Model Checking Safety and Liveness via k-Induction and Witness Refinement with Constraint Generation

Nils Timm*, Stefan Gruner, Madoda Nxumalo, Josua Botha

Department of Computer Science, University of Pretoria, South Africa

Abstract

In this article, we revise our constraint-based abstraction refinement technique for checking temporal logic properties of concurrent software systems. Our technique employs predicate abstraction and SAT-based three-valued bounded model checking. In contrast to classical refinement techniques where a single state space model is iteratively explored and refined with predicates, our approach is as follows: We use a coarsely-abstracted global state space model where we check for abstract witness paths for the property of interest. For each detected abstract witness we construct a local model whose state space is restricted to refinements of the witness only. On the local models we check whether the witness is real or spurious. We eliminate spurious witnesses in the global model via *spurious segment constraints*, which do not increase the state space complexity. Our technique is complete and terminates when a real witness in a local model can be detected, or no more witnesses in the global model exist.

While our technique was originally restricted to the verification of *safety* properties, we extend it here to the verification of *liveness* properties. For this, we make use of the state recording translation of the input system, which reduces liveness model checking to safety checking. Another restriction of our original approach was its incompleteness due to the nature of *bounded* model checking. Here we show how abstraction refinement-based bounded model checking can be combined with the k-induction principle, which enables unbounded model checking. Our approach is iterative with regard to the bound. The extended approach also allows us to define enhanced concepts for *strengthening* the constraints that we use to rule out spurious behaviour and for *reusing* constraints between bound iterations. We demonstrate that our approach enables the complete verification of safety and liveness properties with a reduced state space complexity and a better solving time in comparison to classical abstraction refinement techniques.

Key words: Three-valued abstraction, SAT-based bounded model checking, Constraint generation, k-induction, Liveness

*Corresponding author

Email addresses: ntimm@cs.up.ac.za (Nils Timm), sg@cs.up.ac.za (Stefan Gruner)

1. Introduction

Three-valued abstraction (3VA) [1] is an established technique for reducing the complexity of software verification. It proceeds by generating an abstract representation of an input software system over predicates with the possible truth values *true*, *false* and *unknown*, where the latter value is used to express the loss of information due to abstraction. The state space of an abstract software system can then be represented as a three-valued model. The evaluation of temporal logic properties on such models is known as *three-valued model checking* (3MC) [2]. Under three-valued abstraction both *true* and *false* model checking results can be transferred to the modelled software system, whereas an *unknown* result indicates that the current model is too coarse for a definite outcome. In the latter case a so-called *unconfirmed witness* is produced, which is an execution path in the abstract state space with some *unknown* transitions or predicates that characterises a potential violation of the property of interest. *Witness-guided abstraction refinement* [3] then iteratively adds further predicates to the abstract model until a previously *unconfirmed* witness turns out to be *definite*, or no more witnesses exist. The described approach follows the classical *abstract–check–refine* paradigm [4] where a single model that represents the entire system is iteratively refined. Since each refinement iteration involves an exponential growth of the state space to be explored, this approach can easily suffer from state explosion.

In [5] we introduced a novel abstraction refinement technique that facilitates the verification of safety properties of concurrent software systems with an improved state space complexity. In this approach we make use of two kinds of state space models: We use a *global model* that considers all parts of the underlying system, and we use *local models* that are restricted to previously detected unconfirmed witness paths. Both global and local models are subject to three-valued abstraction. But only the local models are refined by adding predicates, whereas the global model is iteratively pruned via *spurious witness constraints* derived from local models. Our technique proceeds as follows: In the same manner as in a classical *abstract–check–refine* approach, we start with a coarsely-abstracted global model of the input system and we check whether the safety property of interest can be proven or refuted. If the check returns *unknown* along with an unconfirmed witness, then we derive new predicates for refinement. Now instead of refining the global model, we construct a new local model that is narrowed down to refinements of the unconfirmed witness only. Checking the local model either proves the previously unconfirmed witness to be *definite* or to be *spurious*. In the first case we are done. In the latter case we generate a constraint for ruling out the spurious witness. In the subsequent pruning iteration we return to the global model and prune its state space via the generated *spurious witness constraint*. The procedure terminates when either no more witnesses in the global model exist or a definite witness in a local model can be detected. In contrast to standard approaches to constraint reusing in bounded model checking [6] our technique does not only enable the reuse of constraints between bound iterations but also between different levels of abstraction. A constraint generated based on a refined local model is also admissible for the more abstract global model. Our approach reduces the state space complexity in two ways. Refinement is only applied to *local* models whose state space is already strongly limited by being restricted to refinements of a certain unconfirmed witness. The state space of the global model is pruned by spurious witness constraints derived from local models. But the refinement predicates that were used in the local model in order to derive these constraints do *not* have to be added to the global model. Hence, we gain precision in the global model without enlarging its state space. The price that we pay is an increased number of global pruning iterations and local refinement iterations until a definite result can be obtained. The actual number depends on the *strength* of the generated constraints in terms of ruling out spurious behaviour. The spuriousness of a witness typically originates from a *spurious segment* of the path that it represents. A constraint that rules out all paths that exhibit the *spurious segment* is naturally stronger than a constraint that only rules out the spurious witness itself.

As a background technique we use *satisfiability-based three-valued bounded model checking* [7] in order to process the model checking problems to be solved within our abstraction refinement approach. We have shown that the abstracted input systems together with the safety property to be checked can be directly encoded into propositional logic such that the construction of an explicit state space model is avoided [7]. Encoded three-valued bounded model checking problems can be solved via two Boolean satisfiability checks. The first check considers an over-approximating completion of the encoding where all *unknowns* are assumed to be *true*. The second check considers an under-approximating completion where all *unknowns* are assumed to be *false*. If both completions are satisfiable, then the corresponding model checking result is *true*. If both completions are unsatisfiable, then the corresponding model checking result is *false*. Otherwise the result is *unknown*. In [5] we demonstrated that in the satisfiability-based approach a spurious segment of a witness is characterised by a part of an unsatisfiable core of the SAT-encoded problem. The constraint for ruling out the spurious segment corresponds to the negation of this part. This enables us to efficiently generate spurious segment constraints via *unsatisfiable core extraction* [8]. Bounded model checking, as we used it in [3, 5, 7], is inherently incomplete. The bound $k \in \mathbb{N}$ restricts the length of execution paths of the modelled system, which makes this technique only usable for detecting property violations but not for proving their absence. Completeness for finite-state systems can be theoretically established by iterating over the bound until a *completeness threshold* is reached [9]. However, the determination of minimal or tight (close to minimal) completeness thresholds is a computationally hard problem by itself and even tight thresholds are mostly still impractical for efficient verification. In [5] we added a bound iteration loop around our abstraction refinement technique. The loop is *incremental* [10] in the sense that clauses learned by the solver for a particular model M in iteration k can be reused in for pruning the search space of the *same* model in iteration $k + 1$. In addition, our novel approach allows for further pruning based on spurious segment constraints. We proved that *iteration-independent* spurious segment constraints derived from *any* local model can be used for restricting the search space of the global model in all bound iterations. The admissibility of reusing constraints between different models gives us pruning capabilities that are beyond what is feasible with standard incremental SAT solving [10]. The approach allows us to avoid computational overhead caused by repeating constraint generation that has been already conducted based on local models in previous iterations

While our novel refinement approach [5] already revealed promising results in terms of reducing the state space complexity of verification tasks in comparison to classical abstraction refinement [3], it was still subject to a number of drawbacks and limitations. Our original definition of iteration-independence was overly restrictive and resulted in a very limited reusability of constraints. In this extended article we define seven different types of independence (resp. dependence) of spurious segment constraints and we prove an admissible form of reuse for each type. This also includes the additional strengthening of *fully-independent* constraints and the adaptation of *initial state-independent* constraints for reuse in higher bound iterations. The type of independence of a constraint follows immediately from unsatisfiable core extraction. Our enhanced constraint reusing concept facilitates a more extensive pruning of spurious behaviour, and thus, a better verification performance.

A second limitation of [5] is that it is incomplete due to the nature of bounded model checking. Although we use a bound iteration loop, this does not allow us to prove the absence of property violations in large-scale state space models. Completeness of bounded model checking can be established via k*-induction* [11]. This technique was originally introduced for the verification of safety properties of hardware systems. It proceeds as follows: Given a state space model of the system to be analysed and a safety predicate *safe*, it is checked whether all paths of length k that start in an initial state of the model are safe, i.e. whether *safe* holds in each state along the paths. This is the *base case* of k-induction, which is equivalent to standard bounded model checking. If the base case holds, then the *inductive step* is checked: Assuming k consecutive states where *safe* holds in each state, then *safe* also has to hold in every $(k+1)$-st successor state. The inductive step does not restrict the k consecutive states to start in an initial state. If the inductive step holds as well, then it can be concluded that all *unbounded* execution paths of the modelled system are also safe. Otherwise the procedure needs to be repeated with an incremented k. For finite-state models termination is guaranteed and the final bound is typically considerably smaller than a precomputed approximation of a completeness threshold. In [12] we already demonstrated that k-induction is compatible with three-valued bounded model checking. In this article we show that the k-induction technique can be also combined with our novel abstraction refinement approach. For this, we introduce a shared bound iteration loop and within this loop two separate refinement loops, one for the base case and one for the inductive step. This combination enables us to conduct *complete* verification of safety properties via SAT-based model checking. The base case and the inductive step are two distinct problems to be solved. Although they exhibit certain similarity, constraint reuse between the two is not admissible in general. However, we show that generated *initial state-independent* constraints can be reused between the base case and the inductive step, which gives us further pruning capabilities.

k-induction is limited to the verification of *safety* properties and so is our original approach presented in [5]. For concurrent systems also *liveness* properties are of great importance. Liveness model checking under fairness involves a considerably higher complexity than safety checking and many safety checking techniques are not compatible with liveness. In order to facilitate the verification of liveness properties with our approach, we adopt the *state recording translation* [13]. This translation transfers an original state space model into a state recording model, which reduces an original liveness model checking problem to a safety problem. The translation comes at the cost of a quadratic increase of the number of states, but it gives us the benefit to utilise efficient safety checking techniques in order to solve liveness problems. We show that the state recording translation can be already applied to our abstracted systems *before* the corresponding state space models are encoded in propositional logic.

1.1. Contributions and Relation to Previous Work

The main contributions of this work are the establishment of considerably enhanced constraint reusing capabilities for our automatic abstraction refinement approach proposed in [5], the completion of previously incomplete verification techniques [3, 5, 7], and the extension of a pure safety checking technique to a technique that also supports liveness checking under fairness. Table 1 provides an overview on how this article, denoted as SCP 2020, extends and combines our previous work on SAT-based three-valued bounded model checking.

Table 1: Comparison with previous work.

	[7]	[3]	[12]	[5]	SCP 2020
liveness checking	✓	✓	✗	✗	✓
automatic refinement	✗	✓	✓	✓	✓
multi-model approach	✗	✗	✗	✓	✓
reuse of path constraints	✗	✗	✗	✓	✓ (enhanced)
completeness	✗	✗	✓	✗	✓

[7] and [3] support liveness model checking based on an *explicit* encoding of the liveness property to be checked, which turned out to be inefficient in experiments. Thus, in subsequent papers we focussed on safety model checking only. SCP 2020 re-introduces liveness support based on a *reduction* to safety. In [3] we introduced a fully-automatic abstraction refinement technique. Predicates for refinement are derived from unsatisfied clauses. Refined models are generated with the help of the prover Z3 [14]. The originally single-model approach to abstraction refinement has been extended to a multi-model approach

in [5]. This also involved the introduction of path constraint reusing, which we enhance in SCP 2020. Based on thresholds, complete model checking is theoretically conceivable but practically infeasible in [7], [3], [5]. The k-induction approach allows for the feasibility of complete verification in [12] and SCP 2020.

We have implemented our approach. In experiments we demonstrate that our constraint-based refinement technique allows for significant performance improvements in comparison to classical abstraction refinement. Moreover, we show that our implemented tool can compete with the Spin model checker [15] for certain verification tasks.

1.2. Outline

The remainder of this article is organised as follows. In Section 2 we introduce the concurrent software systems that we consider in our approach and the three-valued abstraction technique that we employ. Section 3 provides the background on three-valued bounded model checking. In Section 4 we show how the state recording translation can be applied to our abstracted systems in order to reduce liveness to safety model checking. In Section 5 we show how k-induction can be combined with three-valued bounded model checking in order to establish completeness. Section 6 reviews basic three-valued abstraction refinement. In Section 7 we introduce our novel witness refinement technique with constraint reuse. We show in this section how the novel refinement technique can be combined with k-induction. Moreover, we define the different types of constraints that occur in our approach and we prove an admissible form of reuse and strengthening for each type. In Section 8 we introduce our propositional logic encoding of three-valued bounded model checking problems. Furthermore, we show how constraint types can be determined based on unsatisfiable core extraction. In Section 9 we introduce the implementation of our approach and we present experimental results. Section 10 discusses related work. We conclude this paper in Section 11 and give an outlook on future work.

2. Abstracted Concurrent Software Systems

We start with a brief introduction to the systems that we want to verify and the abstraction technique that we use in our work. Our approach supports integer arithmetic-based concurrent systems with the data types *int*, *bool* and *semaphore* (but no arrays and pointers). Moreover, almost all control structures of the C language are supported, such as *if-then-else*, *while-do*, *for* and *goto*. A *concurrent software system* Sys consists of a number of possibly non-uniform processes P_1 to P_n composed in parallel: $Sys = \|_{i=1}^{n} P_i$. It is defined over a set of variables $Var = Var_{Sys} \cup Var_{PC}$. Var_{Sys} is a set of typed system variables, whereas Var_{PC} is a special set that holds for each process P_i a dedicated program counter variable pc_i ranging over control locations from a set Loc_i. Locations of a process are labelled with conditional operations with regard to system variables and with a reference to the subsequent location.

An example for a concurrent system implementing mutual exclusion is depicted in Figure 1.

y : `semaphore where` $y = 1$;

```
P1 :: [ loop forever do                  ]  || P2 :: [ loop forever do                  ]
      [   [ 0: acquire (y);     ]        ]          [   [ 0: acquire (y);     ]        ]
      [   [ 1: CRITICAL         ]        ]          [   [ 1: CRITICAL         ]        ]
      [   [      release (y);   ]        ]          [   [      release (y);   ]        ]
```

Figure 1: Concurrent system *Sys*.

Here we have two processes operating on a shared semaphore variable y. Processes P_i can be formally represented as *control flow graphs* (CFGs) $G_i = (Loc_i, \delta_i, \tau_i)$ where $Loc_i = \{0, \ldots, |Loc_i - 1|\}$ is a set of control locations given as numbers. We implicitly assume that 0 is the initial location of a control flow graph. $\delta_i \subseteq Loc_i \times Loc_i$ is a transition relation and $\tau_i : \delta_i \to Op$ is a function labelling transitions with operations from a set Op.

Definition 1 (Operations).
Let $Var = \{v_1, \ldots, v_m\}$ *be a set of typed variables. The* set of operations Op *on* Var *consists of all statements of the form* $assume(e) : [v_1 := e_1] \circ \ldots \circ [v_m := e_m]$ *where* e *is a Boolean expression over* Var *that acts as a guard. Moreover,* $\circ$ *is the* append *operator and* $[\ldots] \circ \ldots \circ [\ldots]$ *is a list of type-correct assignments where* $e_1, \ldots e_m$ *are expressions over* Var*.*

Hence, every operation consists of a guard and a list of assignments. For convenience, we sometimes just write e instead of $assume(e)$, we omit the *assume* part completely if the guard is *true*, and we write $v_1 := e_1, \ldots, v_m := e_m$ for an assignment list $[v_1 := e_1] \circ \ldots \circ [v_m := e_m]$. The control flow graphs G_1 and G_2 corresponding to the processes of our example system are depicted in Figure 2. G_1 and G_2 also illustrate the semantics of the operations $acquire(y, 1)$ and $release(y, 1)$.

y : `semaphore where` $y = 1$;

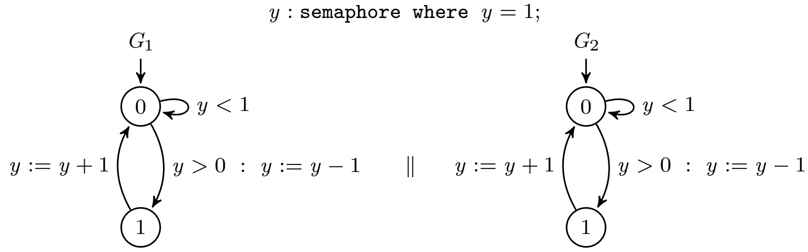


Figure 2: Control flow graphs G_1 and G_2 composed in parallel.

