

# A Configurable CEGAR Framework with Interpolation-Based Refinements

Ákos Hajdu<sup>1,2</sup>, Tamás Tóth<sup>2,\*</sup>, András Vörös<sup>1,2</sup>, and István Majzik<sup>2</sup>

<sup>1</sup> MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

<sup>2</sup> Department of Measurement and Information Systems  
Budapest University of Technology and Economics, Budapest, Hungary  
{hajdua,totht,vori,majzik}@mit.bme.hu

**Abstract.** Correctness of software components in a distributed system is a key issue to ensure overall reliability. Formal verification techniques such as model checking can show design flaws at early stages of development. Abstraction is a key technique for reducing complexity by hiding information, which is not relevant for verification. Counterexample-Guided Abstraction Refinement (CEGAR) is a verification algorithm that starts from a coarse abstraction and refines it iteratively until the proper precision is obtained. Many abstraction types and refinement strategies exist for systems with different characteristics. In this paper we show how these algorithms can be combined into a configurable CEGAR framework. In our framework we also present a new CEGAR configuration based on a combination of abstractions, being able to perform better for certain models. We demonstrate the use of the framework by comparing several configurations of the algorithms on various problems, identifying their advantages and shortcomings.

## 1 Introduction

As critical distributed systems, including safety-critical embedded systems and cloud applications are becoming more and more prevalent, assuring their correct operation is gaining increasing importance. Correctness of software components in a distributed system is a key issue to ensure overall reliability. Formal verification methods such as model checking can show design flaws at early stages of development. However, a typical drawback of using formal verification methods is their high computational complexity. Abstraction is a generic technique for reducing complexity by hiding information which is not relevant for verification. However, finding the proper precision of abstraction is a difficult task. Counterexample-Guided Abstraction Refinement (CEGAR) is an automatic verification algorithm that starts with a coarse abstraction and refines it iteratively until the proper precision is obtained [6]. CEGAR-based algorithms have been successfully applied for both hardware [6], [8] and software [1], [11] verification.

\* This work was partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

CEGAR can be defined for various abstraction types including predicate [6], [13] and explicit value abstraction [1], [8]. There are several refinement strategies as well, many of them being based on Craig [17] or sequence [19] interpolation.

In our paper we describe a configurable CEGAR framework that is able to incorporate both predicate abstraction and explicit value abstraction, along with Craig and sequence interpolation-based refinements. We use this framework to extend predicate abstraction with explicit values at the initial abstraction, producing better results for certain models. We also implemented a prototype of the algorithms and techniques in order to evaluate their performance. In our framework we compare different CEGAR configurations on various (software and hardware) models and identify their advantages and shortcomings.

The rest of the paper is organized as follows. Section 2 introduces the preliminaries of our work. Section 3 presents related work in the field of CEGAR-based model checking. Section 4 describes our framework with our new extension. Section 5 evaluates the algorithms and finally, Section 6 concludes our work.

## 2 Background

This section introduces the preliminaries of our work. First, we present symbolic transition systems as the formalism used in our work (Section 2.1). Then we describe the model checking problem (Section 2.2) and we also introduce interpolation (Section 2.3), a mathematical tool widely used in verification.

### 2.1 Symbolic Transition Systems

In our work we describe models using *symbolic transition systems*, which offer a compact way of representing the set of *states*, *transitions* and *initial states* using first order logic (FOL) variables and formulas. Given a set of variables  $V = \{v_1, v_2, \dots, v_n\}$ , let  $V'$  and  $V_i$  represent the primed and indexed version of the variables, i.e.,  $V' = \{v'_1, v'_2, \dots, v'_n\}$  and  $V_i = \{v_{1,i}, v_{2,i}, \dots, v_{n,i}\}$ . Given a formula  $\varphi$  over  $V$ , let  $\varphi'$  and  $\varphi_i$  denote the formulas obtained by replacing  $V$  with  $V'$  and  $V_i$  in  $\varphi$  respectively, e.g., if  $\varphi = x < y$  then  $\varphi' = x' < y'$  and  $\varphi_2 = x_2 < y_2$ . Given a formula  $\varphi$  over  $V \cup V'$ , let  $\varphi_{i,j}$  denote the formula obtained by replacing  $V$  with  $V_i$  and  $V'$  with  $V_j$  in  $\varphi$ , e.g., if  $\varphi = x' \doteq x + 1$  then  $\varphi_{3,5} = x_5 \doteq x_3 + 1$ . Given a formula  $\varphi$  let  $\text{var}(\varphi)$  denote the set of variables appearing in  $\varphi$ , e.g.,  $\text{var}(x < y + 1) = \{x, y\}$ .

**Definition 1 (Symbolic transition system).** A *symbolic transition system* is a tuple  $T = (V, \text{Inv}, \text{Tran}, \text{Init})$ , where

- $V = \{v_1, v_2, \dots, v_n\}$  is the set of variables with domains  $D_{v_1}, D_{v_2}, \dots, D_{v_n}$ ,
- $\text{Inv}$  is the invariant formula over  $V$ , which must hold for every state,<sup>3</sup>

<sup>3</sup> The invariant formula should not be confused with an invariant property, which is checked whether it holds for every reachable state. The invariant formula only restricts the possible set of states regardless of reachability. For example, an integer variable  $x$  with range  $[2; 5]$  can be defined with domain  $\mathbb{Z}$  and invariant  $2 \leq x \wedge x \leq 5$ .

- *Tran* is the transition formula over  $V \cup V'$ , which describes the transition relation between the actual state ( $V$ ) and the successor state ( $V'$ ),
- *Init* is the initial formula over  $V$ , which defines the set of initial states.

A *concrete state*  $s$  is a (many sorted) interpretation that assigns a value  $s(v_i) = d_i \in D_{v_i}$  to each variable  $v_i \in V$  of its domain  $D_{v_i}$ . A state can also be regarded as a tuple of values  $(d_1, d_2, \dots, d_n)$ . A state with a prime ( $s'$ ) or an index ( $s_i$ ) assigns values to  $V'$  or  $V_i$  respectively. The set of concrete states  $S$ , concrete transitions  $R$  and concrete initial states  $S_0$  (i.e., the *state space*) of a symbolic transition system are defined in the following way.

- $S = \{s \mid s \models \text{Inv}\}$ , i.e.,  $S$  contains all possible interpretations that satisfy the invariant.
- $R = \{(s, s') \mid (s, s') \models \text{Inv} \wedge \text{Tran} \wedge \text{Inv}'\}$ , i.e.,  $s'$  is a successor of  $s$  if assigning  $s$  to the non-primed variables and  $s'$  to the primed variables of the transition formula evaluates to true.
- $S_0 = \{s \mid s \models \text{Inv} \wedge \text{Init}\}$ , i.e.,  $S_0$  is the subset of  $S$  for which the initial formula holds.

