Three-Valued Bounded Model Checking with Cause-Guided Abstraction Refinement

Nils Timm*, Stefan Gruner

Department of Computer Science, University of Pretoria, South Africa

Abstract

We present a technique for verifying concurrent software systems via SAT-based three-valued bounded model checking. It is based on a direct transfer of the system to be analysed and a temporal logic property into a propositional logic formula that encodes the corresponding model checking problem. In our approach we first employ three-valued abstraction which gives us an abstract system defined over predicates with the possible truth values *true*, *false* and *unknown*. The state space of the abstract system is then logically encoded. The verification result of the encoded three-valued model checking problem can be obtained via two satisfiability checks, one for an over-approximation of the encoding and one for an under-approximation. *True* and *false* results can be immediately transferred to the system under consideration.

In case of an *unknown* result, the current abstraction is too imprecise for a definite outcome. In order to achieve the necessary precision, we have developed a novel *cause-guided abstraction refinement* procedure. An *unknown* result always entails a truth assignment that only satisfies the over-approximation, but not the under-approximation. We determine the propositional logic clauses of the under-approximation that are not satisfied under the assignment. These clauses contain *unknown* as a constant. Each *unknown* is associated with a *cause of uncertainty* that refers to missing predicates that are required for a definite model checking result. Our procedure adds these predicates to the abstraction and constructs the encoding corresponding to the refined model checking problem. The procedure is iteratively applied until a definite result can be obtained.

We have integrated our novel refinement approach into a SAT-based three-valued bounded model checker. In an experimental evaluation, we show that our approach allows to automatically and quickly reach the right level of abstraction for solving software verification tasks.

Key words: Three-valued abstraction, Bounded model checking, Cause-guided abstraction refinement, Concurrent software systems, Fairness

*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 a well-established technique in software verification. It proceeds by generating an abstract state space model of the system to be analysed over predicates with the possible truth values *true*, *false* and *unknown*, where the latter value is used to represent the loss of information due to abstraction. For concurrent software systems composed of many processes, 3VA does not only replace concrete variables by predicates. It also abstracts away entire processes by summarising them into a single approximative component [2], which allows for a substantial reduction of the state space. The evaluation of temporal logic properties on models constructed via three-valued abstraction is known as *three-valued model checking* (3MC) [3]. In three-valued model checking there exist three possible outcomes: *true* and *false* results can be immediately transferred to the modelled system, whereas an *unknown* result does not allow to draw any conclusions about the properties of the system.

Verification techniques based on three-valued abstraction and model checking typically assume that an *explicit* three-valued state space model corresponding to the system to be analysed is constructed and explored [3]. However, explicit-state model checking is known for its high memory demands in comparison to *symbolic* model checking techniques like BDD-based model checking [4] and satisfiability-based bounded model checking (BMC) [5]. The benefits of bounded model checking are that its compressed state space representation as a propositional logic formula allows to handle larger systems than explicit-state techniques, and that its performance profits from the advancements in the SAT solver technology. Although there exist a few works on *three-valued bounded model checking*, these approaches are either solely defined for hardware systems [6], or they require an explicit state space model as input which is then symbolically encoded in propositional logic [7]. It is however not efficient to first translate a given system into an *explicit* state space model before encoding it symbolically for bounded model checking.

In [8] we presented a verification technique for concurrent software systems that allows to directly transfer an abstracted input system Sys, a temporal logic property ψ and a bound $b \in \mathbb{N}$ into a propositional logic formula $[\![Sys, \psi]\!]_b$ that encodes the corresponding three-valued bounded model checking problem. $[\![Sys, \psi]\!]_b$ is defined over a set of Boolean atoms and the constants *true*, *false* and *unknown*, where the latter only occurs non-negated in the formula. The result of the encoded model checking problem can be obtained via two satisfiability checks. The first check considers an over-approximation $[\![Sys, \psi]\!]_b^+$ of the encoding where all *unknowns* are assumed to be *true*. The second check considers an under-approximation $[\![Sys, \psi]\!]_b^-$ where all *unknowns* are assumed to be *false*. Unsatisfiability of the over-approximation implies that the bounded model checking result is *false*. Satisfiability of the under-approximation implies that the result is *true*. If only the over-approximation is satisfiable, then the model checking result is *unknown*, which indicates that the current abstraction is too coarse for a definite outcome. While our technique proposed in [8] allows for a compact state space encoding and for the efficient verification of safety and liveness properties under fairness via SAT solving, it does not offer a concept for abstraction refinement in case of an *unknown* result.

In this article, we extend our previous work by introducing a *cause-guided abstraction refinement* procedure for SAT-based three-valued bounded model checking. For this, we enhanced our propositional logic encoding by adding the *cause of uncertainty* to each *unknown* that occurs in the formula $[\![Sys, \psi]\!]_b$. A cause refers to missing information in the current abstraction. We developed a technique for determining whether this information is relevant for the verification task to be solved: If there exists a truth assignment α that satisfies the over-approximation $[\![Sys, \psi]\!]_b^+$ but not the under-approximation $[\![Sys, \psi]\!]_b^-$, then α characterises an unconfirmed witness path for the temporal logic property ψ, i.e. a path with some *unknown* transitions or predicates. Hence, our approach operates with *implicit* paths given by truth assignments. In contrast to most counterexample-guided abstraction refinement (CEGAR) techniques [9], explicit paths do not need to be generated. We next determine all propositional logic clauses of the under-approximation that are not satisfied under the assignment α. Our encoding has the property that these clauses contain at least one *unknown*, and its associated *cause* refers to missing predicates that are required for a definite model checking result. Our cause-guided refinement procedure now adds these predicates to the abstraction and constructs the encoding corresponding to the refined model checking problem. The procedure is iteratively applied until a definite result can be obtained.

We have integrated iterative cause-guided abstraction refinement into our SAT-based three-valued bounded model checking tool TVMC, which is available at `www.github.com/ssfm-up/TVMC`. In an experimental evaluation, we show that our novel refinement approach allows to automatically and quickly reach the right level of abstraction for solving software verification tasks. We also demonstrate in a number of case studies that TVMC outperforms the similar tool 3SPOT [2] in most cases. Moreover, we present two enhancements of our verification technique based on existing work that we have implemented as well: *Temporal induction* [10] allows us to translate our results of bounded model checking of safety properties into *unbounded model checking results*, and *spotlight abstraction* [11] enables us to verify *parameterised systems* composed of arbitrarily many uniform processes.

The remainder of this article is organised as follows. In Section 2 we introduce the concurrent systems that we consider in our software verification approach. Section 3 provides the background on three-valued abstraction and bounded model checking. Section 4 introduces our propositional logic encoding of software verification tasks and presents a theorem which states that the satisfiability result for an encoded verification task is equivalent to the result of the corresponding three-valued bounded model checking problem. In Section 5 we show how our encoding can be augmented with fairness constraints for the verification of liveness properties. Section 6 introduces our novel cause-guided abstraction refinement technique. In Section 7 we present the implementation of our approach, we introduce enhancements that have been implemented as well, and we present experimental results. Section 8 discusses related work. We conclude this paper in Section 9 and give an outlook on future work.

2. Concurrent Software Systems

We start with a brief introduction to the systems that we consider in our work. 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 $Var = Var_s \cup \bigcup_{i=1}^{n} Var_i$ where Var_s is a set of shared variables and $Var_1, \ldots, Var_n$ are sets of local variables associated with the processes $P_1, \ldots, P_n$, respectively. The state space 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. An example for a concurrent system implementing mutual exclusion is depicted in Figure 1.

$$y : \texttt{semaphore where } y = 1;$$

$$P_1 :: \left[\begin{array}{l} \texttt{loop forever do} \\ \left[\begin{array}{l} \texttt{0: acquire } (y,1); \\ \texttt{1: CRITICAL} \\ \quad \texttt{release } (y,1); \end{array}\right] \end{array}\right] \| \; P_2 :: \left[\begin{array}{l} \texttt{loop forever do} \\ \left[\begin{array}{l} \texttt{0: acquire } (y,1); \\ \texttt{1: CRITICAL} \\ \quad \texttt{release } (y,1); \end{array}\right] \end{array}\right]$$

Figure 1: Concurrent system *Sys*.

Here we have two processes operating on a shared counting semaphore variable y. Processes P_i can be formally represented as *control flow graphs* (CFGs) $G_i = (Loc_i, \delta_i, \tau_i)$ where $Loc_i = \{[0]_2, \ldots, [|Loc_i|]_2\}$ is a set of control locations given as binary numbers, $\delta_i \subseteq Loc_i \times Loc_i$ is a transition relation, and $\tau_i : Loc_i \times Loc_i \to Op$ 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 variables. The* set of operations Op *on these variables consists of all statements of the form* $assume(e) : v_1 := e_1, \ldots, v_m := e_m$ *where* $e, e_1, \ldots e_m$ *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 $assume(e)$. Moreover, we omit the *assume* part completely if the guard is *true*. 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 $release(y, 1)$.

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 $Loc = \bigtimes_{i=1}^{n} Loc_i$. G is the product graph of all individual CFGs. We assume that initially all processes of a concurrent system are at location 0. Moreover, we assume that a deterministic initialisation of the system variables is given by an assertion ϕ over Var. In our example we have that $\phi = (y = 1)$. Now, a computation

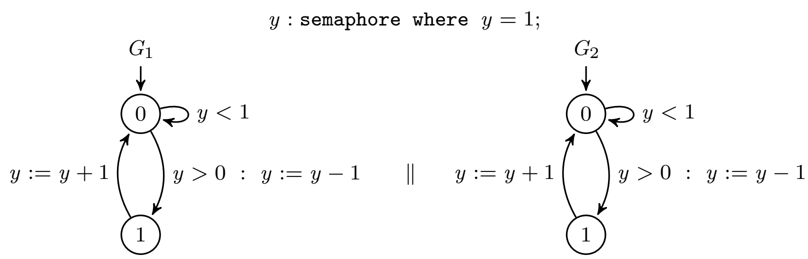

Figure 2: Control flow graphs G_1 and G_2 composed in parallel.

of a concurrent system corresponds to a sequence where in each step one process is non-deterministically selected and the operation at its current location is attempted to be executed. In case the execution is not blocked by a guard, the variables are updated according to the assignment part and the process advances to the consequent control location. For verifying properties of concurrent systems typically only *fair* computations where all processes infinitely often proceed are considered. We will discuss our notion of fairness in more detail in Section 5. The overall state space S of a concurrent system corresponds to the set of states over Var combined with the possible locations, i.e. $S = Loc \times S_{Var}$. Hence, each state in S is a tuple $\langle l, s \rangle$ with $l = (l_1, \dots, l_n) \in Loc$ and $s \in S_{Var}$.

Control flow graphs allow 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* [2]. 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 [12]. In abstract systems operations do not refer to concrete variables but to predicates $Pred = \{p_1, \dots, p_m\}$ over Var with the three-valued domain $\{true, unknown, false\}$. *Unknown*, typically abbreviated by $\perp$, is a valid truth value as we operate with the three-valued *Kleene logic* $\mathcal{K}_3$ [13] whose semantics is given by the truth tables in Figure 3.

$\wedge$	*true*	$\perp$	*false*
true	*true*	$\perp$	*false*
$\perp$	$\perp$	$\perp$	*false*
false	*false*	*false*	*false*

$\vee$	*true*	$\perp$	*false*
true	*true*	*true*	*true*
$\perp$	*true*	$\perp$	$\perp$
false	*true*	$\perp$	*false*

$\neg$	
true	*false*
$\perp$	$\perp$
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 $\perp \leq_{\mathcal{K}_3} true$, $\perp \leq_{\mathcal{K}_3} false$, and *true*, *false* incomparable. Operations in abstract systems are of the following form:

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

where $a, b, a_1, b_1, \ldots, a_m, b_m$ are logical expressions over *Pred* and $choice(a, b)$-expressions have the following semantics:

Definition 2 (Choice Expressions).
Let s be a state over a set of three-valued predicates Pred. Moreover, let a and b be logical expressions over Pred. Then

$$s\left(choice\left(a,b\right)\right) = \begin{cases} \mathit{true} & \text{\textit{if, and only if,}} \;\; s(a) \text{ is } \mathit{true}, \\ \mathit{false} & \text{\textit{if, and only if,}} \;\; s(b) \text{ is } \mathit{true}, \\ \perp & \text{\textit{else.}} \end{cases}$$

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: $s(a) = \mathit{true} \;\Rightarrow\; s(b) = \mathit{false}$ and $s(b) = \mathit{true} \;\Rightarrow\; s(a) = \mathit{false}$. In particular, this implies that $s(a)$ and $s(b)$ are never both *true*. Moreover, the following equivalences hold:

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

A three-valued expression $choice(a, b)$ over *Pred* approximates a Boolean expression e over *Var*, written $choice(a, b) \preceq e$, if, and only if, a logically implies e and b logically implies $\neg e$. The three-valued approximation relation can be straightforwardly extended to operations as described in [2]. An abstract system Sys' approximates a concrete system Sys, written $Sys' \preceq Sys$, if the systems have isomorphic CFGs and the operations in the abstract system approximate the corresponding ones in the concrete system. An example for an abstract system that approximates the concrete system in Figure 2 is depicted in Figure 4. For illustration: the abstract operation $(y > 0) := choice((y > 0), \mathit{false})$ 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 $S = Loc \times S_{Pred}$ where S_{Pred} is the set of all possible valuations of the three-valued predicates in *Pred*. The state space corresponding to the abstraction of our example is thus $S =$

$$\begin{array}{lll} \{\langle(0,0),(y>0)=\mathit{true}\rangle, & \langle(0,0),(y>0)=\perp\rangle, & \langle(0,0),(y>0)=\mathit{false}\rangle \\ \;\langle(1,0),(y>0)=\mathit{true}\rangle, & \langle(1,0),(y>0)=\perp\rangle, & \langle(1,0),(y>0)=\mathit{false}\rangle \\ \;\langle(0,1),(y>0)=\mathit{true}\rangle, & \langle(0,1),(y>0)=\perp\rangle, & \langle(0,1),(y>0)=\mathit{false}\rangle \\ \;\langle(1,1),(y>0)=\mathit{true}\rangle, & \langle(1,1),(y>0)=\perp\rangle, & \langle(1,1),(y>0)=\mathit{false}\rangle\}. \end{array}$$

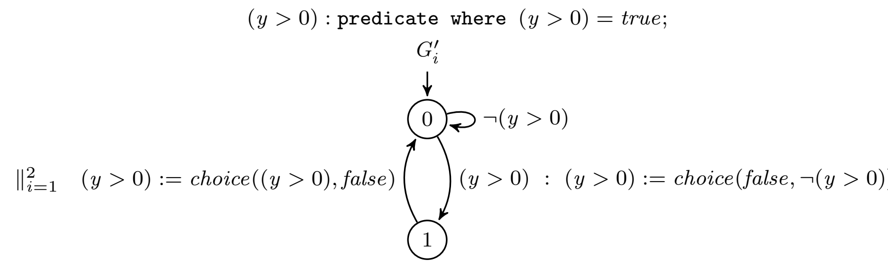

Figure 4: Abstract system represented by control flow graphs G'_1 and G'_2 corresponding to the concrete control flow graphs G_1 and G_1. Transitions are labelled with abstract operations over $Pred = \{(y > 0)\}$.

