Parameterisation of Three-Valued Abstractions

Nils Timm and Stefan Gruner

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

Abstract. Three-valued abstraction is an established technique in software model checking. It proceeds by generating a state space model over the values *true*, *false* and *unknown*, where the latter value is used to represent the loss of information due to abstraction. Temporal logic properties can then be evaluated on such models. In case of an *unknown* result, the abstraction is iteratively refined. In this paper, we introduce *parameterised three-valued model checking*. In our new type of models, unknown parts can be either associated with the constant value *unknown* or with expressions over boolean parameters. Our parameterisation is an alternative way to state that the truth value of certain predicates or transitions is actually not known and that the checked property has to yield the same result under each possible parameter instantiation. A novel feature of our approach is that it allows for establishing logical connections between parameters: While *unknown* parts in pure three-valued models are never related to each other, our parameterisation approach enables to represent facts like 'a certain pair of transitions has unknown but complementary truth values', or 'the value of a predicate is unknown but remains constant along all states of a certain path'. We demonstrate that such facts can be automatically derived from the system to be verified and that covering these facts in an abstract model can be crucial for the success and efficiency of checking temporal logic properties. Moreover, we introduce an automatic verification framework based on counterexample-guided abstraction refinement and parameterisation.

1 Introduction

Predicate abstraction [2] is an established technique for reducing the complexity of temporal logic model checking. It proceeds by generating a state space model of the software system to be analysed. In this model, concrete states of the system are mapped to abstract states over a finite set of predicates, and admissible executions of the system are represented by sequences of transitions between states. Traditional predicate abstraction techniques are based on a boolean domain for predicates and on an over-approximation of the concrete state space. Thus, only universal properties are preserved under this form of abstraction. If checking a universal property for an abstract model yields *false*, it cannot be concluded that the original system violates this property as well. In this case, model checking additionally returns an *abstract counterexample* - a path in the model that refutes the property. In order to gain certainty about whether this counterexample is spurious or corresponds to a real path, it has to be simulated on

the original system. The simulation of counterexamples involves a partial exploration of the concrete state space, and thus, can be exceedingly costly. Spurious counterexamples are typically ruled out via *counterexample-guided abstraction refinement* (CEGAR) [4]: Further predicates over the variables of the system are iteratively added to the model until a level of abstraction is reached where the property can be either definitely proved or a real counterexample can be found. The application of CEGAR does, however, not guarantee that eventually a model can be constructed that is both precise enough for a definite outcome and small enough to be manageable with the available computational resources.

More recent approaches [3, 18, 13] to abstraction refinement for model checking are based on a domain for predicates with the truth values *true, false* and *unknown*. Corresponding three-valued models with the additional value *unknown* enable to explicitly model the loss of information due to abstraction. In comparison to boolean abstractions, the three-valued approach is capable of preserving universal *and* existential properties. Hence, all definite results in three-valued model checking can be directly transferred to the original system. Only an *unknown* result necessitates iterative refinement. In the latter case, an *unconfirmed counterexample* – a potential error path in the model with *unknown* transitions and predicates – is returned. Unconfirmed counterexamples directly hint at necessary refinement steps. Thus, the costly simulation of counterexamples on the original system is not required in the three-valued setting. Model checking three-valued abstractions can be conducted at the same cost as checking boolean abstractions, but it additionally comes along with the aforementioned advantages.

Continuative work in this field has shown that the precision of model checking three-valued abstractions can be increased by the concept of *generalised model checking* (GMC) [7]. While standard three-valued model checking (3MC) [3, 18, 13] is based on a special *three-valued* semantics that enables the direct evaluation of temporal logic formulae on three-valued models, the idea of GMC is to construct *all* boolean concretisations of a three-valued model. Then classical two-valued model checking is applied to each concretisation and it is checked whether the results are consistent, i.e. whether either all results are *true* or whether all are *false*. In case of consistency, the result can be transferred to the original system. GMC generally yields more definite results than 3MC. Hence, the application of GMC instead of 3MC can reduce the number of necessary refinement iterations in abstraction-based verification. However, the 3MC problem is PSPACE-complete, whereas the GMC problem is even EXP-complete: Number and size of concretisations can be exponential in the size of the three-valued model. Thus, GMC is rather of theoretical than of practical interest. Most existing three-valued abstraction-based verification frameworks, e.g. [13, 8, 14], rely on standard 3MC and try to compensate the lack of precision with additional refinement steps.

Here, we introduce *parameterised three-valued model checking* (PMC) which is a hybrid of three-valued and generalised model checking. Predicates and transitions in our parameterised three-valued models can be either associated with the values *true*, *false* or *unknown* – or with expressions over boolean parame-

ters. Our parameterisation is an alternative way to state that the truth value of certain predicates or transitions is actually not known and that the checked property has to yield the same result under each parameter instantiation. PMC is thus conducted via evaluating a temporal logic formula under all parameter instantiations and checking whether the results are consistent. In contrast to GMC, parameterised three-valued model checking reduces to multiple instances of standard three-valued model checking, since the instantiation only affects parameters but not the explicit truth value *unknown*. Sizes of instantiations are always linear in the size of the parameterised three-valued model. Moreover, parameterisation particularly allows to establish logical connections between *unknown*s in the abstract model: While *unknown* parts in 3MC and GMC are never related to each other, our parameterisation approach enables to represent facts like 'a certain pair of transitions has unknown but complementary truth values', or 'the value of a predicate is unknown but remains constant along all states of a certain path'. We demonstrate that such facts can be automatically derived from the software system to be verified and that covering these facts in an abstract model can be crucial for the success and efficiency of checking temporal logic properties. In particular, we introduce an automatic verification framework for concurrent systems based on parameterised three-valued model checking: Starting with pure three-valued abstraction, in each iteration either classical refinement or parameterisation of *unknown* parts is applied until a definite result in verification can be obtained. The decisions for refinement or parameterisation are automatically made based on unconfirmed counterexamples. For several verification tasks our hybrid approach can significantly outperform the pure three-valued approach. Our work includes the definition of parameterisation rules for three-valued abstractions and a proven theorem which states that PMC is sound if parameterisation is applied according to the rules.