A *concrete path* is a (finite, loop-free) sequence of concrete states  $\pi = (s_1, s_2, \dots, s_n)$  for which  $(s_1, s_2, \dots, s_n) \models \text{Init}_1 \wedge \bigwedge_{1 \leq i \leq n} \text{Inv}_i \wedge \bigwedge_{1 \leq i < n} \text{Tran}_{i,i+1}$  holds. In other words, the first state is initial, all states satisfy the invariant and successor states satisfy the transition formula. A concrete state  $s$  is *reachable* if a path  $\pi = (s_1, s_2, \dots, s_n)$  exists with  $s = s_n$  for some  $n$ .

## 2.2 Model Checking

*Model checking* [7] is a formal verification technique to automatically determine whether a system meets a given requirement by explicitly or implicitly analyzing its behaviors (i.e., paths starting from initial states). Requirements are usually given using *temporal logics* [7]. In our work we focus on *safety properties*, where a FOL formula  $\varphi$  is given over  $V$  that must hold for every reachable state. When the system does not meet the safety property, a path  $\pi = (s_1, s_2, \dots, s_n)$  can be found where  $s_n \not\models \varphi_n$ . Such paths are called *counterexamples*.

## 2.3 Interpolation

Craig interpolation is a technique from logic that can produce for two inconsistent formulas an *interpolant*, which generalizes the first formula, while still contradicting the second one. The interpolant can be interpreted as an explanation of the contradiction.

**Definition 2 (Craig interpolant).** *Let  $A$  and  $B$  be FOL formulas such that  $A \wedge B$  is unsatisfiable. The formula  $I$  is a Craig interpolant (or simply an interpolant) for  $A, B$  if the following properties hold [17]:*

- $A$  implies  $I$ ,

- $I \wedge B$  is unsatisfiable,
- $I$  only contains symbols common in  $A$  and  $B$  (excluding symbols of the logic).

William Craig showed that an interpolant always exists for FOL formulas  $A$  and  $B$  with at least one symbol in common and  $A \wedge B$  being unsatisfiable [9].

Interpolation can be generalized from two formulas to a sequence of formulas, for which an *interpolation sequence* is calculated instead of a single interpolant.

**Definition 3 (Interpolation sequence).** Let  $A_1, A_2, \dots, A_n$  be a sequence of FOL formulas such that  $A_1 \wedge A_2 \wedge \dots \wedge A_n$  is unsatisfiable. The sequence of formulas  $I_0, I_1, \dots, I_n$  is an *interpolation sequence* for  $A_1, A_2, \dots, A_n$  if the following properties hold [19]:

- $I_0 = \top$ ,  $I_n = \perp$ ,
- $I_j \wedge A_{j+1}$  implies  $I_{j+1}$  for  $0 \leq j < n$ ,
- $I_j$  only contains symbols common in  $A_1, \dots, A_j$  and  $A_{j+1}, \dots, A_n$  for  $0 < j < n$  (excluding symbols of the logic).

### 3 Related Work and Contributions

*Counterexample-Guided Abstraction Refinement* (CEGAR) is a widely used abstraction-based approach to tackle the complexity of real-life software and hardware systems [6]. CEGAR-based algorithms usually have the following four main steps.

1. The first step is to create an abstract model that over-approximates the concrete model and is easier to handle computationally.
2. The abstract model is then checked by a model checking algorithm. Due to the behavior of over-approximation, if the abstract model satisfies the requirement, then it also holds in the concrete model.<sup>4</sup>
3. On the other hand, if the abstract model violates the requirement, an abstract counterexample is produced by the model checker. The third step is to check the feasibility of the abstract counterexample in the concrete model. If a concrete counterexample exists, it is a witness that the original model also violates the requirement.
4. If the abstract counterexample is not feasible, the abstraction has to be refined and the process has to be repeated from Step 2, until either the requirement holds for the abstract model or a concrete counterexample is found.

*Types of Abstraction.* CEGAR can work with different types of abstractions, including *predicate abstraction* [13] and *explicit value abstraction* [8]. There has also been work on a combination of the former two approaches for configurable program analysis [2]. We also propose a combination of predicate abstraction and explicit values at the initial abstraction, but instead of program analysis, we focus on symbolic transition systems (Section 4.1).

<sup>4</sup> This relation holds for ACTL\* properties [6], including safety properties.

*Refinement Strategies.* Interpolation is often used to infer new predicates that refine the abstraction. Craig interpolation yields a single predicate [4], [14], while its extension, sequence interpolation produces a sequence of predicates [1], [11], [16]. Our approach is similar to the one presented in [11], however in our framework the initial abstraction can be defined by arbitrary predicates and explicit variables (Section 4.4). As a special case, choosing the program counter as the only explicit variable yields a similar approach to the one presented in [11].

*Contributions.* In our work we make the following novel contributions. (1) We describe a *CEGAR framework for symbolic transition systems*, where refinement is based on splitting abstract states. We show that both predicate abstraction and explicit value abstraction can be incorporated into this framework along with Craig and sequence interpolation-based refinement strategies. This allows us to experiment with several algorithm configurations and their extensions. (2) As a first result, we used this framework to develop a *new configuration of CEGAR* that extends predicate abstraction with explicit values at the initial abstraction based on domain knowledge or heuristics. (3) We also use this framework to *evaluate different CEGAR configurations* (including our extended one) on various models, including industrial PLC codes and hardware.

In the following section, we present the CEGAR framework with our new configuration, this way also discussing the integration of the different algorithmic components.

## 4 A Configurable CEGAR Framework

This section presents the steps of our configurable CEGAR framework: initial abstraction (Section 4.1), model checking (Section 4.2) with an incremental optimization (Section 4.5), counterexample concretization (Section 4.3) and abstraction refinement (Section 4.4).

### 4.1 Initial Abstraction

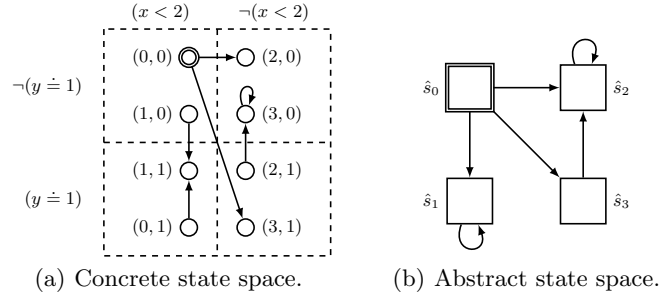
The algorithms are based on the *existential abstraction* framework of Clarke et al. [6], *predicate abstraction* [13] and *explicit-value abstraction* [8].