So far we have seen how concurrent systems can be formally represented and abstracted. Next we will take a look on how 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* 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 3 (Three-Valued Kripke Structure).
A three-valued Kripke structure *over a set of atomic predicates AP is a tuple $M = (S, \langle l^0, s^0\rangle, R, L)$ where*

- *S is a finite set of states,*
- *$\langle l^0, s^0\rangle \in S$ is the initial state,*
- *$R : S \times S \to \{true, \bot, false\}$ is a transition function with $\forall \langle l, s\rangle \in S : \exists \langle l', s'\rangle \in S : R(\langle l, s\rangle, \langle l', s'\rangle) \in \{true, \bot\}$,*
- *$L : S \times AP \to \{true, \bot, false\}$ is a labelling function that associates a truth value with each atomic predicate in each state.*

A simple example for a three-valued Kripke structure M over $AP = \{p\}$ is depicted in Figure 5.

A path π of a Kripke structure M is a sequence of states $\langle l^0, s^0\rangle\langle l^1, s^1\rangle \ldots$ with $R(\langle l^k, s^k\rangle, \langle l^{k+1}, s^{k+1}\rangle) \in \{true, \bot\}$. $\pi(k)$ denotes the k-th state of π, whereas π^k denotes the k-th suffix $\pi(k)\pi(k+1)\pi(k+2)\ldots$ of π. By Π_M we denote the set of all paths of M starting in the initial state. Paths are considered for the evaluation of temporal logic properties of Kripke structures.

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

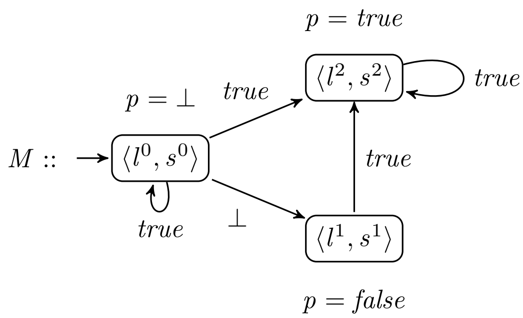

Figure 5: Three-valued Kripke structure.

Definition 4 (Concurrent System as Three-Valued Kripke Structure). Let $Sys = \|_{i=1}^{n} P_i$ over *Var* be a concurrent system given by a composite control flow graph $G = (Loc, \delta, \tau)$ and an initial state predicate ϕ. Moreover, let *Pred* be a set of predicates over *Var*. The corresponding three-valued Kripke structure is a tuple $M = (S, \langle l^0, s^0 \rangle, R, L)$ over a set of atomic predicates $AP = Pred \cup \{(loc_i = j) \mid i \in [1..n],\ j \in Loc_i\}$, where $(loc_i = j)$ denotes that the process P_i is currently at control location j, with

- $S := Loc \times S_{Pred}$,
- $\langle l^0, s^0 \rangle := \langle (0, \ldots, 0), s \rangle$ *where* $s \in S$ *with* $s(\phi) = true$,
- $R(\langle l, s \rangle, \langle l', s' \rangle) := \bigvee_{i=1}^{n} R_i(\langle l, s \rangle, \langle l', s' \rangle) :=$
  $\bigvee_{i=1}^{n} (\delta_i(l_i, l'_i) \wedge \bigwedge_{i' \neq i} (l_{i'} = l'_{i'}) \wedge s(choice(a, b)) \wedge \bigwedge_{j=1}^{m} s'(p_j) = s(choice(a_j, b_j)))$
  assuming that l_i *is the single location of* P_i *in the composite location* l *and*
  $\tau_i(l_i, l'_i) = assume(choice(a, b)) : p_1 := choice(a_1, b_1), \ldots, p_m := choice(a_m, b_m)$,
- $L(\langle l, s \rangle, p) := s(p)$ for each $p \in Pred$,
- $L(\langle l, s \rangle, (loc_i = j)) := \begin{cases} true & if \quad l_i = j \\ false & else \end{cases}$
  assuming that l_i *is the single location of* P_i *in the composite location* l.

In [2] it has been shown that there is a one-to-one correspondence between computations of a concurrent system *Sys* and paths of a three-valued Kripke structure M modelling the state space of *Sys*. Moreover, we get Lemma 1 from [2] that establishes a relation between different abstract models of a system.

Lemma 1
Let $Sys = \|_{i=1}^{n} P_i$ *over Var be a concurrent system. Let* AP_a *and* AP_r *be sets of atomic predicates over Var with* $AP_a \subset AP_r$. *Moreover, let* $M_a =$

$(S_a, \langle l_a^0, s_a^0 \rangle, R_a, L_a)$ be the three-valued Kripke structure modelling the state space of Sys abstracted over AP_a, and let $M_r = (S_r, \langle l_r^0, s_r^0 \rangle, R_r, L_r)$ be the three-valued Kripke structure modelling the state space of Sys abstracted over AP_r. Then the following holds:

1. *For every path $\pi_a \in \Pi_{M_a}$ there exists a path $\pi_r \in \Pi_{M_r}$ with $\forall\, k \in \mathbb{N} : R_a(\pi_a(k), \pi_a(k+1)) = \mathit{true} \;\Rightarrow\; R_r(\pi_r(k), \pi_r(k+1)) = \mathit{true}$ and $\forall\, p \in AP_a : L_a\big(\pi_a(k), p\big) \leq_{\mathcal{K}_3} L_r\big(\pi_r(k), p\big)$*

2. *For every path $\pi_r \in \Pi_{M_r}$ there exists a path $\pi_a \in \Pi_{M_a}$ with $\forall\, k \in \mathbb{N} : R_r(\pi_r(k), \pi_r(k+1)) \neq \mathit{false} \;\Rightarrow\; R_a(\pi_a(k), \pi_a(k+1)) \neq \mathit{false}$ and $\forall\, p \in AP_a : L_a\big(\pi_a(k), p\big) \leq_{\mathcal{K}_3} L_r\big(\pi_r(k), p\big)$*

Hence, for each path π_a in the more abstract model M_a there is a path π_r in the finer model M_r such that π_r is a refinement of π_a in terms of the logic $\mathcal{K}_3$. Moreover, for each path π_r in the finer model M_r there is a path π_a in the more abstract model M_a such that π_a is an abstraction of π_r. An example of two paths with such an abstraction-refinement relation is depicted in Figure 6. As we can see, every labelling with a definite value in π_a has the same definite value in π_r, whereas labellings with *unknown* values in π_a may have different values in π_r. Furthermore, every definite transition in π_a is also definite in π_r, whereas *unknown* transitions in π_a may be either still *unknown* or definite π_r.

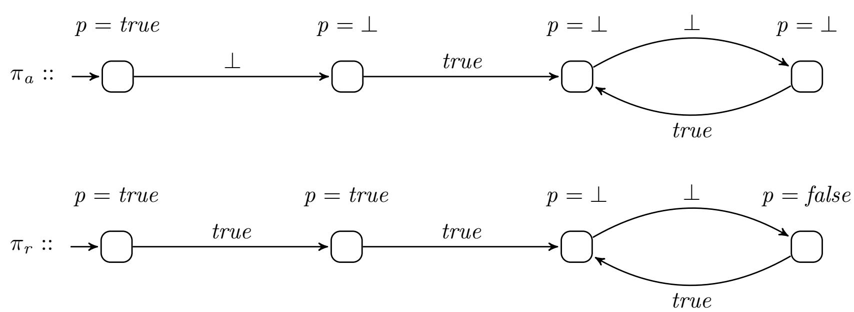

Figure 6: Paths with an abstraction-refinement relation.

According to Lemma 1, definite path information is preserved if we consider a refinement of an abstract model. Subsequently, we will see that this extends to temporal logic properties evaluated on abstract and refined models.

The number of states of a Kripke structure modelling a given system is exponential in the number of its locations and variables. State explosion is the major challenge in software model checking. One approach to cope with the state explosion problem is to use a symbolic and therefore more compact representation of the Kripke structure. In SAT-based bounded model checking [5] all possible path prefixes up to a bound $b \in \mathbb{N}$ are encoded in a propositional logic formula. The formula is then conjuncted with an encoding of the temporal logic property to be checked. In case the overall formula is satisfiable, the satisfying truth assignment characterises a witness path of length b for the property in the state space of the encoded system. Hence, bounded model checking can be performed via satisfiability solving. We now briefly recapitulate the syntax and bounded semantics of the linear temporal logic (LTL):

Definition 5 (Syntax of LTL).
Let AP be a set of atomic predicates and $p \in AP$. The syntax of LTL formulae *ψ is given by*

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

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). Based on $\mathcal{K}_3$, LTL formulae can be evaluated on b-bounded path prefixes of three-valued Kripke structures. Such finite prefixes $\pi(0)\ldots\pi(b)$ can still represent infinite paths if the prefix has a *loop*, i.e. the last state $\pi(b)$ has a successor state that is also part of the prefix.

Definition 6 (b-Loop).
Let π be a path of a three-valued Kripke structure M and let $r, b \in \mathbb{N}$ with $r \leq b$. Then π has a (b, r)-loop if $R(\pi(b), \pi(r)) \in \{true, \bot\}$ and π is of the form $v \cdot w^\omega$ where $v = \pi(0)\ldots\pi(r-1)$ and $w = \pi(r)\ldots\pi(b)$. π has a b-loop if there exists an $r \in \mathbb{N}$ with $r \leq b$ such that π has a (b, r)-loop.

An example for a path with a loop is depicted in Figure 7. Let $b = 3$ be then bound. As we can see, the path π has a $(3, 1)$-loop and therefore a 3-loop.

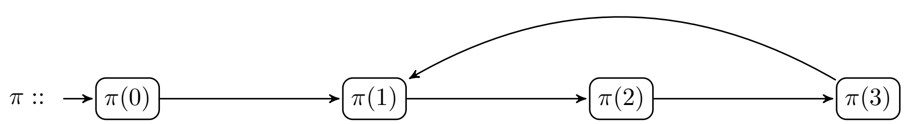

Figure 7: Path with a b loop.

For the bounded evaluation of LTL formulae on paths of Kripke structures we have to distinguish between paths *with* and *without* a b-loop.

Definition 7 (Three-Valued Bounded Evaluation of LTL).
Let $M = (S, \langle l^0, s^0\rangle, R, L)$ over AP be a three-valued Kripke structure. Moreover, let $b \in \mathbb{N}$ and let π be a path of M with *a b-loop. Then the b*-bounded evaluation *of an LTL formula ψ on π, written $\left[\pi \models_b^k \psi\right]$ where $k \leq b$ denotes the current position along the path, is inductively defined as follows:*

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

If π is a path without *a b-loop then the b*-bounded evaluation *of ψ is defined as:*

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

The other cases are identical to the case where π has a b-loop. The universal bounded evaluation of ψ on an entire Kripke structure M is $[M \models_{U,b} \psi] \equiv \bigwedge_{\pi \in \Pi_M} [\pi \models_b^0 \psi]$. The existential bounded evaluation of ψ on a Kripke structure is $[M \models_{E,b} \psi] \equiv \bigvee_{\pi \in \Pi_M} [\pi \models_b^0 \psi]$.

Checking temporal logic properties for three-valued Kripke structures is what is known as three-valued model checking [3]. Universal model checking can always be transformed into existential model checking based on the equation

$$[M \models_{U,b} \psi] \ = \ \neg\, [M \models_{E,b} \neg\psi]\,.$$

From now on we only consider the existential case, since it is the basis of satisfiability-based bounded model checking. Bounded model checking [5] is typically performed incrementally, i.e. b is iteratively increased until the property can be either proven or a completeness threshold [14] is reached. In the three-valued scenario there exist three possible outcomes: *true*, *false* and $\bot$. For our example Kripke structure M we have that $[M \models_{E,0} \mathbf{F}p]$ evaluates to $\bot$ and $[M \models_{E,1} \mathbf{F}p]$ evaluates to *true*, which is witnessed by the 1-bounded path prefix $\langle l^0, s^0 \rangle \langle l^2, s^2 \rangle$.

By combining Lemma 1 with the definitions of LTL we get Corollary 1:

Corollary 1
Let $Sys = \|_{i=1}^n P_i$ over Var be a concurrent system. Let AP_a and AP_r be sets of atomic predicates over Var with $AP_a \subset AP_r$. Let $M_a = (S_a, \langle l_a^0, s_a^0 \rangle, R_a, L_a)$ be the three-valued Kripke structure modelling the state space of Sys abstracted over AP_a, and let $M_r = (S_r, \langle l_r^0, s_r^0 \rangle, R_r, L_r)$ be the three-valued Kripke structure modelling the state space of Sys abstracted over AP_r. Moreover, let ψ be an LTL formula and $b \in \mathbb{N}$ be a bound. Then the following holds:

1. $[M_a \models_{E,b} \psi] = \mathit{true} \ \Rightarrow\ [M_r \models_{E,b} \psi] = \mathit{true}$
2. $[M_a \models_{E,b} \psi] = \mathit{false} \ \Rightarrow\ [M_r \models_{E,b} \psi] = \mathit{false}$

Hence, all definite model checking results obtained under three-valued abstraction can be immediately transferred to any refined model, and thus, also the concrete system *Sys* modelled by the three-valued Kripke structure, whereas an *unknown* result tells us that the current level of abstraction is too coarse. For the latter case we will present an automatic refinement procedure in Section 6 that refines the abstraction by adding predicates to AP.

In the next section we define a propositional logic encoding of three-valued bounded model checking tasks for abstracted concurrent systems. Our encoding allows to immediately transfer verification tasks into a propositional logic formulae that can be then processed via a SAT solver. Thus, the expensive construction of an explicit Kripke structure is not required in our approach. The state space of the system under consideration as well as the property to be checked will be implicitly contained in the propositional logic encoding, and the model checking result will be equivalent to the result of the corresponding satisfiability tests.

4. Propositional Logic Encoding

In our previous work [15] we showed that the three-valued bounded model checking problem $[M \models_{E,b} \psi]$, where M is given as an *explicit* Kripke structure, can be reduced to two classical SAT problems. Here we show that for a given system *Sys* abstracted over *Pred*, a temporal logic property ψ, and a bound $b \in \mathbb{N}$, it is not even necessary to consider the corresponding model checking problem. We can immediately construct a propositional logic encoding $[\![Sys, \psi]\!]_b$ and perform two SAT checks. One check considers an *over-approximating completion* of the encoding, marked with '+', where all $\perp$'s are assumed to be *true*:

$$[\![Sys, \psi]\!]_b^+ \;:=\; [\![Sys, \psi]\!]_b[\perp \mapsto \mathit{true}]$$

and the second check considers an *under-approximating completion*, marked with a '−', where all $\perp$'s are assumed to be *false*:

$$[\![Sys, \psi]\!]_b^- \;:=\; [\![Sys, \psi]\!]_b[\perp \mapsto \mathit{false}].$$

Here $[\perp \mapsto z]$ with $z \in \{\mathit{true}, \mathit{false}\}$ denotes the assumption that $\perp$ mapped z. We will show that the following holds:

$$[M \models_{E,b} \psi] \;=\; \begin{cases} \mathit{true} & \mathit{if} \quad \mathrm{SAT}([\![Sys, \psi]\!]_b^-) = \mathit{true} \\ \mathit{false} & \mathit{if} \quad \mathrm{SAT}([\![Sys, \psi]\!]_b^+) = \mathit{false} \\ \perp & \mathit{else} \end{cases}$$

Hence, it is not required to construct and explore an explicit Kripke structure M modelling the state space of *Sys*. All we need to do is to construct $[\![Sys, \psi]\!]_b$ and check the satisfiability of its under- and over-approximation in order to obtain the result of the corresponding three-valued model checking problem.

