Constraint Reusing and k-Induction for Three-Valued Bounded Model Checking

Nils Timm(✉), Stefan Gruner, and Matthias Harvey

Department of Computer Science, University of Pretoria, Pretoria, South Africa
{ntimm,sgruner}@cs.up.ac.za

Abstract. Refinement-based model checking is an approach to software verification: Starting with an abstract software model, the model is iteratively refined until it is precise enough to prove or refute the property of interest. A downside is that it typically takes several iterations until the necessary precision is reached, and thus, resources are spent on repeating work that has already been performed in previous iterations. We tackle this by introducing a concept for reusing information between refinement iterations in order to reduce the computational overhead. Our approach extends our previous work on three-valued abstraction (3VA) and bounded model checking (BMC). 3VA allows to translate a verification problem into a SAT-encoded three-valued BMC problem that can be checked via a SAT solver. While there was formerly no information sharing between refinement iterations, we now show that logic constraints learned by the solver in the current iteration are also valid in future iterations. Reusing such constraints enables to prune the search space of SAT which leads to a speed-up of the iterative approach. Since we previously used standard BMC, the technique was incomplete and could be only used for detecting property violations but not for proving their absence. Here we combine three-valued BMC with k-induction, which makes the approach complete for model checking safety properties.

1 Introduction

Three-valued abstraction refinement (3VA) [15] is a technique for reducing the complexity of software verification. It proceeds by generating an abstract software model over predicates with the possible truth values *true*, *false* and *unknown*, where the latter is used to represent the loss of information due to abstraction. The model is iteratively refined by adding predicates until it is precise enough to prove or refute some temporal logic property. The evaluation of properties on such models is known as *three-valued model checking* (3MC) [3]. In 3VA both *true* and *false* results can be immediately transferred to the modelled system, whereas *unknown* indicates that the current abstract model is too coarse for a definite outcome. The advantage of 3VA is that it allows to gradually adjust the level of abstraction until the right balance between simplicity and precision is reached in order to verify the property. The downside is

T. Massoni and M. R. Mousavi (Eds.): SBMF 2018, LNCS 11254, pp. 126–143, 2018.
https://doi.org/10.1007/978-3-030-03044-5_9

that it typically takes several iterations until this happens, and thus, computational resources are spent on repeating work that has already been performed in earlier iterations. Here, we tackle this drawback by introducing a concept for sharing gathered information between refinement iterations in order to reduce the computational overhead.

Our approach extends previous work where we presented a verification technique for concurrent systems based on 3VA and *three-valued bounded model checking* (3BMC) [17,18]. Three-valued abstraction allows to translate verification problems into SAT-encoded 3BMC problems. Thus, verification is reduced to SAT solving. In each refinement iteration, a propositional formula that encodes the 3BMC problem for the current level of abstraction is generated and processed via a SAT solver. While there was formerly no information sharing between iterations, we now show that logic constraints learned by the solver in the current iteration are also valid in future iterations. SAT solvers employ *conflict-driven clause learning* [2] while processing a propositional formula, which generates constraint clauses that are used to prune the search space of the *current* SAT check. We prove that certain constraints that have been learned for our model checking encodings correspond to definite temporal logic properties of the encoded system. Since definite properties are preserved under three-valued abstraction refinement, it is permissible to *reuse* the associated constraints among iterations. Our inter-refinement iteration constraint reusing concept enables to considerably reduce the computational effort of 3VA-based verification.

In standard bounded model checking the bound $k \in \mathbb{N}$ restricts the length of execution paths of the modelled system, which makes the technique incomplete and only usable for detecting property violations but not for proving their absence. While our previous approach [17,18] has this limitation, we now establish completeness by integrating *k-induction* [16] into our approach. k-induction was originally introduced for verifying safety of hardware systems. It proceeds as follows: Given a state transition model of the system to be analysed and a state predicate $Safe$, it is checked whether all paths of length k that start in an initial state of the model are safe, i.e. whether $Safe$ holds in each state along the paths. This is the *base case* of k-induction, which is equivalent to standard bounded model checking. If the base case holds, then the *inductive step* is checked: Assuming k consecutive states where $Safe$ holds in each state, then $Safe$ also has to hold in every $(k+1)$-st successor state. The inductive step does not restrict the k consecutive states to start in an initial state. If the inductive step holds as well, then it can be concluded that *all* possible execution paths of the system are safe. Otherwise the procedure needs to be repeated with an incremented k.

Since hardware systems naturally correspond to state transition models, the application of k-induction is straightforward. In our software verification approach, we generate an *implicit* state transition model by applying abstraction and by encoding the state space of the abstract system in propositional logic. For the integration of k-induction, we define the base case and the inductive step of

our verification tasks as 3BMC problems. Moreover, we combine our refinement procedure with a bound incrementation procedure. We use a top-level bound incrementation loop and therein two refinement loops, one for the base case and one for the step, that we independently abstract and refine. Each iteration is now characterised by a bound and a level of abstraction of the base case and the inductive step. Depending on the SAT-based verification outcome in an iteration, either the bound is incremented, refinement is applied, or the procedure terminates with a definite result that can be transferred to the input system. Learned constraints are reused between refinement iterations based on our novel clause reusing concept. Furthermore, we reuse constraints between bound iterations, which is permissible due to the *incremental bounded model checking principle* [11]. In experiments, we demonstrate that our approach enables the complete verification of concurrent systems within linear integer arithmetic and we show that our constraint reusing leads to significant performance improvements.

2 Concurrent Software Systems

Our approach focusses on linear integer concurrent systems. Almost all control structures of the C language, concurrency and the variable types *bool* and *int* are supported. There is currently no support for arrays and pointers. A system Sys consists of processes P_1 to P_n composed in parallel: $Sys = \|_{j=1}^{n} P_j$. It is defined over a set of variables $Var = Var_{Sys} \cup Var_{PC}$. Var_{Sys} is a set of arbitrary system variables and Var_{PC} is a special set that holds for each P_j a program counter pc_j ranging over the binary control locations $Loc_j = \{00, 01, ...\}$ of P_j. Locations of a process are labelled with guarded commands over system variables and with a reference to the next location. The form of a guarded command is $assume(e) : v_1 := e_1,\ ...,\ v_m := e_m$ where $v_1, ..., v_m \in Var_{Sys}$ and $e, e_1, ... e_m$ are expressions over Var_{Sys}. The state space over Var corresponds to the set S_{Var} of all variable valuations. Given a $s \in S_{Var}$ and an expression e over Var, $s(e)$ denotes the valuation of e in s. An example system for mutual exclusion is:

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

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

We have two processes operating on a counting semaphore y. The semantics of the operations are: $acquire(y, 1) = assume(y > 0) : y := y - 1$ and $release(y, 1) = assume(true) : y := y + 1$. We assume that for any Sys a deterministic initialisation of Var is given by a predicate $Init$, e.g. $Init = (y = 1) \wedge (pc_1 = 0) \wedge (pc_2 = 0)$. A computation of Sys corresponds to a sequence of commands where in each step one process is nondeterministically selected and the command at its current location is attempted to be executed. If the execution is not blocked by a guard, the variables are updated according to the assignment part and the process advances to the

next location. A computation can be likewise considered as a state sequence $\pi = s_0 s_1 s_2 \ldots$ where the transition from s_i to s_{i+1} correctly characterises the execution of the associated command. A computation of our example system is: $\pi = \langle y = 1, pc_1 = 0, pc_2 = 0\rangle\langle y = 0, pc_1 = 1, pc_2 = 0\rangle\langle y = 1, pc_1 = 0, pc_2 = 0\rangle \ldots$ Explicit-state verification constructs transition models that represent all possible computations of the analysed system. In our approach, we construct a propositional encoding that represents the computations *implicitly*. Before we introduce our encoding, we look at the temporal properties that we want to verify.

3 Checking Safety via k-Induction

Verification involves checking all possible computations of a system with regard to correctness requirements. Of particular interest are safety properties, which require that all states reached in a computation satisfy some predicate $Safe$. For our example system *mutual exclusion* is a safety property that corresponds to $Safe = \neg(pc_1 = 1) \vee \neg(pc_2 = 1)$. It requires that not both processes are at their critical location 1 at the same time. Verification means to prove or refute that for all computations starting in an initial state $Safe$ always holds, or formally:

$$[Sys, Init \models_\forall \text{always } Safe] \quad := \quad \forall \pi = (s_0 s_1 s_2 \ldots) : s_0(Init) \rightarrow \textstyle\bigwedge_{i=0}^{\infty} s_i(Safe)$$

A method to address such verification problems is *k-induction* [16]: Let Sys be a system with computations in terms of state sequences and let $k \in \mathbb{N}$. In the *base case* it is checked if for all computations starting in an initial state the first k states are $Safe$. In the *induction step* it is checked if, assuming a computation consisting of a sequence of k *Safe* states, also any successor state is $Safe$. In contrast to the base case, the step does not contain a constraint on the initial state. This is necessary for the soundness of k-induction. These universal problems, referring to the safety of all computations, can be transformed into complementary existential problems referring to the existence of unsafe computations; as we can see, only the base case contains the initial state constraint $s_0(Init)$:

$$\begin{aligned} [Sys, Init \models_\exists Base]_k &:= \exists \pi = (s_0 \ldots s_k) : s_0(Init) \wedge \textstyle\bigvee_{i=0}^{k} s_i(\neg Safe) \\ [Sys, true \models_\exists Step]_{k+1} &:= \exists \pi = (s_0 \ldots s_{k+1}) : \textstyle\bigwedge_{i=0}^{k} s_i(Safe) \wedge s_{k+1}(\neg Safe) \end{aligned}$$

Hence, proving the universal problems is equivalent to disproving the existential ones. The latter can be efficiently done via SAT or SMT solving. k-induction is typically performed incrementally with regard to k. Thus, when checking the base case for some k we can assume that all shorter base cases have already been proven to be safe, and we can add these facts as constraints to the problem to be solved. Furthermore, in order to make k-induction complete, i.e. terminating for finite-state systems, it is necessary to restrict the inductive step to *loop-free* computations [16]. This gives us a slightly revised base case and step, for simplicity we abbreviate the verification problems by just $[Base]_k$ and $[Step]_{k+1}$:

$$\begin{aligned} [Base]_k &:= \exists \pi = (s_0 \ldots s_k) : s_0(Init) \wedge \textstyle\bigwedge_{i=0}^{k-1} s_i(Safe) \wedge s_k(\neg Safe) \\ [Step]_{k+1} &:= \exists \pi = (s_0 \ldots s_{k+1}) : \pi(LoopFree) \wedge \textstyle\bigwedge_{i=0}^{k} s_i(Safe) \wedge s_{k+1}(\neg Safe) \end{aligned}$$

where $\pi(LoopFree) = \bigwedge_{0 \leq i < j \leq k+1}(s_i \neq s_j)$ assuming that $\pi = (s_0, \ldots, s_{k+1})$. k-induction is still applicable to systems with loop computations in terms of recurring states, but for checking safety it is sufficient to only consider loop-free ones. The procedure below illustrates the principle of incremental k-induction:

```
1 for k = 0 to ∞ do
2 |   if ([Base]_k holds) then
3 |   |   return "safety property fails"
4 |   if ([Step]_{k+1} does not hold) then
5 |   |   return "safety property holds"
```

k-induction allows to reduce an unbounded verification problem to two bounded ones: the base case and the inductive step. Base case and step can be formulated as bounded model checking problems. Bounded model checking requires a state transition model of the system to be analysed. For hardware, such models can be straightly encoded in propositional logic and verification can be done via SAT solving. k-induction-based hardware verification has been applied in [7,16]. For software it is significantly harder to capture its complex features in propositional logic. Therefore, most k-induction approaches to software verification use an SMT solver [6,10]: The input system is transformed into a k-bounded one, where k typically refers to the number of unrollings of loops. The bounded system is then fed into an SMT solver to check the base case and the step of the verification problem. In our new approach, we use a combination of SMT and SAT: Via SMT solving we generate a three-valued abstraction of the system. Due to the reduced complexity the abstract system can be straightforwardly encoded in propositional logic and verification can be efficiently done via SAT solving. Next, we give a brief introduction to three-valued abstraction.

