Input-based Three-Valued Abstraction Refinement

Jan Onderka[1,2,3] and Stefan Ratschan[2]

[1] Faculty of Engineering, University of Freiburg
Freiburg, Germany
onderka@cs.uni-freiburg.de
[2] Institute of Computer Science, The Czech Academy of Sciences
Prague, Czech Republic
stefan.ratschan@cs.cas.cz
[3] Faculty of Information Technology, Czech Technical University in Prague
Prague, Czech Republic

Abstract Unlike Counterexample-Guided Abstraction Refinement (CEGAR), Three-Valued Abstraction Refinement (TVAR) is able to verify all properties of the μ-calculus. We present a novel algorithmic framework for TVAR that employs a simulator-like approach to build and refine the abstract state space with input-based splitting. This leads to a state space formalism that is much simpler than in previous TVAR frameworks, which use modal transitions. We implemented the framework in our open-source tool **machine-check** and verified properties of machine-code systems for the AVR architecture, showing the ability to verify systems and μ-calculus properties not verifiable by naïve model checking or CEGAR, respectively. This is the first practical use of TVAR for machine-code verification.

Keywords: Model checking · Abstraction · Partial Kripke Structure · μ-calculus

1 Introduction

Abstraction-refinement methodologies are ubiquitous in formal verification, the foremost being Counterexample-guided Abstraction Refinement (CEGAR) [8, 10]. Unfortunately, CEGAR does not support the whole propositional μ-calculus, and indeed not even Computation Tree Logic (CTL). This leaves a large class of potentially crucial non-linear-time properties unverifiable. Three-valued Abstraction Refinement (TVAR) is able to verify full μ-calculus. However, previous TVAR frameworks refined abstract states, necessitating state space formalisms based on modal transitions. Frameworks based on simple formalisms [19, 47] are not monotone: previously provable properties may no longer be provable after refinement. Intricate monotone formalisms [28, 48, 52] were devised, their specialised semantics complicating the use of standard model-checking algorithms.

Another problem common to model-checking with abstraction is that the abstract state space cannot be built using a system simulator, which is possible for naïve explicit-state model checking [45]. This is a problem especially with complex systems such as processors executing machine code.

Contribution. Combining the ideas of TVAR and system simulators, we present a novel TVAR framework based on simulator-like generation of the abstract state space and performing refinements by splitting abstract inputs. Our framework does not use modal transitions, leading to simpler and more intuitive reasoning compared to the previous frameworks. Furthermore, it allows simple building and refinement of the abstract state space based on abstract simulators. We prove that the introduced framework is sound, monotone, and complete for μ-calculus properties and existential abstraction domains, provided simple requirements are met. Arbitrary digital systems formalised as automata can be verified, and the choices of abstraction domains and refinement strategy can be tailored to the specific use-case. Unlike the previous abstract-simulator approaches, our framework can be used for the full μ-calculus.

We implemented an instance of our framework in our formal verification tool **machine-check**[4], where the systems are described as simulable finite-state machines in a subset of the Rust language, translated to abstract and refinement analogues used for generating and refining the state space [38]. The ability to refine removes the need for hints such as where to use abstraction [25].

We designed **machine-check** especially for machine-code verification, and were able to verify non-toy programs for the AVR architecture and find a bug in a simplified version of a real-life program using a non-linear-time property not verifiable by CEGAR. To our knowledge, this is the first use of TVAR and verification of μ-calculus properties for machine-code systems.

2 Previous Work

In this section, we list previous relevant work on TVAR in roughly chronological order, with additional information available in summarising papers [15, 18]. After that, we discuss relevant work on abstract simulator approaches to model checking.

Example 1. Consider the finite-state machine in Figure 1, representing e.g. a controller of aircraft landing gear retraction: if the most significant bit (*msb*) of the state is 0, the landing gear is extended; if 1, it is retracted. The system is required to follow a single-bit input from the gear lever, with some slack for responses. There is a critical bug, occurring if the aircraft loses power in flight when the landing gear is retracted and the controller restarts in the state 000 after regaining power: as the input is set to retraction (1), the controller proceeds through 011 to states where *msb* remains 1 forever, a **total loss of capability** to extend the gear again unless the controller is turned off and on again.

The bug is not just dangerous, but also sneaky, as it does not occur during a normal start with lever input 0. To protect ourselves against it, we can verify the property "from every reachable system state, it should be possible to reach a state where the landing gear is extended" holds in the system. This is formalised by a *recovery property* $\mathbf{AG}[\mathbf{EF}[\neg msb]]$ in CTL, not possible to check using CEGAR.

[4] Free and open-source, official website https://machine-check.org/. In this paper, we discuss version 0.6.1 published at https://crates.io/crates/machine-check/0.6.1.

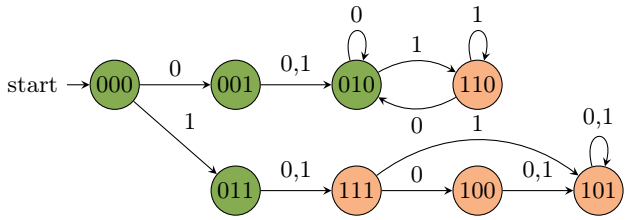


Figure 1. Example system expressed as a finite-state machine. The states where $\neg msb$ holds are drawn green while the states where msb holds are drawn orange.

Since the system is buggy, the property should be disproved[5]. For clarity, we will instead reason about proving a dual property, $\mathbf{EF}[\mathbf{AG}[msb]]$, i.e. from the start state, there exists a path (**E**) where from some state on the path (**F**), all paths (**A**) have every state (**G**) fulfilling msb. We will prove this on Figure 1 bottom-up as usual for CTL. Clearly, msb holds in the state 101. Since 101 just loops on itself, $\mathbf{AG}[msb]$ holds in it. As 101 is reachable from 000 by the path (000, 011, 111, 100, 101), we conclude $\mathbf{EF}[\mathbf{AG}[msb]]$ holds, finding the bug. While such reasoning is easy for simple systems, in real life, the controller may have billions of possible states, requiring us to abstract some information away[6].

Partial Kripke Structures (PKS). In verification on *partial state spaces* [1], some information is disregarded to produce a smaller state space. PKS enrich standard Kripke structures (KS) by allowing unknown state labellings.

Definition 1. *A partial Kripke structure (PKS) is a tuple* (S, S_0, R, L) *over a set of atomic propositions* $\mathbb{A}$ *with the elements*

- S *(the* set of states*),*
- $S_0 \subseteq S$ *(the* set of initial states*),*
- $R \subseteq S \times S$ *(the* transition relation*),*
- $L : S \times \mathbb{A} \to \{0, 1, \bot\}$ *indicating for each atomic proposition whether it holds, does not hold, or its truth value is unknown (the* labelling function*).*

A Kripke Structure (KS) is a PKS with L *restricted to* $S \times \mathbb{A} \to \{0, 1\}$.

Example 2. While Figure 1 shows a finite-state machine, it can be converted to a Kripke structure by discarding the inputs, with $S = \{000, 001, \ldots, 111\}$, $S_0 = \{000\}$, and R given by the transitions in Figure 1. L labels msb in states $\{000, 001, 010, 011\}$ as 0, and in $\{100, 101, 110, 111\}$ as 1.

For proving $\mathbf{EF}[\mathbf{AG}[msb]]$, such a KS is unnecessarily detailed. Using PKS, we could e.g. combine 010 and 110 into a single *abstract* state where it is unknown what the value of the most significant bit is, and the labelling of msb is $\bot$.

[5] In our terminology, *proving* the property determines it holds in the system. *Disproving* it determines it does not. *Verification* aims to either prove or disprove it.

[6] The example is directly inspired by a bug we found, discussed in Section 5.

Existential abstraction. In TVAR, *existential abstraction* [9] is used, where the abstract states in set $\hat{S}$ are related to the original concrete states in S by a function $\gamma : \hat{S} \to 2^S$, the abstract state $\hat{s} \in \hat{S}$ representing some (not fixed) concrete state in $s \in \gamma(\hat{s})$ in each system execution instant. This is a generalisation of domains usable for Abstract Interpretation [11, 12], also allowing non-lattice domains such as wrap-around intervals [16].

Example 3. In the examples, we will use the three-valued bit-vector domain, where each element is a tuple of three-valued bits, each with value '0' (definitely 0), '1' (definitely 1), or 'X' (possibly 0, possibly 1). Except for figures, we write three-valued bit-vectors in quotes, e.g. $\gamma(\text{“0X1”}) = \{001, 011\}$. The bits can also refer to a predicate rather than a specific value. For example, '1' could mean that $v > 5$ holds, '0' that its negation holds, and 'X' that we do not know.

2.1 Previous TVAR Frameworks

Building on the work of Bruns & Godefroid [1,2,3], Godefroid et al. [19] introduced TVAR by refining the abstract state set, using a state space formalism based on modal transitions. Early TVAR approaches [19, 20, 22, 23, 47] were based on Kripke Modal Transition Structures (KMTS) and did not guarantee previously provable properties stay provable after refinement, i.e. were not monotone.