**Predicate Abstraction.** Predicate abstraction maps concrete states to abstract states based on their evaluation on a set of FOL predicates. Given a symbolic transition system  $T = (V, Inv, Tran, Init)$  and a set of FOL predicates  $\mathcal{P}$  over  $V$ , there are  $2^{|\mathcal{P}|}$  possible *abstract states*, denoted by  $\hat{S}$ . An abstract state  $\hat{s} \in \hat{S}$  is a set of predicates, where for each  $p_i \in \mathcal{P}$ ,  $\hat{s}$  contains either  $p_i$  or  $\neg p_i$ . Given an abstract state  $\hat{s} \in \hat{S}$ , let its label be  $Label(\hat{s}) = \bigwedge_{p \in \hat{s}} p$ , i.e., the conjunction of predicates (or their negations) in  $\hat{s}$ . A concrete state  $s$  is mapped to  $\hat{s}$  if  $s \models Label(\hat{s})$ .

In existential abstraction the *abstract transition relation*  $\hat{R}$  and the set of *abstract initial states*  $\hat{S}_0$  are defined in the following way [6].

- $\hat{R} = \{(\hat{s}, \hat{s}') \in \hat{S} \times \hat{S} \mid \exists s, s'. (s, s') \models \text{Inv} \wedge \text{Inv}' \wedge \text{Label}(\hat{s}) \wedge \text{Label}(\hat{s}')' \wedge \text{Tran}\}$ , i.e., concrete successor states  $(s, s')$  exist, with  $s$  mapped to  $\hat{s}$  and  $s'$  to  $\hat{s}'$ .
- $\hat{S}_0 = \{\hat{s} \in \hat{S} \mid \exists s. s \models \text{Inv} \wedge \text{Init} \wedge \text{Label}(\hat{s})\}$ , i.e., a concrete initial state  $s$  exists, which is mapped to  $\hat{s}$ .

*Example 1.* Consider a symbolic transition system  $T$  with  $V = \{x, y\}$ ,  $D_x = D_y = \mathbb{Z}$ ,  $\text{Inv} = (0 \leq x \wedge x \leq 3 \wedge 0 \leq y \wedge y \leq 1)$ ,  $\text{Init} = (x \doteq 0 \wedge y \doteq 0)$  and  $\text{Tran} = (x + y \doteq 0 \wedge x' - y' \doteq 2) \vee (x + y \doteq 1 \wedge x' \doteq 1 \wedge y' \doteq 1) \vee (x + y \doteq 3 \wedge x' \doteq 3 \wedge y' \doteq 0)$ . The concrete state space of  $T$  can be seen in Figure 1(a), where circles denote concrete states  $(x, y)$ , the double circle denotes the initial state and edges denote transitions. Suppose, that  $\mathcal{P} = \{x < 2, y \doteq 1\}$ , which means that there are  $2^{|\mathcal{P}|} = 4$  abstract states. Partitioning by  $\mathcal{P}$  is indicated by dashed lines in Figure 1(a), while the corresponding abstract transition system  $(\hat{S}, \hat{R}, \hat{S}_0)$  can be seen in Figure 1(b).



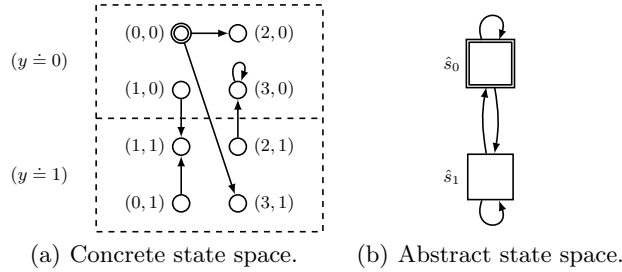
**Fig. 1.** Predicate abstraction example.

An *abstract path* is a (finite, loop-free) sequence of abstract states  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$  with  $\hat{s}_1 \in \hat{S}_0$  and  $(\hat{s}_i, \hat{s}_{i+1}) \in \hat{R}$  ( $1 \leq i < n$ ). An abstract path  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$  is *concretizable* if a sequence of states  $\pi = (s_1, s_2, \dots, s_n)$  exists for which  $(s_1, s_2, \dots, s_n) \models \text{Init}_1 \wedge \bigwedge_{1 \leq i \leq n} \text{Label}(\hat{s}_i)_i \wedge \bigwedge_{1 \leq i \leq n} \text{Inv}_i \wedge \bigwedge_{1 \leq i < n} \text{Tran}_{i, i+1}$ . In other words,  $\pi$  is a concrete path where the  $i$ th concrete state is mapped to the  $i$ th abstract state.

**Explicit Value Abstraction.** In explicit value abstraction, the variables  $V$  of the system are divided into two disjoint sets: *visible* ( $V_V$ ) and *invisible* ( $V_I$ ) sets of variables. Concrete states are mapped to abstract states based on their evaluation on visible variables. Given a symbolic transition system  $T = (V, \text{Inv}, \text{Tran}, \text{Init})$  and the set of visible variables  $V_V \subseteq V$ , there are  $\prod_{v_i \in V_V} |D_{v_i}|$  possible abstract states, denoted by  $\hat{S}$ . An abstract state  $\hat{s} \in \hat{S}$  is a (many sorted) interpretation that assigns a value  $\hat{s}(v_i) = d_i \in D_{v_i}$  to each visible variable  $v_i \in V_V$  of its domain  $D_{v_i}$ . A concrete state  $s$  is mapped to  $\hat{s}$  if  $s(v_i) = \hat{s}(v_i)$  for each visible variable  $v_i \in V_V$ . The label of an abstract state  $\hat{s}$  in explicit value

abstraction can be defined by  $Label(\hat{s}) = \bigwedge_{v_i \in V_V} (v_i \doteq \hat{s}(v_i))$ , i.e., a conjunction of the assignments. Transitions and initial states are mapped as in predicate abstraction.

*Example 2.* Recall the symbolic transition system of Example 1 and suppose, that  $V_V = \{y\}$ ,  $V_I = \{x\}$ . The concrete state space and the partitioning by  $V_V$  is indicated in Figure 2(a), while the corresponding abstract transition system  $(\hat{S}, \hat{R}, \hat{S}_0)$  can be seen in Figure 2(b).