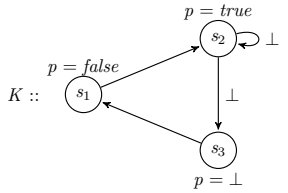
2 Background: Three-Valued Model Checking

We start with a brief introduction to three-valued state space models, here three-valued Kripke structures, and the evaluation of temporal logic properties on them. The key feature of these Kripke structures is a third truth value $\bot$ (i.e. *unknown*) for transitions and labellings, which can be used to model uncertainty.

Definition 1 (Three-Valued Kripke Structure). *A* three-valued Kripke structure *over a set of atomic predicates AP is a tuple $K = (S, R, L, \mathbb{F})$ where*

- *S is a finite set of states,*
- *$R : S \times S \to \{true, \bot, false\}$ is a transition function with $\forall\, s \in S : \exists\, s' \in S : R(s, s') \in \{true, \bot\}$,*
- *$L : S \times AP \to \{true, \bot, false\}$ is a labelling function that associates a truth value with each predicate in each state,*
- *$\mathbb{F} \subseteq \mathcal{P}(R^{-1}(\{true, \bot\}))$ is a set of fairness constraints where each constraint $F \in \mathbb{F}$ is a set of* non-false *transitions.*

An example for a Kripke structure K over a set $AP = \{p\}$ is depicted below.



A path π of a three-valued Kripke structure K is an infinite sequence of states $s_1 s_2 s_3 \ldots$ with $R(s_i, s_{i+1}) \in \{true, \bot\}$. π_i denotes the i-th state of π, whereas π^i denotes the i-th suffix $\pi_i \pi_{i+1} \pi_{i+2} \ldots$ of π. A path π is fair if it takes infinitely often a transition from every fairness constraint $F \in \mathbb{F}$. By $\Pi(K, s)$ we denote the set of all fair paths of K starting in $s \in S$. Paths are considered for the evaluation of temporal logic properties of Kripke structures. Here we use the linear temporal logic (LTL) for specifying properties.

Definition 2 (Syntax of LTL). *Let AP be a set of atomic predicates and $p \in AP$. The* syntax of LTL formulae *ψ is given by*

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

Due to the extended domain for truth values in three-valued Kripke structures, the evaluation of LTL formulae is not based on classical two-valued logic. In three-valued model checking we operate under the three-valued Kleene logic $\mathbb{K}_3$ [6] whose semantics is given by the truth tables below.

$\wedge$	*true*	$\bot$	*false*
true	*true*	$\bot$	*false*
$\bot$	$\bot$	$\bot$	*false*
false	*false*	*false*	*false*

$\vee$	*true*	$\bot$	*false*
true	*true*	*true*	*true*
$\bot$	*true*	$\bot$	$\bot$
false	*true*	$\bot$	*false*

$\neg$	
true	*false*
$\bot$	$\bot$
false	*true*

For $\mathbb{K}_3$ we have a reflexive *information ordering* $\leq_{\mathbb{K}_3}$ (in words: 'less or equal definite than') with $\bot \leq_{\mathbb{K}_3}$ *true*, $\bot \leq_{\mathbb{K}_3}$ *false*, and *true*, *false* incomparable. Based on $\mathbb{K}_3$, linear temporal logic formulae can be evaluated on paths of three-valued Kripke structures according to the following definition.

Definition 3 (Three-Valued Evaluation of LTL). *Let $K = (S, R, L, \mathbb{F})$ over AP be a three-valued Kripke structure. Then the* evaluation *of an LTL formula ψ on a fair path π of K, written $[\pi \models \psi]$, is inductively defined as follows*

$$\begin{aligned}
[\pi \models p] &:= L(\pi_1, p) \\
[\pi \models \neg\psi] &:= \neg\,[\pi \models \psi] \\
[\pi \models \psi \vee \psi'] &:= [\pi \models \psi] \;\vee\; [\pi \models \psi'] \\
[\pi \models \mathbf{X}\psi] &:= R(\pi_1, \pi_2) \wedge \left[\pi^2 \models \psi\right] \\
[\pi \models \mathbf{G}\psi] &:= \textstyle\bigwedge_{i \in \mathbb{N}} \left(R(\pi_i, \pi_{i+1}) \wedge \left[\pi^i \models \psi\right]\right) \\
[\pi \models \mathbf{F}\psi] &:= \textstyle\bigvee_{i \in \mathbb{N}} \left(\left[\pi^i \models \psi\right] \wedge \bigwedge_{0 \leq j < i} R(\pi_i, \pi_{i+1})\right) \\
[\pi \models \psi \mathbf{U} \psi'] &:= \textstyle\bigvee_{i \in \mathbb{N}} \left(\left[\pi^i \models \psi'\right] \wedge \bigwedge_{0 \leq j < i} \left(R(\pi_j, \pi_{j+1}) \wedge [\pi^j \models \psi]\right)\right)
\end{aligned}$$

The evaluation of LTL formulae on entire three-valued Kripke structures is what we call *three-valued model checking* [3].