A concurrent system given by n individual control flow graphs $G_1, \ldots, G_n$ can be modelled by one composite CFG $G = (Loc, \delta, \tau)$ where $Loc = \times_{i=1}^{n} Loc_i$. G is the product graph of all individual CFGs. We assume that for any *Sys* a deterministic initialisation of all its variables is given in terms of a predicate expression *Init*, e.g., $Init = (y = 1) \wedge (pc_1 = 0) \wedge (pc_2 = 0)$ for our example. A computation of a concurrent system corresponds to a sequence where in each step one process is non-deterministically selected and an operation at its current location that is not blocked by the guard is executed, i.e. the variables are updated according to the assignment part and the process advances to the consequent control location. We assume that for each step and each control location of a process there will be always at least one non-blocked operation that can be executed. This might be an explicit idling step, such as the waiting for the semaphore to become available at location 0 in Figure 2. We assume that all assignments of an operation are executed simultaneously and that an operation assigns to each variable at most once. The state space of a system over *Var* corresponds to the set S_{Var} of all type-correct valuations of the variables. Given a state $s \in S_{Var}$ and an expression e over Var, then $s(e)$ denotes the valuation of e in s. Thus, a computation can be likewise considered as sequence of states $s_0 s_1 s_2 \ldots$ where the transition from s_i to s_{i+1} correctly characterises the execution of the associated operation. For verifying properties of concurrent software systems typically only *fair* computations are considered. Our notion of fairness for concurrent systems is as follows: In an infinite computation, each process executes an operation infinitely often.

Control flow graphs allow us to model concurrent systems formally. For an efficient verification it is additionally required to reduce the state space complexity. For this purpose, we use *three-valued predicate abstraction* [1]. Such an abstraction is an approximation in the sense that all definite verification results (*true*, *false*) obtained for an abstract system can be transferred to the original system. Only *unknown* results necessitate abstraction refinement [16]. For a given set A_{Sys} of atomic predicates over system variables, the corresponding three-valued abstraction of the system can be automatically constructed via a theorem prover. In our approach we employ the prover Z3 [14]. In abstract systems operations do not refer to concrete system variables from Var_{Sys} but to predicates from a set A_{Sys} with the three-valued domain $\mathfrak{Z} = \{true, false, unknown\}$ which we typically abbreviate by $\{t, f, u\}$. *Unknown* is used to represent the loss of information due to abstraction and is a valid truth value as we operate with the three-valued *Kleene logic* $\mathcal{K}_3$ [17] whose semantics is given by the truth tables in Figure 3.

$\wedge$	*true*	*u*	*false*
true	*true*	*u*	*false*
u	*u*	*u*	*false*
false	*false*	*false*	*false*

$\vee$	*true*	*u*	*false*
true	*true*	*true*	*true*
u	*true*	*u*	*u*
false	*true*	*u*	*false*

$\neg$	
true	*false*
u	*u*
false	*true*

Figure 3: Truth tables for the three-valued Kleene logic $\mathcal{K}_3$.

The *information order* '$\leq_{\mathcal{K}_3}$' of the Kleene logic is defined as $u \leq_{\mathcal{K}_3} true$, $u \leq_{\mathcal{K}_3} false$, and *true*, *false* incomparable. Operations in abstract systems are of the following form:

$$assume(choice(a, b)) \ : \ p_1 := choice(a_1, b_1), \ \ldots, \ p_m := choice(a_m, b_m)$$

where $\{p_1, \ldots, p_m\} = A_{Sys}$ is a set of predicates and $a, b, a_1, b_1, \ldots, a_m, b_m$ are logical expressions over A_{Sys}. $choice(a, b)$-expressions have the following semantics:

Definition 2 (Choice Expressions).
Let s be a state over a set of three-valued predicates A_{Sys}. Moreover, let a and b be logical expressions over A_{Sys} that may evaluate to true false or unknown in s. Then

$$s\left(choice\left(a, b\right)\right) = \begin{cases} true & \textit{if, and only if,} \;\; s(a) \text{ is } true, \\ false & \textit{if, and only if,} \;\; s(a) \text{ is not } true \text{ and } s(b) \text{ is } true, \\ u & \textit{else.} \end{cases}$$

Thus, predicates in A_{Sys} may be set to *unknown* by an abstract operation. Given a concrete operation $assume(e) \;:\; v_1 := e_1, \ldots, v_m := e_m$ and a set of predicates A_{Sys}, a corresponding abstract operation $assume(choice(a, b)) \;:\; p_1 := choice(a_1, b_1),\ \ldots,\ p_{m'} := choice(a_{m'}, b_{m'})$ has to satisfy the following implications $a \models e$, $b \models \neg e$, $a_i \models wp_{op}(p_i)$, and $b_i \models \neg wp_{op}(p_i)$ with $1 \leq i \leq m'$ where a, b, a_i and b_i are logical expressions over A_{Sys} and $wp_{op}(p_i)$ is the weakest precondition of p_i with regard to op. The abstraction of operations can be performed fully-automatically. In our approach we use the prover Z3 [14] for generating abstract operations.

While our abstraction technique reduces the complexity induced by system variables, it preserves the original control flow. For this, the set of 2-valued predicates $A_{PC} = \{(pc_i = l_i) \mid i \in [1, n],\ l_i \in Loc_i\}$ is used that covers all locations of the system. A state s with $s(pc_i = l_i) = true$ means that in s process P_i is at control location l_i. Since a process can be only at one location at a time, $s(pc_i = l'_i)$ must be *false* for all $l'_i \neq l_i$. The overall set of predicates is $A = A_{Sys} \cup A_{PC}$.

The application of three-valued predicate abstraction ensures that for any state s and for any expression $choice(a, b)$ in an abstract control flow graph the following holds: $s(a) = true \;\Rightarrow\; s(b) = false$ and $s(b) = true \;\Rightarrow\; s(a) = false$. Moreover, the following equivalences hold:

$$\begin{array}{lcl} choice(true, false) & \equiv & true \\ choice(false, true) & \equiv & false \\ choice(false, false) & \equiv & u \\ choice(a, \neg a) & \equiv & a \\ choice(\neg a, a) & \equiv & \neg a \\ choice(a, b) & \equiv & (a \vee \neg b) \wedge (a \vee b \vee u) \\ choice(b, a) & \equiv & \neg choice(a, b) \end{array}$$

A three-valued expression $choice(a, b)$ over A_{Sys} approximates a Boolean expression e over Var, written $choice(a, b) \preceq e$, if, and only if, a implies e and

b implies $\neg e$:

$$choice(a, b) \preceq e \;\equiv\; \big(a \models e\big) \;\wedge\; \big(b \models \neg e\big).$$

The three-valued approximation relation can be extended to operations as follows [1]:

Definition 3 (Approximation of Operations).
Let Var be a set of variables and let A_{Sys} be a set of predicates over Var. Moreover, let

$$op \;=\; assume(e) \;:\; v_1 := e_1, \ldots, v_m := e_m$$

be a concrete operation over Var and let

$$op' \;=\; assume(choice(a, b)) \;:\; p_1 := choice(a_1, b_1),\; \ldots,\; p_{m'} := choice(a_{m'}, b_{m'})$$

be an abstract operation over A_{Sys}. Then op' approximates op, written $op' \preceq op$, if and only if

$$choice(a, b) \preceq e \;\wedge\; \bigwedge_{i=1}^{m'} \big(choice(a_i, b_i) \preceq wp_{op}(p_i)\big)$$

where $wp_{op}(p_i)$ is the weakest precondition of p_i with respect to op.

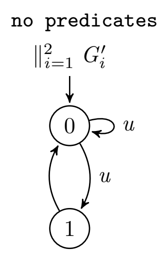
An abstract system Sys' over A_{Sys} approximates a concrete system Sys over Var, written $Sys' \preceq Sys$, if the systems have isomorphic control flow graphs and the operations in the abstract system approximate the corresponding ones in the concrete system. In the same manner we can also define the approximation relation for pairs of abstract systems Sys' over A' and Sys'' over A'' with $A' \subseteq A''$.

Figure 4 depicts different degrees of abstraction of the concrete system Sys in Figure 2. We have that $Sys' \preceq Sys'' \preceq Sys''' \preceq Sys$. For illustration: the abstract operation $(y > 0) := choice((y > 0), false)$ in Sys'' sets the predicate $(y > 0)$ to *true* if $(y > 0)$ was *true* before, and it never sets the predicate to *false*. This is a sound three-valued approximation of the concrete operation $y := y + 1$ over the predicate $(y > 0)$.

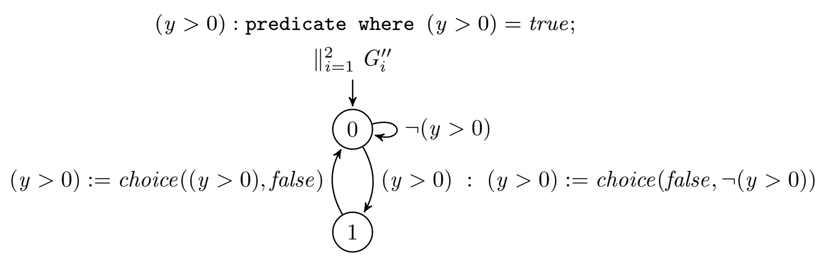
The state space of an abstract system is defined as $S = S_{A_{Sys}} \times S_{A_{PC}}$ where $S_{A_{Sys}}$ is the set of all possible valuations of the three-valued predicates in A_{Sys} and $S_{A_{PC}}$ is the set of all possible valuations of the two-valued predicates in A_{PC}. So far we have seen how concurrent systems can be formally represented and abstracted. Next we will take a look on how bounded model checking of abstracted systems is defined.

3. Three-Valued Bounded Model Checking

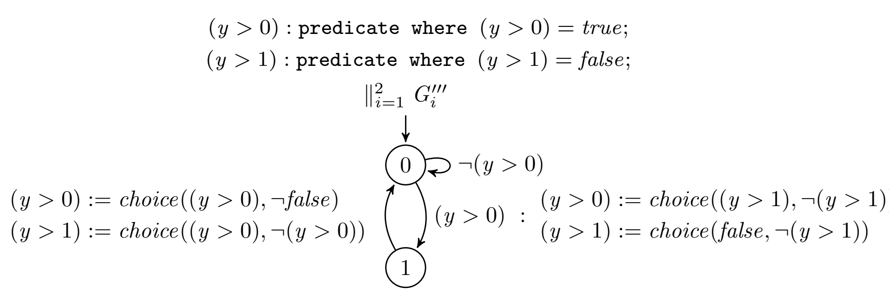
CFGs allow us to model the *control flow* of a concurrent system. The verification of a system additionally requires to explore a corresponding *state space*



(a) CFGs representing an abstract system Sys' over $A_{Sys'} = \varnothing$.



(b) CFGs representing an abstract system Sys'' over $A_{Sys''} = \{(y > 0)\}$.



(c) CFGs representing an abstract system Sys''' over $A_{Sys'''} = \{(y > 0), (y > 1)\}$.

Figure 4: Control flow graphs representing different degrees of abstraction of the concrete system Sys.

model. Since we use three-valued abstraction, we need a model that incorporates the truth values *true*, *false* and *unknown*. *Three-valued Kripke structures* are models with a three-valued domain for transitions and labellings of states:

Definition 4 (Three-Valued Kripke Structure).
A *three-valued Kripke structure* over a set of atomic predicates A is a tuple $M = (S, I, R, L, F)$ where

- S is a finite set of states,
- $I \subseteq S$ is a set of initial states,
- $R : S \times S \to \mathfrak{Z}$ is a transition function such that $\forall\, s \in S : \exists\, s' \in S$ with $R(s, s') \in \{\mathit{true}, \mathit{unknown}\}$,
- $L : S \times A \to \mathfrak{Z}$ is a labelling function that associates a truth value with each atomic predicate in each state,
- $F \subseteq \mathcal{P}(\{(s, s') \mid R(s, s') \in \{\mathit{true}, \mathit{unknown}\}\})$ is a set of fairness constraints where each constraint is a set of *non-false* transitions.

A concurrent system $\mathit{Sys} = \|_{i=1}^{n} P_i$ abstracted over a set of system predicates A_{Sys} can be represented as a three-valued Kripke structure according to the following definition:

Definition 5 (Concurrent System as Three-Valued Kripke Structure).
Let $\mathit{Sys} = \|_{i=1}^{n} P_i$ over Var be a concurrent system abstracted over a set of system predicates A_{Sys} and given by a composite control flow graph $G = (\mathit{Loc}, \delta, \tau)$ and an initial state predicate Init. The corresponding three-valued Kripke structure is a tuple $M = (S, I, R, L, F)$ over the set of atomic predicates $A = A_{\mathit{Sys}} \cup A_{PC}$ where $A_{PC} = \{(pc_i = l_i) \mid i \in [1..n],\ l_i \in \mathit{Loc}_i\}$ with

- $S := S_{A_{\mathit{Sys}}} \times S_{A_{PC}}$,
- $I := \{s \in S \mid s(\mathit{Init}) = \mathit{true}\}$,
- $R(s, s') := \bigvee_{i=1}^{n} R_i(s, s') :=$
  $\bigvee_{i=1}^{n}(\delta_i(l_i, l_i') \wedge \bigwedge_{i' \neq i}(l_{i'} = l_{i'}') \wedge s(\mathit{choice}(a, b)) \wedge \bigwedge_{j=1}^{m} s'(p_j) = s(\mathit{choice}(a_j, b_j)))$
  assuming that l_i is the location of P_i with $s(pc_i = l_i) = \mathit{true}$,
  l_i' is the location of P_i with $s'(pc_i = l_i') = \mathit{true}$ and
  $\tau_i(l_i, l_i') = \mathit{assume}(\mathit{choice}(a, b)) : p_1 := \mathit{choice}(a_1, b_1), \ldots, p_m := \mathit{choice}(a_m, b_m)$,
- $L(s, p) := s(p)$ for each $p \in A$,
- $F := \big\{\{(s, s') \mid R_i(s, s') \in \{\mathit{true}, \mathit{unknown}\}\} \mid i \in [1..n]\big\}$,
  i.e. for each P_i one constraint that contains all transitions caused by P_i.

The abstraction technique that we use guarantees that all states of the resulting three-valued Kripke structure have a unique labelling. Hence, we can assume that a pair of states $s, s' \in S$ with $\forall\, p \in A : L(s,p) = L(s',p)$ implies that $s = s'$.

A three-valued Kripke structure representing the state space of our example system can be defined over the predicate set $A = \{(pc_1 = 0), (pc_1 = 1), (pc_2 = 0), (pc_2 = 1)\}$. Here we only have predicates over the control flow, but so far no predicates over the semaphore variable y. The corresponding Kripke structure is depicted in Figure 5. For the sake of simplicity, only the predicates that evaluate to *true* are shown for each state. Each transition is labelled with its truth value and the identifier of the process that causes it. As we can see, the lack of predicates over y leads to several *unknown* transitions in the model.

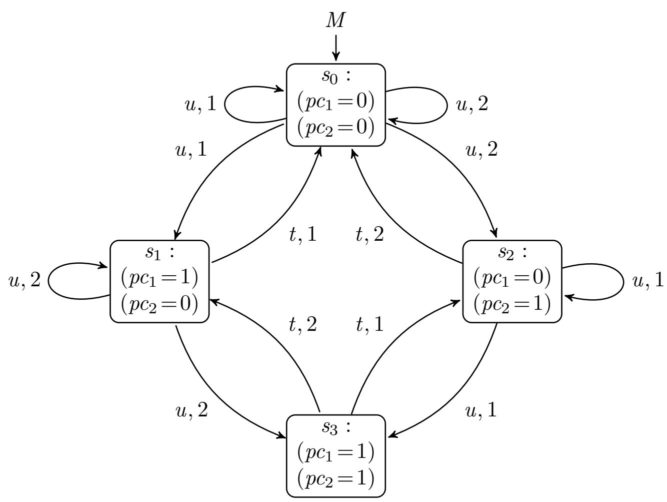


Figure 5: Three-valued Kripke structure corresponding to abstracted mutual exclusion system.

Computations of a modelled system correspond to paths of a Kripke structure:

Definition 6 (Path).
Let $M = (S, I, R, L, F)$ be a three-valued Kripke structure. A *path* π of M is a sequence of states $s_0 s_1 s_2 \dots$ with $s_0 \in I$ and $\forall\, i : R(s_i, s_{i+1}) \in \{\mathit{true}, \mathit{unknown}\}$. π_i denotes the i-th state of π and π^i denotes the i-th suffix $s_i s_{i+1} \dots$ of π. Π_M denotes the set of all paths in M and Π_M^F denotes the set of all infinite paths in M that are fair with regard to F.

A fair path of our example Kripke structure must take infinitely many transitions associated with P_1 and infinitely many transitions associated with P_2. While paths can be generally infinitely long, bounded model checking only looks at finite k-prefixes $\pi_0 \dots \pi_k$ of paths π where $k \in \mathbb{N}$ is the so-called bound. A finite prefix can still represent an infinite path if the prefix has a k-loop.

Definition 7 (k-Loop).
Let π be a path of a three-valued Kripke structure M and let $l, k \in \mathbb{N}$ with $l \leq k$. Then π has a (k,l)-loop if $R(\pi_k, \pi_l) \in \{true, unknown\}$ and π is of the form $v \cdot w^\omega$ where $v = \pi_0 \ldots \pi_{l-1}$ and $w = \pi_l \ldots \pi_k$. π has a k-loop if there exists an $l \leq k$ such that π has a (k,l)-loop.

On prefixes with or without a loop we can evaluate temporal logic properties. Here we use the *linear-time temporal logic* LTL.