**Fig. 2.** Explicit value abstraction example.

**Extending Predicate Abstraction with Explicit Values (Combined Abstraction).** We observed that both abstraction types have advantages and shortcomings. For example, a variable with an infinite domain cannot be tracked explicitly. On the other hand, a variable appearing in different equalities (e.g.,  $x \doteq 1, x \doteq 2, \dots$ ) may yield a handful of predicates and refinement iterations. In such cases it is more efficient to keep track of the variable explicitly. Therefore, we also developed a combined method that extends predicate abstraction with explicit values when creating the initial abstract model. Formally, let  $T = (V, Inv, Tran, Init)$  be a symbolic transition system with variables  $V = \{v_1, v_2, \dots, v_n\}$ ,  $\mathcal{P}$  be a set of FOL predicates over  $V$  and  $V_E \subseteq V$  be the set of *explicit variables*. Without the loss of generality, in the following it is assumed that explicit variables are represented by the first  $k$  indices ( $0 \leq k \leq n$ ), i.e.,  $V_E = \{v_1, v_2, \dots, v_k\}$ . We combine predicate abstraction with explicit values in the following way. An abstract state  $\hat{s} \in \hat{S}$  is a set of predicates, where

- for each  $p_i \in \mathcal{P}$ ,  $\hat{s}$  contains either  $p_i$  or  $\neg p_i$ ,
- for each  $v_i \in V_E$ ,  $\hat{s}$  contains a predicate of the form  $v_i \doteq d_i$ , where  $d_i \in D_{v_i}$ .

Consequently, there are  $|\hat{S}| = 2^{|\mathcal{P}|} \cdot |D_{v_1}| \cdot |D_{v_2}| \cdot \dots \cdot |D_{v_k}|$  possible abstract states. The abstract transition relation  $\hat{R}$  and the initial states  $\hat{S}_0$  can be calculated similarly to predicate abstraction. The initial set of predicates and explicit values can be determined by domain knowledge or by simple heuristics (see Section 5).

*Example 3.* Suppose, that  $V = \{x, y\}$  with  $D_x = D_y = \{0, 1\}$ , the only predicate is  $\mathcal{P} = \{x < y\}$  and the only explicit variable is  $V_E = \{x\}$ . There are thus four abstract states  $\hat{s}_1 = \{x < y, x \doteq 0\}$ ,  $\hat{s}_2 = \{x < y, x \doteq 1\}$ ,  $\hat{s}_3 = \{\neg(x < y), x \doteq 0\}$  and  $\hat{s}_4 = \{\neg(x < y), x \doteq 1\}$ .

## 4.2 Model Checking

An abstract state  $\hat{s} \in \hat{S}$  violates the safety property  $\varphi$  if  $\text{Label}(\hat{s}) \wedge \text{Inv} \wedge \neg\varphi$  is satisfiable, i.e., a concrete state exists, which is mapped to  $\hat{s}$  but violates  $\varphi$ . The model checking problem on the abstract transition system is to check if an abstract state  $\hat{s}$  violating  $\varphi$  is reachable, i.e., whether an abstract path  $\hat{\varphi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$  exists with  $\hat{s}_n = \hat{s}$ .

*Example 4.* Recall Example 1 and suppose that the safety property is  $\varphi = (x \neq 3 \vee y \neq 0)$ , i.e., only the concrete state  $(3, 0)$  violates  $\varphi$ . Consequently,  $\hat{s}_2$  also violates  $\varphi$  and the paths  $\hat{\pi}_1 = (\hat{s}_0, \hat{s}_3, \hat{s}_2)$  and  $\hat{\pi}_2 = (\hat{s}_0, \hat{s}_2)$  are abstract counterexamples.

CEGAR can work with different kinds of model checkers as long as they are capable of providing a counterexample. Our framework is currently equipped with an incremental explicit model checker. Incrementality relies on the refinement strategy (Section 4.4), therefore it is presented afterwards (Section 4.5).

## 4.3 Counterexample Concretization

An abstract counterexample  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$  for the safety property  $\varphi$  is *concretizable* if a sequence of states  $\pi = (s_1, s_2, \dots, s_n)$  exists for which  $(s_1, s_2, \dots, s_n) \models \text{Init}_1 \wedge \bigwedge_{1 \leq i \leq n} \text{Label}(\hat{s}_i)_i \wedge \bigwedge_{1 \leq i \leq n} \text{Inv}_i \wedge \bigwedge_{1 \leq i < n} \text{Tran}_{i,i+1} \wedge \neg\varphi_n$  holds. In other words,  $\hat{\pi}$  is concretizable as a path and in addition the last state violates the safety property. A concretizable counterexample is a witness that the concrete model also violates the requirement, while a non-concretizable counterexample is called *spurious*.

In order to avoid finding the spurious counterexample again, the abstraction has to be refined. The longest concretizable prefix of the counterexample provides useful information for the refinement. Therefore, an abstract counterexample  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$  is concretized iteratively with the following  $n + 1$  formulas.

$$F_i = \begin{cases} \text{Init}_1 \wedge \text{Inv}_1 \wedge \text{Label}(\hat{s}_1)_1 & \text{if } i = 1 \\ \text{Inv}_i \wedge \text{Label}(\hat{s}_i)_i \wedge \text{Tran}_{i-1,i} & \text{if } 1 < i \leq n \\ \neg\varphi_n & \text{if } i = n + 1 \end{cases}$$

The formula  $F_1 \wedge F_2 \wedge \dots \wedge F_n$  describes concrete paths mapped to  $\hat{\pi}$  (similarly to bounded model checking [3]), while  $F_{n+1}$  ensures that the last state violates the property. If  $F_1 \wedge F_2 \wedge \dots \wedge F_{n+1}$  is satisfiable, the counterexample is concretizable. Otherwise, let  $1 \leq f \leq n$  be the largest index for which  $F_1 \wedge F_2 \wedge \dots \wedge F_f$  is satisfiable. The state  $\hat{s}_f$  is then called the *failure state* since a concrete path leads there but it cannot be extended.



*Example 5.* Recall Example 4 with the abstract counterexamples  $\hat{\pi}_1 = (\hat{s}_0, \hat{s}_3, \hat{s}_2)$  and  $\hat{\pi}_2 = (\hat{s}_0, \hat{s}_2)$ . It can be seen that  $\hat{\pi}_1$  is spurious since  $\hat{s}_2$  cannot be reached by a concrete path. The longest concretizable prefix is  $(\hat{s}_0, \hat{s}_3)$ , hence the failure state is  $\hat{s}_3$ . The abstract counterexample  $\hat{\pi}_2$  is concretizable as a path with  $((0, 0), (2, 0))$ , but  $(2, 0)$  fulfills the property, thus the failure state is  $\hat{s}_2$ .

#### 4.4 Abstraction Refinement

The set of concrete states mapped to the failure state  $\hat{s}_f$  are partitioned into the following three groups: states that can be reached from an initial state are *dead-end*, states having a transition to  $\hat{s}_{f+1}$  or violating  $\varphi$  are *bad*, while other states are *irrelevant*. It is clear that a state cannot be dead-end and bad at the same time since then  $\hat{s}_f$  would not be a failure state [6].