Definition 4 (Three-Valued LTL Model Checking). *Let $K = (S, R, L, \mathbb{F})$ over AP be a three-valued Kripke structure. Moreover, let ψ be an LTL formula over AP. The value of ψ in a state s of K, written $[K, s \models \psi]$, is defined as*

$$[K, s \models \psi] \;:=\; \bigwedge_{\pi \in \Pi(K,s)} [\pi \models \psi]$$

In three-valued model checking there exist three possible outcomes: *true*, *false* and $\bot$. Three-valued model checking reduces to classical two-valued model checking if the Kripke structure K is actually two-valued, i.e. $R^{-1}(\bot) = \varnothing$ and $L^{-1}(\bot) = \varnothing$. In this case, only the outcomes *true* and *false* are possible. For our example Kripke structure $[K, s_1 \models \mathbf{G}p]$ yields *false*, whereas $[K, s_1 \models \mathbf{GF}p]$ yields *unknown*. $\mathbf{G}p$ is a temporal logic formula that characterises a typical *safety* property, while $\mathbf{GF}p$ characterises a *liveness* property. Safety and liveness are the most vital requirements in software verification. In our approach, we therefore particularly focus on these two kinds of properties.

For the sake of completeness, we also briefly review generalised model checking (for more details see [7]). Under GMC, $[K, s \models \psi]$ yields *true* iff $[K', s \models \psi]$ is *true* for all concretisations K' of K, where a concretisation is a two-valued K' such that $[K, s \models \psi] \;\leq_{\mathbb{K}_3}\; [K', s \models \psi]$ for all LTL formulae ψ. The definition of $[K, s \models \psi] = \mathit{false}$ is analogous. In all remaining cases $[K, s \models \psi]$ yields $\bot$.

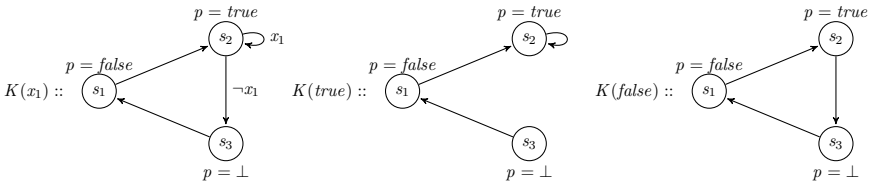
3 Parameterised Three-Valued Model Checking

State space models constructed by three-valued abstraction techniques [13, 8, 14] are typically represented as (pure) three-valued Kripke structures. Here we introduce a generalisation called *parameterised three-valued Kripke structures*, and we define model checking for these structures. Later we will see that *parameterised three-valued model checking* (PMC) for three-valued abstractions can significantly enhance the precision of verification.

Definition 5 (Parameterised Three-Valued Kripke Structure). *A* parameterised three-valued Kripke structure *over AP and a set of boolean parameters $X = \{x_1, \ldots, x_m\}$ is a parameterised tuple $K(\vec[m]{x}) = (S, R(\vec[m]{x}), L(\vec[m]{x}), \mathbb{F}(\vec[m]{x}))$ where*

- *S is a finite set of states,*
- *$R(\vec[m]{x}) : S \times S \to \{\mathit{true}, \bot, \mathit{false}\} \cup BE(X)$ is a transition function with $\forall\, s \in S : \exists\, s' \in S : R(\vec[m]{x})(s, s') \in \{\mathit{true}, \bot\} \cup BE(X)$ where $BE(X)$ denotes the set of boolean expressions over X,*
- *$L(\vec[m]{x}) : S \times AP \to \{\mathit{true}, \bot, \mathit{false}\} \cup BE(X)$ is a labelling function that associates a truth value or a parameter expression with each predicate in each state,*
- *$\mathbb{F}(\vec[m]{x}) \subseteq \mathcal{P}(R^{-1}(\vec[m]{x})(\{\mathit{true}, \bot\} \cup BE(X)))$ is a set of fairness constraints where each constraint $F \in \mathbb{F}(\vec[m]{x})$ is a set of* non-false *transitions.*

Note that $(\overset{m}{x})$ is an abbreviation for the parameter tuple $(x_1, \ldots, x_m)$. An instantiation of a parameterised three-valued Kripke structure $K(\overset{m}{x})$ is a *pure* three-valued Kripke structure $K(\overset{m}{a})$ where $(\overset{m}{a}) \in \{true, false\}^m$. Hence, all parameters are substituted by *boolean* truth values. However, predicates and transitions that were not parameterised in $K(\overset{m}{x})$ may still hold the value *unknown* in $K(\overset{m}{a})$. If the current tuple of parameters or truth values is clear from the context, we will not explicitly mention it, i.e. we will just refer to R, L and $\mathbb{F}$. An example for a parameterised three-valued Kripke structure together with all its pure three-valued instantiations is shown in the figure below.



For evaluating temporal logic formulae on parameterised three-valued Kripke structures we consider all possible instantiations.

Definition 6 (Parameterised Three-Valued LTL Model Checking). *Let $K(\overset{m}{x}) = (S, R(\overset{m}{x}), L(\overset{m}{x}), \mathbb{F}(\overset{m}{x}))$ be a parameterised three-valued Kripke structure over AP and $X = \{x_1, \ldots, x_m\}$. Moreover, let ψ be an LTL formula over AP. The value of ψ in a state s of $K(\overset{m}{x})$, written $[K(\overset{m}{x}), s \models \psi]$, is defined as*

$$\left[K(\overset{m}{x}), s \models \psi\right] := \begin{cases} true & \text{if } \bigwedge_{(\overset{m}{a}) \in \{t,f\}^m} \left(\left[K(\overset{m}{a}), s \models \psi\right] = true\right) \\ false & \text{if } \bigwedge_{(\overset{m}{a}) \in \{t,f\}^m} \left(\left[K(\overset{m}{a}), s \models \psi\right] = false\right) \\ \bot & else \end{cases}$$

Thus, if checking a temporal logic property yields *true* for all instantiations, the result is transferred to the parameterised Kripke structure. The same holds for *false* results for all instantiations. In all other cases PMC returns *unknown*. For our recent example, we get $[K(x_1), s_1 \models \mathbf{GF}p] = true$ since $\mathbf{GF}p$ holds for both $K(true)$ and $K(false)$. In contrast to our example from Section 2, the two outgoing transitions of state s_2 are no longer *unknown* but parameterised. Moreover, we capture the fact that the associated transition values are *complementary*, which gives us the necessary precision for a definite result in verification.