Definition 8 (Syntax of LTL).
Let A be a set of atomic predicates. The syntax of LTL formulae *ψ is inductively defined as*

$$\psi \ ::= \ p \mid \neg p \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{G}\psi \mid \mathbf{F}\psi \mid \mathbf{X}\psi,$$

where p denotes arbitrary atomic predicates from A.

The temporal operator $\mathbf{G}$ is read as *globally*, $\mathbf{F}$ is read as *finally* (or *eventually*), and $\mathbf{X}$ is read as *next*. For the sake of simplicity, we omit the temporal operator $\mathbf{U}$ (*until*). Due to the extended domain of truth values in three-valued Kripke structures, the bounded evaluation of LTL formulae is based on the Kleene logic $\mathcal{K}_3$ (compare Section 2). For the bounded evaluation of LTL formulae on paths of three-valued Kripke structures we have to distinguish between prefixes *with* and *without* a k-loop.

Definition 9 (Three-Valued Bounded Evaluation of LTL).
Let $M = (S, I, R, L, F)$ over A be a three-valued Kripke structure. Moreover, let $k \in \mathbb{N}$ and let π be a path of M with *a k-loop. Then the* bounded evaluation *of an LTL formula ψ on the k-prefix of π, written $[\pi \models \psi]_k^i$ where $i \leq k$ denotes the current position along the path, is inductively defined as follows:*

$$\begin{array}{lcl}
[\pi \models p]_k^i & \equiv & L(\pi_i, p) \\
[\pi \models \neg p]_k^i & \equiv & \neg L(\pi_i, p) \\
[\pi \models \psi \vee \psi']_k^i & \equiv & [\pi \models \psi]_k^i \vee [\pi \models \psi']_k^i \\
[\pi \models \psi \wedge \psi']_k^i & \equiv & [\pi \models \psi]_k^i \wedge [\pi \models \psi']_k^i \\
[\pi \models \mathbf{G}\psi]_k^i & \equiv & \bigwedge_{j \geq i}([\pi \models \psi]_k^j \wedge R(\pi_j, \pi_{j+1})) \\
[\pi \models \mathbf{F}\psi]_k^i & \equiv & \bigvee_{j \geq i}([\pi \models \psi]_k^j \wedge \bigwedge_{j'=i}^{j-1} R(\pi_j, \pi_{j+1})) \\
[\pi \models \mathbf{X}\psi]_k^i & \equiv & [\pi \models \psi]_k^{i+1} \wedge R(\pi_i, \pi_{i+1})
\end{array}$$

Let π be a path of M without *a k-loop. Then the* bounded evaluation *of an LTL formula ψ on the k-prefix of π is defined as follows:*

$$\begin{array}{lcl}
[\pi \models \mathbf{G}\psi]_k^i & \equiv & \mathit{false} \\
[\pi \models \mathbf{F}\psi]_k^i & \equiv & \bigvee_{j=i}^{k}([\pi \models \psi]_k^j \wedge \bigwedge_{j'=i}^{j-1} R(\pi_j, \pi_{j+1})) \\
[\pi \models \mathbf{X}\psi]_k^i & \equiv & \mathit{if}\ i < k\ \mathit{then}\ [\pi \models \psi]_k^{i+1} \wedge R(\pi_i, \pi_{i+1})\ \mathit{else\ false}
\end{array}$$

The other cases are identical to the case where π has a k-loop.

Note that in contrast to standard LTL, the *three-valued* evaluation of LTL formulae requires to take the transition relation into account: Transitions in three-valued models are either *true* or *unknown* and the value of a transition may affect the result of model checking. Moreover, note that in the *no k-loop* case of bounded model checking any formula of the form $\mathbf{G}\,\psi$ always evaluates to *false* because ψ might not hold at the successor position $k+1$ of the considered path.

For convenience, we introduce the following abbreviation that allows for a shorter representation of certain temporal logic properties. Let ψ be an arbitrary LTL formula then:

$$\psi_i \;:=\; \mathbf{X}^i\,\psi \;=\; \underbrace{\mathbf{X}\ldots\mathbf{X}}_{i\text{-times}}\,\psi.$$

Hence, ψ_i states that ψ has to hold for the i-suffix of a considered path.

The bounded evaluation of temporal logic properties on entire three-valued Kripke structures is what is known as *three-valued bounded model checking* with the possible outcomes *true*, *false* and *unknown*. Here we distinguish between universal and existential model checking.

Definition 10 (Three-Valued Bounded Model Checking).
Let $M = (S, I, R, L, F)$ be a three-valued Kripke structure over A, let $k \in \mathbb{N}$ and let ψ be an LTL formula. The corresponding universal model checking problem is

$${}_A[M, I \models^F_\forall \psi]_k \;=\; \bigwedge\nolimits_{\pi\in\Pi^F_M,\pi_0\in I} [\pi \models \psi]^0_k$$

and the corresponding existential model checking problem is

$${}_A[M, I \models^F_\exists \psi]_k \;=\; \bigvee\nolimits_{\pi\in\Pi^F_M,\pi_0\in I} [\pi \models \psi]^0_k.$$

Note that model checking *without* fairness constraints is a special case of model checking under fairness with $F = \varnothing$. For certain verification tasks fairness is not relevant and thus can be ignored in order to avoid an unnecessary growth of complexity. If we neglect fairness, we will simply write $M = (S, I, R, L)$ for a Kripke structure and ${}_A[M, I \models_Q \psi]_k$ with $Q \in \{\forall, \exists\}$ for a model checking problem.

In practice, universal model checking is of major interest as it allows to show that *all* computations of a modelled system satisfy certain temporal logic properties. Certain model checking techniques such as satisfiability-based bounded model checking are only defined for the existential case. However, universal model checking can always be transformed into existential model checking based on the equation ${}_A[M, I \models^F_\forall \psi]_k \;=\; \neg\, {}_A[M, I \models^F_\exists \neg\psi]_k$, which also makes existential approaches generally applicable. In this work we will particularly focus on the existential case. In [3] Lemma 1 has been proven:

Lemma 1
Let Sys over Var be a concurrent system and let A_a and A_r be sets of pred-

icates over Var with $A_a \subset A_r$. Let Sys_a over A_a and Sys_r over A_r be abstract systems with $Sys_a \preceq Sys$, $Sys_r \preceq Sys$ and $Sys_a \preceq Sys_r$. Moreover, let $M_a = (S_a, I_a, R_a, L_a, F_a)$ be the three-valued Kripke structure modelling the state space of Sys_a, let $M_r = (S_r, I_r, R_r, L_r, F_r)$ be the structure modelling the state space of Sys_r, let ψ be an LTL formula and $k \in \mathbb{N}$ be a bound. Then the following holds:

1. ${}_{A_a}[M_a, I_a \models_{\exists}^{F} \psi]_k = \mathit{true} \;\Rightarrow\; {}_{A_r}[M_r, I_r \models_{\exists}^{F} \psi]_k = \mathit{true}$
2. ${}_{A_a}[M_a, I_a \models_{\exists}^{F} \psi]_k = \mathit{false} \;\Rightarrow\; {}_{A_r}[M_r, I_r \models_{\exists}^{F} \psi]_k = \mathit{false}$

Hence, all definite (*true* and *false*) bounded model checking results obtained under three-valued abstraction of *Sys* over a predicate set A_a can be transferred to any refined abstraction over an extended predicate set A_r with $A_a \subset A_r$. Since the most refined abstraction represents the concrete state space of *Sys*, definite results can be also transferred to the concrete system. An *unknown* result indicates that the current level of abstraction is too coarse and further predicates need to be added.

Bounded model checking is inherently incomplete as it only considers paths up to a length k. The existence of a k-prefix that satisfies a formula ψ allows us to conclude that ψ is also existentially satisfied in the unbounded case. But the non-existence of such a k-prefix does not allows us to conclude that ψ is not satisfied in the unbounded case. A straightforward but typically impracticable way of making bounded model checking complete is to iterate over all possible bounds up to a completeness threshold ct [9] with:

$$ {}_{A}[M, I \models_{\exists}^{F} \psi]_{ct} \;=\; {}_{A}[M, I \models_{\exists}^{F} \psi]_{\infty} $$

Two types of temporal logic properties are of particular interest in model checking concurrent software systems: *safety* and *liveness*. Checking safety properties such as mutual exclusion can be done via reachability analysis for which many efficient techniques exist. In particular, only loop-free paths need to be considered and fairness can be neglected for safety model checking. However, many requirements of concurrent systems cannot be formulated as safety properties. For instance, the requirement that the processes of a system make continuous progress refers to a liveness property. Checking liveness requires the consideration of fairness and the exploration of paths with loops, which is significantly more complex than reachability analysis.

In the following we exemplify one safety and one liveness model checking problem that we will consider throughout this work. For our running example, we can define the universal safety model checking problem ${}_{A}[M, I \models_{\forall} \mathbf{G}\, \mathit{safe}]_k$ without fairness constraints where $\mathit{safe} = \neg(pc_1 = 1) \vee \neg(pc_2 = 1)$. This characterises the mutual exclusion requirement that globally at most one process shall be at the critical location 1 at the same time. A *counterexample* to this requirement can be always given by a finite loop-free prefix that reaches a state were both processes are at the critical location. The complementary existential model checking problem is ${}_{A}[M, I \models_{\exists} \mathbf{F}\, \neg \mathit{safe}]_k$ where we check for the existence of a path prefix that violates mutual exclusion. In existential model checking we call a path that satisfies the temporal logic property a *witness*. A counterexample for a universal problem always corresponds to a witness for the complementary existential problem and vice versa. A completeness threshold for model checking safety properties is the *diameter* of the underlying Kripke structure, i.e. the longest distance between any two states. In incremental bounded model checking that iterates over k from 0 up to a completeness threshold, we can assume that in all previous iterations $k' < k$ no witness has been detected (otherwise model checking would have terminated). This allows us to slightly re-formulate the property to be checked. The formula

$$\mathbf{O}^k \neg \mathit{safe} \;\equiv\; \mathit{safe}_0 \wedge \mathit{safe}_1 \wedge \ldots \wedge \mathit{safe}_{k-1} \wedge \neg \mathit{safe}_k$$

where $\mathit{safe}_i = \mathbf{X}^i \mathit{safe}$, characterises that *safe* is *only* ($\mathbf{O}$) violated in the k-th state of a k-prefix. Model checking such a conjunctive formula is generally more efficient than checking a disjunctive formula, since the conjunctive form involves a stronger restriction of the search space.

In the three-valued scenario, we have to distinguish between two kinds of witnesses, *definite* ones when the model checking result is *true* and *unconfirmed* ones when the result is *unknown*.

Definition 11 (Definite Witness for Safety).
Let ${}_A[M, I \models_\exists \mathbf{O}^k \neg \mathit{safe}]_k$ be a three-valued bounded model checking problem with regard to safety where *safe* is a propositional logic expression over A. A *definite witness* for $\mathbf{O}^k \neg \mathit{safe}$ is a prefix $\omega = \pi_0 \ldots \pi_k$ of a path $\pi \in \Pi_M$, $\pi_0 \in I$ with

$$\pi_k(\mathit{safe}) = \mathit{false} \text{ and } \;\; \forall\, 0 \leq i < k : R(\pi_i, \pi_{i+1}) = \mathit{true}$$

A definite witness implies that a safety violation has been detected, and thus, no further model checking runs are required. In our example Kripke structure from Figure 5 there exists no definite witness for $\mathbf{O}^k \neg \mathit{safe}$.

An *unknown* result in three-valued bounded model checking indicates that there exists a path $\pi \in \Pi_M$ such that its k-prefix $\omega = \pi_0 \ldots \pi_k$ is an *unconfirmed witness* for the safety formula $\mathbf{O}^k \neg \mathit{safe}$.

Definition 12 (Unconfirmed Witness for Safety).
Let ${}_A[M, I \models_\exists \mathbf{O}^k \neg \mathit{safe}]_k$ be a three-valued bounded model checking problem with regard to safety where *safe* is a propositional logic expression over A. An *unconfirmed witness* for $\mathbf{O}^k \neg \mathit{safe}$ is a prefix $\omega = \pi_0 \ldots \pi_k$ of a path $\pi \in \Pi_M$, $\pi_0 \in I$ with either

$$\begin{aligned} &\pi_k(\mathit{safe}) = \mathit{unknown}, \text{ or} \\ &\pi_k(\mathit{safe}) = \mathit{false} \text{ and } \;\; \exists\, 0 \leq i < k \text{ with } R(\pi_i, \pi_{i+1}) = \mathit{unknown} \end{aligned}$$

For our Kripke structure from Figure 5 the path prefix $s_0 s_1 s_3$ is an unconfirmed witness for $\mathbf{O}^k \neg \mathit{safe}$.

A fair universal liveness model checking problem for our running example is $_A[M, I \models_\forall^F \mathbf{GF}\, \mathit{progress}]_k$ where $\mathit{progress} = (pc_1 = 1) \vee (pc_2 = 1)$. This characterises the requirement that in a fair computation always eventually some process will reach the critical location. A counterexample would be a path that reaches an infinite loop where each process continues to execute operations but no process will be in the critical location ever again. The complementary existential model checking problem is $_A[M, I \models_\exists^F \mathbf{FG}\, \neg \mathit{progress}]_k$. A witness for the existential problem corresponds to a counterexample to the complementary universal problem. Again, we can distinguish between definite and unconfirmed witnesses for liveness. In our example, the prefix $s_0 \xrightarrow{u,1} s_0 \xrightarrow{u,2} s_0$ which has a fair $(2, 0)$-loop is an unconfirmed witness for the liveness formula $\mathbf{FG}\, \neg \mathit{progress}$. Thus, the corresponding three-valued bounded model checking result is *unknown*.

So far, our three-valued bounded model checking approach fails for our running example because of the following two reasons. Firstly, we are only able to obtain *unknown* results. Hence, three-valued abstraction refinement is necessary for which we will present a technique in Section 6. Secondly, bounded model checking is incomplete and completeness thresholds for liveness properties are hard to compute and typically impracticable for efficient verification. We approach this limitation in two ways. We adopt the *state recording* technique of [13] that allows to translate liveness model checking problems into safety problems, for which smaller completeness thresholds exist. Moreover, we employ the *k-induction* technique [11], which allows to make safety bounded model checking complete without the need of a predetermined threshold. We start with the reduction of liveness to safety in the subsequent section.

4. From Liveness to Safety via State Recording

In this section, we review the state recording technique originally introduced in [13] which allows to reduce model checking liveness to model checking safety. In particular, we show that state recording can be implemented based on a transformation of our concurrent systems to be verified. For the purpose of an easier understanding, we divide the transformation into two steps: a first step that enables *loop detection* and a second step that allows for model checking *liveness under fairness*. The first step translates the input system into a state recording system for the purpose of loop detection. A looped path corresponds to a finite prefix where the last state is identical to an arbitrary previous state. Hence, the search for a loop can be performed by "recording" an arbitrary state along an explored path and checking whether this state will be reached again. The recording creates a copy of the current state. Thus, a state recoding system introduces a copy p^c of each original predicate p in order to represent the recorded state.

Definition 13 (State Recording System).
Let $Sys = \|_{i=1}^n P_i$ be a concurrent system abstracted over a set of system predicates A_{Sys} and given by n control flow graphs $G_i = (Loc_i, \delta_i, \tau_i)$. Moreover,

let $A = A_{Sys} \cup A_{PC}$ with $A_{PC} = \{(pc_i = l_i) \mid i \in [1..n],\ l_i \in Loc_i\}$ be the overall set of predicates and let $Init$ be the initial state predicate of Sys. Then the corresponding *state recoding system* Sys^{sr} is defined over the predicate set A^{sr} with initial state predicate $Init^{sr}$ and given by the control flow graphs $G_i^{sr} = (Loc_i, \delta_i, \tau_i^{sr})$ where

- $A^{sr} := A \cup A^c \cup Aux$

  where $A^c = \{p^c \mid p \in A\}$ is a set of copies of the original predicates in A

  and $Aux = \{record, recorded\}$ is a set of Boolean auxiliary predicates,

- $Init^{sr} := Init \wedge \bigwedge_{p^c \in A^c}(p^c \leftrightarrow p) \wedge \neg recorded$,

- $\forall (l_i, l_i') \in \delta_i \ : \ \tau_i^{sr}(l_i, l_i') :=$

$$\tau_i(l_i, l_i') \circ [record := *] \tag{1}$$

$$\circ\ [recorded := record \vee recorded] \tag{2}$$

$$\circ\ [\forall\, p^c \in A^c \ : \ p^c := (firstRecord \rightarrow p) \wedge (\neg firstRecord \rightarrow p^c)] \tag{3}$$

  where $*$ denotes the non-deterministic choice between *true* and *false*

  and $firstRecord = record \wedge \neg recorded$.

Hence, the state recording translation adds a copy p^c of each original predicate $p \in A$ and two Boolean auxiliary predicates *record* and *recorded* to the system. The copies receive the same initialisation as the original predicates. While there is no restriction with regard to the initialisation of *record*, the predicate *recorded* is initially set to *false*. The control flow and the operations of the original system are preserved, but each original operation gets extended by assignments to the new predicates. Remember that the list of assignments associated with an operation is executed in an *atomic* manner. Thus, if an operation changes the value of *record* from *false* to *true*, then on the right-hand side of the assignment $recorded := record \vee recorded$ the predicate *record* is still evaluated with *false*.