*Example 6.* Recall Example 5 and Figure 1 with  $\hat{\pi}_1 = (\hat{s}_0, \hat{s}_3, \hat{s}_2)$  and  $\hat{\pi}_2 = (\hat{s}_0, \hat{s}_2)$ . The failure state of  $\hat{\pi}_1$  is  $\hat{s}_3$ , where  $(3, 1)$  is a dead-end state and  $(2, 1)$  is bad. The failure state of  $\hat{\pi}_2$  is  $\hat{s}_2$ , where  $(2, 0)$  is dead-end and  $(3, 0)$  is bad.

The purpose of abstraction refinement is to map dead-end and bad states to different abstract states so that the spurious counterexample cannot occur in the next iteration. Predicate abstraction and our combined method uses predicate refinement to obtain new predicates, while explicit value abstraction employs explicit value refinement to make some previously invisible variables visible.

**Predicate Refinement.** Our framework supports both Craig and sequence interpolation to infer new predicates and it also utilizes lazy abstraction, i.e., only a subset of the abstract states is refined.

*Craig Interpolation.* Dead-end and bad states can be characterized with formulas  $D$  and  $B$  respectively in the following way.

$$D = Init_1 \wedge \bigwedge_{1 \leq i \leq f} Inv_i \wedge \bigwedge_{1 \leq i \leq f} Label(\hat{s}_i)_i \wedge \bigwedge_{1 \leq i < f} Tran_{i,i+1}$$

$$B = \begin{cases} Inv_{f+1} \wedge Label(\hat{s}_{f+1})_{f+1} \wedge Tran_{f,f+1} & \text{if } f < n \\ \neg \varphi_n & \text{if } f = n \end{cases}$$

In other words,  $D$  describes paths mapped to the prefix  $(\hat{s}_1, \hat{s}_2, \dots, \hat{s}_f)$ , while  $B$  describes either transitions from  $\hat{s}_f$  to  $\hat{s}_{f+1}$  or states violating  $\varphi$ . It is clear that  $D \wedge B$  is unsatisfiable, otherwise a longer prefix could be found or  $\hat{\pi}$  would be concretizable. Consequently, Craig interpolation can be applied, yielding an interpolant  $I$  with the following properties.

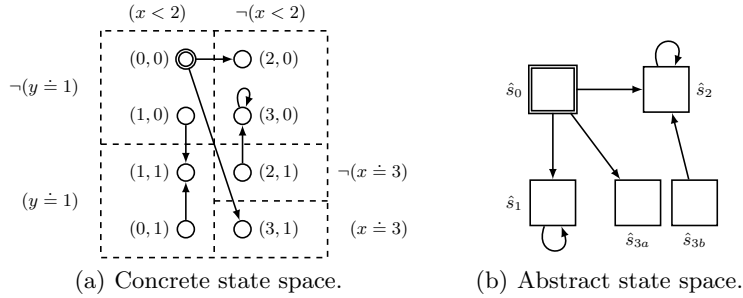
- $D \Rightarrow I$ , i.e.,  $I$  is a generalization of dead-end states,
- $I \wedge B$  is unsatisfiable, i.e., bad states cannot satisfy  $I$ ,
- $I$  refers to common symbols of  $D$  and  $B$ , which are variables with index  $f$ .

Therefore, removing the indices from the variables in  $I$  yields a new predicate that separates dead-end and bad states mapped to  $\hat{s}_f$ . We refine the abstraction by replacing  $\hat{s}_f$  with  $\hat{s}_{f1}$  and  $\hat{s}_{f2}$  obtained by adding  $I$  and  $\neg I$  to the predicates of  $\hat{s}_f$ , i.e.,  $\hat{s}_{f1} = \hat{s}_f \cup \{I\}$  and  $\hat{s}_{f2} = \hat{s}_f \cup \{\neg I\}$ . This approach, namely splitting only a subset of states is similar to *lazy abstraction* [15].

*Example 7.* Recall Example 6 and the spurious counterexample  $\hat{\pi}_1 = (\hat{s}_0, \hat{s}_3, \hat{s}_2)$ , where  $\hat{s}_3$  is the failure state. Thus,  $D$  and  $B$  are defined in the following way for Craig interpolation.

- $D = \text{Init}_0 \wedge \text{Inv}_0 \wedge \text{Inv}_3 \wedge \text{Label}(\hat{s}_0)_0 \wedge \text{Label}(\hat{s}_3)_3 \wedge \text{Tran}_{0,3}$ ,
- $B = \text{Inv}_2 \wedge \text{Label}(\hat{s}_2)_2 \wedge \text{Tran}_{3,2}$ .

The formula  $I = (x_3 \doteq 3)$  is an interpolant for  $D$  and  $B$ , which can be used to split  $\hat{s}_3$  (Figure 3(a)). The refined abstract state space can be seen in Figure 3(b), where the spurious behavior of  $\hat{\pi}_1$  is eliminated. However,  $\hat{\pi}_2 = (\hat{s}_0, \hat{s}_2)$  is still a spurious counterexample that needs another refinement iteration.



**Fig. 3.** Predicate refinement example with Craig interpolation.

*Sequence Interpolation.* Craig interpolation can be generalized to sequence interpolation [11] in order to split multiple states along the spurious counterexample  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$ . Formally,  $A_1, A_2, \dots, A_{n+1}$  is defined in the following way.

$$A_i = \begin{cases} \text{Init}_1 \wedge \text{Inv}_1 \wedge \text{Label}(\hat{s}_1)_1 & \text{if } i = 1 \\ \text{Inv}_i \wedge \text{Label}(\hat{s}_i)_i \wedge \text{Tran}_{i-1,i} & \text{if } 1 < i \leq n \\ \neg \varphi_n & \text{if } i = n + 1 \end{cases}$$

In other words, the formula  $A_1$  describes initial states mapped to  $\hat{s}_1$ , while  $A_2, A_3, \dots, A_n$  describe reachable states mapped to  $\hat{s}_2, \hat{s}_3, \dots, \hat{s}_n$  respectively. Finally,  $A_{n+1}$  describes states violating the safety property. It is clear that  $A_1 \wedge A_2 \wedge \dots \wedge A_{n+1}$  is unsatisfiable, since  $\hat{\pi}$  is spurious. Hence, an interpolation sequence  $I_0, I_1, \dots, I_{n+1}$  exists with the following properties:

- $I_0 = \top$ ,  $I_{n+1} = \perp$ , i.e., interpolants that do not correspond to any state in the counterexample carry no information,
- $I_j \wedge A_{j+1} \Rightarrow I_{j+1}$  for  $0 \leq j \leq n$ , i.e., the interpolants together generalize dead-end states and contradict bad states,
- $I_j$  refers only to the common symbols of  $A_1, \dots, A_j$  and  $A_{j+1}, \dots, A_{n+1}$ , i.e., variables with index  $j$ .