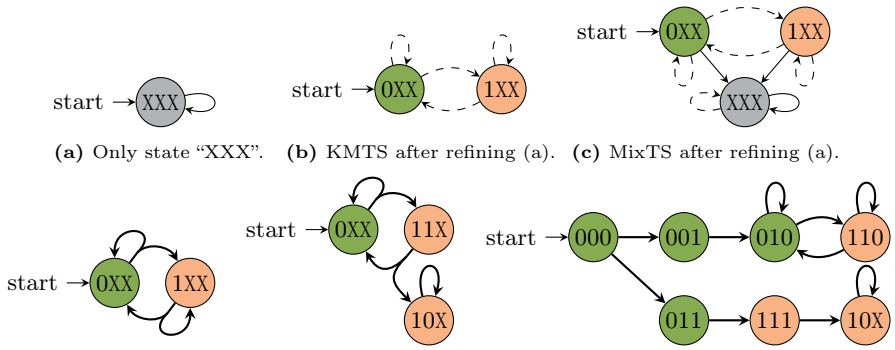
Definition 2. *A Kripke Modal Transition Structure (KMTS) is a five-tuple* $(S, S_0, R^{may}, R^{must}, L)$ *where* S, S_0, *and* L *follow Definition 1, and*

- $R^{may} \subseteq S \times S$ *is the set of transitions which may be present,*
- $R^{must} \subseteq R^{may}$ *is the set of transitions which are definitely present.*

Intuitively, KMTS allow for transitions with unknown presence ($R^{\text{may}} \setminus R^{\text{must}}$). PKS can be trivially converted to KMTS by setting $R^{\text{may}} = R^{\text{must}} = R$. While it is possible to convert a KMTS to an equally expressive PKS by moving the transition presence into the states [21], this requires the set of states to be modified.

Monotone frameworks. Godefroid et al. recognised non-monotonicity as a problem and suggested keeping previous states when refining [19, p. 3-4]. However, Shoham & Grumberg showed the approach was not sufficient, since in certain cases, it prevents verification of additional properties after refinement. As a remedy, they introduced another monotone TVAR framework using Generalized KMTS for CTL [48], later extended to µ-calculus [49]. Gurfinkel & Chechik introduced a framework for verification of CTL properties on Boolean programs using Mixed Transition Systems [28], later extended to lattice-based domains [30][7].

[7] An instance of the framework is implemented in the tool Yasm [29], available online at the time of writing [26]. However, it does not support common programming language elements such as bitwise-operation statements (for example, $y = x \,\&\, 1$) nor full µ-calculus, which our tool can handle without problems.



(a) Only state "XXX". **(b)** KMTS after refining (a). **(c)** MixTS after refining (a).

(d) GKMTS after refining (a). **(e)** GKMTS after refining "1XX" in (d). **(f)** GKMTS after refining "0XX" in (e) by splitting to "000", "001", "010", and "011".

Figure 2. State-based refinement with hyper-transitions based on Generalised KMTS. Implied may-transitions, present in all sub-figures except for (c), are not drawn. The states where it is unknown whether *msb* or $\neg msb$ holds are drawn grey.

Wei et al. introduced a TVAR framework using Reduced Inductive Semantics for μ-calculus under which the results of model-checking GKMTS and MixTS are equivalent [52].

Definition 3. *A Generalized KMTS (GKMTS) is a tuple* $(S, S_0, R^{may}, R^{must}, L)$ *where* S, S_0, R^{may}, *and* L *follow Definition 2 and* $R^{must} : S \times 2^S$ *is the set of* hyper-transitions*, where* $\forall (a, B) \in R^{must} \,.\, \forall b \in B \,.\, (a, b) \in R^{may}$.

Definition 4. *A Mixed Transition System (MixTS) is a tuple* $(S, S_0, R^{may}, R^{must}, L)$ *where* S, S_0, R^{may}, *and* L *follow Definition 2 and* $R^{must} \subseteq S \times S$.

Example 4. We will prove **EF**[**AG**[*msb*]] using the previous state-based TVAR frameworks over the system from Figure 1, starting with abstract state set {"XXX"}. Clearly, we both **may** and **must** transition from "XXX" to "XXX", visualised in Figure 2a. Since *msb* is unknown in "XXX", the model-checking result is unknown and we refine.

Suppose we decide to refine by splitting the abstract state set to {"0XX", "1XX"}, starting in "0XX". From "0XX", we **may** transition either to "0XX" (e.g. by $000 \to 001$ or $010 \to 010$) or "1XX" (e.g. by $010 \to 110$), but cannot conclude that e.g. a transition from "0XX" to itself **must** exist: $011 \in \gamma(\text{"0XX"})$ only transitions to $111 \notin \gamma(\text{"0XX"})$.

PKS cannot be used as they cannot describe unknown-presence transitions. KMTS allow this, producing Figure 2b. However, it is not possible to prove e.g. **EX**[**true**], which was possible in Figure 2a, i.e. the refinement is not monotone. Using MixTS, we retain "XXX" and the must-transitions to it, producing a *forced choice* in Figure 2c. Using GKMTS, we obtain Figure 2d instead. In both, it is possible to prove **EX**[**true**], but not **EF**[**AG**[*msb*]]. Refining further using GKMTS, we obtain Figure 2e, where it is still not possible to prove **EF**[**AG**[*msb*]]:

the hyper-transitions do not imply that the path ("0XX","11X","10X") corresponds to a concrete path. The property is only proven after additional refinement to Figure 2f. While the GKMTS in Figure 2f trivially corresponds to a KMTS or a PKS, fewer refinements and final states may be needed in general when using GKMTS or MixTS as they may guide the refinement better due to monotonicity.

Model checking. µ-calculus properties can be model-checked on PKS and KMTS by a simple conversion to two KS, applying standard model-checking algorithms, and combining the results [2,20]. Similar conversions are also possible for multi-valued logics [27,31]. It is also possible to model-check directly without conversion. A multi-valued model-checker was previously used for the MixTS approach [28]. Game-based model-checking was used for full µ-calculus [22,23].

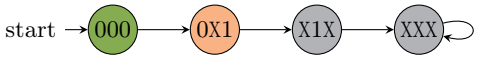
Discussion of previous TVAR frameworks. It was recognised early on that using KMTS with non-monotone refinement is problematic [19,47]. GKMTS seem more susceptible to exponential explosion as MixTS can make use of abstract domains. However, specialised algorithms must be used for MixTS to obtain GKMTS-equivalent results [52]. A drawback of all mentioned approaches is their conceptual complexity which, in our opinion, is the main reason for the dearth of available TVAR tools and test sets, compared to CEGAR. This has also made the analysis of these methods difficult, as illustrated by the subtle differences in definitions of expressiveness identified by Gazda & Willemse [17].

2.2 Abstract Simulator Approaches

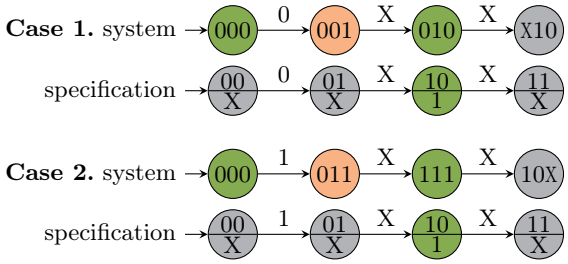
Naïve explicit-state model checking generates a next state from every state and input combination, with the ability to use a system simulator to perform each step. This simulator can be written in an imperative language such as C. Unfortunately, for machine-code systems where state sizes are in kilobytes even for simple microcontrollers and a single port read can produce e.g. 2^8 next states, this results in exponential explosion for all but the simplest toy programs. We will now discuss approaches where the state space is abstract, but still built by generating the next states using an *abstract simulator*.

Trajectory Evaluation. Bryant used three-valued logic simulators for formal verification of hardware circuits [4,6]. He showed linear-time properties (expressed by specification machines or circuit assertions) can be proven using a set of three-valued input sequences that together cover all concrete inputs [4, p. 320] by generating permissible state sequences (*trajectories*). *Symbolic trajectory evaluation (STE)* is an extension that allows parametrisation of the introduced *trajectory formulas* [5]. However, the STE formalism drops the distinction between inputs and states. We refer to Melham [32] for a discussion of STE and extensions. Notably, Generalized STE [53] allows verification of properties corresponding to linear-time µ-calculus [14]. While manual refinement was originally needed, automatic refinement was proposed for both STE [51] and GSTE [7].

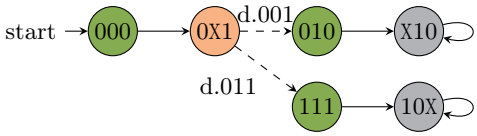
Delayed Nondeterminism. Noll & Schlich [36] verified machine-code programs by model-checking an abstract state space generated by a simulation-based approach. Each input bit was read as 'X' and split to '0' and '1' only when it was decided to in a subsequent step (e.g. if it was an argument of a branch instruction). This allowed e.g. splitting only one bit of a read 8-bit port if the other bits were masked out by a constant first, allowing soundly proving (but not disproving) properties in path-universal logics such as LTL and ACTL.



(a) Abstract state space corresponding to the system in Figure 1, computed by simulation with unconstrained inputs ('X')



(b) Trajectory evaluation: The verification is split into cases, ensuring the abstract input sequences together cover all possible concrete input sequences. Unlike Bryant, we use initial states for consistency with other approaches.



(c) Delayed nondeterminism augmented with must-transitions: the 'X' in state "0X1" is split to '0' and '1' before computing the successor

Figure 3. Simulation-based approaches proving $\mathbf{A}[\mathbf{X}[\mathbf{X}[msb \Leftrightarrow lsb]]]$. (a) can be considered PKS or KMTS, and (c) KMTS. (b) contains four trajectories. Specially in this figure, system states are drawn green if $msb \Leftrightarrow lsb$ holds, orange if it does not, and grey if it is unknown. Specification states are coloured according to the output.