4 Three-Valued Predicate Abstraction and Refinement

To make SAT-based k-induction applicable to software verification, we follow the abstraction refinement paradigm: We employ SMT-based *three-valued predicate abstraction* [15] to our concrete systems, which yields abstract systems over predicates that can take the values *true*, *false* and *unknown*. *Unknown* represents loss of details due to abstraction. Three-valued abstraction generates an approximation in the sense that all definite verification results (*true*, *false*) obtained for an abstract system can be transferred to the concrete system. Only *unknown* results necessitate refinement for which we developed an automatic procedure [17]. Later we show that for an abstract system and a safety property any base case or step of k-induction can be reduced to two Boolean SAT problems. We now briefly outline three-valued abstraction. Details can be found in [15]. Our approach is based on the Kleene logic $\mathcal{K}_3$ [8] where *unknown*, abbreviated by u, is a used as a third truth value. In abstract systems guarded commands do not refer to concrete variables but to abstract predicates A_{Sys} over Var_{Sys}. Predicates in A_{Sys} may be set to u due to the execution of an abstract command. While our

three-valued abstraction reduces the complexity induced by system variables, it preserves the original control flow. For this, we use the set of two-valued predicates $A_{PC} = \{(pc_j = b_m \dots b_0) \mid pc_j \in Var_{PC},\ b_m \dots b_0 \in Loc_j\}$ that covers all possible locations of the system. The overall predicate set is $A = A_{Sys} \cup A_{PC}$. Given a concrete system Sys over Var, an initialisation $Init$ and a property ψ, we refer to the corresponding concrete verification task by ${}_{Var}[Sys, Init \models_Q \psi]$ with $Q \in \{\forall, \exists\}$. Additionally given a predicate set A over Var we refer to the abstract verification task by ${}_{A}[Sys, Init \models_Q \psi]$. From [15] we get the theorem:

Theorem 1 (Property Preservation under Three-Valued Abstraction). *Let Sys, Var, $Init$, A and Q as above. Moreover, let ψ be a property that is expressible in linear temporal logic (LTL) and let $z \in \{true, false\}$. Then:*

$$ {}_{A}[Sys, Init \models_Q \psi] = z \ \Rightarrow\ {}_{Var}[Sys, Init \models_Q \psi] = z $$

Since the properties in our k-induction approach are expressible in LTL, we can make use of Theorem 1. Definite results under abstraction can be transferred to the concrete system. For *unknown* results we have our refinement technique [17] that yields an extended predicate set $A^{r+1} = A^r \cup \{p | p \text{ predicate over } Var, p \notin A^r\}$ where $r = 0, 1, \dots$ denotes the current refinement iteration. We get:

Corollary 1. *Let Sys, $Init$, ψ, A^r, A^{r+1}, Q and z as above. Then:*

$$ {}_{A^r}[Sys, Init \models_Q \psi] = z \ \Rightarrow\ {}_{A^{r+1}}[Sys, Init \models_Q \psi] = z $$

Thus, definite properties are also preserved under abstraction refinement.

5 Three-Valued Bounded Model Checking

So far, we have defined verification tasks and corresponding abstractions. To practically perform verification, we need a computational model. Abstract state spaces can be defined as three-valued Kripke structures and safety properties can be formalised in temporal logic. On this basis, verification tasks can be expressed as three-valued bounded model checking (3BMC) problems.

Definition 1 (Three-Valued Kripke Structure). *A* three-valued Kripke structure *over a set of atomic predicates A is a tuple $M = (S, S_0, R, L)$ where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $R : S \times S \to \{true, u, false\}$ is a transition function, and $L : S \times A \to \{true, u, false\}$ is a labelling function.*

We assume that Kripke structures are *complementary-closed*, i.e. for each $p \in A$ there is a complementary $\overline{p} \in A$ such that $\forall s \in S : L(s, p) = \neg L(s, \overline{p})$. A path π is a sequence of states $s_0 s_1 s_2 \dots$ with $\forall i : R(s_i, s_{i+1}) \in \{true, u\}$. $\pi(i)$ denotes the i-th state of π. By Π_M we denote the set of all paths of M starting in an initial state. Paths are considered for the evaluation of temporal properties. Here we use the bounded temporal logic (BTL) which is a fragment of LTL.

Definition 2 (Syntax of Bounded Temporal Logic (BTL)). *Let A be a predicate set and $k \in \mathbb{N}$ a bound. The set of BTL formulas BTL over A and k is*

- *if $p \in A$ and $i \in [0,k]$ then $p_i \in BTL$ and $\neg p_i \in BTL$,*
- *if $\psi \in BTL$ then $\neg\psi \in BTL$,*
- *if $\psi \in BTL$ and $\psi' \in BTL$ then $\psi \vee \psi' \in BTL$ and $\psi \wedge \psi' \in BTL$.*

Definition 3 (Three-Valued Evaluation of BTL). *Let $M = (S, S_0, R, L)$ be over A and let π be a path of M. Let $k \in \mathbb{N}$ and $i \in [0,k]$. Then the evaluation of a k-bounded BTL formula ψ on π, written $[\pi \models \psi]_k$, is inductively defined as:*

$$
\begin{array}{lcll}
[\pi \models p_i]_k & = & L(\pi(i), p) \wedge \bigwedge_{j=0}^{i-1} R(\pi(j), \pi(j+1)) & \\
[\pi \models \neg p_i]_k & = & L(\pi(i), \overline{p}) \wedge \bigwedge_{j=0}^{i-1} R(\pi(j), \pi(j+1)) & \\
[\pi \models \psi \circ \psi']_k & = & [\pi \models \psi]_k \circ [\pi \models \psi']_k & \text{with } \circ \in \{\wedge, \vee\} \\
[\pi \models \neg(\psi \circ \psi')]_k & = & [\pi \models \neg\psi]_k \;\overline{\circ}\; [\pi \models \neg\psi']_k & \text{with } \overline{\wedge} = \vee \text{ and } \overline{\vee} = \wedge
\end{array}
$$

The universal evaluation of a BTL formula ψ on M over A is ${}_A[M, S_0 \models_\forall \psi]_k = \bigwedge_{\pi \in \Pi_M} [\pi \models \psi]_k$. The existential one is ${}_A[M, S_0 \models_\exists \psi]_k = \bigvee_{\pi \in \Pi_M} [\pi \models \psi]_k$.