The formula $[\![Sys, \psi]\!]_b$ is defined over a set of Boolean atoms and over *true*, *false* and $\perp$. We now give a step-by-step description on how $[\![Sys, \psi]\!]_b$ can be constructed for a concurrent system $Sys = \|_{i=1}^{n} P_i$ abstracted over a set of predicates *Pred* and given by a number of control flow graphs $G_i = (Loc_i, \delta_i, \tau_i)$ with $1 \leq i \leq n$, a temporal logic property $\psi \in$ LTL, and a bound $b \in \mathbb{N}$. The construction of $[\![Sys, \psi]\!]_b$ is divided into the translation of the abstract system into a formula $[\![Sys]\!]_b$ and the translation of the property ψ into a formula $[\![\psi]\!]_b$.

We start with the encoding of the system, which first requires to encode its states as propositional logic formulae. Since a state of a concurrent system is a tuple $\langle l, s\rangle$ where l is a composite control flow location and s is a valuation of all predicates in *Pred*, we encode l and s separately. First, we introduce a set of Boolean atoms for the encoding of locations. A composite location $(l_1, \ldots, l_n) \in Loc$ is a list of single locations $l_i \in Loc_i$ where $Loc_i = \{0, \ldots, |Loc_i|\}$ and i is the identifier of the associated process P_i. Each l_i is a binary number from the domain $\{[0]_2, \ldots, [|Loc_i|]_2\}$. We assume that all these numbers have d_i digits where d_i is the number required to binary represent the maximum value $|Loc_i|$. We introduce the following set of Boolean atoms:

$$LocAtoms := \{l_i[j] \ \mid \ i \in [1..n],\ j \in [1..d_i]\}$$

Hence, for each process P_i of the system we introduce d_i Boolean atoms, each referring to a distinct digit along the binary representation of its locations. The atoms now allow us to define the following encoding of locations:

Definition 8 (Encoding of Locations).
Let the location $l_i \in \{0, \ldots, |Loc_i|\}$ be given as a binary number. Moreover, let $l_i(j)$ be a function evaluating to true *if the j-th digit of l_i is 1, and to* false *otherwise. Then l_i can be encoded in propositional logic as follows:*

$$enc(l_i) \ := \ \textstyle\bigwedge_{j=1}^{d_i}((l_i[j] \wedge l_i(j)) \vee (\neg l_i[j] \wedge \neg l_i(j)))$$

Let $l = (l_1, \ldots, l_n)$ be a composite location. Then $enc(l) \ := \ \bigwedge_{i=1}^{n} enc(l_i)$.

Note that since the function $l_i(j)$ evaluates to *true* or *false* an encoding $enc(l_i)$ can be always simplified to a conjunction of literals over *LocAtoms*. For instance, the initial location $(0, 0)$ of our example system from Section 2 will be encoded to $\neg l_1[1] \wedge \neg l_2[1]$ and the location $(0, 1)$ will be encoded to $\neg l_1[1] \wedge l_2[1]$.

Next, we encode the predicate part of states. Let $s \in S_{Pred}$ where $Pred = \{p_1, \ldots, p_m\}$. We introduce the following set of Boolean atoms:

$$PredAtoms := \{p[j] \ \mid \ p \in Pred,\ j \in \{u, t\}\}$$

Hence, for each three-valued predicate p we introduce two Boolean atoms. The atom $p[u]$ will let us indicate whether p evaluates to *unknown*, and $p[t]$ will let us indicate whether it evaluates to *true* or *false*:

Definition 9 (Encoding of States over Predicates).
Let $p \in Pred$ and let $val \in \{true, \perp, false\}$. Then $(p = val)$ can be logically

encoded follows:

$$enc(p = val) \; := \; \begin{cases} \neg p[u] \wedge p[t] & \text{if} \quad val = true \\ \neg p[u] \wedge \neg p[t] & \text{if} \quad val = false \\ \;\; p[u] & \text{if} \quad val = \bot \end{cases}$$

Let s be a state over Pred. Then $enc(s) := \bigwedge_{p \in Pred} enc(p = s(p))$.

For an overall state $\langle l, s\rangle \in S$ we consequently get

$$enc(\langle l, s\rangle) \; := \; enc(l) \wedge enc(s).$$

Since $enc(\langle l, s\rangle)$ yields a conjunction of literals, there exists exactly one satisfying truth assignment $\alpha : LocAtoms \cup PredAtoms \to \{true, false\}$ for a state encoding. We denote the assignment characterising an encoded state $\langle l, s\rangle$ by $\alpha_{\langle l,s\rangle}$. For instance, the initial state $\langle (0, 0), (y > 0) = true\rangle$ of our abstracted example system will be encoded to

$$Init = \neg l_1[1] \wedge \neg l_2[1] \wedge \neg p[u] \wedge p[t]$$

where $p = (y > 0)$, i.e. we abbreviate $(y > 0)$ by p. The assignment characterising $Init$ is

$$\alpha_{\langle (0,0),(y>0)=true\rangle} : l_1[1] \mapsto false, l_2[1] \mapsto false, p[u] \mapsto false, p[t] \mapsto true.$$

.

The encoding function enc can be extended to *logical expressions* in negation normal form (NNF), which we require for our later transition encoding:

Definition 10 (Encoding of Logical Expressions).
Let $p \in Pred$ and e, e' logical expressions in NNF over $Pred \cup \{true, \bot, false\}$. Let $val \in \{true, \bot, false\}$. Then the encoding of a logical expression is inductively defined as follows:

$$\begin{array}{lcl}
enc(val) & := & val \\
enc(\neg val) & := & \neg val \\
enc(p) & := & (p[u] \wedge \bot) \vee (\neg p[u] \wedge p[t]) \\
enc(\neg p) & := & (p[u] \wedge \bot) \vee (\neg p[u] \wedge \neg p[t]) \\
enc(e \wedge e') & := & enc(e) \wedge enc(e') \\
enc(e \vee e') & := & enc(e) \vee enc(e') \\
enc(choice(e, e')) & := & enc((e \vee NNF(\neg e')) \wedge (e \vee e' \vee \bot))
\end{array}$$

Next, we take a look at how the transition relation of an abstracted system can be encoded. We will construct a propositional logic formula

$$[\![Sys]\!]_b = Init_0 \wedge Trans_{0,1} \wedge \ldots \wedge Trans_{b-1,b}$$

that exactly characterises path prefixes of length $b \in \mathbb{N}$ in the state space of the system *Sys* abstracted over *Pred*. Since we consider states as parts of such prefixes, we have to extend the encoding of states by index values $k \in \{0, \dots, b\}$ where k denotes the position along a path prefix. For this we introduce the notion of indexed encodings. Let F be a propositional logic formula over $Atoms = LocAtoms \cup PredAtoms$ and *true*, *false* and $\perp$. Then F_k stands for $F[a/a_k \mid a \in Atoms]$. Our overall encoding will be thus defined over the set $Atoms_{[0,b]} = \{a_k \,|\, a \in Atoms, 0 \leq k \leq b\}$. An assignment $\alpha_{\langle l,s \rangle}$ to the atoms in a subset $Atoms_{[k,k]} \subseteq Atoms_{[0,b]}$ thus characterises a state $\langle l, s \rangle$ at position k of a path prefix, whereas an assignment $\alpha_{\langle l^0,s^0 \rangle \dots \langle l^b,s^b \rangle}$ to the atoms in $Atoms_{[0,b]}$ characterises an entire path prefix $\langle l^0, s^0 \rangle \dots \langle l^b, s^b \rangle$. Since all execution paths start in the initial state of the system, we extend its encoding by the index 0, i.e. we get

$$Init_0 = \neg l_1[1]_0 \wedge \neg l_2[1]_0 \wedge \neg p[u]_0 \wedge p[t]_0.$$

The encoding of all possible state space transitions from position k to $k+1$ is defined as follows:

Definition 11 (Encoding of Transitions).
Let $Sys = \|_{i=1}^{n} P_i$ *over* $Pred$ *be an abstracted concurrent system given by the single control flow graphs* $G_i = (Loc_i, \delta_i, \tau_i)$ *with* $1 \leq i \leq n$*. Then all possible transitions for position* k *to* $k+1$ *can be encoded in propositional logic as follows:*

$$Trans_{k,k+1} :=$$

$$\bigvee_{i=1}^{n} \bigvee_{(l_i,l'_i) \in \delta_i} \left(enc(l_i)_k \wedge enc(l'_i)_{k+1} \wedge \bigwedge_{i' \neq i} (idle(i')_{k,k+1}) \wedge enc(\tau_i(l_i, l'_i))_{k,k+1} \right)$$

where

$$idle(i')_{k,k+1} := \bigwedge_{j=1}^{d_{i'}} \left(l_{i'}[j]_k \leftrightarrow l_{i'}[j]_{k+1} \right)$$

and

$$\begin{aligned} enc(\tau_i(l_i, l'_i))_{k,k+1} := \quad & \quad enc(choice(a, b))_k \\ & \wedge \quad \bigwedge_{j=1}^{m} \big(\; (enc(a_j)_k \wedge enc(p_j = true)_{k+1}) \\ & \qquad\quad \vee (enc(b_j)_k \wedge enc(p_j = false)_{k+1}) \\ & \qquad\quad \vee (enc(\neg a_j \wedge \neg b_j)_k[\perp \mapsto true] \wedge enc(p_j = \perp)_{k+1}) \big) \end{aligned}$$

assuming that $\tau_i(l_i, l'_i) = assume(choice(a, b)) : p_1 := choice(a_1, b_1), \dots, p_m := choice(a_m, b_m)$.

Thus, we iterate over the system's processes P_i and over the processes' control flow transitions $\delta_i(l_i, l'_i)$. Now we construct the k-indexed encoding of a source location l_i and conjunct it with the $(k+1)$-indexed encoding of a destination location l'_i. This gets conjuncted with the sub formula $\bigwedge_{i' \neq i} idle(i')_{k,k+1}$ which encodes that all processes different to the currently considered process P_i are idle, i.e. do not change their control flow location, while P_i proceeds. The last part of the transition encoding concerns the operation associated with the control flow transition $\delta_i(l_i, l'_i)$: The sub formula $enc(\tau_i(l_i, l'_i))_{k,k+1}$ evaluates to *true* for assignments $\alpha_{\langle l,s\rangle\langle l',s'\rangle}$ to the atoms in $Atoms_{[k,k+1]}$ that characterise pairs of states s and s' over *Pred* where the guard of the operation $\tau_i(l_i, l'_i)$ is *true* in s and the execution of the operation in s definitely results in the state s'. The operation encoding evaluates to $\perp$ for states s and s' where the guard of the operation is $\perp$ in s or where it is *unknown* whether the execution of the operation in s results in the state s'. In all other cases $enc(\tau_i(l_i, l'_i))_{k,k+1}$ evaluates to *false*. Our transition encoding requires that an operation $\tau_i(l_i, l'_i)$ assigns to all predicates in *Pred*: Thus, if a predicate p is not modified by the operation we assume that $p := p$ is part of the assignment list.

The encoding of the control flow transition $\delta_1(0,1)$ of our abstract example system with

$$\tau_1(0,1) \;=\; (assume(p) \;:\; p := choice(false, \neg p)),$$

where p abbreviates $(y > 0)$, yields the following:

$$\begin{array}{lcl}
enc(0)_k & = & \neg l_1[1]_k \\
\wedge & & \wedge \\
enc(1)_{k+1} & = & l_1[1]_{k+1} \\
\wedge & & \wedge \\
idle(2)_{k,k+1} & = & (l_2[1]_k \leftrightarrow l_2[1]_{k+1}) \\
\wedge & & \wedge \\
enc(\tau_1(0,1))_{k,k+1} & = & ((p[u]_k \wedge \perp) \vee (\neg p[u]_k \wedge p[t]_k))\wedge \\
 & & ((false \wedge (\neg p[u]_{k+1} \wedge p[t]_{k+1})) \\
 & & \vee(((p[u]_k \wedge \perp) \vee (\neg p[u]_k \wedge \neg p[t]_k)) \wedge (\neg p[u]_{k+1} \wedge \neg p[t]_{k+1})) \\
 & & \vee(((p[u]_k \wedge true) \vee (\neg p[u]_k \wedge p[t]_k)) \wedge (p[u]_{k+1})))
\end{array}$$

The encoding of the operation only evaluates to *true* for assignments to the atoms in $Atoms_{[k,k+1]}$ that characterise a predicate state s at position k with $s(p) = true$ and a state s' at position $k+1$ with $s'(p) = \perp$. An overall satisfying assignment for this encoding is

$$\begin{array}{lcl}
\alpha_{\langle(0,0),(y>0)=true\rangle\langle(1,0),(y>0)=\perp\rangle} \;:\; & l_1[1]_k & \mapsto & false, \\
 & l_2[1]_k & \mapsto & false, \\
 & l_1[1]_{k+1} & \mapsto & true, \\
 & l_2[1]_{k+1} & \mapsto & false, \\
 & p[u]_k & \mapsto & false, \\
 & p[t]_k & \mapsto & true, \\
 & p[u]_{k+1} & \mapsto & true
\end{array}$$

characterising the definite transition between the states $\langle(0,0),(y > 0) = true\rangle$

and $\langle(1,0),(y>0)=\perp\rangle$. The assignments

$$\alpha_{\langle(0,l_2),(y>0)=true\rangle\langle(1,l_2),(y>0)=false\rangle},$$
$$\alpha_{\langle(0,l_2),(y>0)=\perp\rangle\langle(1,l_2),(y>0)=false\rangle},$$
$$\alpha_{\langle(0,l_2),(y>0)=\perp\rangle\langle(1,l_2),(y>0)=\perp\rangle}$$

with $l_2 \in \{0,1\}$ yield *unknown* for the encoding and hereby correctly characterise $\perp$-transitions in the abstract state space. All other assignments yield *false* indicating that corresponding pairs of states do not characterise valid transitions.

The encoding definitions now allow us to construct the propositional logic formula

$$[\![Sys]\!]_b = Init_0 \wedge Trans_{0,1} \wedge \ldots \wedge Trans_{b-1,b}$$

that characterises all possible path prefixes of length $b \in \mathbb{N}$ in the state space of the encoded system. Each assignment $\alpha : Atoms_{[0,b]} \rightarrow \{true, false\}$ that satisfies the formula characterises a definite path prefix, whereas an assignment that makes the formula evaluate to *unknown* characterises a prefix with some $\perp$-transitions.