The extension allows to reduce the detection of a looped path to a reachability problem on the level of the corresponding Kripke structure. For now we can ignore fairness. Let $M = (S, I, R, L)$ over A be the Kripke structure modelling the original system *Sys* and let $M^{sr} = (S^{sr}, I^{sr}, R^{sr}, L^{sr})$ over A^{sr} be the Kripke structure modelling Sys^{sr}. The set of states of M^{sr} is $S^{sr} = S_A \times S_{A^c} \times S_{Aux}$ and each state is a triple $s^{sr} = \langle s, s^c, s^{aux}\rangle$ where s refers to the state of Sys. M^{sr} still comprises all the behaviour of M with regard to original predicate set A, i.e. for each path π in M there exists a path π^{sr} in M^{sr} with $\forall\, i \geq 0 : \forall\, p \in A : L(\pi_i, p) = L(\pi_i^{sr}, p)$ and vice versa. The new predicates and extended operations add a state recording mechanism to the modelled system, which works as follows. Each execution of an extended operation along a computational path now sets the value of the new predicate *record* non-deterministically to either *true* or *false* (1). When *record* evaluates to *true* for the first time, this indicates that the current state of the original system is selected to be recorded. In order to ensure that a recording only happens once, the predicate *recorded* is used. While initially evaluated with *false*, *recorded* is set to *true* after a *record* has been triggered and then remains *true* in all further computational steps (2). The actual state recording now has the condition that $\mathit{firstRecord} = \mathit{record} \wedge \neg \mathit{recorded}$ holds. The recording happens by assigning the predicate copies p^c to the current values of the original predicates p (3).

While classical state space exploration is memoryless, state recording allows to check whether a previously recorded state can be reached again, which reduces loop detection to a reachability problem. Let $\mathit{looped} = \mathit{recorded} \wedge \bigwedge_{p \in A} (p \leftrightarrow p^c)$, then the model checking problem ${}_{A^{sr}}[M^{sr}, I^{sr} \models_{\exists} \mathbf{F}\, \mathit{looped}]_k$ is equivalent to the question of whether there exits a looped path of length $k-1$ in the original system. Note that if the state recording model checking problem has the bound k, then only paths of length $k-1$ will be considered. This is due to the fact that under state recording a detected loop indicates that the state s_k is identical to some previously reached state s_l. Hence, the associated loop is actually from position $k-1$ to l.

The prefix of a path π^{sr} depicted in Figure 6 illustrates reachability-based loop detection in a state recording Kripke structure M^{sr} corresponding to some original Kripke structure M.

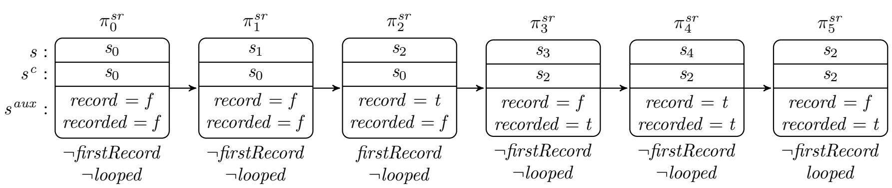


Figure 6: Witness for a looped path.

As we can see, π^{sr} comprises the looped path $\pi = s_0 s_1 \cdot (s_2 s_3 s_4)^\omega$ of some original model. The execution of the operation associated with the transition from π_1^{sr} to π_2^{sr} sets *record* to *true* for the first time. Hence, the original state s_2 is the one chosen to be recorded. Note that this transition only indicates the *choice* of the state to be recorded. The *actual recording* happens by the execution of the operation associated with the subsequent transition, as only in π_2^{sr} the condition *firstRecord* for state recording evaluates *true*. In the overall state π_3^{sr} the state copy s^c has been set to s_2, while the state of the original system has already changed to s_3. In π_4^{sr} the predicate *record* is *true* again, but since *firstRecord* does not hold any more the state copy will not be overwritten in the subsequent state. The path finally reaches the state π_5^{sr} where $s = s^c$ as well as *recorded* holds. Thus, we can conclude that the prefix of π^{sr} characterises a looped path of the original model. Since we use *existential* model checking, *any* reachable original state will be considered for recording. Hence, if the state recording model checking problem ${}_{A^{sr}}[M^{sr}, I^{sr} \models_{\exists} \mathbf{F}\, \mathit{looped}]_k$ returns *false*, then no looped path of length $k-1$ exists in the original model.

So far, the state recording approach only allows to detect looped paths but not to perform liveness model checking. In Definition 14, we further extend state recording systems in order to enable model checking liveness properties under fairness assumptions.

Definition 14 (State Recording System with Liveness Extension). Let Sys^{sr} over $A^{sr} = A \cup A^c \cup Aux$ be a state recording system given by n control flow graphs $G_i^{sr} = (Loc_i, \delta_i, \tau_i^{sr})$. Moreover, let $\mathbf{FG}\,\neg progress$ be a temporal logic formula characterising a liveness violation where *progress* is a predicate expression over A. Then the corresponding *state recoding system with liveness extension* Sys^{le} is defined over the predicate set A^{le} with initial state predicate $Init^{le}$ and given by the control flow graphs $G_i^{le} = (Loc_i, \delta_i, \tau_i^{le})$ where

- $A^{le} := A^{sr} \cup \mathbb{F} \cup \{live\}$

  where $\mathbb{F} = \{fair_i \mid i \in [1..n]\}$ is a set of Boolean fairness predicates

  and *live* is a Boolean auxiliary predicate,
- $Init^{le} := Init^{sr} \wedge \bigwedge_{i\in[1..n]} \neg fair_i \wedge \neg live$,
- $\forall (l_i, l_i') \in \delta_i \;:\; \tau_i^{le}(l_i, l_i') :=$

$$\begin{aligned}\tau_i^{sr}(l_i, l_i') \;\circ\; & [fair_i := loopStarted] && (1)\\ \circ\; & [live := live \vee (loopStarted \wedge progress)] && (2)\end{aligned}$$

  where $loopStarted = record \vee recorded$.

For each process P_i the liveness extension adds a predicate $fair_i$ to the system. Moreover, a predicate *live* is added. All new predicates are initially assigned to *false*. The predicate $fair_i$ is used to indicate whether a computation is fair with regard to P_i in the sense that the process infinitely often executes an operation. On looped execution paths this is the case if P_i executes an operation after the loop has started. Hence, an operation $\tau_i^{le}(l_i, l_i')$ by P_i sets $fair_i$ to *true* if the condition $loopStarted = record \vee recorded$ is satisfied (1). The predicate expression *loopStarted* is used to indicate whether the starting point of a *potential* loop has already been reached and recorded. (If the recorded starting point belongs to an *actual* loop is determined by solving the model checking problem defined in Corollary 1.) Once a computation of a state recording system leads to a state where *loopStarted* holds, this expression will also evaluate to *true* in all future states resulting from the computation (Definition 13). Thus, a predicate $fair_i$ will also evaluate to *true* forever once it has been *true* for the first time along a computational path. The predicate *live* is used to indicate whether *progress* holds infinitely often, i.e. at some point after the loop has started (2). By combining the results of [13] with our definitions, we get Corollary 1:

Corollary 1
Let *Sys* be a concurrent system and M be the Kripke structure modelling the

state space of Sys abstracted over A. Let *progress* be a predicate expression over A. Moreover, let Sys^{le} be the state recording system with liveness extension corresponding to Sys and M^{le} be the Kripke structure modelling the state space of Sys^{le} over A^{le}. Then the following holds:

$$_A[M, I \models^F_\exists \mathbf{FG}\, \neg progress]_k \;\Leftrightarrow\; _{A^{le}}[M^{le}, I^{le} \models_\exists \mathbf{F}\,(looped \wedge fair \wedge \neg live)]_{k+1}$$

where $looped = recorded \wedge \bigwedge_{p \in A}(p \leftrightarrow p^c)$ and $fair = \bigwedge_{i=1}^{n} fair_i$.

Hence, liveness model checking under fairness can be reduced to simple safety model checking.

The prefix of a path π^{le} depicted in Figure 7 illustrates reachability-based detection of liveness violations in a state recording Kripke structure with liveness extension M^{le} corresponding to some original Kripke structure M.

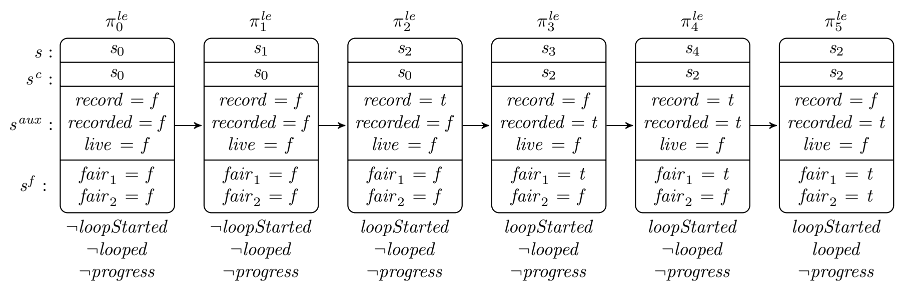


Figure 7: Witness for liveness violation.

As we can see, π^{le} comprises the looped path $\pi = s_0 s_1 \cdot (s_2 s_3 s_4)^\omega$ of some original model M. The reachability of the state π_5^{le} allows us to conclude that there exists a looped path satisfying fairness but violating liveness. Similar as proposed in Section 2, an incremental approach to bounded model checking allows us to re-formulate the property to be checked to $\mathbf{O}^k(looped \wedge fair \wedge \neg live)$ which gives us a more restricted search space.

With the state recording translation we are able to detect liveness violations in our abstracted system via safety model checking. The price to be paid for the reduction from liveness to safety is a quadratic increase of the number of states in the state recording Kripke structure, i.e. $| S^{le} | = \mathcal{O}(| S |^2)$ [13]. However, for safety model checking many efficient algorithms and typically smaller completeness thresholds exist. While the determination of tight thresholds for checking safety is also challenging, an alternative approach to the completeness of bounded model checking is *k-induction* [11]. We review k-induction in Section 5 and show how we have integrated it into our abstraction-based three-valued bounded model checking technique in Section 6.

5. Unbounded Model Checking via k-Induction

In the previous section we have shown that liveness model checking problems can be translated into safety model checking problems. Hence, we can assume that all our verification problems to be solved are of the form

$$_A[M, I \models_\forall \mathbf{G}\, \mathit{safe}]_\infty$$

where *safe* is an arbitrary predicate expression over the set of atomic predicates A. This unbounded model checking problem requires the consideration of all infinite paths of the model in order to prove that no property violations exist. The k-induction approach [11] allows to reduce an unbounded safety model checking problem into two bounded model checking problems: In the *base case* it is checked whether all k-prefixes of paths starting in an initial state $s \in I$ of M are *safe*, i.e.

$$_A[M, I \models_\forall \textstyle\bigwedge_{i=0}^{k} \mathit{safe}_i]_k.$$

In the *inductive step* it is checked whether, assuming a path prefix of k *safe* states, also any successor state is *safe*, i.e.

$$_A[M, S \models_\forall \big(\textstyle\bigwedge_{i=0}^{k} \mathit{safe}_i\big) \to \mathit{safe}_{k+1}]_{k+1}.$$

Note that in the inductive step the considered prefixes can start in an arbitrary state $s \in S$. As proven in [11], if there exists a k for which the base case fails, then the property $\mathbf{G}\, \mathit{safe}$ is violated. And, if there exists a k for which both the base case and the inductive step hold, then the property $\mathbf{G}\, \mathit{safe}$ holds for the model. k-induction is typically performed *incrementally* with regard to the bound. Thus, if the base case holds but the inductive step fails then k gets incremented and the model checking problems corresponding to the increased bound are solved.

The universal problems above refer to the safety of *all* paths. However, model checking via satisfiability solving is based on the existential case. We have already seen that each universal model checking problem can be transformed into a complementary existential problem referring to the existence of an unsafe path:

$$_A[M, I \models_\exists \mathbf{F}\, \neg\mathit{safe}]_\infty$$

The corresponding existential base case is

$$_A[M, I \models_\exists \textstyle\bigvee_{i=0}^{k} \neg\mathit{safe}_i]_k$$

and the existential inductive step is

$$_A[M, S \models_\exists \big(\textstyle\bigwedge_{i=0}^{k} \mathit{safe}_i\big) \wedge \neg\mathit{safe}_{k+1}]_{k+1}.$$

Now an unsafe path exists if for some k the existential base case holds, whereas all paths are safe if both the existential base case and the existential inductive step fail. We want to follow an incremental approach with regard to the bound. Thus, when checking the base case for some k we can assume that all shorter base cases have already been proven to be safe, and we can add these facts as constraints to the problem to be solved:

$$_A[base]_k \ := \ _A[M, I \models_\exists \underbrace{\big(\bigwedge_{i=0}^{k-1} \mathit{safe}_i\big) \wedge \neg \mathit{safe}_k}_{=\ \mathbf{O}^k \neg \mathit{safe}}]_k$$

This strengthening of the temporal logic formula to be checked involves a restriction of the state space to be explored, and thus, allows for model checking with an improved efficiency.

In order to make the k-induction approach *complete*, i.e. terminating for finite-state systems, it is necessary to restrict the inductive step to *loop-free* computations [11]. This gives us a slightly revised inductive step:

$$_A[step]_{k+1} \ := \ _A[M, S \models_\exists \underbrace{\big(\bigwedge_{i=0}^{k} \mathit{safe}_i\big) \wedge \neg \mathit{safe}_{k+1}}_{=\ \mathbf{O}^{k+1} \neg \mathit{safe}} \wedge \mathit{loopFree}_{0,k+1}]_{k+1}$$

where $\mathit{loopFree}_{0,k+1} \ = \ \bigwedge_{0 \leq i < j \leq k+1} \Big(\bigvee_{p \in AP} \big((p_i \wedge \neg p_j) \vee (\neg p_i \wedge p_j)\big)\Big)$.

Adding the loop-free constraint to the inductive step is an implicit way of determining whether the current bound is a completeness threshold of the model checking problem to be solved. Hence, a threshold does not have to be explicitly computed in the k-induction approach. Algorithm 1 illustrates the basic principle of incremental k-induction:

Algorithm 1: Incremental k-induction.

```
for k = 0 to ∞ do
    if (A[base]_k = true) then
        return "safety property violated"
    if (A[base]_k = false and A[step]_{k+1} = false) then
        return "safety property holds"
```

Hence, incremental k-induction iterates over the bound until either a property violation can be detected or until the bound is sufficiently large to conclude that the property holds. In [11] it has been proven that the k-induction approach yields the correct unbounded model checking result.

However, Algorithm 1 does not take into account that our model checking problems have a *three-valued* domain, i.e. solving the base case or the inductive step might yield *unknown*. In order to handle this case, we combine k-induction with three-valued abstraction refinement, which we discuss in the next section.

6. Basic Three-Valued Abstraction Refinement

Solving a three-valued bounded model checking problem ${}_{A}[M, I \models_{\exists} \psi]_{k}$, where the temporal logic formula ψ characterises the violation of a safety property, has the possible outcomes *false*, *true* and *unknown*. A *false* result indicates that there exists no k-prefix in M that is a witness for ψ. A *true* result indicates that there exists a k-prefix ω in M that is a *definite witness* for ψ (Definition 11). An *unknown* result indicates that there exists a k-prefix ω in M that is an *unconfirmed witness* for ψ (Definition 12). Model checking tools are typically not only capable of returning the result of the input problem, but also the corresponding witness in case the result is *true* or *unknown* [3].

Our example Kripke structure in Figure 5 abstracts the mutual exclusion system over the predicate set $A = \{(pc_1 = 0), (pc_1 = 1), (pc_2 = 0), (pc_2 = 1)\}$. Assuming a bound of $k = 2$ and the temporal logic formula $\psi = \mathbf{O}^k \neg safe$ with $safe = \neg\big((pc_1 = 1) \wedge (pc_2 = 1)\big)$, three-valued bounded model checking will return *unknown* along with the corresponding unconfirmed witness shown in Figure 8.

$$\omega \;=\; s_0 s_1 s_3 \;=\; (0,0) \xrightarrow{u} (1,0) \xrightarrow{u} (1,1)$$

Figure 8: Unconfirmed witness.

In this representation, a tuple (l_1, l_2) denotes a state where process P_1 is at location l_1 and P_2 is at location l_2. Moreover, $\xrightarrow{u}$ denotes an *unknown* transition between states. The witness ω is *unconfirmed* because it reaches the abstract state $(1,1)$ where safety is definitely violated, but *unknown* transitions are taken in order to reach this state. An unconfirmed witness implies that the current level of abstraction, characterised by the predicate set A, is too coarse for a definite model checking result. In this case, refinement in the sense of extending the set A is required. In [3] we defined the function *analyseWitness*. It takes an unconfirmed witness $\omega = \pi_0 \dots \pi_k$ for a model checking problem ${}_{A}[M, I \models_{\exists} \mathbf{O}^k safe]_k$ as an input where M models the state space of a system *Sys* abstracted over A. Based on the concrete operations of *Sys* and the weakest precondition calculus *analyseWitness* derives suitable predicates for refinement, which works as follows:

1. Let $p = safe$. One possible reason for $\omega = \pi_0 \dots \pi_k$ being an unconfirmed witness is that $\pi_k(p) = unknown$.
    - Determine the largest index i, $0 \leq i < k$, with $\pi_i(p) \neq unknown$.
    - Determine the concrete operation op of *Sys* that is associated with the transition (π_i, π_{i+1}).
    - Let $p' = wp_{op}(p)$ be the weakest precondition of p with respect to op. If $p' \notin A$ then return p' and $\overline{p}'$ as predicates for refinement. Else repeat Step 1 for p'.