Checking BTL properties of three-valued Kripke structures is *three-valued bounded model checking* [18] with the possible outcomes *true*, *false*, *u*. The state space of an abstracted system Sys can be modelled as a Kripke structure M such that there is a one-to-one correspondence between the states of Sys and M, and the transitions of M correspond to the execution of guarded commands of Sys. Hence, paths of M represent computations of Sys. From [15] we get the theorem:

Theorem 2. *Let Sys be abstracted over A and let M be the three-valued Kripke structure representing the abstract state space of Sys. Moreover, let $Q \in \{\forall, \exists\}$. Then for all linear temporal logic properties ψ the following holds:*

$$ {}_A[Sys, Init \models_Q \psi]_k \;\equiv\; {}_A[M, S_0 \models_Q \psi]_k $$

Thus, verification is equivalent to solving the corresponding 3BMC problem. This is an important fact since 3BMC can be reduced to propositional satisfiability and thus effectively performed via SAT solving. We now define the base case and the step of k-induction-based verification as 3BMC problems. For our example with $Safe_i = \neg(pc_1 = 1)_i \vee \neg(pc_2 = 1)_i$ and $(pc_1 = 1), (pc_2 = 1) \in A$ we get

$$
\begin{array}{lcl}
{}_A[Base]_k & \equiv & {}_A[M, S_0 \models_\exists \bigwedge_{i=0}^{k-1} Safe_i \wedge \neg Safe_k]_k \\
{}_A[Step]_{k+1} & \equiv & {}_A[M, S \models_\exists \bigwedge_{i=0}^{k} Safe_i \wedge \neg Safe_{k+1} \wedge LoopFree(0..k+1)]_{k+1}
\end{array}
$$

with $LoopFree(0..k+1) \;=\; \bigwedge_{0 \le i < j \le k+1} \left(\bigvee_{p \in A} \left((p_i \wedge \neg p_j) \vee (\neg p_i \wedge p_j) \right) \right)$. The *loop-free* property expresses that all states along a prefix are pairwise different. Note that in the 3BMC problem representing the inductive step the set of initial states is simply S, i.e. an arbitrary state can be the initial state. Next, we take a look on how these 3BMC problems can be encoded in propositional logic.

6 Propositional Logic Encoding

In [18] we showed how a 3BMC problem ${}_A[M, S_0 \models_\exists \psi]_k$ corresponding to a system Sys abstracted over A and a property ψ can be encoded as a propositional formula ${}_A[\![M, \psi]\!]_k$. The encoding can be directly constructed based on the abstract system. It corresponds to an implicit representation of the model checking problem such that the construction and exploration of an explicit state transition model is avoided. ${}_A[\![M, \psi]\!]_k$ is defined over a set of Boolean atoms $Atoms$, the constants $true$, $false$, and a special atom $\bot$ that we use to represent the *unknowns* due to abstraction. $\bot$ occurs solely non-negated in ${}_A[\![M, \psi]\!]_k$. 3BMC can now be performed via two SAT checks. One check considers an *over-approximating completion*, marked with '+', where all $\bot$'s are assumed to be $true$:

$$ {}_A[\![M, \psi]\!]_k^+ \;:=\; {}_A[\![M, \psi]\!]_k[\bot \mapsto true] $$

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

$$ {}_A[\![M, \psi]\!]_k^- \;:=\; {}_A[\![M, \psi]\!]_k[\bot \mapsto false]. $$

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

Definition 4 *(sat_3). Let ${}_A[\![M, \psi]\!]_k$ over Atoms be the propositional encoding of ${}_A[M, S_0 \models_\exists \psi]_k$, let $\{\mathcal{A}\mathcal{A} : Atoms \to \{true, false\}\}$ be the set of all possible truth assignments to the atoms in Atoms. Then sat_3 is defined as:*

$$ sat_3({}_A[\![M, \psi]\!]_k) \;=\; \begin{cases} false & \text{if } \forall \mathcal{A} : \mathcal{A}({}_A[\![M, \psi]\!]_k^+) = false \\ true & \text{if } \exists \mathcal{A} : \mathcal{A}({}_A[\![M, \psi]\!]_k^-) = true \\ unknown & \text{else} \end{cases} $$

In [18] the following theorem has been proven:

Theorem 3. *Let ${}_A[\![M, \psi]\!]_k$ and ${}_A[M, S_0 \models_\exists \psi]_k$ be as above. Then:*

$$ sat_3({}_A[\![M, \psi]\!]_k) \;=\; {}_A[M, S_0 \models_\exists \psi]_k $$

Hence, the sat_3 result obtained for the encoding corresponds to the model checking result. We now briefly explain how the translation into Boolean satisfiability works. The details can be found in [18]. Remember that paths of Kripke structures as well as BTL properties correspond to expressions over $A = A_{Sys} \cup A_{PC}$ indexed with $i \in [0, k]$ where i denotes a position along a k-prefix. Thus, our encoding is inductively defined over indexed expressions. Predicates in A_{Sys} have a *three-valued* domain, whereas the encoding is *two-valued* and we use the special atom $\bot$ to represent the '*unknown*'. In order to reduce a three-valued problem to a two-valued one, we use two Boolean atoms for each $p \in A_{Sys}$ and $i \in [0, k]$:

$$ Atoms_{Sys} \;:=\; \{p[u]_i, p[b]_i \mid p \in A_{Sys},\ i \in [0, k]\} $$

Atom $p[u]_i$ will let us indicate whether p evaluates to *unknown* or to a definite value at position i, and $p[b]_i$ will let us indicate whether p evaluates to *true* or *false*. For encoding counter predicates $(pc_j = l_m \dots l_0) \in A_{PC}$ we use the set

$$Atoms_{PC} \;:=\; \{l_j[m]_i, \dots, l_j[0]_i \mid (pc_j = l_m \dots l_0) \in A_{PC},\; i \in [0,k]\}$$

Since a program counter location $l_m \dots l_0$ is a binary number, it can be straightforwardly encoded as a conjunction of literals over $l_j[m]_i, \dots, l_j[0]_i$. The overall set of atoms of our encoding is $Atoms = Atoms_{Sys} \cup Atoms_{PC}$. Now we can define the Boolean encoding of arbitrary indexed expressions over A and $[0,k]$:

Definition 5 (Encoding of Logical Expressions). *Let $A = A_{Sys} \cup A_{PC}$ be a predicate set with $p \in A_{Sys}$ and $(pc_j = l_m \dots l_0) \in A_{PC}$. Let $k \in \mathbb{N}$. The encoding of expressions e over A indexed over $[0,k]$, written $[\![e]\!]_k$, is defined as:*

$$\begin{array}{lcl}
[\![p_i = unknown]\!]_k & := & p[u]_i \\
[\![p_i = true]\!]_k & := & \neg p[u]_i \wedge p[b]_i \\
[\![p_i = false]\!]_k & := & \neg p[u]_i \wedge \neg p[b]_i \\
[\![p_i]\!]_k & := & [\![p_i = true]\!]_k \vee ([\![p_i = unknown]\!]_k \wedge \bot) \\
[\![\neg p_i]\!]_k & := & [\![p_i = false]\!]_k \vee ([\![p_i = unknown]\!]_k \wedge \bot) \\
[\![(pc_j = l_m \dots l_0)_i]\!]_k & := & \bigwedge_{d=0}^{m} (\mathit{if}\ l_d = 1\ \mathit{then}\ l_j[d]_i\ \mathit{else}\ \neg l_j[d]_i) \\
[\![\neg(pc_j = l_m \dots l_0)_i]\!]_k & := & \neg [\![(pc_j = l_m \dots l_0)_i]\!]_k
\end{array}$$

The encoding of $e \vee e'$, $e \wedge e'$, $\neg(e \vee e')$ and $\neg(e \wedge e')$ is trivial and thus omitted.

