE(), whose pseudocode is laid out in Alg. 2. As suggested

```
procedure TRAVERSE(e: ExpTree)
    procedure GETPATHS(e': Exp list)
        paths = [ ]
        if e' is not empty then
            for x in e' do
                add TRAVERSE(x) to paths
            end for
        end if
        return cartesian product of all p in paths
    end procedure
    x = add (e.condition) to each p in GETPATHS(e.true)
    y = add ¬(e.condition) to each p in GETPATHS(e.false)
    return concat(x, y)
end procedure
```

```
procedure SC(b,cs)
    paths = [ ], result = [ ]
    partition cs into (X,Y) s.t. X contains expression trees and Y all other expressions
    for x in X do
        add TRAVERSE(x) to paths
    end for
    let k = cartestian product of all p in paths
    let $nCn$ = [∧(negate all x in k)]
    let $nCn\_minus\_1$ = X
    let first = concat($nCn$, $nCn\_minus\_1$)
    let second = all possible combinations for Y
    for c in second do
        for c' in first do
            if sat ([b,c,c']) then  add to result
            end if
        end for
    end for
end procedure
```

Algorithm 2: Pseudocode for function SC using Expression Trees

by its name, this function, which takes as parameter an expression, traverses the expression tree and returns a list of expressions, all of which are mutually exclusive since they are all products of the same expression tree. If at any point during traversal, more than one child is encountered along either a *true* or an *else* branch, the cartesian product of all the resulting paths from each is calculated since all the paths resulting from different children (at the same branch) can be combined together.

▶ **Example 8**. To better demonstrate how the function TRAVERSE() works, consider the expression trees depicted in Fig. 13. ◀

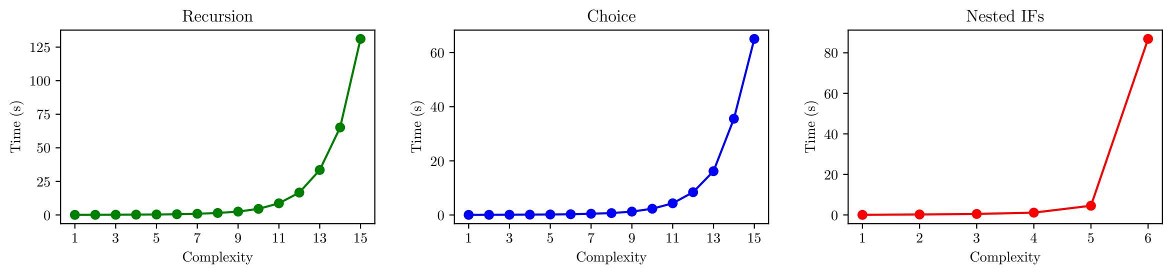


Figure 14: Mean Running Time for Different Monitors

#RelevantConditions	#SatCombinations	#OptimisedSatCombinations
2	4.000000e+00	2
6	6.400000e+01	8
11	2.048000e+03	40
17	1.310720e+05	240
24	1.677722e+07	1680
32	4.294967e+09	13440
41	2.199023e+12	120960
51	2.251800e+15	1209600
62	4.611686e+18	13305600

Table 1: Number of combinations using sc and osc w.r.t. number of relevant conditions

References

[1] K. Sen, A. Vardhan, G. Agha, and G. Rosu, "Efficient Decentralized Monitoring of Safety in Distributed Systems," in *Proceedings. 26th International Conference on Software Engineering*, pp. 418–427, IEEE, 2004.

[2] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An Overview of the MOP Runtime Verification Framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[3] A. Francalanza, "Consistently-Detecting Monitors," in *28th International Conference on Concurrency Theory (CONCUR 2017)* (R. Meyer and U. Nestmann, eds.), vol. 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 8:1–8:19, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[4] D. Sangiorgi, *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

[5] A. Bondi, "Characteristics of Scalability and Their Impact on Performance," pp. 195–203, 01 2000.