Abstraction is refined by replacing each  $\hat{s}_i$  ( $1 \leq i \leq n$ ) with  $\hat{s}_{i1}$  and  $\hat{s}_{i2}$  obtained by adding  $I_i$  and  $\neg I_i$  to the predicates of  $\hat{s}_i$  respectively. Formally,  $\hat{s}_{i1} = \hat{s}_i \cup \{I_i\}$  and  $\hat{s}_{i2} = \hat{s}_i \cup \{\neg I_i\}$ . It may occur that  $I_i = \top$  or  $I_i = \perp$  for some  $1 \leq i \leq n$ . In this case the corresponding abstract state  $\hat{s}_i$  is not split.

The motivation behind sequence interpolation is twofold. On the one hand, splitting multiple states in a single step can eliminate more spurious behavior, yielding fewer refinement iterations. On the other hand, we observed that separating dead-end and bad states with a single formula (Craig interpolant) may render the formula long and complex. Sequence interpolation in contrast, can produce more, but less complex formulas. Furthermore, it also makes concretization easier, since the failure state  $\hat{s}_f$  does not have to be determined.

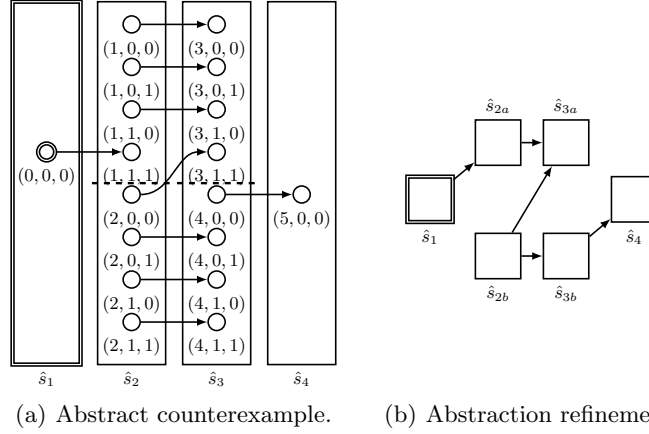
*Example 8.* As an example, consider a symbolic transition system with variables  $V = \{x, y, z\}$ . Suppose, that the safety property is  $\varphi = x \neq 5$ , for which the abstract counterexample  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4)$  shown in Figure 4(a) is produced by the model checker. It can be seen that  $\hat{\pi}$  is spurious, since  $(5, 0, 0)$  cannot be reached from  $(0, 0, 0)$ . Therefore,  $A_1, A_2, \dots, A_5$  is defined in the following way:

- $A_1 = \text{Init}_1 \wedge \text{Inv}_1 \wedge \text{Label}(\hat{s}_1)_1$ ,
- $A_2 = \text{Inv}_2 \wedge \text{Label}(\hat{s}_2)_2 \wedge \text{Tran}_{1,2}$ ,
- $A_3 = \text{Inv}_3 \wedge \text{Label}(\hat{s}_3)_3 \wedge \text{Tran}_{2,3}$ ,
- $A_4 = \text{Inv}_4 \wedge \text{Label}(\hat{s}_4)_4 \wedge \text{Tran}_{3,4}$ ,
- $A_5 = \neg \varphi_4$ .

It can be checked that  $I_0 = \top$ ,  $I_1 = \top$ ,  $I_2 = (x_2 < 2)$ ,  $I_3 = (x_3 < 4)$ ,  $I_4 = \perp$ ,  $I_5 = \perp$  is an interpolation sequence for  $A_1, A_2, \dots, A_5$ . Hence,  $\hat{s}_1$  and  $\hat{s}_4$  are not split,  $\hat{s}_2$  is split with the predicate  $(x < 2)$  and  $\hat{s}_3$  with  $(x < 4)$  as the dashed lines indicate. The abstract states after the refinement can be seen in Figure 4(b). It is clear that the spurious counterexample is eliminated. It can also be seen that both splits are required.

Suppose now, that Craig interpolation is applied for the same problem. The failure state is  $\hat{s}_2$ , where  $(1, 1, 1)$  is a dead-end state and all the others are bad. Therefore,  $(1, 1, 1)$  has to be separated from the others with a single formula. This requires all three variables (e.g.,  $I = (x_2 \doteq 1 \wedge y_2 \doteq 1 \wedge z_2 \doteq 1)$ ), since  $(1, 1, 1)$  is not distinguishable with two or less variables. In contrast, sequence interpolation could be solved with two predicates containing only  $x$ .

**Explicit Value Refinement.** As in predicate abstraction, the purpose of refinement is to map dead-end and bad states to different abstract states. In pure



**Fig. 4.** Predicate refinement example with sequence interpolation.

explicit value analysis this can be done by making a subset  $V'_I \subseteq V_I$  of the previously invisible variables visible, i.e.,  $V_V \leftarrow V_V \cup V'_I$  and  $V_I \leftarrow V_I \setminus V'_I$  [8]. In contrast to predicate abstraction, visible variables are common for each state, which means that each abstract state is split in the new iteration [8].<sup>5</sup> In our framework we generate  $V'_I$  with interpolation in the following way. Recall that we defined the label of an abstract state in explicit value abstraction as a conjunction of assignments ( $Label(\hat{s}) = \bigwedge_{v_i \in V_V} (v_i \doteq \hat{s}(v_i))$ ). Thus, for a spurious counterexample  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$  a Craig interpolant  $I$  or an interpolation sequence  $I_0, I_1, \dots, I_{n+1}$  can be calculated in the same way as in predicate refinement. Then  $V'_I = \text{var}(I) \cap V_I$  with Craig interpolation, or  $V'_I = \bigcup_{1 \leq i \leq n} \text{var}(I_i) \cap V_I$  with sequence interpolation. In other words, new visible variables are the invisible variables appearing in the interpolants. Again, sequence interpolation can generate simpler formulas, keeping  $V'_I$  (and thus, the abstract state space) smaller.

The reason for a spurious abstract counterexample is that dead-end and bad states are mapped to the same abstract state  $\hat{s}_f$ , where  $\hat{s}_f$  assigns the same values to visible variables  $V_V$ . Interpolants distinguish dead-end states and bad states, which means that they must contain at least one invisible variable. This ensures that  $V'_I \neq \emptyset$  and that the spurious counterexample is eliminated.