The second part of the encoding concerns the LTL property to be checked. The three-valued bounded LTL encoding has been defined in [15] before. Here we adjust it to our encodings of predicates and locations. Again, we distinguish the cases where the property is evaluated on a path prefix with and without a loop. The LTL encoding for the evaluation on prefixes with a loop is defined as:

Definition 12 (LTL Encoding with Loop).
Let p and $(loc_i = l_i) \in AP$, ψ and ψ' LTL formulae, and $b, k, r \in \mathbb{N}$ with $k, r \leq b$ where k is the current position, b the bound and r the destination position of the b-loop. Then the LTL encoding with a loop, ${}_r[\![\psi]\!]_b^k$, is defined as follows:

$$\begin{array}{lcl}
{}_r[\![(loc_i = l_i)]\!]_b^k & \equiv & enc(l_i)_k \\
{}_r[\![\neg(loc_i = l_i)]\!]_b^k & \equiv & \neg enc(l_i)_k \\
{}_r[\![p]\!]_b^k & \equiv & enc(p)_k \\
{}_r[\![\neg p]\!]_b^k & \equiv & enc(\neg p)_k \\
{}_r[\![\psi \vee \psi']\!]_b^k & \equiv & {}_r[\![\psi]\!]_b^k \vee {}_r[\![\psi']\!]_b^k \\
{}_r[\![\psi \wedge \psi']\!]_b^k & \equiv & {}_r[\![\psi]\!]_b^k \wedge {}_r[\![\psi']\!]_b^k \\
{}_r[\![\mathbf{G}\psi]\!]_b^k & \equiv & \bigwedge_{k'=min(k,r)}^{b} {}_r[\![\psi]\!]_b^{k'} \\
{}_r[\![\mathbf{F}\psi]\!]_b^k & \equiv & \bigvee_{k'=min(k,r)}^{b} {}_r[\![\psi]\!]_b^{k'} \\
{}_r[\![\mathbf{X}\psi]\!]_b^k & \equiv & {}_r[\![\psi]\!]_b^{succ(k)}
\end{array}$$

where $succ(k) = k+1$ if $k < b$ and $succ(k) = r$ else.

For a path prefix without a loop the LTL encoding is defined as:

Definition 13 (LTL Encoding without Loop).
Let ψ be an LTL formula and $b, k \in \mathbb{N}$ with $k \leq b$ where k is the current position and b the bound. Then the LTL encoding without a loop, $[\![\psi]\!]_b^k$, is defined as follows:

$$\begin{array}{lcl} [\![\mathbf{G}\psi]\!]_b^k & \equiv & \mathit{false} \\ [\![F\psi]\!]_b^k & \equiv & \bigvee_{k'=k}^{b} [\![\psi]\!]_b^{k'} \\ [\![\mathbf{X}\psi]\!]_b^k & \equiv & \mathit{if}\ k < b\ \mathit{then}\ {}_r[\![\psi]\!]_b^{k+1}\ \mathit{else\ false} \end{array}$$

The LTL encoding without a loop of the other cases is identical to the LTL encoding with a loop.

An example encoding is $[\![\mathbf{F}p]\!]_2^0 = enc(p)_0 \vee enc(p)_1 \vee enc(p)_2$ which expresses that a predicate p holds *eventually*, i.e. at some position 0, 1 or 2 along a 2-prefix. Remember that a prefix $\langle l^0, s^0\rangle \ldots \langle l^b, s^b\rangle$ has a b-loop if there exists a transition from $\langle l^b, s^b\rangle$ to a previous state $\langle l^r, s^r\rangle$ along the prefix with $0 \leq r \leq b$. Hence, we can define a loop constraint based on our transition encoding: A prefix characterised by an assignment $\alpha_{\langle l^0,s^0\rangle\ldots\langle l^b,s^b\rangle}$ has definitely resp. maybe a b-loop if the loop constraint

$$\bigvee_{r=0}^{b} \mathit{Trans}_{b,r}$$

evaluates to *true* resp. *unknown* under $\alpha_{\langle l^0,s^0\rangle\ldots\langle l^b,s^b\rangle}$ where $\mathit{Trans}_{b,r}$ is defined according to Definition 11 but with k substituted by b and $k+1$ by r. This now allows us to define the overall encoding of whether a concurrent system Sys satisfies an LTL formula ψ:

$$[\![\mathit{Sys}, \psi]\!]_b \ := \ [\![\mathit{Sys}]\!]_b \wedge [\![\psi]\!]_b$$

with

$$[\![\psi]\!]_b \ := \ [\![\psi]\!]_b^0 \vee \bigvee_{r=0}^{b} (\mathit{Trans}_{b,r} \wedge {}_r[\![\psi]\!]_b^0).$$

We have proven the following theorem that establishes the relation between the satisfiability result for $[\![\mathit{Sys}, \psi]\!]_b$ and the result of the corresponding model checking problem:

Theorem 1
Let M be a three-valued Kripke structure representing the state space of an abstracted concurrent system Sys, let ψ be an LTL formula and $b \in \mathbb{N}$ Then:

$$[M \models_{E,b} \psi] \ \equiv \ \begin{cases} \mathit{true} & \mathit{if} \quad \mathrm{SAT}([\![\mathit{Sys}, \psi]\!]_b^-) = \mathit{true} \\ \mathit{false} & \mathit{if} \quad \mathrm{SAT}([\![\mathit{Sys}, \psi]\!]_b^+) = \mathit{false} \\ \perp & \mathit{else} \end{cases}$$

Proof sketch.
We have proven Theorem 1 by showing the following: (I) For each b-bounded

path π in M there exits an assignment $\alpha_\pi : Atoms_{[0,b]} \rightarrow \{true, false\}$ that exactly characterises π in $[\![Sys]\!]_b$, i.e. the transition values along π and $\alpha_\pi([\![Sys]\!]_b)$ are identical and the labellings along π and $\alpha_\pi([\![Sys]\!]_b)$ are identical as well. (II) The evaluation of an LTL property ψ on π yields the same result as $\alpha_\pi([\![\psi]\!]_b)$. The full proof can be found in [16]. □

Hence, via two satisfiability tests, one where $\perp$ is substituted by *true* and one where it is substituted by *false*, we can determine the result of the corresponding model checking problem. Our encoding can be straightforwardly built based on the concurrent system, which saves us the expensive construction of an explicit state space model. In the next section we show that our encoding can be also easily augmented by fairness constraints, which allows us to check liveness properties of concurrent systems under realistic conditions.

5. Extension to Fairness

Our approach allows to check LTL properties of concurrent software systems via SAT solving. While the verification of safety properties like mutual exclusion does not require any fairness assumptions about the behaviour of the processes of the system, fairness is essential for verifying *liveness* properties under realistic conditions. The most common notions of fairness in verification are *unconditional*, *weak* and *strong* fairness: An unconditional fairness constraint claims that in an infinite computation, certain operations have to be infinitely often executed. A weak fairness constraint claims that in an infinite computation, each operation that is *continuously* enabled has to be infinitely often executed. A strong fairness constraint claims that in an infinite computation, each operation that is *infinitely often* enabled has to be infinitely often executed. All these types of constraints can be straightforwardly expressed in LTL. We now define these constraints for characterising fair, i.e. realistic, behaviour of our concurrent systems $Sys = \|_{i=1}^{n} P_i$ over *Pred*. Our unconditional fairness constraint is defined as:

$$ufair \;\equiv\; \bigwedge_{i=1}^{n} \bigvee_{(l_i, l_i') \in \delta_i} \mathbf{GF}(executed(l_i, l_i'))$$

Hence, for each process some operation has to be executed infinitely often, i.e. each process proceeds infinitely often. Note that we model termination via a location with a self-loop. Thus, terminated processes can still proceed. The expression $executed(l_i, l_i')$ can be easily defined in LTL. For this we extend the set *Pred* by a progress predicate for each process: $Pred := Pred \cup \{progress_i \,|\, i \in [1..n]\}$. Moreover, we extend each operation as follows: $\tau_i(l_i, l_i')$ sets $progress_i$ to *true* and all $progress_{i'}$ with $i' \neq i$ to *false*. Now $executed(l_i, l_i')$ is defined as

$$executed(l_i, l_i') \;\equiv\; (loc_i = l_i) \wedge \mathbf{X}((loc_i = l_i') \wedge progress_i).$$

where $\mathbf{X}$ is the LTL operator 'next'. An example for a system where operations are extended with *progress* statements is given by the parallel composition of control flow graphs depicted in Figure 8. As we can see, each time when a process executes an operation, it sets its own progress predicate to *true* and the progress predicate of the other process to *false*.

$(y > 0)$: `predicate where` $(y > 0) = \mathit{true}$;
$\mathit{progress}_1$: `predicate where` $\mathit{progress}_1 = \mathit{false}$;
$\mathit{progress}_2$: `predicate where` $\mathit{progress}_2 = \mathit{false}$;

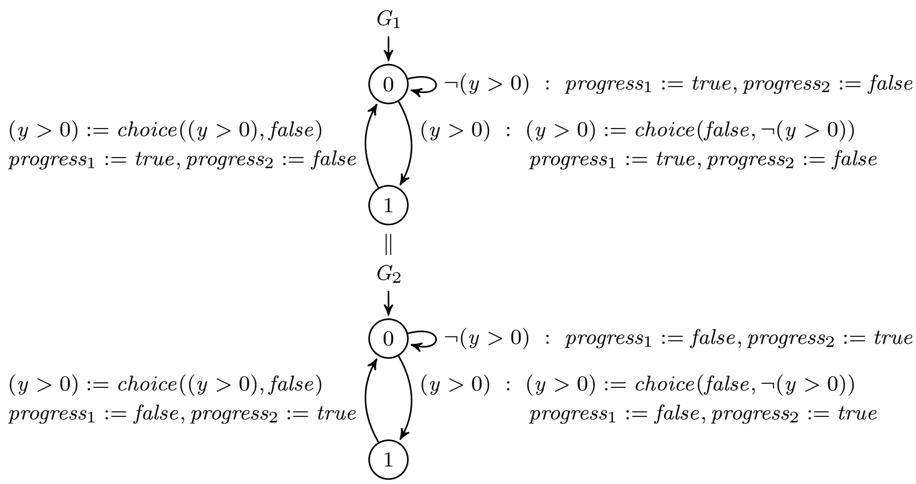

Figure 8: Parallel composition of abstract control flow graphs where operations are extended with progress statements.

An operation associated with a control flow transition (l_i, l_i') is executed if $(\mathit{loc}_i = l_i)$ holds in the current state and $(\mathit{loc}_i = l_i') \wedge \mathit{progress}_i$ holds in the next state. For the control flow graph G_1 in Figure 8 we have for instance $\mathit{executed}(0,1) \equiv (\mathit{loc}_1 = 0) \wedge \mathbf{X}((\mathit{loc}_1 = 1) \wedge \mathit{progress}_1)$. Next, we define our weak fairness constraint:

$$\mathit{wfair} \equiv \bigwedge_{i=1}^{n} \bigwedge_{(l_i,l_i') \in \delta_i} (\mathbf{FG}(\mathit{enabled}(l_i, l_i')) \rightarrow \mathbf{GF}(\mathit{executed}(l_i, l_i')))$$

Hence, for each process, each continuously enabled operation has to be infinitely often executed. Instead of incorporating *each* operation in this type of constraint it is also possible to restrict the operations to crucial ones, which results in a shorter constraint and thus also restrains the complexity of model checking under fairness. For our running example it is for instance appropriate to just incorporate operations in *wfair* that correspond to the successful acquisition of the semaphore. Note that *wfair* can be easily transferred into negation normal form via the common propositional logic transformation rules such that it is conform with the definition of LTL. The expression $\mathit{enabled}(l_i, l_i')$ can be defined as an LTL formula over locations and *Pred* as follows:

$$\mathit{enabled}(l_i, l_i') \equiv (\mathit{loc}_i = l_i) \wedge \mathit{choice}(a, b)$$

assuming that $\tau_i(l_i, l_i') = assume(choice(a,b)) : p_1 := choice(a_1, b_1), \ldots, p_m := choice(a_m, b_m)$. Thus, an operation associated with a control flow transition (l_i, l_i') is enabled if $(loc_i = l_i)$ holds and the guard of the operation holds as well. An example of an *enabled* expression in terms of the control flow graph G_1 in Figure 8 is $enabled(0,1) \equiv (loc_1 = 0) \wedge (y > 0)$. Finally, we define our strong fairness constraint:

$$sfair \equiv \bigwedge_{i=1}^{n} \bigwedge_{(l_i,l_i') \in \delta_i} (\mathbf{GF}(enabled(l_i, l_i')) \rightarrow \mathbf{GF}(executed(l_i, l_i')))$$

Hence, for each process, each operation that is enabled infinitely often has to be executed infinitely often. In model checking under fairness we can either check properties under specific constraints or we can combine all to a general one

$$fair \equiv ufair \wedge wfair \wedge sfair.$$

Existential bounded model checking under fairness is now defined as:

$$[M \models_{E,b}^{fair} \psi] \equiv [M \models_{E,b} (fair \wedge \psi)]$$

Thus, we check whether there exists a b-bounded path that is fair and satisfies the property ψ. Such a model checking problem can be straightforwardly encoded in propositional logic based on our definitions in the previous section. We get

$$[\![Sys, fair \wedge \psi]\!]_b := [\![Sys]\!]_b \wedge [\![fair \wedge \psi]\!]_b,$$

which can be fed into a SAT solver in order to obtain the result of model checking ψ under fairness. Next, we introduce a systematic and fully-automatic approach to the refinement of three-valued abstractions in case the corresponding three-valued model checking problem yields *unknown*.

6. Cause-Guided Abstraction Refinement

In this section we present our approach to the refinement of three-valued abstractions in case the corresponding model checking problem yields an *unknown* result. Our abstractions represent uncertainty in the form of the constant $\perp$. SAT-based three-valued model checking is performed via two satisfiability tests, one where all occurrences of $\perp$ are mapped to *true* in the propositional logic encoding $[\![Sys, \psi]\!]_b$ and one where its occurrences are mapped to *false*. Here we introduce an enhanced encoding that comprises the *causes of uncertainty*: Each $\perp$ in the encoding gets superscripted with a *cause*, which can be missing information about a transition or a predicate. During the satisfiability tests all $\perp$'s are still treated the same, meaning that either all of them are mapped to *true* or all to *false* (compare Theorem 1). Once we have obtained an overall *unknown* model checking result, i.e. $\text{SAT}([\![Sys, \psi]\!]_b^{+}) = true$ for an assignment α and

SAT($[\![Sys, \psi]\!]_b^-$) = *false*, we proceed as follows: We now have that the assignment α characterises an unconfirmed witness path for ψ containing *unknowns*. Thus, this path is not present if all $\perp$'s get mapped to *false*. We determine the unsatisfied clauses of $\alpha([\![Sys, \psi]\!]_b^-)$. All these clauses contain uncertainty in the sense of $\perp$'s and we will see that we can straightforwardly derive the corresponding causes. We then apply our novel *cause-guided abstraction refinement* which rules out the causes of uncertainty by adding new predicates to the abstraction. We will show that our fully-automatic iterative refinement approach enables us to quickly reach the right level of abstraction in order to obtain a definite model checking result.