Example 5. Due to the restrictions of the approaches, we will illustrate proving the property "in two steps from the initial state, the most significant bit corresponds to the least significant bit", i.e. $\mathbf{A}[\mathbf{X}[\mathbf{X}[msb \Leftrightarrow lsb]]]$. Simulating without splitting, we produce Figure 3a, unable to prove the property.

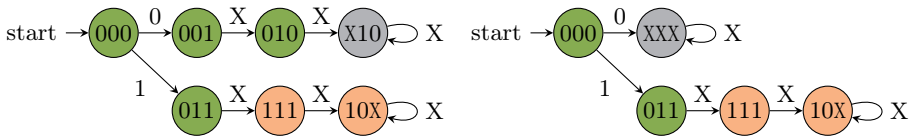
To visualise Bryant's trajectory evaluation approach with explicitly considered inputs [4], we encode the specification as a finite-state machine with two bits containing an initially-zero saturating counter. The system output function is $msb \Leftrightarrow lsb$. The specification outputs '1' iff the counter is 10 and 'X' otherwise. To prove the property, we split verification into two cases based on the value of the first input, and obtain simulated trajectories of both machines in Figure 3b.

The property is proven as the trajectories are long enough (at least 3 for the given property) and the specification output always covers the system output.

To better understand Delayed Nondeterminism, we augment with must-transitions where possible. Splitting "0X1" from Figure 3a, we obtain Figure 3c, where "010" is obtained as a direct successor of "001", and "111" as a direct successor of "011". We cannot augment during the split as 'X' might not generally correspond to a unique input, potentially e.g. being copied before splitting.

3 Input-based Three-Valued Abstraction Refinement

We propose a framework that eliminates the need for modal transitions in TVAR by combining ideas from the discussed approaches: using TVAR, build the abstract state space using an abstract simulator and split **inputs** instead of states.



(a) Refining the input after "000" while keeping all others 'X'.

(b) A smaller reachable state space using a *decayed* step function.

Figure 4. Input-based Three-Valued Abstraction Refinement.

Example 6. We return to the original problem of proving **EF**[**AG**[msb]]. The simulation-based approach initially builds the abstract state space as shown in Figure 3a. After that, we decide (using e.g. a heuristic, machine learning or human guidance) that the input after "000" should be split. We regenerate the abstract state space as shown in Figure 4a. We are immediately able to prove **EF**[**AG**[msb]] holds, meaning the system from Figure 1 contains a bug.

The part of the abstract state space in Figure 4a starting with "001" is unnecessarily large for proving the property, potentially causing exponential explosion problems. To prevent them, we also introduce a way to soundly and precisely regulate the outgoing states of transitions, allowing us to e.g. replace "001" by "XXX" as in Figure 4b when generating the abstract state space, *decaying* to less information. Only one refinement was necessary compared to multiple in Example 4, with the final state space in Figure 4b smaller than in Figure 2f. However, this depends[8] on the correct choice to decay "001" and not "011".

The generated state spaces are PKS, which allows us to use previous work on PKS, KMTS, GKMTS, and MixTS, as PKS are trivially convertible to all. This notably includes model-checking using standard formalisms [2, 20] and TVAR

[8] As with other frameworks, verification performance depends drastically on abstraction, refinement, and implementation choices, further discussed in Sections 4 and 5.

refinement guidance [22, 23, 47], with the caveat that we need to select an input instead of a state to refine. Unlike (G)STE [5,53] and Delayed Nondeterminism [36] which were limited to linear-time or path-universal properties, our approach can be used for the full µ-calculus. Verification can be fully automatic or manually guided, and it is also possible to precisely control the number of reachable abstract states and transitions: we can split inputs up to one by one, and decay any newly reachable states before refining the applied decay.

We will now give the framework formalism and simple requirements for its instances to be sound, monotone, and complete. In Section 4, we will then discuss how reasonable choices of refinements can be made. Finally, in Section 5, we will evaluate an implementation of an instance of our framework in **machine-check**.

3.1 Framework Formalism

We assume that the original Kripke Structure has only one initial state[9], i.e. $K = (S, \{s_0\}, R, L)$. We write the result of model-checking a property ϕ against K as $[\![\phi]\!](K)$, which returns 0 or 1. For a PKS $\hat{K}$, $[\![\phi]\!](\hat{K})$ returns 0, 1, or $\bot$.

We consider the original (concrete) system to be an automaton and will also use automata for abstracting the system, introducing the formalism of *generating automata* that can generate partial Kripke structures.

Definition 5. *A* generating automaton *(GA) is a tuple* $G = (S, s_0, I, q, f, L)$ *with the elements*

- S *(the* set of automaton states*),*
- $s_0 \in S$ *(the* initial state*),*
- I *(the* set of all step inputs*),*
- $q : S \to 2^I \setminus \{\emptyset\}$ *(the* input qualification function*),*
- $f : S \times I \to S$ *(the* step function*),*
- $L : S \times \mathbb{A} \to \{0, 1, \bot\}$ *(the* labelling function*).*

Definition 6. *For a generating automaton* (S, s_0, I, q, f, L)*, we define the PKS-generating function* Γ *as*

$$\Gamma((S, s_0, I, q, f, L)) \stackrel{def}{=} (S, \{s_0\}, R, L) \tag{1}$$

$$\text{where } R = \{(s, f(s, i)) \mid s \in S, i \in q(s)\}. \tag{2}$$

We call a generating automaton $G = (S, s_0, I, q, f, L)$ *concrete* if the labelling function $L : S \times \mathbb{A} \to \{0, 1\}$ (disallowing the value $\bot$) and for all $s \in S$, $q(s) = I$. A concrete GA corresponds to a Moore machine with the output of each state mapping each atomic proposition from $\mathbb{A}$ to either 0 or 1.

Algorithm 1 describes our framework. Given a concrete GA, it abstracts it to an *abstract* generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$, successively refining the input qualification function $\hat{q}$ and step function $\hat{f}$ until the result of model-checking

[9] This is merely a formal choice. For multiple initial states, a dummy initial state can be introduced before them and the verified property ϕ converted to $\mathbf{AX}[\phi]$.

Algorithm 1 Input-based Three-Valued Abstraction Refinement Framework

Require: a concrete generating automaton (S, s_0, I, q, f, L), a µ-calculus property ϕ
Ensure: return $[\![\phi]\!]((S, s_0, I, q, f, L))$ ▷ If requirements are fulfilled, see Corollary 1

$(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L}) \leftarrow \textsc{Abstract}(S, s_0, I, q, f, L)$
while $(r \leftarrow [\![\phi]\!](\Gamma((\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L}))) = \bot$ **do**
  $(\hat{q}, \hat{f}) \leftarrow \textsc{Refine}(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$
end while
return r

is non-$\bot$. $\hat{S}$ and $\hat{I}$ are related to S and I by a state concretization function[10] $\gamma : \hat{S} \to 2^S \setminus \{\emptyset\}$ and an input concretization function $\zeta : \hat{I} \to 2^I \setminus \{\emptyset\}$.

Unlike state-based TVAR, the set of abstract states $\hat{S}$ does not change during refinement. The number of states to be considered is limited by the codomain of $\hat{f}$, allowing structures such as Binary Decision Diagrams to be used. The abstract state space can be built quickly by forward simulation. For backward simulation, care must be taken to pair the states according to inputs.

Example 7. In **machine-check**, states and inputs are composed of bit-vector and bit-vector-array variables, formally represented by flattened $S = \{0,1\}^w$ and $I = \{0,1\}^y$ for finite state width w and finite input width y. A dummy s_0 precedes the actual initial system states, f is a function written in an imperative programming language, and L computes relational operations on state variables.

For the abstract GA, we use three-valued bit-vector abstraction [36, 45] with fast abstract operations [40], abstracting as

$$\gamma^{\text{bit}}(\hat{a}) = \{v \in \mathbb{B} \mid (v = 0 \Rightarrow a \neq \text{`1'}) \wedge (v = 1 \Rightarrow a \neq \text{`0'})\}, \tag{3a}$$

$$\hat{S} = \{\text{`0'}, \text{`1'}, \text{`X'}\}^w,\ \gamma(\hat{s}) = \{s \in S \mid \forall k \in [0, w-1]\ .\ s_k \in \gamma^{\text{bit}}(\hat{s}_k)\}, \tag{3b}$$

$$\hat{I} = \{\text{`0'}, \text{`1'}, \text{`X'}\}^y,\ \zeta(\hat{i}) = \{i \in I \mid \forall k \in [0, y-1]\ .\ i_k \in \gamma^{\text{bit}}(\hat{i}_k)\}. \tag{3c}$$

Again, $\hat{s}_0$ is a dummy state with $\gamma(\hat{s}_0) = \{s_0\}$. We rewrite the step function f into an abstract function $\hat{f}^{\text{basic}} : \hat{S} \times \hat{I} \to \hat{S}$. To formalise the manipulation in Figure 4, we use an *input precision function* $\hat{p}_{\hat{q}} : \hat{S} \to \{0,1\}^y$ and a *step precision function* $\hat{p}_{\hat{f}} : \hat{S} \to \{0,1\}^w$, defining the result of $\hat{q}$ and $\hat{f}$ in each bit k by

$$(\hat{p}_{\hat{q}}(\hat{s})_k = 0 \Rightarrow \hat{q}(\hat{s})_k = \{\text{`X'}\}) \wedge (\hat{p}_{\hat{q}}(\hat{s})_k = 1 \Rightarrow \hat{q}(\hat{s})_k = \{\text{`0'}, \text{`1'}\}), \tag{4a}$$

$$(\hat{p}_{\hat{f}}(\hat{s})_k = 0 \Rightarrow \hat{f}(\hat{s}, \hat{i})_k = \text{`X'}) \wedge (\hat{p}_{\hat{f}}(\hat{s})_k = 1 \Rightarrow \hat{f}(\hat{s}, \hat{i})_k = \hat{f}^{\text{basic}}(\hat{s}, \hat{i})_k). \tag{4b}$$

This allows precise control of the size of the reachable abstract state space. For each $\hat{s} \in \hat{S}$, if $\hat{p}_{\hat{q}}(\hat{s}) = (0)^y$, there is exactly one outgoing transition. Each bit set to 1 increases that up to a factor of 2. If $\hat{p}_{\hat{f}}(\hat{s}) = (0)^w$, there is exactly one outgoing transition to the "most-decayed" state $(\text{`X'})^w$.