We can build the formula ${}_A[\![M, \psi]\!]_k = {}_A[\![M]\!]_k \wedge {}_A[\![\psi]\!]_k$ over $Atoms$ where ${}_A[\![M]\!]_k$ encodes all k-bounded paths of M and ${}_A[\![\psi]\!]_k$ constrains paths to those satisfying ψ. E.g., the property $\bigwedge_{i=0}^{k} Safe_i$ with $Safe_i = \neg(pc_1 = 1)_i \vee \neg(pc_2 = 1)_i$ gets encoded to $\bigwedge_{i=0}^{k}(\neg l_1[0]_i \vee \neg l_2[0]_i)$. Each assignment $\mathcal{A}$ that satisfies ${}_A[\![M, \psi]\!]_k^-$ characterises a path π in M with $[\pi \models \psi] = true$. If there is no such assignment for ${}_A[\![M, \psi]\!]_k^+$ then $\forall \pi$ we have $[\pi \models \psi] = false$. This reduces 3BMC to SAT.

7 Iterative Refinement with Constraint Reusing

Our SAT-based verification technique combines three-valued abstraction with *iterative refinement*. Given a system Sys over Var, a k-bounded property ψ and a predicate set A^r, we construct the encoding ${}_{A^r}[\![M, \psi]\!]_k$ of the corresponding three-valued bounded model checking problem ${}_{A^r}[M \models_E \psi]_k$, where $r = 0, 1, \dots$ denotes the current refinement iteration. In this section, we introduce the concept of *constraint reusing* between refinement iterations. Algorithm $SATBMC$ illustrates our refinement approach and gives a first idea of constraint reusing: $SATBMC$ gets a model checking problem and a predicate set A^h as an input, where $h \in \mathbb{N}$ denotes the refinement level to start with. h is typically 0 but may be also greater when we combine iterative refinement with *bound incrementation* (Sect. 8). Unsatisfiability of the over-approximating completion and satisfiability of the under-approximating one let us immediately derive a corresponding definite model checking result. If this is not possible in iteration r, we apply

Algorithm 1. *SATBMC*$([M, S_0 \models_E \psi]_k, A^h)$

```
definite constraint set ℂ := ∅
for r = h to ∞ do
    if _{A^r}⟦M, ψ⟧_k^+ ∧ ℂ unsatisfiable then
        return _{A^r}[M, S_0 ⊨_E ψ]_k = false
    if _{A^r}⟦M, ψ⟧_k^- ∧ ℂ satisfiable then
        return _{A^r}[M, S_0 ⊨_E ψ]_k = true
    else
        A^{r+1} := A^r ∪ {p | p ∉ A^r}
        add definite constraints learned in current iteration to ℂ
```

our counterexample-guided refinement [17] which yields an extended predicate set $A^{r+1} := A^r \cup \{p | p \text{ predicate over } Var, p \notin A^r\}$ and a corresponding refined encoding ${}_{A^{r+1}}[\![M, \psi]\!]_k$. We then run the necessary SAT tests and repeat these steps until a definite result is obtained. As we can see, the encodings in our algorithm are conjuncted with a constraint set $\mathbb{C}$. A constraint is a clause over the atoms of the encoding that has been inferred by the solver via clause learning [2]. $\mathbb{C}$ is extended with newly learned constraints in each iteration. Thus, constraints learned in the past are reused in future iterations. The motivation for constraint reusing is that adding (valid) constraints to a formula reduces the search space of the corresponding SAT problem, which can improve the solving time.

However, reusing constraints between refinement iterations is not straightforward. A clause learned in iteration r is not necessarily a valid constraint in $r + 1$. The formulas ${}_{A^r}[\![M, \psi]\!]_k$ and ${}_{A^{r+1}}[\![M, \psi]\!]_k$ evidently share a common set of atoms over which they are defined, but their structure is typically completely different: The addition of new predicates by refinement can involve extensive changes of the abstract state space and its encoding. Thus, ${}_{A^{r+1}}[\![M, \psi]\!]_k$ cannot be obtained from ${}_{A^r}[\![M, \psi]\!]_k$ by simply adding more clauses. This makes our refinement generally incompatible with standard *incremental* SAT solving [13] where learned constraints can be reused between consecutive SAT instances without any restriction. Our novel constraint reusing concept for iterative refinement is based on a check of whether a learned constraint is *definite* in terms of the encoded three-valued model checking problem. A definite constraint characterises a temporal property that definitely holds at the current refinement level. Since definite properties are preserved under refinement (Corollary 1), we can prove that definite constraints are also valid at any higher refinement level.

We start with a few basics. A learned constraint is a clause C that is syntactically inferred from a formula F by the solver: $F \vdash C$. The following holds: $(F \vdash C) \Rightarrow (F \models C)$, i.e. a syntactic consequence is also a semantic one. If $F \models C$ holds, we say C is a *valid constraint* of F. We implemented over- and under-approximating completions of $F = {}_{A^r}[\![M, \psi]\!]_k$ as assumptions over $\perp$. This has the effect that all learned constraints are *assumption-independent* [7], i.e. they are logical consequences irrespective of the value assigned to $\perp$. Hence, we say C has been learned for F if it has been learned for F^+ or for F^-. Since

F^+ and F^- only differ in the assumption over $\bot$, we get $(F^+ \vdash C) \Rightarrow (F^- \models C)$. Thus, all C learned for F^+ can be reused when solving F^- in the *same* iteration.