The basis of our refinement technique is an enhanced encoding comprising causes of uncertainty:

Definition 14 (Causes of Uncertainty).
Let $[\![Sys, \psi]\!]_b$ be the propositional logic encoding of a three-valued bounded model checking problem corresponding to a concurrent system $Sys = \|_{i=1}^n P_i$ abstracted over Pred where each process is given by a single control flow graph $G_i = (Loc_i, \delta_i, \tau_i)$ with $1 \leq i \leq n$. Uncertainty is represented in the encoding by the constant $\perp$. Each $\perp$ in the encoding can be associated with a cause *which we define as follows:*

$$cause \in \{p_k, (l_i, l_i')_k\}$$

with $p \in Pred$, $0 \leq k \leq b$ and $l_i, l_i' \in Loc_i$.

We will use p_k in order to denote that the predicate p potentially evaluates to *unknown* at position k of the encoding, and with $(l_i, l_i')_k$ we will denote that missing predicates over the guard of the operation $\tau_i(l_i, l_i')$ potentially cause an *unknown* transition from position k to $k+1$ in the encoding. Note that we refer to *potential* uncertainty in an encoding $[\![Sys, \psi]\!]_b$, since $[\![Sys, \psi]\!]_b$ always characterises *many* possible execution paths. For a *specific* path characterised by an assignment α to the atoms of $[\![Sys, \psi]\!]_b$ will see that we can refer to *actual* uncertainty. Next we show how causes of uncertainty can be integrated into the encoding in the sense of adding them as superscripts to the $\perp$'s. For this we introduce an enhanced encoding of abstract operations:

Definition 15 (Enhanced Encoding of Operations).
Let $Sys = \|_{i=1}^n P_i$ over Pred be an abstracted concurrent system given by the single control flow graphs $G_i = (Loc_i, \delta_i, \tau_i)$ with $1 \leq i \leq n$. Then the encoding of abstract operations $\tau_i(l_i, l_i') = assume(choice(a, b)) : p_1 := choice(a_1, b_1), \ldots, p_m := choice(a_m, b_m)$ comprising the causes of uncertainty *is defined as follows:*

$$\begin{aligned} enc(\tau_i(l_i, l_i'))_{k,k+1} := \quad & enc(choice(a, b))_k \\ & \wedge \textstyle\bigwedge_{j=1}^m (\quad (enc(a_j)_k \wedge enc(p_j = true)_{k+1}) \\ & \qquad \vee (enc(b_j)_k \wedge enc(p_j = false)_{k+1}) \\ & \qquad \vee (enc(\neg a_j \wedge \neg b_j)_k[\perp \mapsto true] \wedge enc(p_j = \perp)_{k+1})) \end{aligned}$$

with

$$enc(choice(a,b))_k \quad := \quad enc((a \vee NNF(\neg b)) \wedge (a \vee b \vee \bot^{\boldsymbol{(l_i,l'_i)_k}}))_k$$

and the following inductive definition of the encoding of logical expressions e, e' *over predicates* $p \in Pred$*.*

$$\begin{array}{lcl} enc(p)_k & := & (p[u]_k \wedge \bot^{\boldsymbol{p_k}}) \vee (\neg p[u]_k \wedge p[t]_k) \\ enc(\neg p)_k & := & (p[u]_k \wedge \bot^{\boldsymbol{p_k}}) \vee (\neg p[u]_k \wedge \neg p[t]_k) \\ enc(e \wedge e')_k & := & enc(e)_k \wedge enc(e')_k \\ enc(e \vee e')_k & := & enc(e)_k \vee enc(e')_k \end{array}$$

This definition enhances our previous encoding Definitions 9 and 10 in terms of superscripting each $\bot$ with a corresponding cause. We will ignore the causes and treat all $\bot$'s the same during the satisfiability checks. Hence, the enhanced encoding is equivalent to the standard encoding. However, in case of an *unknown* model checking result, the causes will become crucial and will allow us to immediately derive expedient refinement steps. For illustration, we encode the following abstract operation:

$$\tau(l_i, l'_i) \;=\; assume(choice(false, false)) \;:\; p := choice(p, \neg p)$$

Remember that $choice(false, false) \equiv \bot$. For the enhanced encoding we get:

$$\begin{gathered} enc(\tau(l_i, l'_i))_{k,k+1} \;= \\ \bot^{\boldsymbol{(l_i,l'_i)_k}} \wedge ((((p[u]_k \wedge \bot^{\boldsymbol{p_k}}) \vee (\neg p[u]_k \wedge p[t]_k)) \wedge \neg p[u]_{k+1} \wedge p[t]_{k+1}) \\ \vee (((p[u]_k \wedge \bot^{\boldsymbol{p_k}}) \vee (\neg p[u]_k \wedge \neg p[t]_k)) \wedge \neg p[u]_{k+1} \wedge \neg p[t]_{k+1})) \end{gathered}$$

Thus, uncertainty may be caused by the unknown guard of the abstract operation $\tau(l_i, l'_i)$ or by the predicate p evaluating to $\bot$ at position k. Actual uncertainty along a path characterised by an assignment α is only present if the $\bot$'s occur in clauses unsatisfied under $\alpha[\bot \mapsto \mathit{false}]$. Then we can utilise the causes attached to the $\bot$'s in order to rule out the uncertainty. We will now introduce our iterative abstraction-based model checking procedure with cause-guided refinement:

Procedure 1 (Iterative Abstraction-Based Model Checking). Let $G = (Loc, \delta, \tau)$ be the concrete control flow graph representing a concurrent system *Sys* defined over a set of variables *Var*. Moreover, let ψ be an LTL formula to be checked for *Sys*. The corresponding bounded model checking problem can be solved via three-valued abstraction refinement and satisfiability solving as follows:

1. Initialise the set of predicates *Pred* with the atomic propositions over *Var* referenced in ψ. Initialise the bound b with 1.
2. Construct the abstract control flow graph $G_a = (Loc_a, \delta_a, \tau_a)$ representing *Sys* abstracted over the current set *Pred* via a three-valued abstractor [2].

3. Encode the three-valued bounded model checking problem $[M(G_a) \models_{E,b} \psi]$ for the current bound b in propositional logic, which yields the formula $[\![Sys(G_a), \psi]\!]_b$.
4. Apply SAT-based three-valued bounded model checking (according to Theorem 1):
   (a) If the result is $[M(G_a) \models_{E,b} \psi] = \mathit{true}$, then there exists a b-bounded witness path for ψ in the state space of Sys. The path is characterised by an assignment α satisfying $[\![Sys(G_a), \psi]\!]_b^-$. Return α.
   (b) If the result is $[M(G_a) \models_{E,b} \psi] = \mathit{false}$, then there does not exist a b-bounded witness path for ψ in the state space of Sys. Terminate if b has reached the completeness threshold [14] of the verification task. Otherwise increment b and go to 3.
   (c) If the result is $[M(G_a) \models_{E,b} \psi] = \perp$, then it is *unknown* whether there exists a b-bounded witness path for ψ in the state space of Sys. An unconfirmed witness path for ψ with *unknowns* is characterised by an assignment α satisfying $[\![Sys(G_a), \psi]\!]_b^+$ but not satisfying $[\![Sys(G_a), \psi]\!]_b^-$. Apply the procedure *Cause-Guided Abstraction Refinement*, which updates *Pred*, and go to 2.

Procedure 2 (Cause-Guided Abstraction Refinement). Let $G = (Loc, \delta, \tau)$ be the concrete composite control flow graph representing a concurrent system $Sys = \|_{i=1}^{n}\ P_i$ defined over a set of variables *Var* and let $G_i = (Loc_i, \delta_i, \tau_i)$ with $1 \leq i \leq n$ the corresponding single CFGs. Let ψ be an LTL formula to be checked for Sys. Moreover, let $G_a = (Loc_a, \delta_a, \tau_a)$ be the abstract composite control flow graph representing Sys abstracted over a set of predicates *Pred* and let $[M(G_a) \models_{E,b} \psi]$ be the corresponding three-valued bounded model checking problem with $[M(G_a) \models_{E,b} \psi] = \perp$. Then for the propositional logic encoding $[\![Sys(G_a), \psi]\!]_b$ the following holds: $\mathrm{SAT}([\![Sys(G_a), \psi]\!]_b^+) = \mathit{true}$ and the solver additionally returns a corresponding satisfying assignment α characterising an unconfirmed witness path. $\mathrm{SAT}([\![Sys(G_a), \psi]\!]_b^-) = \mathit{false}$. Now the abstraction can be automatically refined by updating *Pred* as follows:

1. Determine the set U of clauses of $[\![Sys, \psi]\!]_b$ that are unsatisfied under the assignment $\alpha[\perp \mapsto \mathit{false}]$. (Each $u \in U$ must contain at least one $\perp$ since we have that $\mathrm{SAT}([\![Sys(G_a), \psi]\!]_b^+) = \mathit{true}$.)
2. Determine a set *Causes* such that for each $u \in U$ there exists a $\mathit{cause} \in \mathit{Cause}$ with $\perp^{\mathit{cause}}$ is contained in u.
3. For each $\mathit{cause} \in \mathit{Causes}$:
   (a) If $\mathit{cause} = (l_i, l'_i)_k$ with $l_i, l'_i \in Loc_i$, $1 \leq i \leq n$ and $0 \leq k \leq b$, then the value of the k-th transition along the unconfirmed witness path characterised by α is *unknown*. The transition from k to $k+1$ is associated with an operation $\tau_i(l_i, l'_i) = \mathit{assume}(e) : v_1 := e_1, \ldots, v_m := e_m$ of the concrete control flow graph G_i. $\tau_i(l_i, l'_i)$ can be straightforwardly derived from G_i. Add the atomic propositions occurring in the assume condition e as new predicates to *Pred*.

(b) If $cause = p_k$ with $p \in Pred$ and $0 \leq k \leq b$, then the value of p is *unknown* at position k of the unconfirmed witness path characterised by α, i.e. $\alpha(p[u]_k) = true$. Let $k' < k$ be the last predecessor of k with $\alpha(p[u]_{k'}) = false$, i.e. the last position where the value of p is known. The transition from position k' to $k' + 1$ is associated with an operation $\tau_i(l_i, l'_i) = assume(e) : v_1 := e_1, ..., v_m := e_m$ of a concrete control flow graph G_i. Missing information about this concrete operation in terms of predicates is the cause of uncertainty in the current abstraction. $\tau_i(l_i, l'_i)$ can be straightforwardly derived based on G_i and $\alpha(LocAtoms_{[k',k'+1]})$, which indicates the corresponding control flow locations. Let $wp_{\tau_i(l_i,l'_i)}(p) = p[v_1 \leftarrow e_1, \ldots, v_m \leftarrow e_m]$ be the weakest precondition[1] of p with respect to the assignment part of the operation $\tau_i(l_i, l'_i)$. Add the atomic propositions occurring in $wp_{\tau_i(l_i,l'_i)}(p)$ as new predicates to *Pred*.

Our procedure refines the three-valued abstraction by adding further predicates to the set *Pred*. According to Corollary 1, such a refinement is sound in the sense that it preserves the validity of definite temporal logic properties.

We now exemplify our iterative abstraction refinement approach based on the simple system *Sys* and the corresponding concrete control flow graph G_c depicted in Figure 9.

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

Sys ::

```
0: while(y > 0)
      [ y := y - 1; ]
1: END
```

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

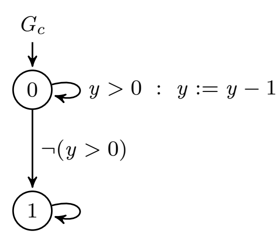

Figure 9: Concurrent system *Sys* and corresponding concrete control flow graph G_c.

Here we have a single process operating on the integer variable y and we want to check whether there exists an execution that finally reaches control flow location 1. Thus, the temporal logic property of interest is $\mathbf{F}(loc = 1)$. In the first iteration, we start with bound $b = 1$ and $Pred = \varnothing$. The corresponding abstract control flow graph G_{a_1}, computable with a three-valued abstractor, is depicted in Figure 10.

In order to solve the corresponding three-valued bounded model checking problem $[M(G_{a_1}) \models_{E,1} \mathbf{F}(loc = 1)]$ we construct the propositional logic encod-

[1]Computable via an SMT solver with built-in linear integer arithmetic theory. In our approach we use Z3 [17].

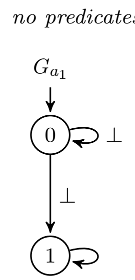

Figure 10: Abstract control flow graph G_{a_1}.

ing that now comprises the causes of uncertainty:

$$[\![Sys(G_{a_1}), \mathbf{F}(loc = 1)]\!]_1 \;=$$

$$\underbrace{(\neg l_0)}_{Init_0} \wedge \underbrace{((\neg l_0 \wedge \neg l_1 \wedge \perp^{(0,0)_0}) \vee (\neg l_0 \wedge l_1 \wedge \perp^{(0,1)_0}) \vee (l_0 \wedge l_1))}_{Trans_{0,1}} \wedge \underbrace{(l_0 \vee l_1)}_{[\![\mathbf{F}(loc=1)]\!]_1^0}$$

Since our system consists of a single process and only one digit is necessary to encode the control flow, we can omit the process- and digit-indices of the atoms. Solely the position index is required for the encoding. Now we run the associated satisfiability tests. We get

$$\mathrm{SAT}([\![Sys(G_{a_1}), \mathbf{F}(loc = 1)]\!]_1^+) = \mathit{true}$$

and the corresponding satisfying truth assignment $\alpha : l_0 \mapsto \mathit{false}, l_1 \mapsto \mathit{true}$. Moreover, we get

$$\mathrm{SAT}([\![Sys(G_{a_1}), \mathbf{F}(loc = 1)]\!]_1^-) = \mathit{false}$$

Hence, $[M(G_{a_1}) \models_{E,1} \mathbf{F}(loc = 1)]$ yields *unknown* and α characterises an unconfirmed witness path

$$\langle 0 \rangle \stackrel{\perp}{\rightarrow} \langle 0 \rangle$$

in the abstract state space where $\stackrel{\perp}{\rightarrow}$ denotes an *unknown* transition between the states. Next we apply the procedure *Cause-Guided Abstraction Refinement*. Remember that SAT solvers always operate on formulae transferred into conjunctive normal form (CNF). The current encoding is equivalent to the following formula in CNF[2]

$$(\neg l_0) \wedge (l_0 \vee \neg l_1 \vee \bot^{\mathbf{(0,1)_0}}) \wedge (l_1 \vee \bot^{\mathbf{(0,0)_0}})$$
$$\wedge (l_0 \vee \bot^{\mathbf{(0,0)_0}} \vee \bot^{\mathbf{(0,1)_0}}) \wedge (l_1 \vee \bot^{\mathbf{(0,0)_0}} \vee \bot^{\mathbf{(0,1)_0}}) \wedge (l_0 \vee l_1)$$

Under the assignment $\alpha[\bot \mapsto \mathit{false}]$ (the assignment α extended by the assignment that maps all $\bot$'s to *false*) we get the following set of unsatisfied clauses for our encoding:

$$U \;=\; \{(l_0 \vee \neg l_1 \vee \bot^{\mathbf{(0,1)_0}}), (l_0 \vee \bot^{\mathbf{(0,0)_0}} \vee \bot^{\mathbf{(0,1)_0}})\}$$

A corresponding set of causes of uncertainty that covers U is

$$\mathit{Causes} \;=\; \{(0,1)_0\}$$

since $\bot^{\mathbf{(0,1)_0}}$ occurs in all clauses of U. $(0,1)_0$ indicates that at the current level of abstraction uncertainty is caused by the missing guard of the operation $\tau(0,1)$. We have that $\tau(0,1) = \neg(y > 0)$ in the concrete system. Hence, we add $(y > 0)$ to the set of predicates: $\mathit{Pred} := \mathit{Pred} \cup \{(y > 0)\}$ and proceed with the next iteration.