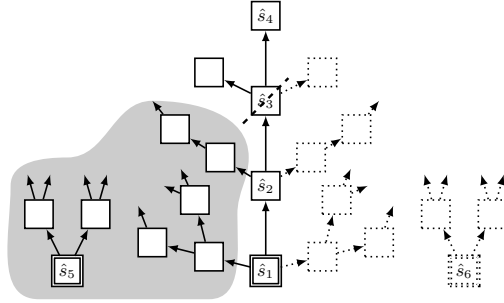
#### 4.5 Incremental Model Checking

A non-incremental explicit model checker loops through each initial abstract state and traverses the set of reachable abstract states using for example depth-first search. If an abstract state violating the safety property is found, the actual abstract path is returned. Our incremental model checker exploits the fact that only a subset of the abstract states are split when using predicate refinement

<sup>5</sup> This splitting is of course, not performed explicitly. The model checker constructs the state space on-the-fly.

(see Section 4.4). Let  $\hat{s}_s$  denote the first state of the abstract counterexample  $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n)$  that was split in the previous iteration, which is the failure state  $\hat{s}_f$  using Craig interpolation or the state with the lowest index  $s$  such that  $I_s \neq \top$  and  $I_s \neq \perp$  using sequence interpolation.

The main idea of our incremental approach is presented in Figure 5. The path  $(\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4)$  represents the actual abstract counterexample, where  $\hat{s}_3$  was the first abstract state to be split. Each state has some successors that were already fully explored (drawn on the left side of the state) and also some successors yet to be explored (drawn on the right side). There can also be abstract initial states that were already fully explored ( $\hat{s}_5$  in the figure) and abstract initial states that will be explored after  $\hat{s}_1$  ( $\hat{s}_6$  in the figure). Abstract states in the gray area were fully explored before  $\hat{s}_3$ . Let this set be denoted by  $\hat{G}$ . It is clear that  $\hat{s}_3$  can only be reached from  $\hat{G}$  through  $\hat{s}_2$ . Otherwise,  $\hat{s}_3$  would first be reached that way and not through  $\hat{s}_2$ . Therefore, splitting  $\hat{s}_3$  does not affect states in  $\hat{G}$ . If exploration is continued with  $(\hat{s}_1, \hat{s}_2)$  on the stack,  $\hat{s}_2$  will “represent” states in  $\hat{G}$ , i.e., if some of the new abstract states could be reached from  $\hat{G}$ , they will be reached from  $\hat{s}_2$ . Therefore, if  $\hat{s}_s$  is the first abstract state that was split ( $\hat{s}_3$  in the example), abstract states explored before  $\hat{s}_s$  do not need to be re-explored and the actual abstract path can be kept until  $\hat{s}_{s-1}$ .



**Fig. 5.** Illustration of incremental model checking.

It may seem that incremental model checking requires extra memory to store the explored states. However, a non-incremental version also has to discover and keep track of the same states. The only difference is that the incremental version keeps the explored states in memory between the refinement iterations and continues the search, while the non-incremental version always re-explores.

## 5 Evaluation

We developed a prototype Java implementation for the framework presented in Section 4. We used Z3 [18] as the underlying logic solver. We compared various configurations on industrial PLC codes (Section 5.1), on a protocol with infinite state space (Section 5.2) and on hardware models (Section 5.3).

## 5.1 Industrial PLC Codes

Programmable Logic Controller (PLC) codes can be represented by an automaton-based model [12], which can then be translated into a symbolic transition system. Table 1 contains results for the following six configurations, corresponding to the main columns: (1) predicate abstraction with Craig interpolation, (2) predicate abstraction with sequence interpolation, (3) combined abstraction with Craig interpolation, (4) combined abstraction with sequence interpolation, (5) explicit value abstraction with Craig interpolation, (6) explicit value abstraction with sequence interpolation.

In predicate abstraction the initial set of predicates is empty, while in explicit value abstraction the initial visible variables are those appearing in the safety property. We observed that the program location variable appears in many equality formulas, e.g.,  $loc \doteq 0, loc \doteq 1, \dots, loc \doteq n$ . With this extra knowledge, we configured the combined approach to track the location variable explicitly and to start with an empty set of initial predicates. Note, that this idea can be generalized to any automaton-based model or such variables can also be detected by a heuristic that analyzes the formulas.

The sub-columns T, #R and #S represent the run time in seconds, the number of refinements and the sum of explored abstract states in each iteration respectively. The  $\checkmark$  or  $\times$  sign before the name of the model indicates whether it meets the property or not. The columns V and L denote the number of variables and locations in the automaton-based model respectively. The shortest run time is indicated by bold font for each model.

It can be seen that explicit value abstraction has the best performance for many models. However, predicate abstraction has shorter run time for models PLC01 and PLC02 (where no reductions were applied to the automata) and the combined approach performs best for models with the largest state space (PLC06 and PLC08). It can also be observed that the combined approach gives a better performance for most of the models compared to pure predicate abstraction. Furthermore, it can be seen that Craig interpolation yields many small steps (many iterations), in contrast to sequence interpolation, which performs fewer, but bigger steps.

## 5.2 Fischer’s Protocol

Fischer’s protocol [10] is a mutual exclusion algorithm for arbitrarily many components (column #C). The model contains clock variables (with domain  $\mathbb{Q}$ ), rendering the state space infinite. Explicit value abstraction fails to verify these models because the clock variables become visible after a few iterations. Table 2 contains results for predicate and combined abstraction. The algorithms start with an empty set of initial states and the combined approach tracks the variables corresponding to the locks explicitly. It can be seen that Craig interpolation outperforms sequence interpolation for these models. It can also be observed that the combined method is more efficient when the model meets the property.