Subsequently, we will see that such facts can be automatically derived from the control flow and program code of the modelled system in the sense that the corresponding parameterisation gives us a sound abstraction. Furthermore, we will show how parameterised three-valued model checking can be effectively integrated into an automatic abstraction refinement-based verification procedure.

4 Application to Three-Valued Abstractions

Three-valued model checking [3] is used in many abstraction-based verification frameworks for software systems [13, 10, 8, 1]. An effective state space reduction

technique for concurrent software systems is *three-valued spotlight abstraction* [12, 14, 15]. In previous works [16, 17], we have demonstrated that verifying concurrent systems via spotlight abstraction and three-valued model checking can significantly outperform approaches based on boolean predicate abstraction [2]. In this section, we give a brief introduction to concurrent systems and spotlight abstraction (for more details see [12]). Moreover, we show how *parameterisation* can be applied to three-valued Kripke structures constructed by spotlight abstraction and how this can increase the efficiency of verification.

4.1 Spotlight Abstraction for Concurrent Systems

A concurrent system Sys consists of a number of asynchronous processes composed in parallel: $Sys = \|_{i=1}^{n} Proc_i$. It is defined over a set of variables $Var = Var_s \cup \bigcup_{i=1}^{n} Var_i$ where Var_s is a set of shared variables and $Var_1, \ldots, Var_n$ are sets of local variables associated with the processes $Proc_1, \ldots, Proc_n$, respectively. A process corresponds to a finite sequence of locations where each location is associated with an operation op on the variables in $Var_s \cup Var_i$. Operations are of the form $op = assume(e) : v_1 := e_1, \ldots, v_k := e_k$ where $e, e_1, \ldots, e_k$ are expressions over $Var_s \cup Var_i = \{v_1, \ldots, v_k\}$. Hence, an operation consists of an assume part, also called *guard*, and a list of assignments. Executing the guard blocks the execution of the assignments until the expression e evaluates to *true*. We omit the guard if e is constantly *true*. The current location of a process $Proc_i$ can be regarded as the value of an additional local counter variable pc_i over the process' locations $Loc_i = \{1_i, \ldots, L_i\}$. Locations may also be associated with compound operations, which consist of one or more sub-operations nested inside a control structure. Compound operations in our systems are, amongst others, *if-then-else* and *while-do*. An example for a concurrent system is depicted below.

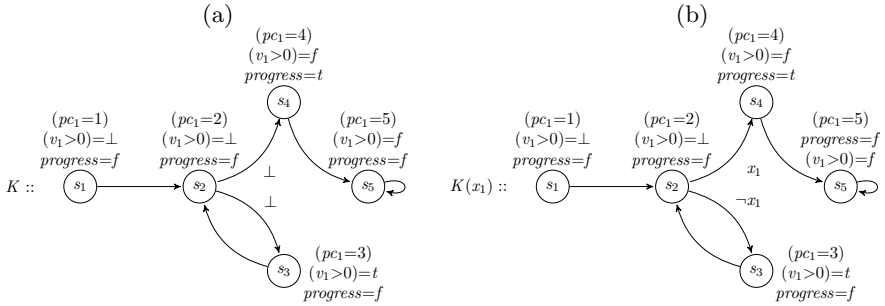
$$v_1, \ldots, v_k \ : \textbf{integer}$$

$$Proc_1 :: \begin{bmatrix} 1: [\ \ldots\ ] \\ 2: \textbf{while}\ (v_1 > 0)\ \textbf{do} \\ \quad 3: [\ \ldots\ ] \\ 4: \ progress \\ 5: [\ \ldots\ ] \end{bmatrix} \ \| \ Proc_2 :: \begin{bmatrix} 1: [\ \ldots\ ] \\ 2: \ v_1 := f(v_2, \ldots, v_k) \\ 3: [\ \ldots\ ] \end{bmatrix} \ \| \ \ldots \ \| \ Proc_n$$

Here we have a composition of n processes operating on the shared variables $v_1, \ldots, v_k$. A liveness property to verify might be whether $Proc_1$ always repeatedly reaches *progress*, which we assume is an arbitrary assertion over $Proc_1$'s variables. Subsequently, we show how this verification task can be approached by three-valued spotlight abstraction.