[10] We forbid abstract elements with no concretizations as they do not represent any concrete element. Practically speaking, this does not disqualify abstract domains with such elements; we just require such elements are not produced by $\hat{s}_0$, $\hat{q}$, or $\hat{f}$.

3.2 Soundness, Monotonicity, and Completeness

In this subsection, we state the requirements sufficient to ensure soundness (the algorithm returns the correct result if it terminates), monotonicity (refinements never lose any information), and completeness (the algorithm always terminates). For reasons of space, we only sketch the proofs in this version of the paper[11].

To intuitively describe the requirements, we formalise the concept of *coverage*. An abstract state $\hat{s}$ or input $\hat{i}$ *covers* a concrete $s \in S$ or $i \in I$ exactly when $s \in \gamma(\hat{s})$ or $i \in \zeta(\hat{i})$, respectively, and it *covers* another abstract state $\hat{s}^* \in \hat{S}$ or input $\hat{i}^* \in \hat{I}$ exactly when $\gamma(\hat{s}^*) \subseteq \gamma(\hat{s})$ or $\zeta(\hat{i}^*) \subseteq \zeta(\hat{i})$, respectively.

We want abstraction to preserve the truth value of μ-calculus properties in the following sense:

Definition 7. *A partial Kripke structure $K^{\uparrow}$ is* sound *with respect to a partial Kripke structure $K^{\downarrow}$ if, for every property ϕ of μ-calculus over the set of atomic propositions $\mathbb{A}$, it holds that*

$$[\![\phi]\!](K^{\uparrow}) \neq \bot \Rightarrow [\![\phi]\!](K^{\downarrow}) = [\![\phi]\!](K^{\uparrow}). \tag{5}$$

Intuitively, $K^{\uparrow}$ can contain less information than $K^{\downarrow}$, turning some non-$\bot$ proposition results to $\bot$. No other differences are possible.

To ensure the soundness of Algorithm 1, we use the following requirements. Soundness is ensured with any refinement heuristic as long as they are met.

Definition 8. *A GA $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ is a* soundness-guaranteeing (γ, ζ)-abstraction *of a concrete GA $G = (S, s_0, I, q, f, L)$ iff*

$$\gamma(\hat{s}_0) = \{s_0\}, \tag{6a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall s \in \gamma(\hat{s}) \,.\, \forall a \in \mathcal{A} \,.\, (\hat{L}(\hat{s}, a) \neq \bot \Rightarrow \hat{L}(\hat{s}, a) = L(s, a)), \tag{6b}$$

$$\forall (\hat{s}, i) \in \hat{S} \times I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, i \in \zeta(\hat{i}), \tag{6c}$$

$$\forall (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \forall (s, i) \in \gamma(\hat{s}) \times \zeta(\hat{i}) \,.\, f(s, i) \in \gamma(\hat{f}(\hat{s}, \hat{i})). \tag{6d}$$

Informally, the four requirements express the following:

(a) **Initial state concretization.** The abstract initial state has exactly the concrete initial state in its concretization.
(b) **Labelling soundness.** Each abstract state labelling must either correspond to the labelling of all concrete states it covers or be unknown.
(c) **Full input coverage.** In every abstract state, each concrete input must be covered by some qualified abstract input.
(d) **Step soundness.** Each result of the abstract step function must cover all results of the concrete step function where its arguments are covered by the abstract step function arguments.

The requirements ensure the soundness of the used abstractions as follows.

[11] The full proofs are given in Appendix A in a version of this paper available at https://arxiv.org/abs/2408.12668.

Theorem 1 (Soundness). *For every generating automaton $\hat{G}$ and concrete generating automaton G, state concretization function γ, and input concretization function ζ such that $\hat{G}$ is a soundness-guaranteeing (γ, ζ)-abstraction of G, the partial Kripke structure $\Gamma(\hat{G})$ is sound with respect to $\Gamma(G)$.*

Proof sketch. Show that $\{(s, \hat{s}) \mid \hat{s} \in \hat{S} \wedge s \in \gamma(\hat{s})\}$ is a modal simulation [15, p. 408] from $\Gamma(G)$ to $\Gamma(\hat{G})$ due to (6). Then, $\hat{G}$ is sound wrt. G due to a previous theorem on preservation of µ-calculus formulas [15, p. 410].

Corollary 1. *Assume that the functions* Abstract *and* Refine *ensure that the generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ in Algorithm 1 is always a soundness-guaranteeing (γ, ζ)-abstraction of (S, s_0, I, q, f, L). Then, if the algorithm terminates, its result is correct.*

Example 8. Continuing from Example 7, (6a) is fulfilled trivially. (6c) is fulfilled due to (3c) and (4a). From (4b), it is apparent that

$$\forall(\hat{s}, \hat{i}) \in (\hat{S}, \hat{I}) \,.\, \gamma(\hat{f}^{\text{basic}}(\hat{s}, \hat{i})) \subseteq \gamma(\hat{f}(\hat{s}, \hat{i})), \tag{7}$$

i.e. results of $\hat{f}$ cover results of $\hat{f}^{\text{basic}}$ that abstracts f. We carefully implemented the translation of f to $\hat{f}^{\text{basic}}$ and $\hat{L}$ so that(6b) and (6d) hold.

Next, we turn to monotonicity, which ensures no algorithm loop iteration loses information. We give the requirements for the refinement to guarantee it.

Definition 9. *A generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ is monotone wrt. (γ, ζ)-coverage iff*

$$\begin{aligned} &\forall(\hat{s}, \hat{s}', a) \in \hat{S} \times \hat{S} \times \mathcal{A} \,. \\ &\qquad ((\gamma(\hat{s}') \subseteq \gamma(\hat{s}) \wedge \hat{L}(\hat{s}, a) \neq \bot) \Rightarrow \hat{L}(\hat{s}', a) = \hat{L}(\hat{s}, a)), \end{aligned} \tag{8a}$$

$$\begin{aligned} &\forall(\hat{s}, \hat{s}', \hat{i}, \hat{i}') \in \hat{S} \times \hat{S} \times \hat{I} \times \hat{I} \,. \\ &\qquad ((\gamma(\hat{s}') \times \zeta(\hat{i}') \subseteq \gamma(\hat{s}) \times \zeta(\hat{i})) \Rightarrow \gamma(\hat{f}(\hat{s}', \hat{i}')) \subseteq \gamma(\hat{f}(\hat{s}, \hat{i})). \end{aligned} \tag{8b}$$

Informally, we require that each abstract state covered by an abstract state with non-$\bot$ labelling has the same labelling, and when abstract step function arguments are covered by some other arguments, its result is also covered by the result computed using the other arguments.

Definition 10. *The generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}', \hat{f}', \hat{L})$ is a (γ, ζ)-monotone refinement of the generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ iff it is monotone wrt. (γ, ζ)-coverage and*

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i}' \in \hat{q}'(\hat{s}) \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}') \subseteq \zeta(\hat{i}), \tag{9a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \exists \hat{i}' \in \hat{q}'(\hat{s}) \,.\, \zeta(\hat{i}') \subseteq \zeta(\hat{i}), \tag{9b}$$

$$\forall(\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \gamma(\hat{f}'(\hat{s}, \hat{i})) \subseteq \gamma(\hat{f}(\hat{s}, \hat{i})). \tag{9c}$$

Informally, in addition to the monotonicity wrt. coverage, we also require:

(a) **New qualified inputs are not spurious.** Each new qualified input is covered by at least one old qualified input.
(b) **Old qualified inputs are not lost.** Each old qualified input covers at least one new qualified input.
(c) **New step function covered by old.** The result of the new step function is always covered by the result of the old step function.

The need for both quantifier combinations in the first two requirements in Equation 9 may be surprising. Their violations correspond to transition addition and removal, respectively, which could make a previously non-$\bot$ property $\bot$.

Theorem 2 (Monotonicity). *If the generating automaton $\hat{G}'$ is a (γ, ζ)-monotone refinement of the generating automaton $\hat{G}$, then for every μ-calculus property ψ for which $[\![\psi]\!](\Gamma(\hat{G})) \neq \bot$, it also holds $[\![\psi]\!](\Gamma(\hat{G}')) \neq \bot$.*

Proof sketch. Show that $\{(\hat{s}', \hat{s}) \mid \hat{s}' \in \hat{S}' \wedge s \in S)\}$ is a modal simulation [15, p. 408] from $\Gamma(\hat{G}')$ to $\Gamma(\hat{G})$ due to (8) and (9). Then, $\hat{G}$ is sound wrt. $\hat{G}'$ due to a previous theorem on preservation of µ-calculus formulas [15, p. 410].