We now describe how constraints can be reused *between* refinement iterations $r < r'$. This is feasible for constraints that characterise temporal properties that definitely hold in iteration r. In order to identify such *definite constraints* we use an enhancement of the encoding ${}_{A^r}[\![M, \psi]\!]_k$ based on the *Tseytin transformation* (TT) [19] where sub-formulas are represented by auxiliary atoms:

$$Atoms_{Aux} \;=\; \{p[t]_i, p[f]_i \mid p \in A^r_{Sys},\ i \in [0,k]\}$$

The enhanced encoding revises Definition 5 in two cases only:

Definition 6 (Enhanced Encoding). *Let $p \in A^r_{Sys}$ and $i \in [0,k]$. Then:*

$$[\![p_i = true]\!]_k \;:=\; p[t]_i \qquad\qquad [\![p_i = false]\!]_k \;:=\; p[f]_i$$

Hence, definite information with regard to a predicate p at position i can now be derived from a *single literal* (e.g. $p[t]_i$), rather than from a *conjunction of literals* (e.g. $\neg p[u]_i \wedge p[b]_i$) as in the original encoding. Note that a constraint is always a *disjunction* of single literals. Thus, only with our enhanced encoding a learned constraint may tell us something definite about a predicate p. We still need to put $p[t]_i$, $p[f]_i$ and $p[u]_i$ into a relation to correctly encode that a three-valued predicate can only hold one truth value at a time. According to TT, we conjunct the overall encoding with the following equivalences:

$$\bigwedge\nolimits_{p \in A^r_{Sys}} \bigwedge\nolimits_{i=0}^{k} \left((p[t]_i \leftrightarrow \neg p[u]_i \wedge p[b]_i) \wedge (p[f]_i \leftrightarrow \neg p[u]_i \wedge \neg p[b]_i) \right)$$

Thus, the single auxiliary atoms represent sub-formulas that indicate definite information with regard to predicates. Let ${}_{A^r}[\![M, \psi]\!]_k$ over *Atoms* be the resulting enhanced encoding. We now define the set of *definite literals* DL over *Atoms* as:

$$\begin{aligned} DL \;=\; & \{p[t]_i, p[f]_i \mid p \in A^r_{Sys},\ i \in [0,k]\} \\ & \cup \{l_j[d]_i, \neg l_j[d]_i \mid (pc_j = l_m \ldots l_0) \in A^r_{PC},\ i \in [0,k],\ d \in [0,m]\} \end{aligned}$$

DL contains all auxiliary atoms, and all program counter atoms and their negations. We denote constraints that are purely composed of literals from DL as *definite constraints*. BTL formulas corresponding to definite constraints are:

Definition 7 (BTL Formulas Corresponding to Definite Constraints). *Let ${}_{A^r}[\![M, \psi]\!]_k$ be the enhanced encoding of ${}_{A^r}[M, S_0 \models_\exists \psi]_k$. Moreover, let $C = c_1 \vee \ldots \vee c_n$ over DL be a definite constraint learned for ${}_{A^r}[\![M, \psi]\!]_k$. Then the BTL formula corresponding to C, written $btl(C)$, is inductively defined as:*

$$\begin{aligned} btl(p[t]_i) \;&:=\; p_i \\ btl(p[f]_i) \;&:=\; \neg p_i \\ btl(l_j[d]_i) \;&:=\; \textstyle\bigvee_{(l_m \ldots l_0) \in Loc_j, l_d = 1} (pc_j = l_m \ldots l_0)_i \\ btl(\neg l_j[d]_i) \;&:=\; \textstyle\bigvee_{(l_m \ldots l_0) \in Loc_j, l_d = 0} (pc_j = l_m \ldots l_0)_i \\ btl(c_1 \vee \ldots \vee c_n) \;&:=\; btl(c_1) \vee \ldots \vee btl(c_n) \end{aligned}$$

We get the following lemma wrt. constraints and corresponding BTL formulas:

Lemma 1. *Let* ${}_{A^r}[\![M, \psi]\!]_k$ *be the enhanced encoding of* ${}_{A^r}[M, S_0 \models_\exists \psi]_k$*. Moreover, let* C *over* DL *be a definite constraint. Then*

$$ {}_{A^r}[\![M, \psi]\!]_k \vdash C \;\Rightarrow\; {}_{A^r}[M, S_0 \models_\forall (\psi \rightarrow btl(C))]_k = true $$

Proof. See http://github.com/ssfm-up/TVMC/raw/unbounded/proofs.pdf.

Hence, even if the current refinement level is too coarse to prove the actual property of interest ψ, a learned definite constraint C tells us that all paths satisfying ψ must also satisfy $btl(C)$. This is a definite result of a three-valued model checking problem with refinement level r. Corollary 1 allows us to transfer this result to all refinement levels $r' > r$: ${}_{A^r}[M, S_0 \models_\forall (\psi \rightarrow btl(C))]_k = true \;\Rightarrow\; {}_{A^{r'}}[M, S_0 \models_\forall (\psi \rightarrow btl(C))]_k = true$. Next, we show that a constraint C associated with a definite property $\psi \rightarrow btl(C)$ is also valid at higher levels.

Lemma 2. *Let* ${}_{A^r}[\![M, \psi]\!]_k$ *be the encoding of* ${}_{A^r}[M, S_0 \models_\exists \psi]_k$ *and let* C *be a definite constraint. Then*

$$ {}_{A^r}[M, S_0 \models_\forall (\psi \rightarrow btl(C))]_k = true \quad \Rightarrow \quad {}_{A^r}[\![M, \psi]\!]_k \models C $$

Proof. See http://github.com/ssfm-up/TVMC/raw/unbounded/proofs.pdf.

We get the following Corollary that establishes the reusability of definite constraints between refinement iterations.

Corollary 2 (Reusability of Definite Constraints). *Let* ${}_{A^r}[\![M, \psi]\!]_k$ *be the encoding of* ${}_{A^r}[M, S_0 \models_\exists \psi]_k$*. Let* C *be a definite constraint and* $r' > r$*. Then*

$$ \begin{array}{ccc} {}_{A^r}[\![M, \psi]\!]_k \vdash C & & {}_{A^{r'}}[\![M, \psi]\!]_k \models C \\ \Downarrow \textit{ Lemma 1} & & \Uparrow \textit{ Lemma 2} \\ {}_{A^r}[M, S_0 \models_\forall (\psi \rightarrow btl(C))]_k = true & \overset{\textit{Cor. 1}}{\underset{\textit{Thm. 2}}{\Longrightarrow}} & {}_{A^{r'}}[M, S_0 \models_\forall (\psi \rightarrow btl(C))]_k = true \end{array} $$

Hence, a definite constraint C learned in iteration r implies that $\psi \rightarrow btl(C)$ universally holds at r and any higher refinement level r' as well. Consequently, C must be also a valid constraint of the encoding in all iterations $r' > r$. We utilise this by determining definite constraints in each iteration and adding them to the set $\mathbb{C}$ that we use as a constraint set of the SAT problems to be solved in $SATBMC$. Our concept is based on TT. Thus, it does not lead to an increased complexity of the SAT problem. In fact, we also use TT to transform the overall encoding into conjunctive normal form, which introduces further auxiliary atoms. This does not affect our constraint reusing since we use the same auxiliary atoms for representing sub-formulas recurring in multiple refinement iterations. After TT, all clauses purely containing definite literals and auxiliary atoms referring to definite constraints can be reused between refinement iterations.