Spotlight abstraction involves the partition of the processes of the system into a *spotlight* and a *shade*. Predicate abstraction is applied to the spotlight, while the shade processes are abstracted away by summarising them in one approximative component. The state space of the resulting abstract system can be straightforwardly modelled as a (pure) three-valued Kripke structure. In our current verification task, the relevant process for the property of interest is $Proc_1$,

which we put into the spotlight: $Spot(Proc) = \{Proc_1\}$, whereas the remaining system is for now kept in the shade: $Shade(Proc) = \{Proc_2, \ldots, Proc_n\}$. Next, a set of so-called *spotlight predicates* over the system variables is selected, here we choose $Spot(Pred) = \{progress, (v_1 > 0)\}$. By applying three-valued predicate abstraction to the spotlight processes, we obtain an abstract process $Proc_1^a$ with the same control flow as $Proc_1$ but with operations abstracted over $Spot(Pred)$. The processes in the shade are summarised to one approximative process $Proc_{Shade}$. Due to the loss of information about the shade, $Proc_{Shade}$ might set predicates over shared variables to the value $\perp$. Our abstract system now looks as follows: $Sys^a = Proc_1^a \parallel Proc_{Shade}$. The state space of Sys^a can be modelled as a pure three-valued Kripke structure over $AP = Spot(Pred) \cup \{(pc_i = j) \mid Proc_i \in Spot(Proc), j \in Loc_i\}$ where $(pc_i = j)$ refers to the program counter of $Proc_i$, and each definite model checking result obtained for this structure can be transferred to the concrete system [12]. A three-valued Kripke structure K corresponding to Sys^a is depicted in part (a) of the figure below. For simplicity, we only show the program counter predicates that are currently *true*.



Note that the control flow of spotlight processes is always preserved under spotlight abstraction. Hence, each transition of K associated with the spotlight matches with a specific operation of the spotlight process $Proc_1$. For K and its set of atomic predicates $AP = \{progress, (v_1 > 0)\} \cup \{(pc_1 = j) \mid j \in Loc_1\}$ we can formalise our property of interest as the LTL formula $\mathbf{GF}progress$ and then apply standard three-valued model checking, i.e. check $[K, s_1 \models \mathbf{GF}progress]$. The current abstraction is not precise enough for a definite result in verification. Since there exist processes in the shade that operate on the shared variable v_1, the value of the predicate $(v_1 > 0)$ in the states s_1 and s_2 is $\perp$. Thus, it is also unknown whether the body of the *while*-loop can be executed via the transition (s_2, s_3), or whether the loop can be eventually left via (s_2, s_4). The automatic abstraction refinement procedure introduced in [17] would now iteratively shift processes from the shade to the spotlight until it can be definitively shown *which* branch of the *while*-loop can be actually taken. However, due to transitive dependencies – $Proc_2$ modifies v_1, but in turn depends on $v_2, \ldots, v_k$ which may be modified by other shade processes as well – such a refinement can be exceedingly costly or can even lead to a failure of verification because of state explosion. A closer look at our simple example structure tell us that, regardless of which branch of the loop will be ever taken, *progress* will never hold repeatedly. Hence, the evaluation of $\mathbf{GF}progress$ on K should yield *false*. However, the standard

three-valued LTL semantics (compare Section 2) does not allow us to draw this conclusion. In the following we will see that automated *parameterisation* can give us the necessary precision for a definite verification result – at considerably less cost than classical abstraction refinement.

4.2 Parameterisation of Three-Valued Abstractions

As we just have seen, $[K, s_1 \models \mathbf{GF}\mathit{progress}]$ yields $\bot$. Nevertheless, a $\bot$-result in 3MC always comes along with an *unconfirmed counterexample* – a potential error path in the Kripke structure with some *unknown* transitions or predicates. Four our running example the path $\pi = s_1 s_2 s_4 s_5 s_5 \ldots$ is an unconfirmed counterexample. Such a path is typically used for *counterexample-guided abstraction refinement* (CEGAR) [4]: In our case, the $\bot$-transition (s_2, s_4) would be identified as the reason for uncertainty, and shade processes that modify the *if*-condition $(v_1 > 0)$ associated with (s_2, s_4) would be iteratively shifted to the spotlight. Now we will show that counterexamples can also be exploited for the parameterisation of three-valued Kripke structures. We first illustrate parameterisation based on our running example and then provide the general rules for it.

Our method detects that the reason for uncertainty, the $\bot$-transition (s_2, s_4) along π, is associated with a *complementary branch* in the original system: a branch of the control flow of a single process with complementary branching conditions – here $(v_1 > 0)$ and $\neg(v_1 > 0)$. Instead of applying classical CEGAR, a fresh boolean parameter x_1 is introduced and the transition is parameterised as follows: $R(s_2, s_4) := x_1$. Next, the complementary transition (s_2, s_3) is identified and parameterised by $R(s_2, s_3) := \neg x_1$. The corresponding parameterised three-valued Kripke structure $K(x_1)$ is depicted in part (b) of the figure on the previous page. Applying parameterised three-valued model checking, i.e. verifying $[K(x_1), s_1 \models \mathbf{GF}\mathit{progress}]$ immediately returns *false*. Thus, for our running example a definite result in verification only requires the introduction of a single parameter and the consideration of the two instantiations $K(\mathit{true})$ and $K(\mathit{false})$ of $K(x_1)$. In contrast, a corresponding pure three-valued approach would require a large number of additional refinement steps and thus would most likely fail due to state explosion. Also the application of the computationally more expensive GMC would not be successful, since it cannot establish the complementary relation between (s_2, s_4) and (s_2, s_3). The following rule generalises the parameterisation of complementary branches in three-valued Kripke structures.

Rule I (Parameterisation of Complementary Branch Transitions). *Let* $\mathit{Sys} = \|_{i=1}^{n} \mathit{Proc}_i$ *be a concurrent system and* $\mathit{Spot} = \mathit{Spot}(\mathit{Proc}) \cup \mathit{Spot}(\mathit{Pred})$ *be a spotlight abstraction for Sys. Let K be a three-valued KS over* $AP = \mathit{Spot}(\mathit{Pred}) \cup \{(\mathit{pc}_i = j) \mid \mathit{Proc}_i \in \mathit{Spot}(\mathit{Proc}) \wedge j \in \mathit{Loc}_i\}$ *that models the abstract state space corresponding to Sys and Spot, and let* s_1 *be a state of K. Moreover, let* ψ *be a safety or liveness LTL formula and checking* $[K, s_1 \models \psi]$ *yields* $\bot$*. Let* π *be the unconfirmed counterexample returned by model checking which runs through a finite number of different transitions. The transitions of K can be parameterised as follows:*

For each transition (s, s') along π with $R(s, s') = \bot$, check if (s, s') is part of a complementary branch, i.e.: (s, s') is associated with a guard operation assume(e) of a spotlight process $Proc_i$, where e is a boolean expression – and moreover, there exists a state s'' such that (s, s'') is associated with a complementary guard operation assume$(\neg e)$ of $Proc_i$. Then introduce a fresh parameter x_j and set $R(s, s') = x_j$ and $R(s, s'') = \neg x_j$.