2. The other possible reason for $\omega = \pi_0 \dots \pi_k$ being unconfirmed is a transition (π_i, π_{i+1}) with $R(\pi_i, \pi_{i+1}) = \mathit{unknown}$ for some index $0 \le i < k$.
   - Determine the concrete operation *op* of *Sys* that is associated with the transition (π_i, π_{i+1}).
   - Let $\mathit{assume}(e)$ be the guard of *op*. If $e \notin A$ then return e and $\overline{e}$ as predicates for refinement. Else repeat Step 1 for e.

For our current example and the unconfirmed witness $\omega = s_0 s_1 s_3$ we get $\mathit{analyseWitness}(s_0 s_1 s_3) = \{(y > 0), (\overline{y > 0})\}$. This is the set containing the guard $(y > 0)$, associated with the *acquire* operation in the mutual exclusion system, as well as its complement $(\overline{y > 0})$[1]. While a detailed description of *analyseWitness* can be found in [3], we subsequently focus on the integration of this function into a fully-automatic abstraction refinement algorithm. Algorithm 2 defines a basic three-valued abstraction refinement algorithm *AR* that utilises *analyseWitness*.

Algorithm 2: $AR({}_A[M, I \models_\exists \psi]_k)$

---

loop forever do /*refinement loop*/
    if ${}_A[M, I \models_\exists \psi]_k = \mathit{false}$ **then**
        return *false*, 'no witness for safety violation of length k exists'
    if ${}_A[M, I \models_\exists \psi]_k = \mathit{true}$ and definite witness ω **then**
        return *true*, 'ω is a definite witness for safety violation'
    if ${}_A[M, I \models_\exists \psi]_k = \mathit{unknown}$ and unconfirmed witness ω **then**
        $A := A \cup \mathit{analyseWitness}(\omega)$

AR takes a three-valued bounded model checking problem ${}_A[M, I \models_\exists \psi]_k$ as an input. We assume that M is the state space model corresponding to the system to be verified abstracted over the initial predicate set A. Within the algorithm the model checking problem is solved. If the outcome is *true* or *false* the algorithm terminates and returns the definite result. In case of an *unknown* result, an unconfirmed witness ω is generated and new predicates are derived via *analyseWitness*. The predicate set of the next iteration is defined by the predicates of the current iteration joined with the newly derived predicates. Now the steps of model checking the Kripke structure corresponding to the extended predicate set and deriving new refinement predicates are repeated until a definite result can be obtained. The termination of *AR* is guaranteed for finite-state systems. In [3] we showed that the result of *AR* correctly characterises the computational behaviour of the system modelled by M. For our running

[1] In contrast to the Boolean predicates over the control flow, predicates over system variables have a *three-valued-valued* domain as they may evaluate to *unknown* due to abstraction. In order to enable the later reduction of three-valued bounded model checking to Boolean satisfiability, there must be a complementary predicate $\overline{p}$ with $\overline{p} \equiv \neg p$ for each predicate p over system variables [18].

example with the fixed bound $k = 2$ AR terminates with a *false* result after one refinement iteration that adds the predicates $(y > 0)$ and $(\overline{y > 0})$. Thus, there does not exist a witness of length 2 that violates mutual exclusion.

AR can be easily integrated into the algorithm for incremental k-induction. For the integration it is advisable to use distinct predicate sets for the base case and for the inductive step. In this way both problems can be independently refined, which is typically more efficient than always forcing the same level of abstraction, i.e. identical predicate sets for both problems. An approach that combines incremental k-induction with abstraction refinement will process our running example as follows: In bound iteration $k = 0$ the base case yields *false* and the inductive step yields *true*. Hence, we move to bound iteration $k = 1$ where the base case yields *false* and the step yields *unknown*. Consequently, the inductive step gets refined by adding the predicates $(y > 0)$ and $(\overline{y > 0})$ to its associated predicate set. After refinement, the inductive step yields *true*. Thus, we move to bound iteration $k = 2$ where we receive an *unknown* result for the base case. We refine the base case by adding $(y > 0)$ and $(\overline{y > 0})$ to its associated predicate set. After refinement, the base case yields *false*. The algorithm continues with the inductive step which yields also *false* for the current bound iteration. Hence, we can conclude that mutual exclusion is generally not violated for the system under consideration. Note that in each bound iteration k, the bound of the inductive step is always $k + 1$. In the next section we introduce our enhanced abstraction refinement technique.

7. Witness Refinement and Constraint Generation

Abstraction refinement-based model checking is still challenged by the state explosion problem. Each additional predicate involves an exponential growth of the state space to be explored. In the following, we introduce an enhanced abstraction refinement algorithm that allows to reduce the number of predicates that are actually considered during model checking. Our enhancement is based on restricting the search space of the model checking problem by *path constraints* that can be formulated as temporal logic formulae:

Definition 15 (Path Constraints).
Let $M = (S, I, R, L)$ be a three-valued Kripke structure defined over a set of atomic predicates A. Moreover, let $\pi_0 \dots \pi_k$ be the k-prefix of a path π in M. Then the corresponding *focussing path constraint* is

$$\sigma(\pi_0 \dots \pi_k) \ := \ \bigwedge_{i=0}^{k} \Big(\big(\bigwedge_{p \in \mathcal{T}(\pi_i)} p_i \big) \wedge \big(\bigwedge_{p \in \mathcal{F}(\pi_i)} \neg p_i \big) \Big)$$

and the corresponding *excluding path constraint* is

$$\overline{\sigma}(\pi_0 \dots \pi_k) \ := \ \bigvee_{i=0}^{k} \Big(\big(\bigvee_{p \in \mathcal{T}(\pi_i)} \neg p_i \big) \vee \big(\bigvee_{p \in \mathcal{F}(\pi_i)} p_i \big) \Big)$$

where $\mathcal{T}(\pi_i) = \{p \in A \,|\, L(\pi_i, p) = \mathit{true}\}$ and $\mathcal{F}(\pi_i) = \{p \in A \,|\, L(\pi_i, p) = \mathit{false}\}$.

Hence, a focussing path constraint is of the form $\phi_0 \wedge \ldots \wedge \phi_k$ and an excluding path constraint is of the form $\phi_0 \vee \ldots \vee \phi_k$ where each ϕ_i is a propositional logic expression over i-indexed predicates ($\phi_i = \mathbf{X}^i \phi$).

Given a prefix $\pi_0 \ldots \pi_k$, the corresponding focussing constraint $\sigma(\pi_0 \ldots \pi_k)$ is an LTL formula that is only satisfied for paths π' that have the same definite properties as $\pi_0 \ldots \pi_k$. Such a constraint can be especially useful in the context of three-valued abstraction where refinement always preserves definite properties but may also make previously *unknown* properties definite. For the unconfirmed witness ω depicted in Figure 8 we can construct the corresponding path constraint $\sigma(\omega)$.

$$\begin{aligned}\sigma(\omega) \;=\; & \quad (pc_1\!=\!0)_0 \wedge \neg(pc_1\!=\!1)_0 \wedge (pc_2\!=\!0)_0 \wedge \neg(pc_2\!=\!1)_0 \\ & \wedge \neg(pc_1\!=\!0)_1 \wedge (pc_1\!=\!1)_1 \wedge (pc_2\!=\!0)_1 \wedge \neg(pc_2\!=\!1)_1 \\ & \wedge \neg(pc_1\!=\!0)_2 \wedge (pc_1\!=\!1)_2 \wedge \neg(pc_2\!=\!0)_2 \wedge (pc_2\!=\!1)_2\end{aligned}$$

Now after a refinement step that adds the predicate $(y > 0)$, the prefix

$$\omega' \;=\; (0, 0, (y > 0) = t) \;\longrightarrow\; (1, 0, (y > 0) = u) \;\xrightarrow{u}\; (1, 1, (y > 0) = f)$$

satisfies the constraint $\sigma(\omega)$, since ω' follows the same control flow as ω, whereas the prefix

$$\omega'' \;=\; (0, 0, \,\cdot\,) \;\longrightarrow\; (0, 1, \,\cdot\,) \;\longrightarrow\; (1, 1 \,\cdot\,)$$

where the $\cdot$ denotes an arbitrary valuation of the predicate $(y > 0)$, does not satisfy $\sigma(\omega)$ because the prefixes ω and ω'' differ in their control flow. We will show that focussing path constraints can be used to focus on a particular unconfirmed witness in order to confirm it or to show that it is spurious after refinement. Conversely, we will use excluding path constraints to rule out unconfirmed witnesses that turn out to be spurious. A constraint can be added to a model checking problem by simply conjugating it with the safety property to be checked. We will see that, in particular in the context of satisfiability-based model checking, path constraints can substantially narrow down the search space of the model checking problem to be solved.

While the algorithm AR follows the classical *abstract–check–refine* loop [4], we now introduce an enhanced abstraction refinement algorithm that makes use of path constraints. The new algorithm is based on a loop

$$\textit{abstract}\text{—}\textit{check}\text{—}\big(\textit{refineWitness}\text{—}\textit{checkWitness}\big)^*\text{—}\textit{generateConstraint}$$

where the * denotes that the steps in brackets belong to an internal loop with potentially multiple iterations. The idea of the new approach is as follows. If abstraction-based model checking on a model that covers the global state space returns an unconfirmed witness ω, then we start an internal refinement loop with a local model that is restricted to refinements of the witness ω only. The local model can be straightforwardly obtained by using the focussing path constraint $\sigma(\omega)$, which masks out all prefixes that differ from ω. The *witness refinement* *loop* either results in a definite witness, which means we are done, or it tells us that ω is spurious. In the latter case, we generate the constraint $\overline{\sigma}(\omega)$ that excludes the unconfirmed witness ω from further consideration. In the next overall loop, we return to the global model and we use the constraint $\overline{\sigma}(\omega)$ in order to restrict the state space exploration. But we do not need to add the refinement predicates that we used in the local model in order to generate the constraint. Hence, we have two forms of refinement respectively concretisation here: predicate refinement along unconfirmed witnesses in a local model and the pruning of infeasible paths via constraints in the global model. The latter does not involve any increase of the state space. Algorithm 3 shows our new abstraction refinement algorithm *WRC*.

Algorithm 3: $\mathit{WRC}({}_A[M, I \models_\exists \psi]_k)$

$\Sigma_k := \mathit{true}$ /**cumulative excluding path constraint**/
loop forever do /*global constraint loop*/
    if ${}_A[M, I \models_\exists \Sigma_k \wedge \psi]_k = \mathit{false}$ **then**
        return *false*, ‘no witness for safety violation of length k exists’
    if ${}_A[M, I \models_\exists \Sigma_k \wedge \psi]_k = \mathit{true}$ and definite witness ω **then**
        return *true*, ‘ω is a definite witness for safety violation’
    if ${}_A[M, I \models_\exists \Sigma_k \wedge \psi]_k = \mathit{unknown}$ and unconfirmed witness ω **then**
        $A^\omega := A \cup \mathit{analyseWitness}(\omega)$
        loop forever do /*refinement loop local to ω*/
            if ${}_{A^\omega}[M, I \models_\exists \sigma(\omega) \wedge \psi]_k = \mathit{false}$ **then**
                /* ω *is spurious* */
                $\Sigma_k := \Sigma_k \wedge \overline{\sigma}(\omega)$
                goto 3
            if ${}_{A^\omega}[M, I \models_\exists \sigma(\omega) \wedge \psi]_k = \mathit{true}$ and witness υ **then**
                return *true*, ‘υ is a definite witness for safety violation’
            if ${}_{A^\omega}[M, I \models_\exists \sigma(\omega) \wedge \psi]_k = \mathit{unknown}$ and witness υ **then**
                $A^\omega := A^\omega \cup \mathit{analyseWitness}(\upsilon)$

WRC consists of an outer constraint loop where we operate on a global state space model defined over a global predicate set A and cumulative excluding path constraint Σ_k that is initially *true*, i.e. no paths are excluded. While A remains constant throughout the execution of the algorithm, Σ_k will be gradually extended with constraints that rule out *spurious witnesses*. The cases where a definite result is obtained in the outer loop are identically handled as in *AR*. If an *unknown* result together with an unconfirmed witness ω is obtained in the outer loop, then the algorithm enters an inner refinement loop *local* to ω. In the inner loop, we use a model defined over the predicate set A^ω. A^ω is initialised as the union of A and the refinement predicates derived from the unconfirmed witness ω. Moreover, the temporal logic formula to be checked is conjugated with the focussing path constraint $\sigma(\omega)$, which restricts the feasible paths to those whose prefix is a refinement of ω. Hence, the model checking problem in the inner loop has a refined state space defined over A^ω, but the employed model is local in the sense that the state space exploration is narrowed down to refinements of ω. In case of a *false* result in the inner loop, we have that ω is a spurious witness. We then extend the cumulative excluding path constraint Σ_k by the constraint $\overline{\sigma}(\omega)$, which excludes ω from further consideration, and we return to the outer loop where we operate again with the global model and the original predicate set A. In case of a *true* result in the inner loop, we obtain a definite witness υ that is a refinement of ω. Thus, *WRC* can terminate. In case of an *unknown* result in the inner loop, we obtain an unconfirmed witness υ that is a refinement of ω. We then derive new predicates from υ and continue with a further refinement iteration local to ω. We get the following theorem with regard to the return values of *AR* and *WRC*:

Theorem 1
Let $_A[M, I \models_\exists \psi]_k$ be a three-valued bounded model checking problem where M is a state space model of a system Sys abstracted over A and ψ is an LTL safety formula defined over A. Then the following holds:

1. $AR(_A[M, I \models_\exists \psi]_k) = \mathit{true}$ *iff* $WRC(_A[M, I \models_\exists \psi]_k) = \mathit{true}$
2. $AR(_A[M, I \models_\exists \psi]_k) = \mathit{false}$ *iff* $WRC(_A[M, I \models_\exists \psi]_k) = \mathit{false}$

Proof. See `http://github.com/ssfm-up/TVMC/raw/master/SCICOProofs.pdf`

Hence, both algorithms return the same result for the same input model checking problem with regard to a system *Sys*. We have already shown in [3] that the result of *AR* correctly characterises the computational behaviour of *Sys*. Thus, we can conclude that the result of *WRC* also correctly characterises the behaviour of *Sys*.

We now illustrate how *WRC* processes verification tasks based on our running example with regard to the mutual exclusion system and a bound of $k = 2$. The initial predicate set is again: $A = \{(pc_1 = 0), (pc_1 = 1), (pc_2 = 0), (pc_2 = 1)\}$. The set is de facto reducible to just $\{(pc_1 = 0), (pc_2 = 0)\}$ by assuming the equivalences $(pc_1 = 1) \equiv \neg(pc_1 = 0)$ and $(pc_2 = 1) \equiv \neg(pc_2 = 0)$, which we effectively do in the implementation of our approach. However, for illustrative purposes we use the expanded set A here. In the first global constraint iteration, *WRC* detects the unconfirmed witness $\omega = (0, 0) \xrightarrow{u} (1, 0) \xrightarrow{u} (1, 1)$. Similar as in our illustration of the basic algorithm, *WRC* now derives the refinement predicates $(y > 0)$ and $(\overline{y > 0})$. But the predicates are added to the *local* predicate set A^ω. Moreover, the focussing path constraint $\sigma(\omega)$ is added to the model checking problem, which gives us a local state space model. Hence, when we are solving the refined problem in the inner loop, the valuation of the control flow predicates in each state along a prefix is now fixed by $\sigma(\omega)$. This means that the complexity of the state space to be explored is solely induced by the predicates over y. Model checking yields that ω is spurious. Consequently, $\overline{\sigma}(\omega)$ is added to the cumulative excluding path constraint Σ_k via conjunction. This excludes any further consideration of ω and its possible refinements. The next iteration detects another unconfirmed witness $\omega' = (0,0) \xrightarrow{u} (0,1) \xrightarrow{u} (1,1)$. *WRC* now enters a refinement loop local to ω'. It detects that ω' is also spurious, and thus, can be ruled out via the excluding path constraint $\overline{\sigma}(\omega')$. In the final constraint iteration *WRC* terminates with the definite result that no witness for safety violation of length $k = 2$ exists.

With *WRC* we are able to reduce the number of predicates that actually contribute to the size of the state space to be explored. In our simple example, *WRC* had to solve model checking problems on global and local models with a maximum number of two predicates, whereas *AR* had to solve a problems with maximum four predicates. The price that we pay is an increased number of model checking runs. In our experiments we will show that the savings due to the reduced number of predicates typically outweigh the extra costs due to additional model checking runs. Similar to *AR*, the enhanced algorithm can be straightforwardly combined with a bound iteration loop ranging over k. The corresponding algorithm is depicted below.