We illustrate our constraint reusing concept based on our running example. Let ${}_{A^0}[\![M, Safe_0 \wedge Safe_1]\!]_1$ with $A^0 = \{(pc_1 = 0), (pc_1 = 1), (pc_2 = 0), (pc_2 = 1)\}$ and $Safe_i = \neg(pc_1 = 1)_i \vee (pc_2 = 1)_i$ be the encoding of the base case of checking safety of the mutual exclusion system. The atom set used for the encoding is $Atoms = \{l_1[0]_0, l_2[0]_0, l_1[0]_1, l_2[0]_1, \}$. SAT solving yields *unknown*, since no information about the semaphore is considered at the current abstraction level. However, the solver infers the following constraint clauses $(\neg l_1[0]_1 \vee \neg l_2[0]_1)$ and $(l_1[0]_1 \vee l_2[0]_1)$ with the corresponding temporal logic formulas:

$$\big((pc_1{=}0)_1 \vee (pc_2{=}0)_1\big) \text{ and } \big((pc_1{=}1)_1 \vee (pc_2{=}1)_1\big)$$

Hence, at position 1 of any execution path either only process 1 or only process 2 is at its critical location. According to our corollary, the constraint clauses that characterise this property can be reused in all future refinement iterations for pruning the search space of SAT. In the next iteration, we add the predicate $p := (y{=}1)$ and SAT solving infers another reusable constraint clause $(p[f]_1)$. It tells us that at position 1 of any path the semaphore will be occupied. Next, we show how we integrated $SATBMC$ into an incremental k-induction procedure.

8 Implementation

We implemented an automatic verification tool for concurrent software systems[1]. Our tool extends our existing SAT-based bounded model checking framework [17] by integrating the k-induction principle with base case and inductive step, which makes our formerly incomplete approach complete. It employs three-valued abstraction in order to reduce the complexity of the state space encodings. Abstraction is combined with iterative refinement. As an input we take a system Sys in a C-like syntax with *int*, *bool* and *semaphore* as data types, an initial state predicate $Init$ and a safety predicate $Safe$. To verify whether $[Sys, Init \models_\forall \text{ always } Safe]$ holds, we determine the abstract model checking problems corresponding to the base case and step of k-induction

$$[Base]_k = [M, S_0 \models_\exists \textstyle\bigwedge_{i=0}^{k-1} Safe_i \wedge \neg Safe_k]_k \text{ over } A^{r_B}$$
$$[Step]_{k+1} = [M, S \models_\exists \textstyle\bigwedge_{i=0}^{k} Safe_i \wedge \neg Safe_{k+1} \wedge LoopFree(0..k{+}1)]_{k+1} \text{ over } A^{r_S}$$

where A^{r_B} denotes the predicate set used for the three-valued abstraction of the base case and A^{r_S} the set used for the inductive step. The bound k and the predicate sets are so far only uninitialised parameters and instead of explicitly constructing the Kripke structure M and exploring its state space, we take the bounded model checking problems as the input of our *k-induction* algorithm:

The variables r_B and r_S indicate the current refinement iteration of the base case and of the inductive step. Both are initialised with 0. The corresponding sets of abstraction predicates A^{r_B} and A^{r_S} are also initialised by the *k-induction*

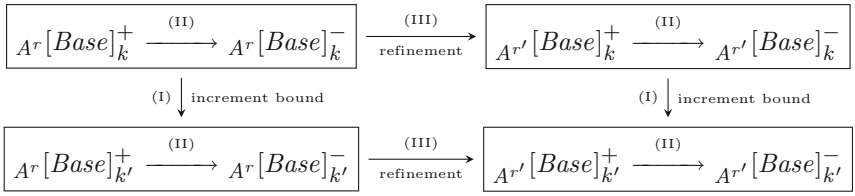
[1] Available at www.github.com/ssfm-up/TVMC/tree/unbounded.

Algorithm 2. $k\text{-}induction([Base]_k, [Step]_{k+1})$

```
1 r_B := 0,  A^{r_B} := initialiseA^{r_B}()
2 r_S := 0,  A^{r_S} := initialiseA^{r_S}()
3 for k = 0 to ∞ do
4 |   if (SATBMC([Base]_k, A^{r_B}) = true) then
5 |   |   return "safety property fails"
6 |   if (SATBMC([Step]_{k+1}, A^{r_S}) = false) then
7 |   |   return "safety property holds"
```

algorithm. Initially, they contain all control flow predicates and potentially further predicates over system variables that are referenced in the property to be checked. After the initialisation, *k-induction* iterates over the bound. In each k-iteration $SATBMC$ is called for the bounded model checking problems associated with the base case and with the step. Within $SATBMC$ we have a further iteration: The set of abstraction predicates is iteratively extended via refinement. Each refinement iteration consists of the propositional encoding of the three-valued model checking problem for the current predicate set and the execution of the corresponding SAT checks. $SATBMC$ terminates once a refinement level is reached where a definite model checking result can be obtained. k-induction terminates when it can be either proven or refuted that safety holds for the system. Termination is guaranteed for finite-state systems.

As another new feature, our tool supports constraint reusing on three levels. (I): Constraints are reused between bound iterations $k < k'$ based on incremental SAT with assumption literals [13]. (II): Similarly, we reuse assumption-independent constraints between the over- and the under-approximating completion in each refinement iteration r. (III): Finally, we reuse definite constraints between refinement iterations $r < r'$ based on the results from Sect. 7. The diagram below illustrates the directions of constraint reusing for the base case:



9 Experimental Results

We experimentally investigated the impact of our novel clause reusing concepts (II) and (III) on the verification time. While detecting safety violations in faulty systems was generally very fast, we focused in our case study on proving safety of *correct* systems: We verified deadlock-freedom of a semaphore-based dining philosophers algorithm and we proved mutual exclusion of Dijkstra's mutex algorithm [5]. In each benchmark we considered systems with increasing

numbers of processes. A general optimisation that we used was checking the over-approximation of the base case always first and in case of a *false* result skipping the then redundant under-approximation check (see Definition 4 and Theorem 3). Analogously, we always checked the under-approximation of the step first and in case of a *true* result skipped the over-approximation check. The intuition behind this is that for safe systems we can always expect a sequence of iterations where the base case fails and the step holds until in the final iteration the step fails, which corresponds to a correctness result. The experimental results are depicted below.