In the second iteration, we have $b = 1$ and $\mathit{Pred} = \{(y > 0)\}$. The corresponding abstract control flow graph G_{a_2} is depicted in Figure 11.

$(y > 0)$: **predicate where** $(y > 0) = \mathit{true}$;

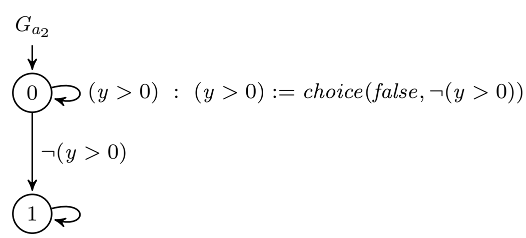

Figure 11: Abstract control flow graph G_{a_2}.

In order to solve $[M(G_{a_2}) \models_{E,1} \mathbf{F}(\mathit{loc} = 1)]$ we construct the following encoding ($(y > 0)$ abbreviated by p):

$$[\![\mathit{Sys}(G_{a_2}), \mathbf{F}(\mathit{loc} = 1)]\!]_1 \;=$$
$$(\neg l_0 \wedge \neg p[u]_0 \wedge p[t]_0) \wedge ((\neg l_0 \wedge \neg l_1 \wedge \mathit{enc}(\tau(0,0))_{0,1})$$
$$\vee (\neg l_0 \wedge l_1 \wedge \mathit{enc}(\tau(0,1))_{0,1}) \vee (l_0 \wedge l_1 \wedge \mathit{enc}(\tau(1,1))_{0,1})) \wedge (l_0 \vee l_1)$$

[2]For the sake of simplicity we use a standard CNF transformation in this illustrating example. Note that in our implementation we use the more compact Tseitin CNF transformation which introduces additional auxiliary atoms. Hence, we would get a slightly different CNF formula and unsatisfied clauses. Nevertheless, these clauses would hint at exactly the same causes of uncertainty.

with

$$
\begin{aligned}
enc(\tau(0,0))_{0,1} \;=&\\
((p[u]_0 \wedge \bot^{\boldsymbol{p_0}}) \vee (\neg p[u]_0 \wedge p[t]_0))&\\
\wedge(((p[u]_0 \wedge \bot^{\boldsymbol{p_0}}) \vee (\neg p[u]_0 \wedge \neg p[t]_0)) \wedge (\neg p[u]_1 \wedge \neg p[t]_1))&\\
\vee((p[u]_0 \vee (\neg p[u]_0 \wedge p[t]_0)) \wedge (p[u]_1)))&
\end{aligned}
$$

and $\tau(0,1))_{0,1}$ and $\tau(1,1))_{0,1}$ encoded analogously. As we can see, uncertainty is now potentially caused by predicate p evaluating to $\bot$ at position 0. Now we run the associated satisfiability tests. We get

$$\mathrm{SAT}([\![Sys(G_{a_2}), \mathbf{F}(loc = 1)]\!]_1^+) = \mathit{false}$$

and

$$\mathrm{SAT}([\![Sys(G_{a_2}), \mathbf{F}(loc = 1)]\!]_1^-) = \mathit{false}$$

Hence, $[M(G_{a_2}) \models_{E,1} \mathbf{F}(loc = 1)]$ yields *false*, which indicates that there does not exist a 1-bounded witness path for $\mathbf{F}(loc = 1)$. Consequently, we increment the bound: $b := b + 1$ and proceed with the next iteration.

In the third iteration, we have $b = 2$ and still $Pred = \{(y > 0)\}$. Hence, we continue with the abstract the control flow graph G_{a_2}. In order to solve $[M(G_{a_2}) \models_{E,2} \mathbf{F}(loc = 1)]$ we construct the following encoding:

$$
\begin{aligned}
[\![Sys(G_{a_2}), \mathbf{F}(loc = 1)]\!]_2 \;=&\\
(\neg l_0 \wedge \neg p[u]_0 \wedge p[t]_0) \wedge \textstyle\bigwedge_{k=0}^{1}((\neg l_k \wedge \neg l_{k+1} \wedge enc(\tau(0,0))_{k,k+1})&\\
\vee(\neg l_k \wedge l_{k+1} \wedge enc(\tau(0,1))_{k,k+1}) \vee (l_k \wedge l_{k+1} \wedge enc(\tau(1,1))_{k,k+1}))&\\
\wedge(l_0 \vee l_1 \vee l_2)&
\end{aligned}
$$

Now we run the associated satisfiability tests. We get

$$\mathrm{SAT}([\![Sys(G_{a_2}), \mathbf{F}(loc = 1)]\!]_2^+) = \mathit{true}$$

and the corresponding satisfying truth assignment α :

$$
\begin{aligned}
&l_0 \mapsto \mathit{false},\ l_1 \mapsto \mathit{false},\ l_2 \mapsto \mathit{true},\\
&p[u]_0 \mapsto \mathit{false},\ p[t]_0 \mapsto \mathit{true},\\
&p[u]_1 \mapsto \mathit{true},\ p[t]_1 \mapsto \mathit{true},\\
&p[u]_2 \mapsto \mathit{true},\ p[t]_2 \mapsto \mathit{true}.
\end{aligned}
$$

Moreover, we get

$$\mathrm{SAT}([\![Sys(G_{a_2}), \mathbf{F}(loc = 1)]\!]_2^-) = \mathit{false}$$

Hence, $[M(G_{a_2}) \models_{E,2} \mathbf{F}(loc = 1)]$ yields *unknown* and α characterises an unconfirmed witness path

$$\langle 0, p = true \rangle \rightarrow \langle 0, p = \perp \rangle \overset{\perp}{\rightarrow} \langle 1, p = \perp \rangle$$

in the abstract state space. Next we apply the procedure *Cause-Guided Abstraction Refinement*. After deriving the set U of clauses of $[\![Sys(G_{a_2}), \mathbf{F}(loc = 1)]\!]_2$ that are unsatisfied under the assignment $\alpha[\perp \mapsto false]$, we determine a corresponding set of causes covering U. We get:

$$Causes = \{p_1\}$$

p_1 indicates that at the current level of abstraction uncertainty is caused by the predicate p evaluating to *unknown* at position 1 of the witness path characterised by α. Now we determine the last predecessor position where the value of p is known, that is the greatest $k < 1$ with $\alpha(p[u]_k) = false$. This holds for $k = 0$. Hence, the transition from position 0 to 1 along the witness path characterised by α makes p *unknown*. We have that $\alpha(l_0) = false$ and $\alpha(l_1) = false$, which indicates that the transition from position 0 to 1 is associated with the operation $\tau(0,0)$. From the concrete control flow graph we get that the assignment part of $\tau(0,0)$ is $y := y - 1$. Thus, the weakest precondition of $p = (y > 0)$ with respect to $\tau(0,0)$ is

$$wp_{\tau(0,0)}(y > 0) \;=\; (y > 0)[y \leftarrow y - 1] \;=\; (y - 1 > 0) \;=\; (y > 1).$$

Hence, we add $(y > 1)$ to the set of predicates and proceed with the next iteration.

In the forth iteration, we have $b = 2$ and $Pred = \{(y > 0), (y > 1)\}$. The corresponding abstract control flow graph G_{a_3} is depicted in Figure 12.

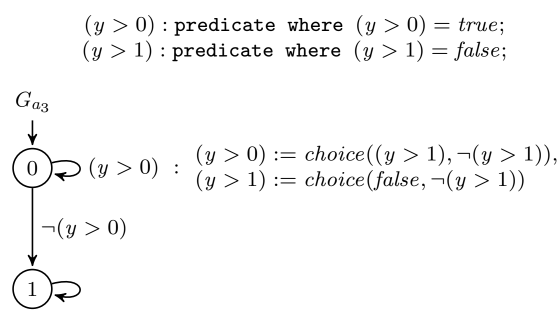

Figure 12: Abstract control flow graph G_{a_3}.

In order to solve $[M(G_{a_3}) \models_{E,2} \mathbf{F}(loc = 1)]$ we construct the encoding $[\![Sys(G_{a_3}), \mathbf{F}(loc = 1)]\!]_2$ and run the associated satisfiability tests. We get

$$\mathrm{SAT}([\![Sys(G_{a_3}), \mathbf{F}(loc = 1)]\!]_2^+) = true$$

and

$$\mathrm{SAT}([\![Sys(G_{a_3}), \mathbf{F}(loc = 1)]\!]_2^-) = true$$

and the corresponding satisfying truth assignment α :

$$\begin{aligned}
&l_0 \mapsto false, \quad l_1 \mapsto false, \ l_2 \mapsto true, \\
&p[u]_0 \mapsto false, \ p[t]_0 \mapsto true, \\
&p[u]_1 \mapsto false, \ p[t]_1 \mapsto false, \\
&p[u]_2 \mapsto false, \ p[t]_2 \mapsto false, \\
&q[u]_0 \mapsto false, \ q[t]_0 \mapsto false, \\
&q[u]_1 \mapsto true, \ q[t]_1 \mapsto false, \\
&q[u]_2 \mapsto true, \ q[t]_2 \mapsto false
\end{aligned}$$

where p abbreviates $(y > 0)$ and q abbreviates $(y > 1)$. We can immediately conclude that $[M(G_{a_3}) \models_{E,2} \mathbf{F}(loc = 1)]$ yields *true*, which indicates that α characterises a definite 2-bounded witness path

$$\langle 0, p = true, q = false\rangle \to \langle 0, p = false, q = \bot\rangle \to \langle 1, p = false, q = \bot\rangle$$

for $\mathbf{F}(loc = 1)$. This outcome completes our verification task. Within four iterations of cause-guided abstraction refinement resp. bound incrementation we have automatically proven that the property of interest holds for the system.

Thus, given a software verification task to be solved, our cause-guided refinement approach enables us to automatically reach the right level of abstraction in order to obtain a definite result in verification. Next, we present the implementation of our encoding-based model checking technique and we report on experimental results.

7. Implementation, Enhancements and Experiments

In this section we introduce the implementation of our theoretical concepts and we present enhancements based on existing work on *temporal induction* [10] as well as on *symmetry-based parameterised verification* [11]. Moreover, we discuss several experimental results.

7.1. The TVMC *Tool*

We have implemented a SAT-based bounded model checker called TVMC for three-valued abstractions of concurrent software systems.[3] TVMC employs a three-valued abstractor [2] that builds abstract control flow graphs for a given concurrent system *Sys* and a set of predicates *Pred*. It supports almost all control structures of the C language as well as *int*, *bool* and *semaphore* as data

[3] available at `www.github.com/ssfm-up/TVMC`

types. Based on the CFGs and an input LTL formula ψ, our tool automatically constructs an encoding $[\![Sys, \psi]\!]_b$ of the corresponding verification task. TVMC iteratively refines the abstraction in case of an *unknown* result and increments the bound in case of a *false* result. It terminates once a *true* result is obtained or a *false* result is obtained for a predefined threshold of the bound: In each iteration the two instances of the encoding are processed by a solver thread of the SAT solver Sat4j [18]. A *true* result for $[\![Sys, \psi]\!]_b^-$ can be immediately transferred to the corresponding model checking problem $[M \models_{E,b} \psi]$. The same holds for a *false* result for $[\![Sys, \psi]\!]_b^+$ if b represents a completeness threshold of the verification task [14]. In case of an *unknown* result we apply cause-guided abstraction refinement as defined in the previous section. For *true* and *unknown* results, we additionally output a definite resp. unconfirmed witness path for the property ψ in the form of an assignment satisfying $[\![Sys, \psi]\!]_b$. The tool chain of TVMC is depicted in Figure 13.

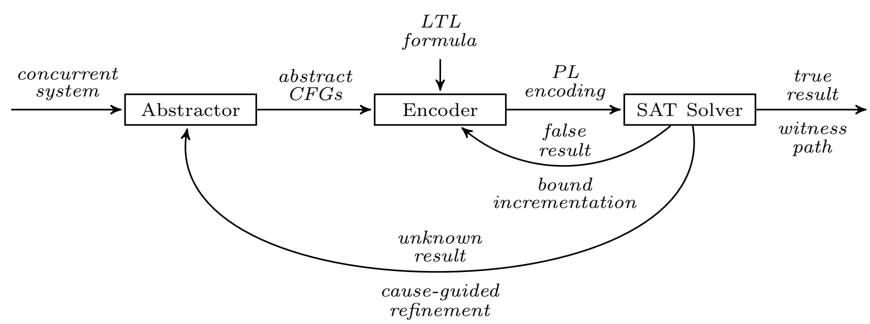

Figure 13: TVMC tool chain.

We now illustrate how our tool systematically solves verification tasks via three-valued abstraction and cause-guided refinement. An example system $Sys = \|_{i=1}^{n} P_i$ implementing a solution to the dining philosophers problem is depicted in Figure 14. Here we have $n \in \mathbb{N}$ philosopher processes and the same number of binary semaphore variables modelling the forks. Processes P_i with $i < n$ continuously attempt to first acquire the semaphore y_i and second the semaphore y_{i+1}, whereas process P_n attempts to acquire first y_1 and then y_n. Hence, all philosophers will always pick up the lower-numbered fork first and the higher-numbered fork second. Once a process has successfully acquired both semaphores it consecutively releases them and attempts to acquire them again.

$$y_1, \ldots, y_n : \texttt{binary semaphore where } y_1 = \mathit{true};\ \ldots;\ y_n = \mathit{true};$$

$$\|_{i=1}^{n-1}\ P_i :: \left[\begin{array}{l} \texttt{loop forever do} \\ \left[\begin{array}{l} \texttt{00: acquire } (y_i); \\ \texttt{01: acquire } (y_{i+1}); \\ \texttt{10: CRITICAL} \\ \quad\ \texttt{release } (y_i); \\ \texttt{11: release } (y_{i+1}); \end{array}\right] \end{array}\right] \ \|\ P_n :: \left[\begin{array}{l} \texttt{loop forever do} \\ \left[\begin{array}{l} \texttt{00: acquire } (y_1); \\ \texttt{01: acquire } (y_n); \\ \texttt{10: CRITICAL} \\ \quad\ \texttt{release } (y_1); \\ \texttt{11: release } (y_n); \end{array}\right] \end{array}\right]$$

Figure 14: Dining philosophers system *Sys*.