Corollary 2. *If the update in the loop of Algorithm 1 performs a (γ, ζ)-monotone refinement of the generating automaton $(\hat{S}, \hat{s}_0, \hat{q}, \hat{f}, \hat{L})$ and for a μ-calculus property ψ, it held that $[\![\psi]\!](\Gamma((\hat{S}, \hat{s}_0, \hat{q}, \hat{f}, \hat{L}))) \neq \bot$ before the loop iteration, then this is also the case after the iteration.*

Now we turn to termination. We use the following requirements to ensure that the algorithm makes progress.

Definition 11. *The generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}', \hat{f}', \hat{L})$ is a* strictly (γ, ζ)*-monotone refinement of the generating automaton $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ if it is a monotone refinement and either*

$$\exists \hat{s} \in \hat{S} \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \forall \hat{i}' \in \hat{q}'(\hat{s}) \,.\, \exists i \in \zeta(\hat{i}) \,.\, i \notin \zeta(\hat{i}'), \tag{10}$$

$$\text{or } \exists (\hat{s}, \hat{i}) \in \hat{S} \times \hat{I} \,.\, \exists s \in \gamma(\hat{f}(\hat{s}, \hat{i})) \,.\, s \notin \gamma(\hat{f}'(\hat{s}, \hat{i})). \tag{11}$$

Informally, progress in monotone refinements is ensured either by an abstract state where some old qualified input is not fully covered by any new qualified input or by an abstract state-input combination where the old step function result has at least one concretization absent in the new result concretizations.

Example 9. Continuing from Example 8, during each refinement, we set at least one bit of $\hat{p}_{\hat{q}}$ and/or $\hat{p}_{\hat{f}}$ to 1, and prohibit resetting them to 0, fulfilling (9). We implemented $\hat{L}$ so that (8a) holds. (8b) holds due to (4b). As setting bits in $\hat{p}_{\hat{f}}$ does not necessarily mean $\hat{f}$ changes, we set them until R in $\Gamma(\hat{G})$ changes. This means $\hat{q}$ or $\hat{f}$ change as per (10) or (11), and the refinement is strictly monotone.

Theorem 3 (Completeness). *If $\hat{S}$ and $\hat{I}$ are finite, there is no infinite sequence of generating automata that are soundness-guaranteeing (γ, ζ)-abstractions of some G such that all subsequent pairs in the sequence are strictly (γ, ζ)-monotone refinements.*

Proof sketch. Assume such a sequence exists. For each sequence element $\hat{G}$, define a function $M_{\hat{G}}(\hat{s}) = (\{\zeta(\hat{i}) \mid \hat{i} \in \hat{q}_{\hat{G}}(\hat{s})\}, \{(\hat{i} \in \hat{I}, \gamma(\hat{f}_{\hat{G}}(\hat{s}, \hat{i})))\})$. Show that $M_{\hat{G}}$ must be different for different sequence elements. As there is only a finite number of different functions $M_{\hat{G}}$, the sequence cannot be infinite.

Corollary 3. *If $\hat{S}, \hat{I}$ are finite, the functions* Abstract *and* Refine *in Algorithm 1 ensure that $(\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ always is a* soundness-guaranteeing (γ, ζ)-abstraction *of (S, s_0, I, q, f, L), and calls of* Refine *perform strict (γ, ζ)-monotone refinements, then the algorithm returns the correct result in finite time.*

Fulfilling the requirements of Corollary 3 is not trivial. We must exclude the situation when no strict (γ, ζ)-monotone refinement is possible any more while a non-$\bot$ result has not yet been reached. We propose Lemma 1 for easier reasoning.

Definition 12. *A generating automaton $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$ is (γ, ζ)-terminating wrt. a concrete generating automaton $G = (S, s_0, I, q, f, L)$ if it is a (γ, ζ)-abstraction of G monotone wrt. (γ, ζ)-coverage and*

$$\forall (s, \hat{s}) \in S \times \hat{S} \,.\, (\gamma(\hat{s}) = \{s\} \Rightarrow \hat{L}(\hat{s}) = L(s)), \tag{12a}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall \hat{i} \in \hat{q}(\hat{s}) \,.\, \exists i \in I \,.\, \zeta(\hat{i}) = \{i\}, \tag{12b}$$

$$\forall \hat{s} \in \hat{S} \,.\, \forall i \in I \,.\, \exists \hat{i} \in \hat{q}(\hat{s}) \,.\, \zeta(\hat{i}) = \{i\}, \tag{12c}$$

$$\forall (\hat{s}, \hat{i}, s, i) \in \hat{S} \times \hat{I} \times S \times I . ((\gamma(\hat{s}), \zeta(\hat{i})) = (\{s\}, \{i\}) \Rightarrow \gamma(\hat{f}(\hat{s}, \hat{i})) = \{f(s, i)\}). \tag{12d}$$

Informally, (12a) requires that each abstract state with a single concretization has the same labelling as the corresponding concrete state. (12b) requires that every qualified abstract input has a single concretization, while (12c) requires that every concrete input corresponds to some qualified abstract input. (12d) requires that the step function applied on single-concretization abstract state-input combinations produces the correct single-concretization result.

Lemma 1. *If $\hat{G}$ is (γ, ζ)-terminating wrt. G, then for every μ-calculus property ψ, $[\![\psi]\!](\Gamma(\hat{G})) = [\![\psi]\!](\Gamma(G))$. Furthermore, if $\hat{G}$ is a soundness-guaranteeing (γ, ζ)-abstraction of G for which some (γ, ζ)-monotone refinement is (γ, ζ)-terminating wrt. G, then $\hat{G}$ itself is (γ, ζ)-terminating wrt. G or the refinement is strict.*

Proof sketch. Show that $\{(\hat{s}, s) \mid \hat{s} \in \hat{S} \wedge s \in \gamma(\hat{s})\}$ is a modal simulation from $\Gamma(\hat{G})$ to $\Gamma(G)$ due to (6) and (12). As $[\![\phi]\!](\Gamma(G)) \neq \bot$, $[\![\psi]\!](\Gamma(\hat{G})) = [\![\psi]\!](\Gamma(G))$. Then, consider that if the $\hat{G}$ in the second part is not (γ, ζ)-terminating, at least one of (12a), (12b), (12c), (12d) does not hold for $\hat{G}$. Show on a case-by-case basis this is a contradiction for (12a) and forces the refinement to be strict otherwise.

Corollary 4. *If every generating automaton constructed in Algorithm 1 fulfils the conditions of Lemma 1, calls to* Refine *can always perform a* strict (γ, ζ)-*monotone refinement to a soundness-guaranteeing (γ, ζ)-abstraction.*

Example 10. Continuing from Example 9, we implemented $\hat{L}, \hat{f}^{\text{basic}}$ so that they fulfil (12a) and (12d). Since (12b) and (12c) hold due to (4a), $\hat{G}^*$ obtained by $\hat{p}_{\hat{q}} = (1)^y, \hat{p}_{\hat{f}} = (1)^w$ is (γ, ζ)-terminating. Inspecting (9), $\hat{G}^*$ is monotone wrt. all other applicable GA, and Corollary 4 holds. As soundness was already ensured in Example 8 and $\hat{S}$, $\hat{I}$ are finite by definition, Corollary 3 holds.

4 Making Reasonable Refinement Choices

A sound, monotone, and complete instantiation of the framework can e.g. refine randomly while fulfilling the requirements from Subsection 3.2. However, for fast verification, refinements must be chosen carefully. We will discuss how reasonable refinements can be made, inspired by the `FindFailure` algorithms to find the cause of an unknown result of verification of CTL [48, p. 551-552] and game-based verification of µ-calculus [23, p. 1145].

The goal is to find the root cause of an unknown verification result, an unknown atomic labelling in an abstract state reached through the abstract state space, which we will also call the *culprit*. For this, we extend the semantics of three-valued µ-calculus on PKS, a simplification of three-valued µ-calculus on KMTS [23,49], so that it returns the culprit instead of the unknown value $\bot$. For simplicity, we base this definition on the standard syntax of µ-calculus in positive normal form. Assuming, in addition to the set of atomic labellings $\mathbb{A}$, a set of µ-calculus variables $\mathbb{V}$, a formula then has the form

$$\phi ::= a \mid \neg a \mid Z \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle\phi\rangle \mid [\phi] \mid \mu Z \,.\, \phi \mid \nu Z \,.\phi \tag{13}$$

where $a \in \mathbb{A}$, $Z \in \mathbb{V}$. The language of µ-calculus then consists of all closed-form formulas given by this definition. We also assume variables are well-bound, i.e. every variable must be bound by μ or ν exactly once.