Algorithm 4: $k\text{-}IND({}_{A_b}[base]_k, {}_{A_s}[step]_k)$

for $k = 0$ **to** ∞ **do**
  if $WRC({}_{A_b}[base]_k) = true$ **then**
    return "safety property violated"
  else if $WRC({}_{A_s}[step]_{k+1}) = false$ **then**
    return "safety property holds"

Here the input ${}_{A_b}[base]_k$ is a three-valued bounded model checking problem corresponding to the base case of the k-induction approach where A_b is the initial set of predicates and k is the bound parameter which gets initialised within the algorithm. Likewise, ${}_{A_s}[step]_k$ is a three-valued bounded model checking problem corresponding to the inductive step with initial predicate set A_s.

7.1. Constraint Strengthening and Reuse Between Bound Iterations

In general, it is not admissible to reuse generated constraints $\overline{\sigma}(\omega)$ for ruling out spurious witnesses between bound iterations and between the base case and the inductive step. For instance, a witness $\omega = \pi_0 \dots \pi_k$ might be spurious in iteration k, but it might be still the prefix of a definite witness in some later iteration $k + j$. However, we have identified different *types* of spurious witness constraints. Depending on the type, a spurious witness constraint may be straightforwardly reused or adapted for reuse in a higher bound iteration.

The algorithm *WRC* identifies an unconfirmed witness ω to be spurious if ${}_{A^\omega}[M, I \models_\exists \sigma(\omega) \wedge \psi]_k$ yields *false* (line 10). We can re-formulate this model checking problem as ${}_{A^\omega}[M, S \models_\exists I_0 \wedge \sigma(\omega) \wedge \psi]_k$ with $I_0 = \bigvee_{s \in I} \sigma(s)$. This gives us an equivalent model checking problem where the initial state constraint is part of the temporal logic formula to be checked. A sufficient condition for ${}_{A^\omega}[M, S \models_\exists I_0 \wedge \sigma(\omega) \wedge \psi]_k = false$ is that any conjunct or combination of conjuncts of the overall formula $I_0 \wedge \sigma(\omega) \wedge \psi$ is not satisfied by M. We denote such a conjunct (combination) as a *cause of violation* of the model checking problem:

Definition 16 (Cause of Violation).
Let ${}_A[M, S \models_\exists \phi \wedge \phi']_k$ be a bounded model checking problem that yields *false*. Then ϕ is a *cause of violation* of the problem if ${}_A[M, S \models_\exists \phi]_k = \mathit{false}$.

Evidently, for every model checking problem that yields *false*, the overall LTL formula is always a cause of violation itself. However, smaller causes may exist. For our abstraction refinement approach and the model checking problem ${}_{A^\omega}[M, S \models_\exists I_0 \wedge \sigma(\omega) \wedge \psi]_k$ local to an unconfirmed witness ω a cause of violation is also a cause of spuriousness of the witness. While the cumulative constraint Σ_k of the *current* bound iteration is simply extended by the spurious witness constraint $\overline{\sigma}(\omega)$ (line 12), $\overline{\sigma}(\omega)$ may have to be adapted for reuse in a *higher* bound iteration $k + j$. In our context, adaptation means to increment the index values that occur in $\overline{\sigma}(\omega)$ according to the higher bound. We define a j-increment of the indices occurring in a constraint as

$$\overline{\sigma}(\omega)_j \ := \ \overline{\sigma}(\omega)[i \leftarrow i + j \ \mid \ 0 \leq i \leq k].$$

Thus, a j-increment substitutes i-indexed predicates p_i in $\overline{\sigma}(\omega)$ by $(i + j)$-indexed predicates p_{i+j}. While the incrementation of indices may be *necessary* for reusing a constraint in a higher iteration, it may be additionally *feasible* (but not necessary) to strengthen a generated constraint. Note that since the focussing path constraint $\sigma(\omega)$ is a pure conjunction, a cause of violation may only contain a sub-formula $\sigma(\omega)^{sub}$ of $\sigma(\omega)$. We will see that in this case the stronger formula $\overline{\sigma}(\omega)^{sub}$ is a feasible constraint for excluding spuriousness. We denote such a $\overline{\sigma}(\omega)^{sub}$ as a *spurious segment constraint.*

We have proven Theorem 2 that assigns to the possible causes of violation of ${}_{A^\omega}[M, S \models_\exists I_0 \wedge \sigma(\omega) \wedge \psi]_k$ admissible extensions of the constraints Σ_{k+j} with $j \in \mathbb{N}$.

Theorem 2
Let ϕ_x be a cause of violation of the three-valued bounded model checking problem ${}_{A^\omega}[M, I \models_\exists \sigma(\omega) \wedge \psi]_k$ local to an unconfirmed witness ω. Then in bound iteration $k+j$ with $j \in \mathbb{J}$ it is admissible to extend the cumulative path constraint Σ_{k+j} of the corresponding global model checking problem ${}_A[M, I \models_\exists \Sigma_{k+j} \wedge \psi]_{k+j}$ as follows: $\Sigma_{k+j} := \Sigma_{k+j} \wedge \varphi_x$ where

$\phi_1 = I_0 \wedge \sigma(\omega)^{sub} \wedge \psi$	$\varphi_1 = \overline{\sigma}(\omega)^{sub}$	$\mathbb{J} = \{0\}$
$\phi_2 = I_0 \wedge \sigma(\omega)^{sub}$	$\varphi_2 = \overline{\sigma}(\omega)^{sub}$	$\mathbb{J} = \mathbb{N}$
$\phi_3 = \sigma(\omega)^{sub} \wedge \psi$	$\varphi_3 = \overline{\sigma}(\omega)^{sub}_j$	$\mathbb{J} = \mathbb{N}$
$\phi_4 = I_0 \wedge \psi$	$\varphi_4 = \mathit{false}$	$\mathbb{J} = \{0\}$
$\phi_5 = \sigma(\omega)^{sub}$	$\varphi_5 = \bigwedge_{l=0}^{j} \overline{\sigma}(\omega)^{sub}_l$	$\mathbb{J} = \mathbb{N}$
$\phi_6 = \psi$	$\varphi_6 = \mathit{false}$	$\mathbb{J} = \{0\}$
$\phi_7 = I_0$	$\varphi_7 = \mathit{false}$	$\mathbb{J} = \mathbb{N}$

Proof. See `http://github.com/ssfm-up/TVMC/raw/master/SCICOProofs.pdf`

Here the set $\mathbb{J}$ denotes the range of bound iterations for which the constraint φ_x is admissible. If $\mathbb{J} = \{0\}$ then φ_x is an admissible constraint in iteration $k+0$ only. If $\mathbb{J} = \mathbb{N}$ then φ_x admissible in all iterations $k+0, k+1, k+2, \ldots$.

In our context, *admissible* means that the added spurious segment constraint $\overline{\sigma}(\omega)^{sub}$ will only rule out paths that would anyway turn out to be spurious after the refinement of the global model checking problem ${}_A[M, I \models_\exists \Sigma_{k+j} \wedge \psi]_{k+j}$ over some extended predicate set $A' \supset A$. However, with our approach we are able to exclude all paths that exhibit the spurious segment *without* refining the global problem. For each distinct cause of violation we defined a descriptive *type* of the associated constraint:

cause of violation	**constraint type**
ϕ_1	*full-dependent*
ϕ_2	*property-independent*
ϕ_3	*initial state-independent*
ϕ_4	*redundant I*
ϕ_5	*fully-independent*
ϕ_6	*redundant II*
ϕ_7	*redundant III*

We now illustrate certain interesting cases of causes of violation resp. constraint types and the consequent reuse of constraints. Remember that ψ is of the form $\mathbf{O}^k \neg safe$. Hence, in bound iteration k we check for safety violation at position k whereas in a higher bound iteration $k+j$ we check for safety violation at position $k+j$.

- If the cause of violation is of the form ϕ_1, this tells us that there exists no k-prefix $s_0 \ldots s_k$ satisfying $\sigma(\omega)^{sub}$ that starts in an initial state $s_0 \in I$ and ends in a state s_k in which *safe* is violated. Hence, the cause of violation is linked to an initial state at position 0 *and* to an error state at position k. We denote the corresponding constraint as *fully-dependent* on the current bound iteration. The excluding path constraint $\overline{\sigma}(\omega)^{sub}$ is admissible in the current iteration k. However, in any bound iteration $k+j$ with $j > 0$ we search for a path prefix from an initial state to an error state at position $k+j$ rather than at position k. Such a prefix may still satisfy $\sigma(\omega)^{sub}$, and thus, $\overline{\sigma}(\omega)^{sub}$ is not an admissible constraint in iteration $k+j$.

- If the cause of violation is of the form ϕ_2, this tells us that there exists no k-prefix $s_0 \ldots s_k$ satisfying $\sigma(\omega)^{sub}$ that starts in an initial state $s_0 \in I$. Hence, the cause of violation is linked to an initial state at position 0 but *not* to any error state. We denote the corresponding constraint as (error) *property-independent*. If there exists no k-prefix satisfying $\sigma(\omega)^{sub}$ in iteration k then there is also not such a prefix in any iteration $k+j$. Thus, it is admissible to reuse the excluding path constraint $\overline{\sigma}(\omega)^{sub}$ in higher bound iterations.

- If the cause of violation is of the form ϕ_3, this tells us that there exists no k-prefix $s_0 \dots s_k$ satisfying $\sigma(\omega)^{sub}$ ends in a state s_k in which *safe* is violated. Since the cause of violation is *not* linked to an initial state such a k-prefix may start in an *arbitrary* state. The cause is however linked to an error state at position k. We denote the corresponding constraint as *initial state-independent*. In bound iteration $k+j$ with $j > 0$ we search for a path prefix $s_0 \dots s_{k+j}$ from an initial state to an error state at position $k+j$. Note that the j-suffix $s_j \dots s_{k+j}$ of such a prefix is a k-prefix leading to an error state. Hence, $\sigma(\omega)^{sub}$ must be violated for this j-suffix. In order to exclude corresponding paths it is admissible to use the j-incremented constraint $\overline{\sigma}(\omega)^{sub}_j$ in iteration $k+j$.

- If the cause of violation is of the form ϕ_5, this tells us that there exists no k-prefix $s_0 \dots s_k$ satisfying $\sigma(\omega)^{sub}$. Hence, the cause of violation is neither linked to an initial state nor to an error state. We denote the corresponding constraint as *fully-independent*. We have that $\sigma(\omega)^{sub}$ is violated for arbitrary sequences of states $s_0 \dots s_k$ in the model. Thus, in any bound iteration $k+j$ all l-incremented constraints $\overline{\sigma}(\omega)^{sub}_l$ with $0 \leq l \leq j$ are admissible.

As we can see in Theorem 2, there are also causes of violation were we can immediately conclude that the model checking problem yields *false*, regardless of the constraint. In this cases we denote the corresponding constraint as *redundant*. In case of a *redundant I* or *redundant II* constraint the *false* result only holds for the current bound iteration. In the latter case the *false* result is additionally transferable from the base case to the corresponding inductive step. In case of a *redundant III* constraint the *false* result can be even transferred to higher bound iterations.

Based on a similar argumentation we are also able to characterise admissible constraint reuse between the base case and the inductive step: If the cause of violation of the base case does not contain the initial state condition I_0 then the corresponding constraint is initial state-independent and can be reused for the inductive step (because there is no initial state condition in the step). And, if the cause of violation of the inductive step does not contain the loop-free condition $\mathit{loopFree}_{0,k+1}$ then the corresponding constraint can be reused for the base case (because there is no loop-free condition in the base case). Based on the results of Theorem 2 constraint reusing can be easily integrated into the algorithm *k-IND* that invokes the algorithm *WRC* for ascending bounds.

Remember our running example where we detected in bound iteration $k = 2$ of the base case that the unconfirmed witness $\omega = (0,0) \xrightarrow{u} (1,0) \xrightarrow{u} (1,1)$ is spurious. A corresponding cause of violation is

$$\sigma(\omega)^{sub} \;=\; (pc_1 = 0)_0 \wedge (pc_2 = 0)_0 \wedge (pc_1 = 1)_2 \wedge (pc_2 = 1)_2$$

The cause is of the form ϕ_5 as introduced in Theorem 2. Hence, the corresponding spurious witness constraint $\overline{\sigma}(\omega)^{sub}$ is full-independent, i.e. neither linked to an initial state nor to an error state, and thus, reusable in all future bound iterations $k + j$ of the base case and the inductive step. Moreover, all index increments $\overline{\sigma}(\omega)_l^{sub}$ with $0 \leq l \leq j$ are also admissible constraints in iterations $k + j$. Note that the cause of violation $\sigma(\omega)^{sub}$ is a real sub-formula of $\sigma(\omega)$, which positively affects the strength of the spurious segment constraint $\overline{\sigma}(\omega)^{sub}$ and its index increments. The constraint reveals that there does not exist any sequence of two transitions leading from $(0, 0)$ to $(1, 1)$ – independent of which state will be reached between $(0, 0)$ and $(1, 1)$. After reusing this constraint for the inductive step, the algorithm *k-IND* will immediately terminate with the result that the safety properties holds.

We have outlined the benefits of cause-based constraint reusing between bound iterations in terms of ruling out spurious behaviour. However, so far we have not discussed how causes of violation of a model checking problem can be automatically and efficiently computed. In the next section we introduce the propositional logic encoding of three-valued bounded model checking. The encoding enables us to solved model checking problems via Boolean satisfiability (SAT) solving. Moreover, we will see that the SAT-based approach reduces the detection of causes of violation to the extraction of *unsatisfiable cores* of a propositional logic formula, for which efficient tools exist.

8. Reduction to Propositional Logic Satisfiability

In our previous work [7] we showed how a three-valued bounded model checking problem ${}_A[M, I \models_\exists \psi]_k$ can be encoded as a propositional logic formula ${}_A[\![M, I, \psi, k]\!]$. The encoding corresponds to an implicit problem representation such that the construction of an explicit Kripke structure is avoided. The formula ${}_A[\![M, I, \psi, k]\!]$ is defined over a set of Boolean atoms *Atoms*, the constants *true*, *false*, and a special atom $\perp$ that is used to represent the *unknowns* due to abstraction. The atom $\perp$ occurs solely non-negated in ${}_A[\![M, I, \psi, k]\!]$. Based on the encoding, three-valued bounded model checking can be performed via two satisfiability checks. The first check considers an *under-approximating completion*, marked with '$-$', where all $\perp$'s are assumed to be *false*:

$${}_A[\![M, I, \psi, k]\!]^- := {}_A[\![M, I, \psi, k]\!][\perp \mapsto \mathit{false}]$$

and the second check considers an *over-approximating completion*, marked with '$+$', where all $\perp$'s are assumed to be *true*:

$${}_A[\![M, I, \psi, k]\!]^+ := {}_A[\![M, I, \psi, k]\!][\perp \mapsto \mathit{true}].$$

Here $[\perp \mapsto z]$, $z \in \{\mathit{true}, \mathit{false}\}$ denotes the assumption that the special atom $\perp$ is assigned to z. This gives us the notion of *3-valued satisfiability* $\mathbf{sat_3}$:

Definition 17 ($\mathbf{sat_3}$).
Let ${}_A[\![M, I, \psi, k]\!]$ over *Atoms* be the propositional logic encoding of a three-valued bounded model checking problem ${}_A[M, I \models_\exists \psi]_k$. Then $\mathbf{sat_3}$ is defined

as:

$$\mathbf{sat_3}\big({}_A[\![M,I,\psi,k]\!]\big) \;=\; \begin{cases} \mathit{true} & \mathit{if} \quad \mathbf{sat}\big({}_A[\![M,I,\psi,k]\!]^-\big) = \mathit{true} \\ \mathit{false} & \mathit{if} \quad \mathbf{sat}\big({}_A[\![M,I,\psi,k]\!]^+\big) = \mathit{false} \\ \mathit{unknown} & \mathit{else} \end{cases}$$

Thus, a $\mathbf{sat_3}$ problem is reducible to two Boolean satisfiability problems. In [7] the following lemma has been proven:

Lemma 2 *Let ${}_A[\![M,I,\psi,k]\!]$ over Atoms be the propositional logic encoding of a three-valued bounded model checking problem ${}_A[M,I \models_\exists \psi]_k$. Then:*

$${}_A[M,I \models_\exists \psi]_k \;=\; \mathbf{sat_3}\big({}_A[\![M,I,\psi,k]\!]\big)$$

Hence, by solving $\mathbf{sat_3}$ we obtain the result of the encoded three-valued bounded model checking problem. If the results of the two Boolean satisfiability checks are $\mathbf{sat}\big({}_A[\![M,\psi,k]\!]^-\big) = \mathit{false}$ and $\mathbf{sat}\big({}_A[\![M,\psi,k]\!]^+\big) = \mathit{true}$, then we can conclude that the result of the encoded problem is *unknown*. In this case, a truth assignment $\mathcal{A} : \mathit{Atoms} \to \{\mathit{true}, \mathit{false}\}$ that satisfies ${}_A[\![M,\psi,k]\!]^+$ characterises an unconfirmed witness ω. Thus, witness generation in the SAT-based approach is straightforward. The details on how the formula ${}_A[\![M,I,\psi,k]\!]$ is built and on how witnesses ω can be derived from satisfying assignments $\mathcal{A}$ can be found in [7]. ${}_A[\![M,I,\psi,k]\!]$ is in *conjunctive normal form* (CNF) and we assume a representation of the CNF formula as a set of sets of literals $\big\{\{l,\ldots,l'\},\ldots,\{l'',\ldots,l'''\}\big\}$. The construction of ${}_A[\![M,I,\psi,k]\!]$ is divided into the encoding of initial states I, the encoding of k unrollings of the transition relation of M and the encoding of the temporal logic formula ψ for bound k:

$${}_A[\![M,I,\psi,k]\!] \;=\; [\![M,k]\!] \cup [\![I]\!] \cup [\![\psi,k]\!]$$

In the expanded representation, we omit the reference to the associated predicate set A, as this is clear from the context. Since path constraints Σ_k and $\sigma(\omega)$ are also temporal logic formulae, the propositional logic encoding of model checking problems with constraints, as used in the refinement algorithm *WRC*, is straightforward. For a global model checking problem ${}_A[M,I \models_\exists \Sigma_k \wedge \psi]_k$ with a cumulative path constraint Σ_k we get

$${}_A[\![M,I,\Sigma_k,\psi,k]\!] \;=\; [\![M,k]\!] \cup [\![I]\!] \cup [\![\Sigma_k]\!] \cup [\![\psi,k]\!]$$

and for a model checking problem ${}_{A^\omega}[M,I \models_\exists \sigma(\omega) \wedge \psi]_k$ local to an unconfirmed witness ω we get

$${}_{A^\omega}[\![M,I,\sigma(\omega),\psi,k]\!] \;=\; [\![M,k]\!] \cup [\![I]\!] \cup [\![\sigma(\omega)]\!] \cup [\![\psi,k]\!].$$

This allows us to redefine the algorithm *WRC* as a satisfiability-based version *SATWRC*.