TVMC generally searches for violations of desirable properties. For an instantiation of the dining philosophers system with $n = 2$ the violation of deadlock-freedom can be expressed in LTL as

$$\psi \ = \ \mathbf{F}((loc_1 = 01) \wedge (loc_2 = 01)).$$

Hence, we want to check whether a state is reachable where each philosopher has picked up one fork and is waiting for the other fork. Starting with $Pred = \varnothing$ and $b = 1$ our tool automatically constructs the corresponding abstract control flow graphs and the encoding $[\![Sys, \psi]\!]_b$. Next, it iteratively increments the bound and refines the initial abstraction. For our example the bound will be increased until an unconfirmed witness path for the property of interest will be detected at $b = 2$:

$$\langle 00, 00\rangle \stackrel{\perp}{\rightarrow} \langle 01, 00\rangle \stackrel{\perp}{\rightarrow} \langle 01, 01\rangle$$

Then cause-guided refinement will add the predicates y_1 and y_2 in a single step , which yields the level of abstraction characterised by the abstract control flow graphs depicted in Figure 15. Finally the bound will be further increased until a completeness threshold is reached, which is the case for $b = 64$ for this verification task. A technique for computing over-approximations of completeness thresholds is introduced in [14]. Completeness thresholds for checking LTL properties that are restricted to the temporal operators $\mathbf{F}$ and $\mathbf{G}$ are linear in the size of the abstraction, i.e. in the number of abstract states.

$$y_1, y_2 : \texttt{predicate where } y_1 = \mathit{true};\ y_2 = \mathit{true};$$

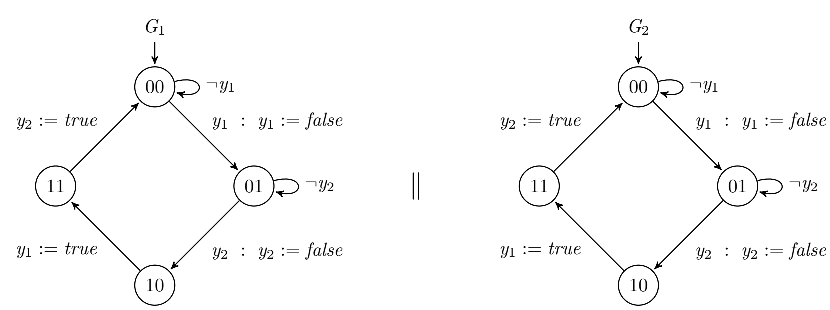

Figure 15: Abstraction of the dining philosophers system with $n = 2$.

Via SAT solving we obtain a *false* result in the final iteration, which allows us to conclude that the instantiation of the dining philosophers system with 2 processes is *safe* in terms of deadlock-freedom.

A distinct feature of our approach is the verification of *liveness* properties of concurrent systems under fairness assumptions. The formula

$$\psi' \;=\; \bigvee_{i=1}^{n} \mathbf{F}(\mathbf{G}\neg(loc_i = 10))$$

characterises the violation of a liveness property regarding our dining philosophers system. It states that eventually some philosopher process will nevermore reach its critical location. For the instantiation of the system with $n = 2$ processes and starting with $Pred = \{progress_1, progress_2\}$ (fairness predicates only) and $b = 1$ our tool constructs the encoding $[\![Sys, wfair \wedge \psi']\!]_b$. Within four iterations over b and a refinement step adding the predicates y_1 and y_2 we can already detect a satisfying assignment for the encoding that characterises a weak fair path with a loop where P_2 never reaches its critical location:

$$\begin{array}{c}
\langle 00, 00, y_1 = true, y_2 = true, progress_1 = false, progress_2 = false\rangle \\
\downarrow \\
\langle 01, 00, y_1 = false, y_2 = true, progress_1 = true, progress_2 = false\rangle \\
\downarrow \\
\langle 01, 00, y_1 = false, y_2 = true, progress_1 = false, progress_2 = true\rangle \\
\downarrow \\
\langle 11, 00, y_1 = false, y_2 = false, progress_1 = true, progress_2 = false\rangle \\
\downarrow \\
\langle 00, 00, y_1 = true, y_2 = true, progress_1 = true, progress_2 = false\rangle
\end{array}$$

Thus, we have proven that liveness of *Sys* is violated under *weak* fairness. With our tool we could also prove that liveness of *Sys* holds under *strong* fairness, which required to set the bound to the completeness threshold of the verification task. Moreover, we could successfully verify generalisations of the dining philosophers system with more philosophers and semaphores, which we will discuss in the experimental results section.

The previous examples illustrate the basic functionality of the Tvmc tool. Next, we introduce an extension that allows for an improved verification of safety properties, without setting the bound to the completeness threshold.

7.2. Complete Safety Verification via Temporal Induction

Bounded model checking is inherently incomplete since the bound restricts the length of the explored paths. Hence, it is predominantly used for for detecting property violations rather than for proving their absence. As we previously outlined, a simple way of making bounded model checking complete is to determine a completeness threshold of a verification task and to set the bound to this threshold. However, determining small or minimal completeness thresholds is costly and for many verification tasks even the minimal threshold might be to large for efficient verification. An alternative way of establishing completeness is the combination of bounded model checking with *temporal induction* [10]. Temporal induction allows to reduce an unbounded model checking problem to two bounded model checking problems: the *base case* and the *inductive step*. The approach is defined for checking safety violations of the form $\mathbf{F}(\mathit{unsafe})$ where *unsafe* is a predicate expression. In the base case it is checked whether there exists a b-bounded path, starting in the initial state, that witnesses the safety violation. In the inductive step it is checked whether there exists a loop-free $(b+1)$-bounded path, starting in an arbitrary state, where *unsafe* only holds in the $(b+1)st$ state. If there exists a bound b for which neither in the base case nor in the step such witnesses can be detected, then it can be concluded that safety is globally not violated. Both the base case and the inductive step can be straightforwardly expressed as three-valued bounded model checking problems of our framework. The base case does not require any change of our encoding:

$$[\![\mathit{Sys}, \mathbf{F}(\mathit{unsafe})]\!]_b^{base} \;:=\; [\![\mathit{Sys}, \mathbf{F}(\mathit{unsafe})]\!]_b.$$

For the step we have to remove the initial state constraint Init_0 and add a constraint *loopFree* that ensures that all states along the considered path prefixes are pairwise different. Moreover, we have to adjust the property encoding such that it is checked whether *unsafe* only holds in the $(b+1)st$ state:

$$[\![\mathit{Sys}, \mathbf{F}(\mathit{unsafe})]\!]_{b+1}^{step} := \bigwedge_{k=0}^{b} \mathit{Trans}_{k,k+1} \wedge \mathit{loopFree} \wedge \neg[\![\mathbf{F}(\mathit{unsafe})]\!]_b^0 \wedge [\![\mathit{unsafe}]\!]_{b+1}^{b+1}$$

with $\mathit{loopFree} = \bigwedge_{0 \leq i < j \leq b+1} \big(\bigvee_{p \in AP} \neg(\mathit{enc}(p)_i \leftrightarrow \mathit{enc}(p)_j)\big)$.

We have integrated the temporal induction approach into Tvmc. Similar to standard three-valued bounded model checking, we have to consider the cases where $\perp$ is mapped to *true* and where $\perp$ is mapped to *false*, but now for both the base case and the inductive step. A *true* result for the base case implies that we have detected a safety violation. A *false* result for both the base case and the step implies that safety is globally not violated. If we obtain an *unknown* result for either the base case or the step, then we apply cause-guided abstraction refinement. If we get *false* for the base case and *true* for the inductive step, then we have to increment the bound. We checked deadlock-freedom for the dining philosophers example in Figure 14 with the induction approach. As the predicate expression *unsafe* we used $(loc_1 = 01) \wedge (loc_2 = 01)$. In comparison with the completeness threshold approach, that required the final bound $b = 64$, temporal induction enabled us to already prove mutual exclusion with bound $b = 3$. In our experimental evaluation we will present a case study on the induction-based approach.

7.3. Parameterised Verification via Spotlight Abstraction

So far, we have seen that our tool can be used for detecting property violations and also for proving their absence. However, in all of our previous examples we verified systems with a *fixed* number of processes. The three-valued abstractor that we use also allows to construct finite abstractions of parameterised systems with an *unbounded* number of uniform processes [11]. A simple example of a parameterised system with uniform processes can be derived from our dining philosophers. Let $\mathit{Sys} = \|_{i=1}^{n} P_i$ be a concurrent system over $\mathit{Var} = \{y_1, y_2\}$ where each P_i is a philosopher that picks up y_1 first and y_2 second. As long as we do not instantiate the number of processes n, this system is *parameterised*. But in contrast to our previous philosopher example, where all processes requested different pairs of forks in different orders, the processes of our modified example are fully uniform in terms of fork requests.

For constructing a finite abstraction of such a parametrerised uniform system, three-valued predicate abstraction is combined with *spotlight abstraction* [2]. The basic concept of spotlight abstraction is to partition a parallel composition of processes into a *spotlight* and a *shade*. The control flow of spotlight processes is then explicitly considered under abstraction, whereas the processes in the shade get summarised into a single abstract process $P_\perp$ that approximates their behaviour with regard to three-valued logic. Hence, predicates over variables that are modified by processes in the shade may be set to *unknown* by $P_\perp$. For our parameterised uniform philosopher system $\mathit{Sys} = \|_{i=1}^{n} P_i$, spotlight abstraction can be applied to the process partition $\mathit{Spotlight} = \{P_1, P_2\}$ and $\mathit{Shade} = \{P_3, \ldots, P_n\}$, i.e. we consider P_1 and P_2 explicitly and summarise the parameterised number of processes P_3 to P_n into $P_\perp$. The property $\psi = \mathbf{F}((\mathit{loc}_1 = 01) \wedge (\mathit{loc}_2 = 01))$ can also be disproven for the abstraction $P_1 \parallel P_2 \parallel P_\perp$ with our tool, which allows us to conclude that the processes P_1 and P_2 will never be in a deadlock situation for *any* instantiation of the uniform philosopher system with $n \geq 2$ processes. The LTL formula ψ characterises a local property since it refers to particular processes of a parameterised system. However, as shown in [11], symmetry arguments enable us to transfer this result to arbitrary pairs of processes in the system. We can conclude that

$$\psi_{global} \;=\; \exists\, 1 \leq i, j \leq n,\ i \neq j : \mathbf{F}((\mathit{loc}_i = 01) \wedge (\mathit{loc}_j = 01))$$

does not hold for any instantiation of the uniform philosophers system, i.e. no pair of processes will ever reach a deadlock. The approach also works for parameterised systems with different (but finite numbers of) classes of uniform processes, e.g. a class of reader processes and a class of writer processes. More details about spotlight abstraction and its combination with symmetry arguments can be found in [11]. In our experimental evaluation we will present a case study on parameterised verification with our tool.

7.4. Experimental Evaluation

We have experimentally evaluated the performance of our tool in a number of case studies. In our experiments, we compared Tvmc with the similar three-valued model checking tool 3Spot [2], which is also designed for the verification of concurrent systems. 3Spot uses the same abstractor as Tvmc that yields abstract control flow graphs. After abstraction, the tool follows a different approach to solve verification tasks: The state space corresponding to the abstract control flow graphs is represented as a three-valued decision diagram (TDD), which is a generalisation of a Boolean decision diagram (BDD). The model checking algorithm of 3Spot is therefore closely related to BDD-based CTL model checking [19]. Hence, we compare a decision diagram-based model checking approach with our SAT-based approach. Similar to Tvmc, 3Spot generates unconfirmed witness paths in case of an *unknown* result and refines the abstraction based on these paths. The witness paths of 3Spot are explicitly generated and explored, whereas our witness paths are implicitly represented by truth assignments. Since 3Spot is a CTL model checker, we focussed in our case studies on temporal logic properties from the common fragment of LTL and CTL. We conducted our experiments on a 1.6 GHz Intel Core i7 system with 8 GB memory. We measured the final bound, the number of refinement steps, the final number of predicates as well as the overall time for encoding and SAT-based model checking in all iterations, and we compared it with the performance results of 3Spot.

In the case study Philosophers, we verified generalisations of the dining philosopher system with $n \leq 2$ processes. Each additional philosopher process involved an exponential growth of the state space complexity. We checked for the existence of a computation where eventually some philosopher will nevermore reach its critical location, which characterises the violation of a liveness property. Verification was performed under the assumption of weak fairness, as this is the only type of fairness constraint that is supported by 3Spot. Since liveness of the philosopher system is not guaranteed under weak fairness, we could always detect a property violation. The performance results of the Philosophers case study are shown in Table 1. We can see that for very small instances the time performance of 3Spot was better, whereas for larger n our Tvmc was significantly faster than 3Spot. Hence, our SAT-based approach scaled considerably better here. Another observation is that both tools required the same number of predicates for completing the verification tasks. However, while 3Spot needed several refinement iterations in order to reach the right level of abstraction, our cause-guided refinement detected and added all necessary predicates within a single step. More specifically, our tool generated an unconfirmed witness path in the first iteration from which all necessary predicates could be immediately derived in order to refine the path to a real witness.

In case study Dijkstra, we verified an implementation of Dijkstra's mutual exclusion algorithm [20]. Again, we considered instances with an increasing number of processes. Dijkstra's algorithm for two processes is depicted in Figure 16. We checked whether there is no violation of mutual exclusion, i.e. whether there will be never more than one process at the critical location at the same time. In order to prove that this safety property holds, we used Tvmc with the temporal induction approach discussed in Section 7.2. Since 3Spot is an unbounded model checker, there was no need to use temporal induction with this tool. The performance results of the Dijkstra case study are also show in Table 1. Similar as in the previous case study we can observe that Tvmc scales

		Tvmc				3Spot		
case study	processes	bound	refinements	predicates	time	refinements	predicates	time
Philosophers	2	3	1	4	$1.13s$	2	4	$0.41s$
	3	5	1	6	$2.12s$	3	6	$0.92s$
	4	7	1	8	$4.69s$	4	8	$47.7s$
	5	9	1	10	$12.4s$	5	10	$696s$
	6	11	1	12	$38.1s$	6	12	$152m$
	7	13	1	14	$379s$	7	14	$> 5h$
	8	15	1	16	$75.0m$	8	16	$> 5h$
Dijkstra	2	12	3	6	$3.32s$	2	2	$0.18s$
	3	16	4	9	$22.2s$	5	5	$3.05s$
	4	21	5	12	$52m$	> 6	> 6	OOM
	5	25	6	15	$158m$	> 6	> 6	OOM
Parameterised	2 classes	4	1	2	$0.85s$	2	2	$0.16s$
	3 classes	6	1	3	$1.17s$	3	3	$0.30s$
	4 classes	8	1	4	$1.82s$	4	4	$0.98s$
	5 classes	10	1	5	$8.63s$	5	5	$55.8s$
	6 classes	12	1	6	$89.2s$	6	6	$799s$

Table 1: Experimental Results of the case studies Philosophers, Dijkstra and Parameterised.

better for larger systems than 3Spot. Our new tool could even successfully prove safety when 3Spot failed due to an out-of-memory exception (OOM). A second observation is that in the cases where both tools succeeded (two and three processes), Tvmc required more predicates than 3Spot. This is due to the fact that proving the inductive step of the temporal induction approach requires some extra predicates because of the arbitrary initial state. Nevertheless, Tvmc still showed a good performance with these extra predicates. If we compare the Philosophers case study where a property violation could be detected with the Dijkstra case study where correctness could be proven, then we can see that in the latter Tvmc requires multiple iterations of cause-guided refinement in order to achieve a definite result. This is due to the fact that in the second case study each refinement iteration rules out a different unconfirmed witness path until no more witnesses exist, whereas in the first case study the first detected unconfirmed witness could be immediately refined to a real one. Another observation that we can make by comparing the two case studies is that showing the absence of errors generally requires more computational effort than detecting errors, which is not surprising.

```
turn : integer where turn = 1;
flag1, flag2 : integer where flag1 = 0; flag1 = 0;

P1 :: [ loop forever do                          || P2 :: [ loop forever do
        [ 0000: flag1 := 1;                               [ 0000: flag2 := 1;
          0001: while(turn ≠ 1)                             0001: while(turn ≠ 2)
          [ 0010: if(turn = 2)                              [ 0010: if(turn = 1)
            [ 0011: if(flag2 = 0)                             [ 0011: if(flag1 = 0)
              [ 0100: turn := 1; ] ] ]                          [ 0100: turn := 2; ] ] ]
          0101: flag1 := 2;                                 0101: flag2 := 2;
          0110: if(flag2 = 2)                               0110: if(flag1 = 2)
          [ 0111: goto 0000; ]                              [ 0111: goto 0000; ]
          1000: CRITICAL                                    1000: CRITICAL
          1001: flag1 := 0; ] ]                             1001: flag2 := 0; ] ]
```

Figure 16: Implementation of Dijkstra's Mutual Exclusion Algorithm with $n = 2$.