Definition 13. *Given a GA $\hat{G} = (\hat{S}, \hat{s}_0, \hat{I}, \hat{q}, \hat{f}, \hat{L})$, an* environment *is a function $\rho : \mathbb{V} \to (\hat{S} \to (\{0,1\} \cup C))$, where C is the set of pairs $(\mathcal{S}, a)$ with $\mathcal{S}$ being a non-empty sequence of states from $\hat{S}$ and $a \in \mathbb{A}$ an atomic labelling.*

Now let $\hat{R}$ be the transition relation of the PKS $\Gamma(\hat{G})$. We define the extended semantics of µ-calculus recursively according to its syntax such that for an environment ρ and a state $\hat{s} \in \hat{S}$,

$$[\![a]\!]^\rho_{\hat{s}} ::= \begin{cases} \hat{L}(\hat{s}, a) & \text{if } \hat{L}(\hat{s}, a) \neq \bot, \\ ((\hat{s}), a) & \text{otherwise,} \end{cases} \qquad [\![\neg a]\!]^\rho_{\hat{s}} ::= \begin{cases} 1 - \hat{L}(\hat{s}, a) & \text{if } \hat{L}(\hat{s}, a) \neq \bot, \\ ((\hat{s}), a) & \text{otherwise,} \end{cases}$$

$$[\![\phi \wedge \psi]\!]^\rho_{\hat{s}} ::= \begin{cases} 1 & \text{if } [\![\phi]\!]^\rho_{\hat{s}} = 1 \text{ and } [\![\psi]\!]^\rho_{\hat{s}} = 1, \\ 0 & \text{if } [\![\phi]\!]^\rho_{\hat{s}} = 0 \text{ or } [\![\psi]\!]^\rho_{\hat{s}} = 0, \\ c \in C \text{ s.t. } c = [\![\phi]\!]^\rho_{\hat{s}} \text{ or } c = [\![\psi]\!]^\rho_{\hat{s}} & \text{otherwise,} \end{cases}$$

$$[\![\langle\phi\rangle]\!]^\rho_{\hat{s}} ::= \begin{cases} 1, \text{ if there is a } \hat{s}^+ \text{ with } (\hat{s}, \hat{s}^+) \in \hat{R} \text{ and } [\![\phi]\!]^\rho_{\hat{s}^+} = 1, \\ 0, \text{ if for all } \hat{s}^+, (\hat{s}, \hat{s}^+) \in \hat{R} \text{ implies } [\![\phi]\!]^\rho_{\hat{s}^+} = 0, \\ ((\hat{s}, \hat{s}_1, \dots), a) \text{ s.t. } (\hat{s}, \hat{s}_1) \in \hat{R} \text{ and } [\![\phi]\!]^\rho_{\hat{s}_1} = ((\hat{s}_1, \dots), a), \text{otherwise,} \end{cases}$$

$$[\![Z]\!]^\rho_{\hat{s}} ::= \rho(Z)(\hat{s}), \qquad [\![\mu Z \,.\, \phi]\!]^\rho_{\hat{s}} ::= \mathit{lfp}(\lambda g \,.\, [\![\phi]\!]^{\rho[Z \mapsto g]}_{\hat{s}}).$$

The least fixed point lfp is defined as usual for the ordering $0 < \bot < 1$, but retains the element of C previously obtained when the environmental value remains $\bot$ between subsequent iterations of fixed-point computation. The operators $\phi \vee \phi$, $[\phi]$, and $\nu Z \,.\phi$ are defined correspondingly to their duals.

As already mentioned, in the case where the result is an element of C, we will call it the *culprit*. The definition is not unique, allowing implementation choice of which culprit to use for refinement. With any choice, the culprit describes a path through the state space ending with a labelling that contributes to the verification result $[\![\phi]\!]_{\hat{s}_0}$ being unknown.

Using the culprit for refinement. If the instance of our framework fulfils the conditions from Corollary 3 then there clearly is at least one application of $\hat{f}$ on the culprit's path that makes the subsequent state cover more than one state, since $\gamma(\hat{s}_0) = \{s_0\}$ and (12a) forbids spuriously unknown labellings. As such, we can decide to only refine $\hat{f}$ from the preceding $\hat{s}$ of such applications. This avoids refinements that definitely would not impact any culprit, unnecessarily increasing the size of the abstract state space. This idea can be extended to the instance details, not refining e.g. inputs that provably do not have an impact on the culprit labelling. The choice of culprit and actual refinement can be fine-tuned according to the typical systems under verification in order to achieve good performance.

Example 11. In **machine-check**, we model-check and obtain the culprit in the spirit of the above discussion. To ensure deterministic behaviour, we prefer the left culprit for operators $\phi \wedge \phi, \phi \vee \phi$ and the culprit continuing with the state with the smallest unique identification number for next-state operators $\langle\phi\rangle, [\phi]$.

For better performance, we use incremental model-checking where we construct a history of environment updates and only update a part of the history after refinement, if possible. To avoid storing the elements of C, where paths can be problematically long, we instead store the evaluation choices made to obtain the given history point from previous history points and atomic labellings. This allows us to reconstruct the culprit if necessary by walking back through history.

After obtaining the culprit $((\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{n-1}, \hat{s}_n), a)$, we compute a cone of influence on a being unknown in $\hat{s}_n$. For each state $\hat{s} \in \{\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{n-1}\}$, we determine candidate bit positions $k \in [0, y-1]$ where $\hat{p}_{\hat{q}}(\hat{s})_k = 0$ and it is possible this has contributed to a being unknown in $\hat{s}_n$, and likewise for $k \in [0, w-1]$ where $\hat{p}_{\hat{f}}(\hat{s})_k = 0$. We then choose the state in which to refine $\hat{p}_f$ or $\hat{p}_q$ deterministically based on the following heuristic:

- If there are candidate bits for refinement of $\hat{p}_f$, we choose to refine $\hat{p}_f$ in the last state on the culprit path where there is some candidate.
- Otherwise, we pre-select only the states containing candidates that have the highest level of indirection (indirection rises for indices to array reads and writes that impact the culprit labelling). We then choose to refine $\hat{p}_q$ using the candidates on the last pre-selected state on the culprit path.

From multiple candidate bits for the refinement in the chosen state, we choose the input variable deterministically. From multiple bit candidates in a bit-vector variable, we refine the most significant bit.

5 Implementation and Experimental Evaluation

We implemented an instance of our proposed framework in our free and open-source tool **machine-check**. In this section, we discuss our implementation choices and show that our framework is able to mitigate exponential explosion.

5.1 Implementation

Our focus is verification of machine-code systems, where the system is composed of a processor and a machine-code program it executes. However, there is no special handling for typical features of machine-code systems such as the Program Counter, call stack, etc., allowing **machine-check** to support verification of arbitrary finite-state digital systems against properties in propositional µ-calculus.

Systems. Systems to be verified are described in a subset of the Rust programming language. The systems can be comprised of bit-vectors that support standard arithmetical, logical, shift, and extension operations and bit-vector arrays that support indexing.

Properties. Properties are described in a Rust-style format that allows µ-calculus and Computation Tree Logic properties to be expressed. Atomic propositions are formed by equality and comparison operations between variables of the system. CTL and µ-calculus can be freely mixed, and CTL operators are converted to µ-calculus before model-checking. For example, the CTL property $\mathbf{AG}[\texttt{value} = 0]$ is translated to $\nu Z.(\texttt{value} = 0) \wedge [Z]$.

Translating the systems. The system description, corresponding to the concrete GA under verification, is compiled together with **machine-check** core to form the system verifier, and the description is automatically translated [38] to representations that serve as an abstract simulator and a cone-of-influence calculator. No other interaction between the system and framework is needed.

Abstraction. Three-valued bit-vector abstraction is used for bit-vectors, with fast abstract operations [40]. Arrays are abstracted using a version of sparse representation where elements that are not stored have the value of the nearest stored element with a lesser index.

Building the state space. The state space is built incrementally using the abstract analogue of the system as the abstract simulator. Abstract state data and a sparse transition graph are retained throughout verification, with garbage collection of abstract states that are no longer reachable.

Model-checking. As noted in Example 11, we use incremental model-checking of µ-calculus. For reassurance, once the final verification result is obtained, it is double-checked non-incrementally.

Verification settings. With default verification settings, inputs are initially unknown but decay is not used, i.e. $\hat{p}_{\hat{q}}(\hat{s}) = (0)_{k=0}^{y-1}$, $\hat{p}_{\hat{f}}(\hat{s}) = (1)_{k=0}^{w-1}$ from all states $\hat{s} \in \hat{S}$. It is possible to enable decay ubiquitously, i.e. $\hat{p}_{\hat{f}}(\hat{s}) = (0)_{k=0}^{w-1}$, but we found this to be slower for machine-code systems due to the need to refine each generated state. A naïve strategy with $\hat{p}_{\hat{q}}(\hat{s}) = (1)_{k=0}^{y-1}$ corresponding to explicit-state verification without abstraction would immediately result in hopeless explosion of the state space due to the amount of initially uninitialised memory and read inputs in machine code systems.

5.2 Evaluation Setup

We evaluated verification of µ-calculus properties on machine-code systems for the AVR ATmega328P microcontroller that we described for use in **machine-check** according to the datasheet [33] and instruction set reference [34].

Benchmark set. As we are not aware of any publicly available and appropriately licensed benchmark sets for formal verification of embedded 8-bit microcontrollers, we used our own set of benchmarks based on machine-code programs for ATmega328P. Our set of benchmarks currently contains 16 programs: 6 programs where we expect a violation of some system-inherent property (e.g. forbidden instruction), 5 simple programs, and 5 more complicated programs, four of which are variations of a simplified version of a program for calibration of a Voltage-Controlled Oscillator used in real life. For C programs, we benchmark the machine code obtained with debug and release builds separately.