As we can see, each three-valued bounded model checking problem to be

Algorithm 5: $SATWRC({}_{A}[M, I \models_{\exists} \psi]_k)$

```
Σ_k := true /*cumulative excluding path constraint*/
loop forever do /*global constraint loop*/
    if sat3( A⟦M, I, Σ_k, ψ, k⟧ ) = false then
        return false, ‘no witness for safety violation of length k exists’
    if sat3( A⟦M, I, Σ_k, ψ, k⟧ ) = true and definite witness ω then
        return true, ‘ω is a definite witness for safety violation’
    if sat3( A⟦M, I, Σ_k, ψ, k⟧ ) = unknown and unconfirmed witness ω
     then
        A^ω := A ∪ analyseWitness(ω)
        loop forever do /*refinement loop local to ω*/
            if sat3( A^ω⟦M, I, σ(ω), ψ, k⟧ ) = false then
                /* ω is spurious */
                Σ_k := Σ_k ∧ σ̄(ω)
                goto 3
            if sat3( A^ω⟦M, I, σ(ω), ψ, k⟧ ) = true and witness υ then
                return true, ‘υ is a definite witness for safety violation’
            if sat3( A^ω⟦M, I, σ(ω), ψ, k⟧ ) = unknown and witness υ then
                A^ω := A^ω ∪ analyseWitness(υ)
```

solved is substituted by a $\mathbf{sat_3}$ problem. Thus, the soundness of *SATWRC* in terms of returning the correct model checking result follows from Theorem 1 and Lemma 2. The algorithm extends the cumulative path constraint Σ_k in the same manner as *WRC*. The extended constraint then gets encoded as part of ${}_{A}[\![M, I, \Sigma_k, \psi, k]\!]$. So far, none of the improvements that we introduced in the previous section are integrated into the algorithm. Remember that our improvements with regard to constraint strengthening and constraint reusing were based on causes of violation of a model checking problem. We will now show that there is a strong correspondence between a *cause of violation* of an explicit model checking problem and an *unsatisfiable core* of the propositional logic encoding of a model checking problem.

Definition 18 (Unsatisfiable Core).
Let α be a propositional logic formula in conjunctive normal form with $\mathbf{sat_3}(\alpha) = \mathit{false}$. An *unsatisfiable core* is a subset $\alpha_{uc} \subseteq \alpha$ of clauses of α with $\mathbf{sat_3}(\alpha_{uc}) = \mathit{false}$.

State-of-the-art SAT solvers support the efficient extraction of small or even minimal unsatisfiable cores [19]. Thus, when *SATWRC* detects that $\mathbf{sat_3}\big({}_{A^\omega}[\![M, I, \sigma(\omega), \psi, k]\!]\big)$ yields *false*, i.e. that ω is a spurious witness, then we can extract an unsatisfiable core of the encoded model checking problem

$$ {}_{A^\omega}[\![M, I, \sigma(\omega), \psi, k]\!] \;=\; [\![M, k]\!] \cup [\![I]\!] \cup [\![\sigma(\omega)]\!] \cup [\![\psi, k]\!] $$

which is a sub-formula

$$ {}_{A^\omega}[\![M, I, \sigma(\omega), \psi, k]\!]_{uc} \;=\; [\![M, k]\!]_{uc} \cup [\![I]\!]_{uc} \cup [\![\sigma(\omega)]\!]_{uc} \cup [\![\psi, k]\!]_{uc} $$

with

$$ [\![M, k]\!]_{uc} \subseteq [\![M, k]\!],\;\; [\![I]\!]_{uc} \subseteq [\![I]\!],\;\; [\![\sigma(\omega)]\!]_{uc} \subseteq [\![\sigma(\omega)]\!] \text{ and } [\![\psi, k]\!]_{uc} \subseteq [\![\psi, k]\!]. $$

Hence, such an unsatisfiable core consists of a model-related part $[\![M, k]\!]_{uc}$, an initial state-related part $[\![I]\!]_{uc}$, a constraint-related part $[\![\sigma(\omega)]\!]_{uc}$ and a property-related part $[\![\psi, k]\!]_{uc}$. For our approach, the constraint related part is of major interest. We have proven the following lemma:

Lemma 3
Let $\mathbf{sat_3}\big({}_{A^\omega}[\![M, I, \sigma(\omega), \psi, k]\!]\big) = \mathit{false}$ and let $[\![\sigma(\omega)]\!]_{uc}$ be the constraint-related part of the unsatisfiable core of ${}_{A^\omega}[\![M, I, \sigma(\omega), \psi, k]\!]$. Then there exists a unique sub-formula $\sigma(\omega)^{uc}$ of the constraint $\sigma(\omega)$ such that $[\![\sigma(\omega)]\!]_{uc} \subseteq [\![\sigma(\omega)^{uc}]\!]$ and $\mathbf{sat_3}\big({}_{A^\omega}[\![M, I, \sigma(\omega)^{uc}, \psi, k]\!]\big) = \mathit{false}$.

Proof. See `http://github.com/ssfm-up/TVMC/raw/master/SCICOProofs.pdf`

Hence, if SAT-based model checking yields that the unconfirmed witness ω is spurious, then the constraint-related part $[\![\sigma(\omega)]\!]_{uc}$ of the unsatisfiable core encodes a sub-formula $\sigma(\omega)^{uc}$ of $\sigma(\omega)$ which already contributes to the violation of the encoded model checking problem. While $\sigma(\omega)$ characterises the *entire* spurious witness ω, $\sigma(\omega)^{uc}$ characterises a *segment* of ω that is already spurious by itself. Both complements $\overline{\sigma}(\omega)$ and $\overline{\sigma}(\omega)^{uc}$ are admissible constraints for ruling out spurious behaviour. However, while the spurious witness constraint $\overline{\sigma}(\omega)$ excludes ω only, the spurious segment constraint $\overline{\sigma}(\omega)^{uc}$ excludes *all* paths that exhibit the spurious segment. We can immediately obtain $\sigma(\omega)^{uc}$ from $[\![\sigma(\omega)]\!]_{uc}$ by applying the inverse $[\![\,\cdot\,]\!]^{-1}$ of the encoding $[\![\,\cdot\,]\!]$: $\sigma(\omega)^{uc} := [\![[\![\sigma(\omega)]\!]_{uc}]\!]^{-1}$.

Beside extracting the stronger spurious fragment constraints, unsatisfiable cores allow for further improvements of *SATWRC*. Note that it is possible that certain parts of an unsatisfiable core $[\![M, k]\!]_{uc} \cup [\![I]\!]_{uc} \cup [\![\sigma(\omega)]\!]_{uc} \cup [\![\psi, k]\!]_{uc}$ are the empty set, e.g. $[\![I]\!]_{uc} = \varnothing$. In the following, we will omit the empty parts of an unsatisfiable core of an encoded model checking problem and we assume that all shown parts are *non-empty*. In Theorem 3 we consider the different cases of unsatisfiable cores of ${}_{A^\omega}[\![M, I, \sigma(\omega), \psi, k]\!]$ with empty parts and we assign an admissible extension of the cumulative constraint Σ_{k+j} with $j \in \mathbb{N}$ to each case.

Theorem 3
Let ϕ_x be an unsatisfiable core of the encoding ${}_{A^\omega}[\![M, I, \sigma(\omega), \psi, k]\!]$ of a three-valued bounded model checking problem local to an unconfirmed witness ω. Then

in bound iteration $k+j$ with $j \in \mathbb{J}$ it is admissible to extend the cumulative path constraint Σ_{k+j} of the encoding ${}_{A}[\![M, I, \Sigma_{k+j}, \psi, k+j]\!]$ of the corresponding global model checking problem as follows: $\Sigma_{k+j} := \Sigma_{k+j} \wedge \varphi_x$ with

$\phi_1 = [\![M,k]\!]_{uc} \cup [\![I]\!]_{uc} \cup [\![\sigma(\omega)]\!]_{uc} \cup [\![\psi,k]\!]_{uc}$	$\varphi_1 = \overline{\sigma}(\omega)^{uc}$	$\mathbb{J} = \{0\}$
$\phi_2 = [\![M,k]\!]_{uc} \cup [\![I]\!]_{uc} \cup [\![\sigma(\omega)]\!]_{uc}$	$\varphi_2 = \overline{\sigma}(\omega)^{uc}$	$\mathbb{J} = \mathbb{N}$
$\phi_3 = [\![M,k]\!]_{uc} \cup [\![\sigma(\omega)]\!]_{uc} \cup [\![\psi,k]\!]_{uc}$	$\varphi_3 = \overline{\sigma}(\omega)^{uc}_j$	$\mathbb{J} = \mathbb{N}$
$\phi_4 = [\![M,k]\!]_{uc} \cup [\![I]\!]_{uc} \cup [\![\psi,k]\!]_{uc}$	$\varphi_4 = \mathit{false}$	$\mathbb{J} = \{0\}$
$\phi_5 = [\![M,k]\!]_{uc} \cup [\![\sigma(\omega)]\!]_{uc}$	$\varphi_5 = \bigwedge_{l=0}^{j} \overline{\sigma}(\omega)^{uc}_l$	$\mathbb{J} = \mathbb{N}$
$\phi_6 = [\![M,k]\!]_{uc} \cup [\![\psi,k]\!]_{uc}$	$\varphi_6 = \mathit{false}$	$\mathbb{J} = \{0\}$
$\phi_7 = [\![M,k]\!]_{uc} \cup [\![I]\!]_{uc}$	$\varphi_7 = \mathit{false}$	$\mathbb{J} = \mathbb{N}$

where $\overline{\sigma}(\omega)^{uc} = \neg[\![[\![\sigma(\omega)]\!]_{uc}]\!]^{-1}$.

Proof. See `http://github.com/ssfm-up/TVMC/raw/master/SCICOProofs.pdf`

As we can see, Theorem 3 is the SAT-based version of Theorem 2. The new theorem allows us to implicitly determine causes of violation of a model checking problem and to derive corresponding constraints via unsatisfiable core extraction. Subsequently, we outline some of the cases that are considered in Theorem 3. If the unsatisfiable core is *initial state-independent* (ϕ_2 with $[\![I]\!]_{uc} = \varnothing$), then the extracted spurious segment constraint $\overline{\sigma}(\omega)^{uc}$ is an admissible constraint in all bound iterations $k+j$. If the unsatisfiable core is *property-independent* (ϕ_4 with $[\![\psi]\!]_{uc} = \varnothing$), then the j-increment of the extracted spurious segment constraint $\overline{\sigma}(\omega)^{uc}$ is an admissible constraint in all bound iterations $k+j$. If the unsatisfiable core is *fully-independent*, i.e. *both* initial state-independent and property-independent (ϕ_5), then all l-increments of the extracted spurious segment constraint $\overline{\sigma}(\omega)^{uc}$ with $0 \leq l \leq j$ are admissible constraints in all bound iterations $k+j$.

The admissible use, reuse and incrementation of spurious segment constraints can be straightforwardly integrated into a combination of the algorithms *k-IND* and *SATWRC* by storing all reusable constraints in a global set Σ_{global} and by initialising Σ_k with Σ_{global} in each bound iteration. Note that the index increments of constraints in Σ_{global} may be parameterised with regard to the current bound. Thus, a constraint $\overline{\sigma}(\omega)^{uc}_j$ in iteration k may be adapted to $\overline{\sigma}(\omega)^{uc}_{j+1}$ in iteration $k+1$. In the subsequent section we introduce the implementation of our approach and we present experimental results.

9. Implementation and Experiments

We have prototypically implemented our novel refinement approach on top of the SAT-based three-valued bounded model checker Tvamcus [2]. Our tool takes a concurrent system *Sys* within integer arithmetic as a first input. It supports almost all control structures of the C language as well as *int*, *bool* and *semaphore* as data types. The second input is a temporal logic formula that is either of the form $\mathbf{G}\, safe$ (safety) or $\mathbf{GF}\, progress$ (liveness) where *safe* and *progress* are predicate expressions over the control flow or the variables of the system. The tool employs a three-valued abstractor [1] that automatically builds abstract control flow graphs corresponding to the system and an initial predicate set A that covers the predicates referenced in the temporal logic formula to be checked. In case the input is a liveness formula, we apply the state recording translation to the abstract control flow graphs, which reduces liveness checking to safety checking. Hence, the formula to be evaluated will be always of the form $\mathbf{G}\, safe$. Tvamcus iterates over the bound starting with $k = 0$. In each bound iteration, the three-valued bounded model checking problems ${}_A[base]_k$ and ${}_A[step]_{k+1}$ corresponding to the inputs are encoded into propositional logic. For each, the base case and the inductive step we run our refinement algorithm until both invocations yield a definite result. Within Tvamcus we use an implementation of MiniSat [20] for processing the satisfiability problems and for extracting unsatisfiable cores. If the base case yields *true*, we have detected a property violation. If both the base case and the inductive step yield *false*, we have proven that no property violation exists. Otherwise our tool proceeds to the next bound iteration and repeats the computations for the incremented bound. Generated spurious segment constraints are reused between bound iterations and between the base case and the inductive step according to our results in terms of the admissible reuse of constraints.

In our experimental evaluation, we considered implementations of mutual exclusion algorithms for multiple processes, namely Dijkstra's mutual exclusion algorithm and Lamport's bakery algorithm. Both algorithms ensure *mutual exclusion* and *progress* in the sense that some process will be always able to enter the critical section. While Dijkstra's algorithm is prone to *starvation*, Lamport's algorithm ensures starvation-freedom. Moreover, we considered a *deadlock*-prone instantiation of a semaphore-based dining philosopher system with ten philosophers and forks. We checked safety and liveness properties of the systems under the assumption of weak fairness in the case of liveness. In experiments we compared our model checker Tvamcus with Spin [15]. Spin is an established model checking tool for verifying temporal logic properties of concurrent systems under fairness. Thus, Tvamcus and Spin focus on similar kinds of verification tasks. We employed our Tvamcus model checker in two different modes. In Tvamcus-AR we used standard abstraction refinement, whereas in Tvamcus-WRC we used our novel refinement approach with mul-

[2] available at `https://github.com/siocnarff/tvamcus`

tiple models and constraint generation. The experiments were conducted on a 2.6 GHz Intel Core i5 system with 8 GB. The experimental results are depicted in Table 2. A "✓" indicates that the property holds for the system, whereas a "✗" indicates that the property does not hold. Moreover, a "-" means that the model checker did not solve the task within 2 hours.

Table 2: Experimental results

		Tvamcus-AR		Tvamcus-WRC		Spin
case study	processes	maximum predicates	time	maximum predicates	time	time
Dijkstra Mutual Exclusion (✓)	2	10	2.0s	8	0.8s	0.1s
	3	15	9.2s	12	3.2s	0.2s
	4	20	19.9s	16	4.8s	3.9s
Dijkstra Progress (✓)	2	16	14.2s	10	1.3s	0.3s
	3	24	227s	15	9.3s	4.8s
	4	32	-	20	178s	166s
Dijkstra Starvation-Freedom (✗)	2	16	14.7s	10	2.8s	0.4s
	3	24	806s	15	20.5s	9.1s
	4	32	-	20	343s	307s
Lamport Progress (✓)	2	18	248s	10	12.6s	17.7s
	3	30	-	18	169s	-
Philosophers Deadlock-Freedom (✗)	10	30	98.1s	20	54.6s	93.6s

If we compare the results for Tvamcus-AR and Tvamcus-WRC we can observe that the multi-model approach allows to significantly reduce the maximum number of predicates in the problems to be solved. The reduced amount of predicates results from the fact that Tvamcus-WRC makes use of a global model that is solely defined over the set of initial predicates, and of local models where the initial predicates are fixed by an unconfirmed witness and only the refinement predicates contribute to the state space complexity. Constraints derived from local models are added to the global model without increasing its state space complexity. Operating on smaller models under Tvamcus-WRC results in considerably faster model checking in all experiments in comparison to Tvamcus-AR. If we compare the results for Tvamcus-WRC and Spin for the Dijkstra case studies we can see that Spin clearly outperforms Tvamcus-WRC for smaller instances with two and three processes. In the case of four processes Spin is still faster than Tvamcus-WRC but the differences are less significant. The Lamport and Philosophers case studies show that Tvamcus-WRC is even capable of outperforming Spin for the specified instances. These promising results demonstrate that our multi-model approach with constraint reuse can compete with existing tools. We have ideas for further enhancing our approach that we discuss in the outlook of this article.