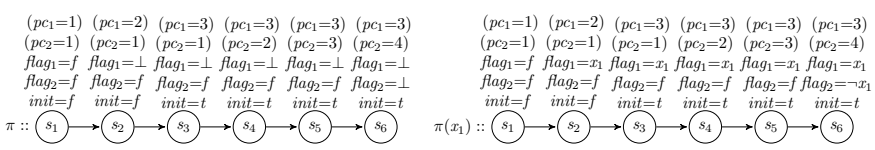
This rule allows to parameterise complementary branches (e.g. *if*- or *while*-operations) in three-valued abstractions. As we have seen in our running example, this can lead to substantial savings in the number of necessary refinement steps for a definite result in verification. In fact, any verification task where the property of interest turns out to be independent from certain branches can profit from such a parameterisation in a similar manner. At the end of this section we will present a theorem which states that the application of Rule I leads to sound abstractions of concurrent systems. Beforehand, we introduce another rule that allows the parameterisation of *predicates* in three-valued abstractions.

In order to illustrate how the parameterisation of predicates works, we consider a second example, the concurrent system *Sys* depicted below. Our property of interest is now *mutual exclusion*, i.e. whether the flag variables $flag_1$ and $flag_2$ are never *true* at the same time.

$$v_1, \ldots, v_k : \textbf{integer};$$
$$flag_1, flag_2, init : \textbf{boolean where } flag_1 = false, flag_2 = false, init = false;$$
$$Proc_1 :: \begin{bmatrix} 1: & flag_1 := f(v_1, \ldots, v_k) \\ 2: & init := true \\ 3: & flag_1 := \neg flag_2 \\ 4: & [\ \ldots \quad] \end{bmatrix} \ \| \ Proc_2 :: \begin{bmatrix} 1: & flag_2 := false \\ 2: & \textbf{await}(init) \\ 3: & flag_2 := \neg flag_1 \\ 4: & [\ \ldots \quad] \end{bmatrix} \ \| \ \ldots \ \| \ Proc_n$$

Applying three-valued spotlight abstraction with classical refinement yields the following spotlight after a number of iterations: $Spot(Proc) = \{Proc_1, Proc_2\}$ and $Spot(Pred) = \{flag_1, flag_2, init\}$. Next, a corresponding pure three-valued Kripke structure K over $AP = \{flag_1, flag_2, init\} \cup \{(pc_i = j) \mid Proc_i \in Spot(Proc) \wedge j \in Loc_i\}$ is constructed, and the mutual exclusion property formalised by the safety LTL formula $\mathbf{G}\neg(flag_1 \wedge flag_2)$ is checked for K. Model checking returns *unknown*, since the assignment to $flag_1$ at location 1 of $Proc_1$ depends on the shared variables $v_1, \ldots, v_k$ which are potentially modified by a large number of processes that are currently in the shade. Thus, with classical abstraction refinement we have to expect a large number of further refinement steps necessary for a definite result in verification: Predicates over the variables $v_1, \ldots, v_k$ as well as processes modifying these variables have to be drawn into the spotlight. Nevertheless, the model checking run based on the current spotlight also returns the unconfirmed counterexample π depicted in part (a) of the figure below.

(a) (b)



The reason for uncertainty is the reachable state s_6 where flag_1 and flag_2 are both $\bot$. The predicate flag_1 is set to $\bot$ by transition (s_1, s_2), since there are not enough predicates and processes in the spotlight in order to abstract the associated operation $\mathit{flag}_1 := f(v_1, \ldots, v_k)$ properly. The predicate flag_2 is set to $\bot$ by (s_5, s_6) because the associated operation $\mathit{flag}_2 := \neg \mathit{flag}_1$ modifies this predicate in relation to the already *unknown* predicate flag_1. In our simple example it is easy to see that flag_1 and flag_2 must have *complementary* values in state s_6 – which would rule out the unconfirmed counterexample π. However, this fact cannot be captured by pure three-valued abstraction since it does not allow to establish connections between predicates that are associated with the value $\bot$.

Our concept of parameterisation enables us to establish such connections. For our running example we proceed as follows: We backtrack to the state s_2 where flag_1 was initially associated with $\bot$. Next, we introduce a fresh parameter x_1 and set $L(s_2, \mathit{flag}_1) := x_1$. Based on the operations associated with the succeeding transitions along π we update the labellings of the states s_3 to s_6. As a consequence, we now can capture that flag_1 constantly keeps the value x_1 along π, flag_2 keeps the value *false* until s_5, and in particular, flag_1 and flag_2 have complementary values in s_6. The resulting path $\pi(x_1)$, which is depicted in part (b) on the previous page, is no longer an unconfirmed counterexample. Thus, checking $\mathbf{G}\neg(\mathit{flag}_1 \wedge \mathit{flag}_2)$ on a corresponding parameterised Kripke structure $K(x_1)$ will immediately return that no counterexample exists, i.e. that the property is satisfied for the modelled system. Again we have seen that parameterisation – here with regard to predicates – can lead to substantial savings in the number of necessary refinement steps for a definite result in verification. The following rule generalises the parameterisation of predicates in three-valued abstractions.