Setup. The evaluation was performed on a Linux machine with an Intel Core i9-12900 processor with 128 GB of RAM. The tool was built with Rust 1.83.0 in release configuration. The default verification strategy was used.

Evaluated properties. We evaluated up to 8 properties on each program[12]. Using w for a propositional formula on atomic properties that varies depending on the program, the properties included, among others:

- Inherent property defined in the processor description, determining if a system with given machine-code is permissible, containing no calls to unimplemented instructions, peripherals, etc. $\mathbf{AG}[w]$, i.e. a safety property.

[12] To ensure the implementation is not faulty, we also test the properties on simple non-machine-code systems in our test suite, e.g. whether the verifier distinguishes between $\mathbf{AFG}[w]$ and $\mathbf{AF}[\mathbf{AG}[w]]$ using the standard example [41, p. 65].

- Recovery property (as discussed in Example 1), checking whether the system can be recovered from every reachable state to the program loop start with initial output values. $\mathbf{AG}[\mathbf{EF}[w]]$ in CTL. Not a linear-time property.
- Program counter (PC) remains within the main program loop in the main function infinitely often on all paths. $\mathbf{FG}[w]$ in LTL, $\mu X \,.\, \nu Y \,.\, [X] \vee (w \wedge [Y])$ in µ-calculus [13, p. 3133]. Not present in CTL.
- PC is never at the start of the main loop during even times (wrt. instructions). $\nu X \,.\, w \wedge [\,[X]\,]$ in µ-calculus, not present in CTL, LTL, nor CTL*.
- The stack pointer stays above a given value, preventing stack overflow problems. $\mathbf{AG}[w]$, i.e. a safety property.

5.3 Evaluation Results

We performed 126 measurements in total on the evaluated machine-code systems and properties, all of which resulted in the property being verified. All results matched our expectations. We will discuss a limited number of the more interesting measurements in detail, demonstrating the applicability of the framework.

Calibration. We used a simplified version of a program for calibration of a Voltage-Controlled Oscillator (VCO) using binary search. We previously used the program in a real device, where the VCO frequency was adjusted by a digital potentiometer based on the measured frequency of the produced wave. For verification, we replaced the interactions with the digital potentiometer and frequency measure by I/O writes and reads, leaving the core algorithm unchanged.

Verifying using **machine-check**, we found a bug in the program using the recovery property: the output value could never recover to zero after an iteration concludes, since the least significant bit was never set to zero during the calibration, causing a calibration precision loss. This bug would have been tricky to find using source-code verification, as the problem occurred in peripheral manipulation. Linear-time properties of forms $\mathbf{F}[w]$ or $\mathbf{FG}[w]$ would not find the bug, as the calibration command may never be given. After fixing the bug, the recovery property holds, as seen in Table 1.

Despite only the simple three-valued bit-vector abstraction used, the final sizes of abstract state spaces are reasonable. The refinement guidance is remarkably well-behaved, arriving at the same state space for the properties where a fast decision was not possible, despite the non-inherent properties having no knowledge about e.g. illegal instructions. Verification using the infinitely-often property is notably slow due to the need to model-check with non-trivial nested quantifiers, but this slowdown is effectively mitigated by using the inherent property first.

To show Input-based TVAR holds up where explicit-state verification would be completely infeasible, we created complicated calibration variants where a 64-bit value is read during initialisation, and unrelated 8-bit volatile reads are performed while doing the calibration, ensuring there are more than 2^{80} concrete states even if not considering uninitialised memory. As seen in Table 1, **machine-check** verifies with at most approx. factor-of-4 slowdown.

Table 1. Selected measurements of verification of the calibration program compiled in debug configuration. Asterisks mark measurements where the inherent property was verified first and verification of the given property progressed from that state space.

Variant	Property name	Holds	Refin.	States	Transitions	CPU t. [s]	Mem. [MB]
Original	Inherent	✓	513	13059	13573	17.09	87.39
Original	Recovery	✗	513	13059	13573	19.88	111.34
Original	Infinitely often	✓	513	13059	13573	672.05	160.58
Original	Infinitely often*	✓	513	13059	13573	17.98	115.34
Original	Even non-starts	✗	0	17	18	<0.01	3.78
Original	Stack above 0x08FD	✓	513	13059	13573	17.08	87.30
Original	Stack above 0x08FE	✗	0	17	18	<0.01	3.94
Fixed	Inherent	✓	513	13059	13573	17.21	87.60
Fixed	Recovery	✓	513	13059	13573	29.16	112.02
Fixed	Infinitely often	✓	513	13059	13573	672.19	161.65
Fixed	Infinitely often*	✓	513	13059	13573	18.07	115.66
Fixed	Even non-starts	✗	0	17	18	<0.01	3.72
Fixed	Stack above 0x08FD	✓	513	13059	13573	17.2	87.56
Fixed	Stack above 0x08FE	✗	0	17	18	<0.01	3.98
Comp. orig.	Inherent	✓	771	17330	18102	54.43	125.83
Comp. orig.	Recovery	✗	770	17333	18104	65.28	159.20
Comp. orig.	Infinitely often*	✓	771	17330	18102	73.29	173.08
Comp. fixed	Inherent	✓	771	17330	18102	51.63	125.71
Comp. fixed	Recovery	✓	771	17330	18102	70.74	159.52
Comp. fixed	Infinitely often*	✓	771	17330	18102	64.2	173.04

Factorial. We implemented a program in C which computes the factorial of an input recursively. In the compiled machine code, the computation remained recursive in the debug build but was optimised to an iterative version in the release build. We verified properties including maximal stack sizes for both versions, as seen in Table 2. Regarding the state space size, we noticed that the call return locations remain known even after the stack is popped and they are no longer relevant, unnecessarily growing the state space. We believe a smarter, selective decay of step precision could alleviate these problems.

Comparison with other tools. While there has been a multitude of previous machine-code verifiers [39, p. 29-37], we are not aware of any that support full µ-calculus. Our work has been inspired by the research on the tool [mc]square / Arcade.µC [24, 36, 37, 42, 45, 46], which could use strategies including Delayed Nondeterminism [36]. However, the tool has been discontinued and is not publicly available. It only supported proving LTL/ACTL properties when abstraction was used, and needed system-based hints for useful abstraction. As for more recent work, verifiers based on theorem proving have been introduced, with the Serval [35, 50] tool supporting automatic verification but not programs with loops, and Islaris [43, 44] requiring manual proof support but supporting loops. Both only support verification of safety properties. In contrast, **machine-check** supports fully automatic verification of µ-calculus properties on machine-code programs that include loops thanks to the introduced framework.

Table 2. Selected measurements of verification of the factorial program. Everything is as expected. The recovery property does not hold as the factorial program never outputs zero after the first computation. The infinitely-often property does not hold due to calls to another function from the main function affecting the Program Counter.

Program	Property name	Holds	Refin.	States	Transitions	CPU t. [s]	Mem. [MB]
Debug	Inherent	✓	576	51595	52940	50.4	359.79
Debug	Recovery	✗	576	51595	52940	76.48	469.88
Debug	Infinitely often	✗	6	1396	1411	5.55	259.89
Debug	Even non-starts	✓	576	51595	52940	46.96	360.40
Debug	Stack above 0x08DD	✓	576	51595	52940	52.77	359.74
Debug	Stack above 0x08DE	✗	3	748	756	0.16	31.58
Release	Inherent	✓	45	4272	4378	0.78	46.84
Release	Recovery	✗	45	4272	4378	1.11	54.58
Release	Infinitely often	✗	6	1006	1021	2.93	148.15
Release	Even non-starts	✗	3	550	558	0.04	17.92
Release	Stack above 0x08FB	✓	45	4272	4378	0.78	46.79
Release	Stack above 0x08FC	✗	0	20	21	< 0.01	3.72

6 Conclusion

We presented a novel input-based Three-valued Abstraction Refinement (TVAR) framework for formal verification of μ-calculus properties using abstraction refinement and gave requirements for soundness, monotonicity, and completeness. Our framework can verify properties not verifiable using Counterexample-guided Abstraction Refinement or (Generalized) Symbolic Trajectory Evaluation, eliminating verification blind spots. Compared to previous TVAR frameworks, our framework does not use modal transitions, allowing monotonicity without formalism complications. We implemented the framework in our tool **machine-check**, which can verify machine-code systems while mitigating exponential explosion.

Acknowledgments. The work of Jan Onderka reported in this paper was supported by an Amazon Research Award (Fall 2023) and Czech Technical University in Prague grant SGS23/208/OHK3/3T/18. Development of **machine-check** was supported by the NGI Zero Core fund established by NLnet Foundation. The work of Stefan Ratschan was supported by the research programme of the Strategy AV21 AI: Artificial Intelligence for Science and Society and by institutional support RVO:67985807l.

Data Availability Statement. The artefact containing detailed results summarised in Section 5 and reproduction data (including the benchmark set) is available at https://doi.org/10.5281/zenodo.17167265.