10. Related Work

Some of the earliest work on the application of three-valued reasoning about software specifications and their properties is [21]. There, however, no reasoning algorithm was provided to deal with the *unknown* in a constructive manner. More recent work on three-valued reasoning about system specifications can be found in [22], whereby techniques of *theorem-proving* are used additionally to deal with *unknown* model checking results. Other interesting applications of three-valued model checking can be found in the domain of *multi-agent systems* [23], whereas model checking of temporal logic properties with *multi-valued* logics is described in [24].

Our verification technique is related to existing approaches for improving the classical *abstract-check-refine* paradigm [4, 25]. *Lazy abstraction* [26, 27, 28, 29] is a concept that builds and refines a single abstract model where different parts of the model exhibit different degrees of precision. This is achieved by adding refinement predicates only at parts where they are required for proving the spuriousness of witnesses. The major difference to our approach is that we work with one full and multiple partial models. Only the partial models are refined in order to *prove* whether a particular witness is spurious or not. In the full model we take proven spuriousness as a *fact* in order to prune the state space. The separation of proving and eliminating spuriousness enables us to conduct verification on smaller models in comparison to lazy abstraction where only a single model is used. Another refinement strategy for SAT-based model checking, which also uses a combination of over- and under-approximations in order to solve reachability problems can be found in [30, 31].

The propositional logic encoding of bounded model checking problems has been initially introduced in [32]. An improved encoding that is linear in both the size of the formula to be checked and the length of the bound has been presented in [33]. Both techniques are incomplete. Liveness model checking is supported by explicitly evaluating liveness formulae on lasso-shaped paths. In our approach we enabled complete liveness checking by reducing liveness to reachability via state recording [13] and by employing induction. Earliest approaches to complete bounded model checking via k-induction can be found in [11, 34] where induction is used for the verification of safety properties of finite-state systems. k-induction for infinite-state systems such as communication protocols and timed automata has been proposed in [35]. In [36] we can find the use of counterexample-guided k-induction for bug detection, where counterexamples produced from over-approximating the loops are exploited to shorten the number of steps that are required to find bugs. Other upper-bound considerations, also with the purpose of making bounded model checking complete, can be found in [37].

Another related approach is *local abstraction refinement* [38] which extends the lazy abstraction idea. The technique also adds predicates only to relevant parts of the model. While a new predicate typically splits an abstract state in two refined states, local abstraction refinement uses heuristics for determining whether a single refined state is sufficient for the underlying verification task.

This enables smaller state spaces. The approach is still based on a single model, and thus, does not have the same state space reduction capabilities as our multi-model approach.

Our work also is related to *conditional model checking* (CMC) [39], which reformulates model checking as follows: If model checking fails (due to state explosion) to fully prove or disprove the property of interest, then it at least returns a condition under which the property holds. This allows for a sequential combination of model checking runs where a first run generates a condition and a second run checks whether the condition holds. Our approach can be regarded as an application and generalisation of the CMC idea in the context of abstraction refinement. We take unconfirmed witnesses as conditions for our partial models and we use conditions for excluding spurious witnesses in the full model.

Whereas our approach exploits control flow knowledge for the purpose of abstraction refinement in the context of SAT solving, the exploitation of control flow knowledge to obtain better results in SMT solving is described in [40], whereas the approach of [41] is designed to verify program properties by way of CHC-solving with constrained Horn clauses. An automatic on-the-fly decomposition of large specifications into their most interesting or most relevant parts can be found in [42].

The practical applicability of SAT-based model checking for various application purposes is well known, for instance [43] in the railway domain. While we are using SAT solvers to reason about temporal logic properties of concurrent systems, it is in principle possible to use SAT solving also for reasoning about other modal logic system properties that are not temporal e.g. [44].

In a wider context one can observe that model checking via SAT solving, with potentially *unknown* results for any α-interpretation of the given logical atoms, is somewhat similar to the path coverage problem in software testing where different value interpretations of the input variables of a program lead to different execution paths being covered, whereby the problem arises which execution paths remain *unknown* for a given set of input values. In [45] we can find an incremental method that generates input data from atomic predicates in software specifications with the aim of achieving full path coverage. Somewhat similar to our approach, distance-based reasoning is applied in [45] to incrementally refine the generation of input data leading to better path coverage in the software model under analysis.

11. Conclusion and Outlook

We presented an iterative abstraction refinement technique for the verification of temporal logic properties of concurrent software systems. The novelty of our approach is that we use separate models for producing abstract witness paths and for checking whether witnesses are definite or spurious. Our local models are restricted to refinements of particular witnesses only. The abstract state space of our global model is pruned via constraints derived from local models. We hereby gain precision in the global model without increasing its state space. Our multi-model approach allows for a significant reduction of the state space complexity in comparison to single-model approaches. It comes at the cost of an increased number of constraint generation iterations. Our *constraint strengthening* concept enables us to diminish this number, which gives us a space- *and* time-efficient verification technique. Our approach employs *satisfiability-based bounded model checking* [46], and thus, profits from the capabilities of today's SAT solvers. Since bounded model checking is inherently incomplete, we integrated the *k-induction principle* [47] into our verification technique. This iterative approach reduces an unbounded model checking problem to two bounded model checking problems, which enables us to perform *complete* verification via satisfiability solving. We showed that constraints for ruling out spurious behaviour can be generated via *unsatisfiable core extraction* [8] and we introduced a concept for the admissible *reuse of constraints* between bound iterations. In general, k-induction is limited to model checking *safety* properties. We additionally enabled *liveness* checking by applying the *state recording translation* [13] to the systems to be verified, which reduces liveness to safety checking. In experiments we received promising performance results with our new approach.

As future work we intend to enhance our constraint strengthening concept based on ideas adopted from *symmetry reduction* [48] and *partial-order reduction* [49]. Since we focus on the verification of concurrent *systems*, these systems typically exhibit a considerable amount of symmetry. If a constraint has been generated that reveals that a *particular* pair of processes will never be at a certain location at the same time, this observation may be transferable to *arbitrary* pairs based on symmetry arguments. Similarly, certain behaviour may be spurious, independent of the order in which the processes execute their operations. Thus, a constraint that does not only exclude one but all orders that lead to spuriousness may be admissible as well. We are working on a concept for detecting symmetric and order-independent constraints that will allow us to rule out spurious behaviour on a larger scale. Moreover, we plan to extend our verification technique to systems beyond linear integer arithmetic and to multi-agent systems.

References

[1] J. Schrieb, H. Wehrheim, D. Wonisch, Three-valued spotlight abstractions, in: A. Cavalcanti, D. R. Dams (Eds.), FM 2009: Formal Methods, Vol. 5850 of LNCS, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 106–122.

[2] G. Bruns, P. Godefroid, Model checking partial state spaces with 3-valued temporal logics, in: N. Halbwachs, D. Peled (Eds.), Computer Aided Verification, Vol. 1633 of LNCS, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 274–287.

[3] N. Timm, S. Gruner, Three-valued bounded model checking with cause-guided abstraction refinement, Science of Computer Programming 175

(2019) 37 – 62. doi:https://doi.org/10.1016/j.scico.2019.02.002. URL http://www.sciencedirect.com/science/article/pii/S0167642319300206

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: E. A. Emerson, A. P. Sistla (Eds.), Computer Aided Verification, Vol. 1855 of LNCS, Springer Berlin Heidelberg, 2000, pp. 154–169.

[5] N. Timm, S. Gruner, Abstraction refinement with path constraints for 3-valued bounded model checking, in: C. Artho, P. C. Ölveczky (Eds.), Formal Techniques for Safety-Critical Systems, Springer International Publishing, Cham, 2019, pp. 139–157.

[6] O. Strichman, Accelerating bounded model checking of safety properties, Formal Methods in System Design 24 (1) (2004) 5–24. doi:10.1023/B:FORM.0000004785.67232.f8. URL https://doi.org/10.1023/B:FORM.0000004785.67232.f8

[7] N. Timm, S. Gruner, M. Harvey, A bounded model checker for three-valued abstractions of concurrent software systems, in: L. Ribeiro, T. Lecomte (Eds.), Formal Methods: Foundations and Applications, Vol. 10090 of LNCS, Springer International Publishing, Cham, 2016, pp. 199–216.

[8] A. Nadel, Boosting minimal unsatisfiable core extraction, in: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, FMCAD '10, FMCAD Inc, Austin, TX, 2010, pp. 221–229.

[9] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, J. Worrell, Linear completeness thresholds for bounded model checking, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 557–572.

[10] A. Nadel, V. Ryvchin, O. Strichman, Ultimately incremental sat, in: C. Sinz, U. Egly (Eds.), SAT 2014, Springer International Publishing, 2014, pp. 206–218.

[11] N. Een, N. Sörensson, Temporal induction by incremental sat solving, Electronic Notes in Theoretical Computer Science 89 (4) (2003) 543 – 560, bMC 2003. doi:https://doi.org/10.1016/S1571-0661(05)82542-3.

[12] N. Timm, S. Gruner, M. Harvey, Constraint reusing and k-induction for three-valued bounded model checking, in: T. Massoni, M. R. Mousavi (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2018, pp. 126–143.

[13] A. Biere, C. Artho, V. Schuppan, Liveness checking as safety checking, Electronic Notes in Theoretical Computer Science 66 (2) (2002) 160 – 177, fMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop).

doi:https://doi.org/10.1016/S1571-0661(04)80410-9.
URL http://www.sciencedirect.com/science/article/pii/S1571066104804109

[14] L. Moura, N. Bjrner, Z3: An efficient SMT solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Vol. 4963 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.
URL http://dx.doi.org/10.1007/978-3-540-78800-3_24

[15] G. J. Holzmann, The model checker spin, IEEE Transactions on software engineering 23 (5) (1997) 279–295.

[16] N. Timm, H. Wehrheim, M. Czech, Heuristic-guided abstraction refinement for concurrent systems, in: T. Aoki, K. Taguchi (Eds.), ICFEM, Vol. 7635 of LNCS, Springer, 2012, pp. 348–363. doi:10.1007/978-3-642-34281-3_25.

[17] M. Fitting, Kleene's three valued logics and their children, Fundamenta Informaticae 20 (1-3) (1994) 113–131.

[18] H. Wehrheim, Bounded model checking for partial Kripke structures, in: J. S. Fitzgerald, A. E. Haxthausen, H. Yenigun (Eds.), Theoretical Aspects of Computing — ICTAC 2008, Vol. 5160 of LNCS, Springer Berlin Heidelberg, 2008, pp. 380–394.

[19] A. Nadel, Boosting minimal unsatisfiable core extraction, in: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, FMCAD '10, FMCAD Inc, Austin, TX, 2010, pp. 221–229.
URL http://dl.acm.org/citation.cfm?id=1998496.1998537

[20] N. Sorensson, N. Een, Minisat v1. 13-a sat solver with conflict-clause minimization, SAT 2005 (53) (2005) 1–2.

[21] D. Kourie, An approach to defining abstractions, refinements and enrichments.

[22] A. Bernasconi, C. Menghi, P. Spoletini, L. D. Zuck, C. Ghezzi, From model checking to a temporal proof for partial models, in: A. Cimatti, M. Sirjani (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2017, pp. 54–69.

[23] A. Lomuscio, J. Michaliszyn, Verifying multi-agent systems by model checking three-valued abstractions, in: Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS '15, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2015, pp. 189–198.
URL http://dl.acm.org/citation.cfm?id=2772879.2772907

[24] Y. Li, M. Droste, L. Lei, Model checking of linear-time properties in multi-valued systems, Information Sciences 377 (2017) 51 – 74. doi:https://doi.org/10.1016/j.ins.2016.10.030. URL http://www.sciencedirect.com/science/article/pii/S0020025516313548

[25] S. Shoham, O. Grumberg, 3-valued abstraction: More precision at less cost, Information and Computation 206 (11) (2008) 1313–1333.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, ACM, New York, NY, USA, 2002, pp. 58–70.

[27] K. Madhukar, M. Srivas, B. Wachter, D. Kroening, R. Metta, Verifying synchronous reactive systems using lazy abstraction, in: 2015 Design, Automation Test in Europe Conference Exhibition (DATE), 2015, pp. 1571–1574.

[28] R. Degiovanni, P. Ponzio, N. Aguirre, M. Frias, Improving lazy abstraction for SCR specifications through constraint relaxation, Software Testing, Verification and Reliability 28 (2) (2018) e1657.

[29] K. Hsu, R. Majumdar, K. Mallik, A. Schmuck, Lazy abstraction-based control for reachability, CoRR abs/1804.02722. arXiv:1804.02722.

[30] J. Li, S. Zhu, Y. Zhang, G. Pu, M. Y. Vardi, Approximate reachability, arXiv preprint arXiv:1611.04946.

[31] J. Li, S. Zhu, Y. Zhang, G. Pu, M. Y. Vardi, Safety model checking with complementary approximations, in: Proceedings of the 36th International Conference on Computer-Aided Design, ICCAD '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 95–100. URL http://dl.acm.org/citation.cfm?id=3199700.3199713

[32] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without bdds, in: W. R. Cleaveland (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 193–207.

[33] T. Latvala, A. Biere, K. Heljanko, T. Junttila, Simple bounded ltl model checking, in: A. J. Hu, A. K. Martin (Eds.), Formal Methods in Computer-Aided Design, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 186–200.

[34] M. Sheeran, S. Singh, G. Stålmarck, Checking safety properties using induction and a sat-solver, in: W. A. Hunt, S. D. Johnson (Eds.), Formal Methods in Computer-Aided Design, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 127–144.

[35] L. de Moura, H. Rueß, M. Sorea, Bounded model checking and induction: From refutation to verification, in: W. A. Hunt, F. Somenzi (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 14–26.

[36] M. Y. R. Gadelha, L. C. Cordeiro, D. A. Nicole, Counterexample-guided k-induction verification for fast bug detection, CoRR abs/1706.02136. arXiv:1706.02136.
URL http://arxiv.org/abs/1706.02136

[37] I. Konnov, H. Veith, J. Widder, On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability, Information and Computation 252 (2017) 95 – 109. doi:https://doi.org/10.1016/j.ic.2016.03.006.
URL http://www.sciencedirect.com/science/article/pii/S0890540116000432

[38] H. Fecher, S. Shoham, Local abstraction-refinement for the mu-calculus, in: D. Bošnački, S. Edelkamp (Eds.), Model Checking Software, Vol. 4595 of LNCS, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 4–23.

[39] D. Beyer, T. A. Henzinger, M. E. Keremoglu, P. Wendler, Conditional model checking: A technique to pass information between verifiers, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, 2012, pp. 57:1–57:11.

[40] J. Chen, F. He, Control flow-guided smt solving for program verification, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, ACM, New York, NY, USA, 2018, pp. 351–361. doi:10.1145/3238147.3238218.
URL http://doi.acm.org/10.1145/3238147.3238218

[41] E. DE ANGELIS, F. FIORAVANTI, A. PETTOROSSI, M. PROIETTI, Predicate pairing for program verification, Theory and Practice of Logic Programming 18 (2) (2018) 126?166. doi:10.1017/S1471068417000497.

[42] S. Apel, D. Beyer, V. Mordan, V. Mutilin, A. Stahlbauer, On-the-fly decomposition of specifications in software model checking, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 349–361. doi:10.1145/2950290.2950349.
URL http://doi.acm.org/10.1145/2950290.2950349

[43] P. James, Sat-based model checking and its applications to train control systems, Swansea University.

[44] T. Caridroit, J.-M. Lagniez, D. Le Berre, T. de Lima, V. Montmirail, A sat-based approach for solving the modal logic s5-satisfiability problem, in: Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[45] P. Zhao, S. Liu, A software tool to support the "vibration" method, in: C. Tian, F. Nagoya, S. Liu, Z. Duan (Eds.), Structured Object-Oriented Formal Language and Method, Springer International Publishing, Cham, 2018, pp. 171–186.

[46] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, Bounded model checking., Handbook of Satisfiability 185 (2009) 457–481.

[47] T. Wahl, The k-induction principle, Northeastern University, College of Computer and Information Science (2013) 1–2.

[48] A. Miller, A. Donaldson, M. Calder, Symmetry in temporal logic model checking, ACM Computing Surveys (CSUR) 38 (3) (2006) 8.

[49] D. Peled, Partial-Order Reduction, Springer International Publishing, Cham, 2018, pp. 173–190. doi:10.1007/978-3-319-10575-8_6. URL https://doi.org/10.1007/978-3-319-10575-8_6