Rule II (Parameterisation of Predicates along Counterexamples). *Let Sys, Spot, K, s_1 and AP be as in Rule I. Moreover, let $\psi = \mathbf{G}\neg(\bigwedge_{i=1}^{m} p_i)$ be a safety LTL formula with $\{p_1, \ldots, p_m\} \subseteq \mathit{Spot}(\mathit{Pred})$ and model checking $[K, s_1 \models \psi]$ yields $\bot$. Let $\pi = s_1 \ldots s_k$ be the unconfirmed counterexample returned by model checking which is a path prefix that ends in a state s_k where all predicates from $\{p_1, \ldots, p_m\}$ are associated with either the value $\bot$ or true. K can be parameterised along π according to the following procedure:*

for $s := s_1$ *to* s_k **do**
  for *each* $p_i \in \{p_1, \ldots, p_m\}$ *with* $L(s_k, p_i) = \bot$ **do**
    if $L(s, p_i) = \bot$ **then**
      if $s = s_1$, *i.e. s is the initial state* **then**
        introduce a fresh parameter x_j and set $L(s, p_i) := x_j$
      else
        let s' be the direct predecessor of s along π, and let *op* be the operation associated with the transition (s', s)
        if *op is not associated with a process in Spot(Proc) or none of the atomic predicates occurring in the weakest precondition*[1] $wp_{op}(p_i)$ *are contained in Spot(Pred)* **then**
          introduce a fresh parameter x_j and set $L(s, p_i) := x_j$
        else
          set $L(s, p_i) :=$
          $wp_{op}(p_i)\,[p/L(s', p) \mid p \in \mathit{Spot}(\mathit{Pred})]\,[p/\bot \mid p \notin \mathit{Spot}(\mathit{Pred})]$,
          i.e. update $L(s, p_i)$ wrt. parameterisations in predecessor s'

[1] Let $op = \mathit{assume}(e) : x_1 := e_1, \ldots, x_m := e_m$ then $wp_{op}(p) = e \wedge p[x_1/e_1, \ldots, x_m/e_m]$.

Parameterisation of predicates is applied in a similar way for model checking liveness formulae, i.e. $[K, s_1 \models \mathbf{GF}(\bigvee_{i=1}^{m} p_i)]$ *with* $\{p_1, \ldots, p_m\} \subseteq Spot(Pred)$*. In case of an unknown result, the model checker additionally returns an unconfirmed counterexample* π *of the form* $(s_1 \ldots s_{l-1}) \circ (s_l \ldots s_k)^\omega$ *and in all states* $s_l \ldots s_k$ *each predicate from* $\{p_1, \ldots, p_m\}$ *is associated with either the value* $\perp$ *or false. The finite prefix* $(s_1 \ldots s_{l-1})$ *of* π *is then parameterised in the same manner as in the case of model checking safety formulae.*

The following theorem establishes the soundness, with respect to the information ordering $\leq_{\mathbb{K}_3}$ (compare Section 2), of parameterised three-valued model checking, provided that parameterisation is applied according to Rule I and II.

Theorem 1. *Let Sys and Spot be as before. Let* K *over* AP *be a two-valued KS modelling the concrete state space of Sys and let* $K^\perp$ *over* $AP^\perp = Spot(Pred) \cup \{(pc_i = j) \mid Proc_i \in Spot(Proc) \wedge j \in Loc_i\}$ *with* $AP^\perp \subseteq AP$ *be a pure three-valued KS modelling the abstract state space corresponding to Spot. Moreover, let* s_1 *and* $s_1^\perp$ *be states representing the initial configuration of Sys in* K *resp.* $K^\perp$*. Then for any parameterisation* $K^\perp(\overset{m}{x})$ *of* $K^\perp$ *obtained by applying the rules I and II, and for any safety or liveness LTL formula* ψ[2] *over* $AP^\perp$ *the following holds:*

$$[K^\perp(\overset{m}{x}), s_1^\perp \models \psi] \quad \leq_{\mathbb{K}_3} \quad [K, s_1 \models \psi]$$

Proof. See `http://www.cs.up.ac.za/cs/ntimm/proof.pdf`

Hence, every definite result in verification obtained for $[K^\perp(\overset{m}{x}), s_1^\perp \models \psi]$ can be directly transferred to the concrete system modelled by K, whereas an *unknown* result for $[K^\perp(\overset{m}{x}), s_1^\perp \models \psi]$ tells us that further abstraction refinement or parameterisation of $K^\perp(\overset{m}{x})$ is required. In the next section, we will show how we have implemented the application of the parameterisation rules within an automatic abstraction refinement procedure for the verification of concurrent systems and how verification can benefit from our parameterisation approach.

5 Automatic Counterexample-Guided Refinement and Parameterisation

We have prototypically implemented a verification framework for concurrent systems based on spotlight abstraction with counterexample-guided refinement and parameterisation. Our framework 3Spot works on top of the three-valued symbolic model checker χChek [5]. 3Spot takes a concurrent system *Sys* over a variable set *Var* and a safety or liveness temporal logic formula ψ over *Sys* as input. The initial spotlight *Spot* is defined by the processes that are referenced in ψ and the atomic predicates over *Var* that are subformulae of ψ. Next, a parameterised three-valued Kripke structure $K^\perp(\overset{m}{x}) = (S, R, L, \mathbb{F})$ corresponding to *Sys* and *Spot* is constructed with a state $s_1 \in S$ representing the initial configuration of *Sys*. The parameter tuple $(\overset{m}{x})$ of $K^\perp(\overset{m}{x})$ is initially empty. In order to check $[K^\perp(\overset{m}{x}), s_1 \models \psi]$, the following procedure is executed:

[2] ψ is either of the form $\mathbf{G}\neg(\bigwedge_{i=1}^{m} p_i)$ or $\mathbf{GF}(\bigvee_{i=1}^{m} p_i)$ with $\{p_1, \ldots, p_m\} \subseteq AP^\perp$.