In case study Parameterised, we evaluated parameterised verification via spotlight abstraction. For this, we looked at parameterised variants of the dining philosopher system with an increasing number of classes of uniform processes. In the simplest case, we considered a system with two classes of philosophers: an unbounded number $n \in \mathbb{N}$ of philosophers that pick up fork y_1 first and fork y_2 second, and an unbounded number $m \in \mathbb{N}$ of philosophers that pick up y_2 first and y_1 second. The corresponding parameterised system is depicted in Figure 17. Moreover, we considered generalised cases e.g. a system with three classes and the three different pick up orders y_1 then y_2, y_2 then y_3, and y_3 then y_1. In these parameterised systems a deadlock in the sense of circular waiting for forks is possible. We used Tvmc with the spotlight abstraction approach discussed in Section 7.3 in order to check this property. Since 3Spot also supports spotlight abstraction, we could compare the performance of the two tools. The experimental results of the Parameterised case study are shown in Table 1. As we can see, our approach is also capable of checking properties of parameterised systems. Again, Tvmc scales better than 3Spot and our cause-guided refinement technique finds all necessary predicates for detecting a real witness path within a single refinement step.

```
y1, y2 : binary semaphore where y1 = true; y2 = true;

||_{i=1}^{n} Pi :: [ loop forever do              || ||_{j=1}^{m} Pj :: [ loop forever do
                     [ 00: acquire (y1);                                [ 00: acquire (y2);
                       01: acquire (y2);                                  01: acquire (y1);
                       10: CRITICAL                                       10: CRITICAL
                           release (y1);                                      release (y2);
                       11: release (y2); ] ]                              11: release (y1); ] ]
```

Figure 17: Parameterised dining philosophers system with two classes of uniform processes.

All in all, our experiments showed that SAT-based three-valued bounded model checking is a feasible approach to the verification of concurrent systems. We were able to detect violations of safety and liveness properties and to prove the absence of such violations. Cause-guided refinement allows to rule out uncertainty in the abstraction within a small number of iterations, especially when property violations can be detected. Moreover, we could enhance the performance of proving the absence of safety violations via temporal induction. Naturally, our approach cannot resolve the state explosion problem in general. Each additional process involves an exponentially greater complexity of the underlying verification task. However, we demonstrated that for larger systems our Tvmc tool scales significantly better than the comparable 3Spot tool. Furthermore, we were able to integrate spotlight abstraction into our approach, which facilitates the verification of parameterised systems composed of uniform processes.

8. Related Work

Our SAT-based software verification technique is related to a number of existing approaches in the field of bounded model checking for software. The bounded model checker CBMC [21] supports the verification of *sequential C programs*. It is based on a Boolean abstraction of the input program and it allows for checking buffer overflows, pointer safety and assertions, but not full LTL properties. A similar tool is F-Soft [22]. This bounded model checker for sequential programs is restricted to the verification of *reachability properties*. While CBMC and F-Soft support a wider range of program constructs like pointers and recursion, our approach focusses on the challenges associated with *concurrency* and the verification of *liveness properties* under fairness. The tool TCBMC [23] is an extension of CBMC for verifying *safety properties* of *concurrent programs*. TCBMC introduces the concept of bounding context switches between processes, which is a special abstraction technique for reducing concurrency. Our approach supports the process summarisation abstraction of 3Spot [2], which allows us to reduce the complexity induced by concurrency in a different way. The verification of concurrent C programs is also addressed in [24]. The authors introduce a tool that translates C programs into a TLA+ [25] specification which is then model checked via an explicit-state approach.

In contrast to the above mentioned tools, we employ *three-valued abstraction*, which preserves *true* and *false* results in verification. *Three-valued bounded model checking* is addressed in [6] and [7]. However, only in the context of *hardware verification* [6] resp. assuming that an *explicit three-valued Kripke structure* is given [7]. To the best of our knowledge, our approach is the first that supports software verification under fairness via an immediate propositional logic encoding and SAT-based BMC.

Abstraction refinement for SAT-based model checking is addressed in [26, 27, 28]. The hardware verification approach presented in [26] employs Boolean abstraction by means of variable hiding. Counterexamples detected in the abstract model are simulated on the concrete model via SAT solving. Unsatisfiability results correspond to spurious counterexamples. In this case abstraction refinement is applied by deriving new variables from the unsatisfiable core. Very similar approaches are used in [27] and [28]. The authors of [27] additionally show that their technique yields an over-approximative abstraction that preserves safety and always has a completeness threshold for not only refuting but also proving properties. The authors of [28] generate refinement interpolants from simulated spurious counterexamples in SAT-based model checking. In our three-valued approach we do not have to simulate abstract counterexamples. In case of an *unknown* result, our path characterising assignment allows us to derive new predicates from the set of unsatisfied clauses that is typically significantly smaller than the unsatisfiable core. Unconfirmed witness paths in a three-valued abstract model are also used for refinement in [2]. However, the proposed approach requires to explicitly generate and analyse paths, whereas our refinement happens based on unsatisfied clauses that implicitly represent uncertainty in the abstraction.

The notion of causality in the context of temporal logic model checking has also been used in [29]. The authors present a technique for detecting property violations in concurrent systems via counterfactual reasoning [30]. While classical model checking techniques generate witness paths resp. counterexamples, the approach of [29] derives *causal factors* that lead to a property violation. These causal factors are orders of occurrences of events in the analysed system and, according to the authors, provide a more concise explanation of why the desired property does not hold. The technique of [29] explores the concrete state space of the system, whereas our approach is based on three-valued predicate abstraction and iterative refinement: The lack of predicates over certain system variables may cause an *unknown* result in verification along with an unconfirmed witness. Hence, in our scenario a *cause of uncertainty* tells us which predicates are missing at the current level of abstraction in order to either confirm or to rule out the unconfirmed witness.

Our work is also related to other analysis techniques for concurrent systems. The authors of [31, 32] propose a construction-based method for ensuring deadlock- and livelock-freedom of process compositions. Hence, the systems to be analysed as well as the desired properties are similar to the ones in our approach. While [31] focusses on communication via message passing, we consider asynchronous systems with shared variables. [31] is based on the correctness-by-construction paradigm, whereas our technique aims at the verification of implementations of software systems. Thus, the two approaches complement each other in the sense that [31] can ensure a correct design, whereas our approach can ensure the correct implementation of a design. In [32] local analysis is used for establishing correctness of the entire system. This is slightly related to the spotlight abstraction that we use in our approach. Spotlight abstraction also facilitates the verification of local properties. For uniform systems such local verification results can be transferred to the global system.

9. Conclusion and Outlook

We introduced a verification technique for concurrent software systems based on three-valued abstraction, cause-guided refinement and SAT-based bounded model checking. We defined a direct propositional logic encoding of software verification tasks and we proved that our encoding is sound in the sense that SAT results can be straightforwardly transferred to the corresponding model checking problem. Hence, the expensive construction and exploration of an explicit state space model is not necessary. Our tool enables the verification of safety and liveness properties under fairness. With cause-guided refinement we introduced an automatic technique for systematically reaching the right level of abstraction in order to obtain a definite outcome in verification. Refinement steps are straightforwardly derived from clauses of the encoding that are unsatisfied under assignments characterising potential witness/error paths. Due to the efficiency of modern SAT solvers we achieve promising performance results with our overall approach.

As future work we plan to optimise our technique by integrating *incremental* SAT solving [33] into the tool and by developing a concept for reusing parts of the encoding between the consecutive refinement iterations. Finally, we want to develop SAT solving *heuristics* tailored to the structure of our encodings [34] in order to further accelerate our approach.

Acknowledgements

We thank Dewald de Jager and Matthias Harvey for help with implementing our approach.

References

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

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

[3] G. Bruns, P. Godefroid, Model checking partial state spaces with 3-valued temporal logics, in: CAV 1999, LNCS, Springer Berlin Heidelberg, 1999, pp. 274–287.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: a new symbolic model checker, Int. Jour. on Softw. Tools for Techn. Transfer 2 (4) (2000) 410–425.

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

[6] O. Grumberg, 3-Valued Abstraction for (Bounded) Model Checking, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 21–21. doi: 10.1007/978-3-642-04761-9_2.

[7] H. Wehrheim, Bounded model checking for partial Kripke structures, in: J. Fitzgerald, A. Haxthausen (Eds.), ICTAC, Vol. 5160 of LNCS, Springer, 2008, pp. 380–394. doi:10.1007/978-3-540-85762-4_26.

[8] N. Timm, S. Gruner, M. Harvey, A Bounded Model Checker for Three-Valued Abstractions of Concurrent Software Systems, Springer International Publishing, Cham, 2016, pp. 199–216. doi:10.1007/978-3-319-49815-7_12.
URL http://dx.doi.org/10.1007/978-3-319-49815-7_12

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: CAV, Vol. 1855 of LNCS, Springer, 2000, pp. 154–169. doi:10.1007/10722167_15.

[10] N. En, N. Srensson, Temporal induction by incremental sat solving, Electronic Notes in Theoretical Computer Science 89 (4) (2003) 543 – 560, bMC'2003, First International Workshop on Bounded Model Checking. doi:https://doi.org/10.1016/S1571-0661(05)82542-3.
URL http://www.sciencedirect.com/science/article/pii/S1571066105825423

[11] N. Timm, H. Wehrheim, On symmetries and spotlights - verifying parameterised systems, in: J. Dong, H. Zhu (Eds.), ICFEM, Vol. 6447 of LNCS, Springer, 2010, pp. 534–548.

[12] 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.

[13] M. Fitting, Kleene's 3-valued logics and their children, Fund. Inf. 20 (1-3) (1994) 113–131.

[14] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, J. Worrell, Linear completeness thresholds for bounded model checking, in: CAV, Springer, 2011, pp. 557–572.

[15] N. Timm, Bounded model checking für partielle Systeme, Masters thesis, University of Paderborn.

[16] N. Timm, S. Gruner, Three-valued bounded model checking with cause-guided abstraction refinement – proofs, Tech. rep., Department of Computer Science, University of Pretoria (August 2018).
URL http://hdl.handle.net/2263/66136

[17] 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

[18] D. Le Berre, A. Parrain, The Sat4j library, release 2.2, Journal on Satisfiability, Boolean Modeling and Computation 7 (2010) 59–64.

[19] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, Symbolic model checking: 1020 states and beyond, Information and Computation 98 (2) (1992) 142 – 170. doi:https://doi.org/10.1016/0890-5401(92)90017-A.
URL http://www.sciencedirect.com/science/article/pii/089054019290017A

[20] E. W. Dijkstra, Solution of a problem in concurrent programming control, Commun. ACM 8 (9) (1965) 569–. doi:10.1145/365559.365617.
URL http://doi.acm.org/10.1145/365559.365617

[21] D. Kroening, M. Tautschnig, CBMC–C bounded model checker, in: TACAS, Springer, 2014, pp. 389–391.

[22] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, P. Ashar, F-Soft: Software verification platform, in: International Conference on Computer Aided Verification, Springer, 2005, pp. 301–306.

[23] I. Rabinovitz, O. Grumberg, Bounded model checking of concurrent programs, in: CAV, Springer, 2005, pp. 82–97.

[24] A. Methni, M. Lemerre, B. Ben Hedia, S. Haddad, K. Barkaoui, Specifying and Verifying Concurrent C Programs with TLA+, Springer International Publishing, Cham, 2015, pp. 206–222. doi:10.1007/978-3-319-17581-2_14.
URL http://dx.doi.org/10.1007/978-3-319-17581-2_14

[25] S. Merz, The Specification Language TLA+, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 401–451. doi:10.1007/978-3-540-74107-7_8.
URL http://dx.doi.org/10.1007/978-3-540-74107-7_8

[26] A. Gupta, O. Strichman, Abstraction Refinement for Bounded Model Checking, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 112–124. doi:10.1007/11513988_11.
URL http://dx.doi.org/10.1007/11513988_11

[27] D. Kroening, Computing over-approximations with bounded model checking, Electron. Notes Theor. Comput. Sci. 144 (1) (2006) 79–92. doi:10.1016/j.entcs.2005.07.021.
URL http://dx.doi.org/10.1016/j.entcs.2005.07.021

[28] C. Y. Wu, C. A. Wu, C.-Y. Lai, C. Y. Huang, A counterexample-guided interpolant generation algorithm for sat-based model checking, in: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), 2013, pp. 1–6.

[29] A. Beer, S. Heidinger, U. Kühne, F. Leitner-Fischer, S. Leue, Symbolic causality checking using bounded model checking, in: B. Fischer, J. Geldenhuys (Eds.), Model Checking Software, Springer International Publishing, Cham, 2015, pp. 203–221.

[30] J. Y. Halpern, J. Pearl, Causes and explanations: A structural-model approach. part i: Causes, The British journal for the philosophy of science 56 (4) (2005) 843–887.

[31] R. Ramos, A. Sampaio, A. Mota, Systematic development of trustworthy component systems, in: A. Cavalcanti, D. R. Dams (Eds.), FM 2009: Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 140–156.

[32] M. S. C. Filho, M. V. M. Oliveira, A. Sampaio, A. Cavalcanti, Local livelock analysis of component-based models, in: K. Ogata, M. Lawford, S. Liu (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2016, pp. 279–295.

[33] A. Nadel, V. Ryvchin, O. Strichman, Ultimately incremental sat, in: C. Sinz, U. Egly (Eds.), Theory and Applications of Satisfiability Testing – SAT 2014, Springer International Publishing, Cham, 2014, pp. 206–218.

[34] N. Timm, S. Gruner, P. Sibanda, Model checking of concurrent software systems via heuristic-guided sat solving, in: M. Dastani, M. Sirjani (Eds.), Fundamentals of Software Engineering, Springer International Publishing, Cham, 2017, pp. 244–259.