Benchmark	Processes	Final bound	Refinements (base/step)	Time with (I) only	Time with (I), (II) and (III)
Philosophers	2	3	1/2	0.45 s	0.44 s
	3	6	2/3	0.75 s	0.66 s
	4	12	3/4	4.67 s	3.85 s
	5	24	4/5	91.1 s	68.5 s
Dijkstra	2	12	1/6	4.80 s	3.32 s
	3	16	1/9	31.87 s	22.21 s
	4	21	2/12	73 min	52 min
	5	25	2/15	244 min	158 min

The experiments were conducted on a 3.4 GHz Core i7 with 8 GB memory. As we can see, our two novel constraint reusing concepts (II) and (III) together could lead to noticeable performance improvements in comparison to only using the established inter-bound constraint reusing (I). The computational savings were more evident for Dijkstra's algorithm where the number of refinement iterations was generally higher, i.e. where there were more capabilities for inter-refinement clause reusing. When we investigated the individual speed-up effect of (II) and of (III), we observed that (II) had a stronger impact when the number of refinements was small, whereas for cases with many refinements the performance impact of the two concepts was nearly equally strong, e.g. for Dijkstra 4: savings of 12 min with (II) only, savings of 14 min with (III) only, and of 21 min both together. This also shows that there is an overlap of savings due to (II) and due to (III). The experiments also revealed that it is beneficial to abstract and refine the base case and the step *individually*. The base case could be always accomplished based on fewer refinements, i.e. a less complex encoding. This advantage would not come into effect with a joint abstraction refinement of base case and step.

10 Related Work

k-induction was first introduced in [16] as a technique for verifying hardware systems that correspond to finite-state transition models. It extends classical

bounded model checking from falsification to verification. It has been combined with incremental SAT, which allows to reuse learned constraints between bound iterations [7]. In comparison to our approach, the *bound* is the only dimension of incrementation. Since hardware is generally simpler than software, there is no concept for abstraction refinement used in the above-mentioned papers. Our software verification technique adds the *level of abstraction* as a second dimension of incrementation and we show that constraint reusing is also feasible between refinement iterations. Our major focus is the verification of safety properties of *concurrent systems*, e.g. mutual exclusion and deadlock-freedom. In the context of software verification, k-induction has been used for checking safety of *loop programs* [1,6,9,10,14]. In these papers, the bound k determines the number of loop unwindings. The unwound program is encoded into a formula that can be processed by an SMT solver. SMT generally allows for more compact encodings than SAT. While [1,6,9,10,14] directly operate on the *concrete program*, we follow the *abstraction refinement paradigm* [4]. SMT-based predicate abstraction [12] allows us to generate a propositional state transition encoding that is compact enough to be efficiently processed by a SAT solver. In particular, our abstraction approach enables to omit details along the explored paths that are not relevant for solving the verification problem. Missing but necessary details are iteratively added by refinement, where our constraint reusing concept alleviates the computational overhead of the iterative approach. In [1,6,9,10,14] the performance of verification is improved by inferring loop invariants that are added as assumptions to the program, which is a particular form of constraint using in the context of loop programs. While earlier works are based on manually specified invariants [6], recent approaches use automatic invariant generation [14] or refine invariants in each bound iteration [1]. Regarding background theories, k-induction approaches to the verification of loop programs range from integer arithmetic [9], real arithmetic and uninterpreted functions [1] to pointers [10].

11 Conclusion

We introduced a safety verification technique for concurrent software systems based on a combination of three-valued abstraction refinement and SAT-based k-induction. The approach extends our prior work on (incomplete) three-valued bounded model checking [17,18]. The main contributions of this paper are as follows: We showed that, after the application of abstraction, base case and inductive step of the k-induction technique can be formulated as bounded model checking problems and encoded in propositional logic, which facilitates *complete* verification. We integrated the k-induction approach into a twofold-iterative verification procedure that enables to reach the necessary bound and the right level of abstraction in order to prove or refute safety properties. We enhanced this iterative approach by adopting k-incremental SAT solving and by extending the idea of reusing logical constraints to two new levels: In our three-valued setting, constraints can be reused between over- and under-approximations and also between refinement iterations. The latter is a non-straightforward concept

that we proved to be sound. In experiments we demonstrated the effectiveness of our approach as formal method for software model checking and we showed that our novel constraint reusing concepts can lead to significant computational savings.

References

1. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_42
2. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Conflict-driven clause learning sat solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, pp. 131–153. IOS Press, Amsterdam (2009)
3. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_25
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
5. Dijkstra, E.W.: Solution of a problem in concurrent programming control. In: Broy, M., Denert, E. (eds.) Software Pioneers, pp. 347–350. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-59412-0_20
6. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_26
7. Een, N., Sörensson, N.: Temporal induction by incremental sat solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003). BMC 2003
8. Fitting, M.: Kleene's 3-valued logics. Fund. Inf. **20**(1–3), 113–131 (1994)
9. Franzén, A.: Using satisfiability modulo theories for inductive verification of lustre programs. ENTCS **144**(1), 19–33 (2006)
10. Gadelha, M., Ismail, H., Cordeiro, L.: Handling loops in bounded model checking of C programs via k-induction. STTT **19**(1), 97–114 (2017)
11. Günther, H., Weissenbacher, G.: Incremental bounded software model checking. In: SPIN 2014, pp. 40–47. ACM (2014)
12. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_39
13. Nadel, A., Ryvchin, V., Strichman, O.: Ultimately incremental SAT. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 206–218. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_16
14. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., Fischer, B.: DepthK: a k-induction verifier based on invariant inference for C programs. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 360–364. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_23
15. Schrieb, J., Wehrheim, H., Wonisch, D.: Three-valued spotlight abstractions. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 106–122. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_8

16. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
17. Timm, N., Gruner, S.: Three-valued bounded model checking with cause-guided abstraction refinement (2018, manuscript submitted for publication)
18. Timm, N., Gruner, S., Harvey, M.: A bounded model checker for three-valued abstractions of concurrent software systems. In: Ribeiro, L., Lecomte, T. (eds.) SBMF 2016. LNCS, vol. 10090, pp. 199–216. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49815-7_12
19. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning, pp. 466–483. Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28