References

1. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D. (eds.) Proceedings on 11th International Conference on Computer Aided Verification, CAV 1999. pp. 274–287. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_25
2. Bruns, G., Godefroid, P.: Generalized model checking: Reasoning about partial state spaces. In: Palamidessi, C. (ed.) Proceedings of the 11th International Conference on Concurrency Theory, CONCUR 2000. pp. 168–182. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_14
3. Bruns, G., Godefroid, P.: Temporal logic query checking. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 409–417. IEEE Computer Society (2001). https://doi.org/10.1109/LICS.2001.932516
4. Bryant, R.E.: A methodology for hardware verification based on logic simulation. J. ACM **38**(2), 299–328 (Apr 1991). https://doi.org/10.1145/103516.103519
5. Bryant, R.E., Seger, C.J.H.: Formal verification of digital circuits using symbolic ternary system models. In: Clarke, E.M., Kurshan, R.P. (eds.) Computer-Aided Verification. pp. 33–43. Springer Berlin Heidelberg, Berlin, Heidelberg (1991). https://doi.org/10.1007/BFb0023717
6. Bryant, R.: Formal verification of memory circuits by switch-level simulation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **10**(1), 94–102 (1991). https://doi.org/10.1109/43.62795
7. Chen, Y., He, Y., Xie, F., Yang, J.: Automatic abstraction refinement for generalized symbolic trajectory evaluation. In: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings. pp. 111–118. IEEE Computer Society (2007). https://doi.org/10.1109/FAMCAD.2007.11
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Proceedings of the 12th International Conference on Computer Aided Verification, CAV 2000. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). https://doi.org/10.1007/10722167_15
9. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (Sep 1994). https://doi.org/10.1145/186025.186051
10. Clarke, E., Gupta, A., Strichman, O.: SAT-based counterexample-guided abstraction refinement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **23**(7), 1113–1123 (2004). https://doi.org/10.1109/TCAD.2004.829807
11. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1979. p. 269–282. Association for Computing Machinery, New York, NY, USA (1979). https://doi.org/10.1145/567752.567778
13. Cranen, S., Groote, J.F., Reniers, M.A.: A linear translation from CTL* to the first-order modal μ-calculus. Theor. Comput. Sci. **412**(28), 3129–3139 (2011). https://doi.org/10.1016/J.TCS.2011.02.034

14. Dam, M.: Fixed points of Büchi automata. In: Shyamasundar, R. (ed.) Foundations of Software Technology and Theoretical Computer Science. pp. 39–50. Springer Berlin Heidelberg, Berlin, Heidelberg (1992). https://doi.org/10.1007/3-540-56287-7_93
15. Dams, D., Grumberg, O.: Abstraction and abstraction refinement. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 385–419. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_13
16. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Interval analysis and machine arithmetic: Why signedness ignorance is bliss. ACM Trans. Program. Lang. Syst. **37**(1) (Jan 2015). https://doi.org/10.1145/2651360
17. Gazda, M., Willemse, T.A.: Expressiveness and completeness in abstraction. Electronic Proceedings in Theoretical Computer Science **89**, 49–64 (Aug 2012). https://doi.org/10.4204/eptcs.89.5
18. Godefroid, P.: May/must abstraction-based software model checking for sound verification and falsification. In: Grumberg, O., Seidl, H., Irlbeck, M. (eds.) Software Systems Safety, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 36, pp. 1–16. IOS Press (2014). https://doi.org/10.3233/978-1-61499-385-8-1
19. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) Proceedings of the 12th International Conference on Concurrency Theory, CONCUR 2001. pp. 426–440. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44685-0_29
20. Godefroid, P., Jagadeesan, R.: Automatic abstraction using generalized model checking. In: Brinksma, E., Larsen, K.G. (eds.) Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002. pp. 137–151. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_11
21. Godefroid, P., Jagadeesan, R.: On the expressiveness of 3-valued models. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2003. pp. 206–222. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-36384-X_18
22. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: Don't Know in the μ-calculus. In: Cousot, R. (ed.) Proceeding of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2005. pp. 233–249. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_16
23. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: When not losing is better than winning: Abstraction and refinement for the full μ-calculus. Information and Computation **205**(8), 1130–1148 (2007). https://doi.org/10.1016/j.ic.2006.10.009
24. Gückel, D.: Synthesis of State Space Generators for Model Checking Microcontroller Code. Dissertation thesis, RWTH Aachen (November 2014), http://aib.informatik.rwth-aachen.de/2014/2014-15.pdf
25. Gückel, D., Kowalewski, S.: Automatic Derivation of Abstract Semantics From Instruction Set Descriptions. In: Brauer, J., Roveri, M., Tews, H. (eds.) 6th International Workshop on Systems Software Verification. Open Access Series in Informatics (OASIcs), vol. 24, pp. 71–83. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2012). https://doi.org/10.4230/OASIcs.SSV.2011.71
26. Gurfinkel, A.: Yasm: Software model-checker, https://www.cs.toronto.edu/~arie/yasm/, retrieved on 2025-01-17.

27. Gurfinkel, A., Chechik, M.: Multi-valued model checking via classical model checking. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003 - Concurrency Theory. pp. 266–280. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45187-7_18
28. Gurfinkel, A., Chechik, M.: Why waste a perfectly good abstraction? In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 212–226. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11691372_14
29. Gurfinkel, A., Wei, O., Chechik, M.: Yasm: A software model-checker for verification and refutation. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification. pp. 170–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11817963_18
30. Gurfinkel, A., Wei, O., Chechik, M.: Model checking recursive programs with exact predicate abstraction. In: Cha, S.S., Choi, J.Y., Kim, M., Lee, I., Viswanathan, M. (eds.) Automated Technology for Verification and Analysis. pp. 95–110. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88387-6_9
31. Konikowska, B., Penczek, W.: Reducing model checking from multi-valued CTL* to CTL*. In: Brim, L., Křetínský, M., Kučera, A., Jančar, P. (eds.) CONCUR 2002 — Concurrency Theory. pp. 226–239. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45694-5_16
32. Melham, T.: Symbolic trajectory evaluation. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 831–870. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_25
33. Microchip Technology Inc.: ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet (October 2018), http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf, dS40002061A
34. Microchip Technology Inc.: AVR Instruction Set Manual (February 2021), https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf, dS40002198B
35. Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with Serval. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 225–242. SOSP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359641
36. Noll, T., Schlich, B.: Delayed nondeterminism in model checking embedded systems assembly code. In: Yorav, K. (ed.) Hardware and Software: Verification and Testing. pp. 185–201. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10/b6jbwx
37. Noll, T., Schlich, B.: Delayed nondeterminism in model checking embedded systems assembly code. In: Yorav, K. (ed.) Proceedings of the 3rd Haifa Verification Conference, HVC 2008. pp. 185–201. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77966-7_16
38. Onderka, J.: Formal verification of machine-code systems by translation of simulable descriptions. In: Proceedings of the 13th Mediterranean Conference on Embedded Computing, MECO 2024. pp. 1–4 (2024). https://doi.org/10.1109/MECO62516.2024.10577942
39. Onderka, J.: Abstraction-Based Machine-Code Program Verification. Doctoral thesis, Czech Technical University in Prague (2025), http://hdl.handle.net/10467/122640

40. Onderka, J., Ratschan, S.: Fast three-valued abstract bit-vector arithmetic. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 242–262. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_12
41. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 27–73. Springer International Publishing, Cham (2018). https://doi.org/10/qdm8
42. Reinbacher, T., Brauer, J., Horauer, M., Schlich, B.: Refining assembly code static analysis for the Intel MCS-51 microcontroller. In: Proceedings of the Fourth IEEE International Symposium on Industrial Embedded Systems, SIES 2009. pp. 161–170 (2009). https://doi.org/10.1109/SIES.2009.5196212
43. Rigorous Engineering of Mainstream Systems Project: Islaris: verification of machine code against authoritative ISA semantics (2023), https://github.com/rems-project/islaris, retrieved 2025-02-23.
44. Sammler, M., Hammond, A., Lepigre, R., Campbell, B., Pichon-Pharabod, J., Dreyer, D., Garg, D., Sewell, P.: Islaris: verification of machine code against authoritative ISA semantics. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 825–840. PLDI 2022, ACM, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523434
45. Schlich, B., Kowalewski, S.: [mc]square: A model checker for microcontroller code. In: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISOLA 2006. pp. 466–473 (2006). https://doi.org/10.1109/ISoLA.2006.62
46. Schlich, B.: Model checking of software for microcontrollers. ACM Transactions on Embedded Computing Systems **9**(4) (April 2010). https://doi.org/10/bm83n7
47. Shoham, S., Grumberg, O.: A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In: Hunt, W.A., Somenzi, F. (eds.) Proceedings of the 15th International Conference on Computer Aided Verification, CAV 2003. pp. 275–287. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_28
48. Shoham, S., Grumberg, O.: Monotonic abstraction-refinement for CTL. In: Jensen, K., Podelski, A. (eds.) Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004. pp. 546–560. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_40
49. Shoham, S., Grumberg, O.: 3-valued abstraction: More precision at less cost. Information and Computation **206**(11), 1313–1333 (2008). https://doi.org/10.1016/j.ic.2008.07.004
50. The UNSAT group: Serval (2019), https://unsat.cs.washington.edu/projects/serval/, retrieved 2025-02-23.
51. Tzoref, R., Grumberg, O.: Automatic refinement and vacuity detection for symbolic trajectory evaluation. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification. pp. 190–204. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11817963_20
52. Wei, O., Gurfinkel, A., Chechik, M.: On the consistency, expressiveness, and precision of partial modeling formalisms. Information and Computation **209**(1), 20–47 (2011). https://doi.org/10.1016/j.ic.2010.08.001
53. Yang, J., Seger, C.J.: Introduction to generalized symbolic trajectory evaluation. In: Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001. pp. 360–365 (2001). https://doi.org/10.1109/ICCD.2001.955052