**Table 1.** Measurement results for PLC codes.

			Pred. (Cr.)			Pred. (seq.)			Comb. (Cr.)			Comb. (seq.)		
Model	V	L	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S
× PLC01	66	36	<b>22.5</b>	33	100	50.2	34	191	42.0	20	452	48.5	1	81
× PLC02	66	36	<b>22.7</b>	33	100	49.4	34	191	41.0	20	452	47.3	1	81
✓ PLC03	29	17	479.2	195	6694	99.2	23	292	28.2	34	629	51.8	6	212
✓ PLC04	29	17	40.2	64	1076	14.4	16	82	17.6	21	353	6.1	2	47
× PLC04	29	17	44.0	65	1069	406.7	31	1198	34.3	35	650	36.1	5	192
✓ PLC05	29	17	42.2	63	1130	21.4	17	98	17.4	21	352	6.3	2	47
✓ PLC06	82	43	1512.8	159	4812	—	—	—	333.1	52	1369	<b>227.5</b>	2	120
✓ PLC07	82	43	190.8	58	552	462.2	66	1057	164.8	26	657	164.8	1	70
× PLC08	82	43	86.1	37	111	—	—	—	46.7	0	43	<b>46.2</b>	0	43
✓ PLC09	23	14	87.4	90	1716	94.6	32	633	61.3	94	1845	35.7	11	193
			Expl. (Cr.)			Expl. (Seq.)								
Model	V	L	T (s)	#R	#S	T (s)	#R	#S						
× PLC01	66	36	36.4	7	1640	211.8	3	758						
× PLC02	66	36	32.2	7	1697	428.5	5	1439						
✓ PLC03	29	17	<b>5.2</b>	1	339	9.9	1	369						
✓ PLC04	29	17	<b>3.3</b>	1	165	3.8	1	165						
× PLC04	29	17	<b>7.6</b>	2	274	38.0	1	209						
✓ PLC05	29	17	<b>3.5</b>	1	167	4.7	1	167						
✓ PLC06	82	43	1254.5	3	20956	—	—	—						
✓ PLC07	82	43	78.1	2	1163	<b>50.9</b>	1	518						
× PLC08	82	43	65.1	2	628	123.0	3	541						
✓ PLC09	23	14	<b>11.8</b>	5	1261	14.5	4	833						

**Table 2.** Measurement results for Fischer’s protocol.

		Pred. (Cr.)			Pred. (seq.)			Comb. (Cr.)			Comb. (seq.)		
Model	#C	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S
✓ fischer	2	1.2	17	69	3.0	15	107	<b>0.8</b>	18	66	1.2	14	78
× fischer	2	<b>0.6</b>	11	41	1.1	9	45	0.8	18	62	1.2	12	58
✓ fischer	3	12.1	97	998	68.1	101	1584	<b>10.3</b>	93	1329	45.8	99	1334
× fischer	3	<b>1.4</b>	19	70	1.5	9	44	1.7	28	121	2.9	21	105

### 5.3 Hardware Models

We also evaluated the algorithms for some of the smaller models of the Hardware Model Checking Competition [5]. Table 3 contains results for predicate abstraction and explicit value abstraction. We did not evaluate the combined approach, since all variables are boolean type, hence it is identical to add a predicate for a variable or to track it explicitly. Predicate abstraction starts with an empty set of initial predicates and only the single output variable is visible using explicit value abstraction. The columns I, L and A correspond to the number of inputs, latches and and-gates respectively. It can be seen that predicate abstraction performs better with Craig interpolation, but explicit value abstraction is more efficient using sequence interpolation.

### 5.4 Summary

Measurements show that all configurations have advantages and shortcomings depending on the types of the models. Predicate abstraction with Craig interpo-

**Table 3.** Measurement results for hardware models.

Model	I	L	A	Pred. (Cr.)			Pred. (seq.)			Expl. (Cr.)			Expl. (Seq.)		
				T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S
✓ mutexp0	11	20	159	<b>10.3</b>	63	494	24.5	43	420	14.3	8	742	22.7	7	806
✓ mutexp0neg	11	20	159	6.1	44	284	<b>3.7</b>	12	82	8.8	9	441	6.7	6	330
× nusmv.syncarb5p2.B	5	10	52	1.3	30	139	3.1	14	132	0.7	6	113	<b>0.2</b>	2	18
× nusmv.syncarb10p2.B	10	20	157	31.6	110	779	117.9	56	1491	239.8	11	5179	<b>1.6</b>	2	32
× pdtpmsarbiter	3	46	209	<b>0.5</b>	6	22	4.6	6	22	5.3	15	130	7.8	13	108
✓ ringp0	15	25	145	16.4	55	300	25.6	19	127	16.1	10	763	<b>14.5</b>	7	657
✓ ringp0neg	15	25	145	<b>7.8</b>	21	83	35.7	31	237	187.5	11	4870	108.2	7	2629
✓ srg5ptimonegnv	30	47	304	<b>0.3</b>	3	9	0.5	4	15	1.7	4	40	1.3	3	36

lation performs well for software and hardware models, explicit value abstraction is efficient for PLC models, while the combined method with sequence interpolation was able to handle the largest state spaces. It can also be observed that extending predicate abstraction with explicit values (the combined method) boosts its performance. As our implementation is only a prototype without optimizations, the developed model checker can not compete with state-of-the-art tools now. Furthermore our current goal was to compare the configurations in the same framework, so also the formerly existing algorithms were reimplemented.

## 6 Conclusions

In our paper we examined various CEGAR-based algorithms for the verification of symbolic transition systems. From the theoretical point of view, we described a configurable framework, which can incorporate the different types of abstractions and refinement strategies. We also proposed a combination of predicate abstraction and explicit values at the initial abstraction, being able to provide better performance based on domain knowledge or heuristics. On the practical side, we examined the efficiency of different configurations of the algorithms on various models, including software and hardware and identified their advantages and shortcomings. Our future plan is to improve our prototype implementation, experiment with further algorithms and develop heuristics for automatically selecting the most efficient configuration based on the model.

**Acknowledgement.** This work was partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP (Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems) project.

## References

1. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Fundamental Approaches to Software Engineering, LNCS, vol. 7793, pp. 146–162. Springer (2013)



2. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Model Checking Software, LNCS, vol. 9232, pp. 20–38. Springer (2015)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 1579, pp. 193–207. Springer (1999)
4. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. In: International Symposium on Fundamentals of Software Engineering, LNCS, vol. 4767, pp. 17–32. Springer (2007)
5. Cabodi, G., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Vendraminetto, D., Biere, A., Heljanko, K., Baumgartner, J.: Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation* 9, 135–172 (2016)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50(5), 752–794 (2003)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (1999)
8. Clarke, E.M., 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)
9. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic* 22(03), 269–285 (1957)
10. Dutertre, B., Sorea, M., et al.: Timed systems in SAL. Tech. rep., SRI International, Computer Science Laboratory (2004)
11. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Verification, Model Checking, and Abstract Interpretation, LNCS, vol. 7148, pp. 186–201. Springer (2012)
12. Fernández Adiego, B., Darvas, D., Blanco Viñuela, E., Tournier, J.C., Bliudze, S., Blech, J.O., González Suárez, V.M.: Applying model checking to industrial-sized PLC programs. *IEEE Transactions on Industrial Informatics* 11(6), 1400–1410 (2015)
13. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification, LNCS, vol. 1254, pp. 72–83. Springer (1997)
14. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 232–244. ACM (2004)
15. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 58–70. ACM (2002)
16. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification, LNCS, vol. 4144, pp. 123–136. Springer (2006)
17. McMillan, K.: Applications of Craig interpolants in model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 3440, pp. 1–12. Springer (2005)
18. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 4963, pp. 337–340. Springer (2008)
19. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2009)