1. **check** $[K^{\perp}(\overset{m}{a}), s_1 \models \psi]$ for all valuations $(\overset{m}{a}) \in \{t,f\}^m$
   if $\forall(\overset{m}{a}) \in \{t,f\}^m : [K^{\perp}(\overset{m}{a}), s_1 \models \psi] = t$ **or** $\forall(\overset{m}{a}) \in \{t,f\}^m : [K^{\perp}(\overset{m}{a}), s_1 \models \psi] = f$ **then**
   property ψ is successfully proved resp. disproved for the concurrent system Sys; stop
   if $\forall(\overset{m}{a}) \in \{t,f\}^m : [K^{\perp}(\overset{m}{a}), s_1 \models \psi] \in \{\perp, t\}$ **or** $\forall(\overset{m}{a}) \in \{t,f\}^m : [K^{\perp}(\overset{m}{a}), s_1 \models \psi] \in \{\perp, f\}$ **then**
   still some *unknown* results; further refinement or parameterisation required; go to 2.
   if $\exists(\overset{m}{a}) \in \{t,f\}^m : [K^{\perp}(\overset{m}{a}), s_1 \models \psi] = t$ **and** $\exists(\overset{m}{a}) \in \{t,f\}^m : [K^{\perp}(\overset{m}{a}), s_1 \models \psi] = f$ **then**
   current parameterisation not expedient; revoke last parameterisation; go to 2.
2. **for** each valuation $(\overset{m}{a}) \in \{t,f\}^m$ with $[K^{\perp}(\overset{m}{a}), s_1 \models \psi] = \perp$ **do**
   generate unconfirmed counterexample $\pi^{\perp}$ for $[K^{\perp}(\overset{m}{a}), s_1 \models \psi]$
   select unconfirmed counterexample $\pi^{\perp}$ with the fewest *unknown* transitions and predicates
   if Rule I is applicable along $\pi^{\perp}$ **then**
   apply Rule I to the corresponding branch in $K^{\perp}(\overset{m}{x})$
   else if Rule II is applicable along $\pi^{\perp}$ **then**
   apply Rule II to the corresponding path prefix in $K^{\perp}(\overset{m}{x})$
   else
   determine cause of indefinite result along $\pi^{\perp}$ and derive corresponding refinement candidate r (see our previous work [17] for an example technique for deriving refinement candidates from unconfirmed counterexamples), which can be a shade process or a predicate; add r to *Spot*
   if r is a predicate **then**
   revoke parameterisation for parameterised branches in $K^{\perp}(\overset{m}{x})$ where the value of r affects the branching condition
   update $K^{\perp}(\overset{m}{x})$ according to changes in 2. and go to 1.

Hence, the procedure terminates if for all instantiations of the current parameterised Kripke structure the same definite result in verification can be obtained. If model checking yields *true* for some instantiations and *false* for others, the last parameterisation step was not expedient: The property of interest is then obviously not independent from the most recent parameterisation. Thus, this step is revoked, which also includes that the same parameterisation will not be admissible in future iterations. In case model checking returns *unknown* for some instantiations, the abstraction has to be further parameterised or refined based on unconfirmed counterexamples obtained for these instantiations. For this purpose we always apply Rule I or II if possible, or use classical refinement (see our previous work [17]) otherwise. Adding a new predicate p to the abstraction may affect parameterised branches: An abstract state s that is the starting point of a complementary branch may be split into two new states s_a and s_b with $L(s_a, p) = \mathit{true}$ and $L(s_b, p) = \mathit{false}$. Thus, in the general case, the parameterisation of the complementary branch starting in s has to be revoked. However, if the branch condition is independent from the value of p then the parameterisation can be kept. Alternatives to the revocation of parameterisations are: Keeping the parameterisation for only one state, either s_a or s_b. Or, introducing a fresh parameter x_j for the second branch starting in s_b. Each iteration ends with the update of the parameterised three-valued Kripke structure according to new parameterisations or additional refinements. In case a new predicate has been added to the abstraction, this update also involves the recalculation of the parameterisation of predicates (compare last step of Rule II).

So far, parameterisation resp. refinement is performed based on the unconfirmed counterexample with the fewest *unknown* transitions and predicates. The intention behind this is to minimise the expected effort to confirm or eliminate the counterexample. Moreover, the attempt to apply the parameterisation rules

or classical refinement is so far always conducted in the fixed order *Rule I, Rule II, refinement*. In the future, we intend to use heuristic guidance for selecting the unconfirmed counterexample and for deciding which rule application or which refinement step is currently most promising in order to achieve a definite result in verification within a small number of iterations. Similar to our previous work on heuristics for pure refinement [17], we plan to base this heuristic approach on the structure of the underlying concurrent system, i.e. on the variable dependencies between the processes of the system.

In preliminary experiments, we applied our procedure to multiple-resource allocation systems[3] with up to 25 processes and 140 variable dependencies, and we checked safety as well as liveness properties. We compared verification under the pure three-valued approach (which has proven to be generally successful for concurrent systems in [17, 14, 15]) with verification under our novel approach with parameterisation. In several cases where the pure three-valued approach failed due to an out-of-memory exception, our new technique was capable of returning a definite verification result. The additional computations for parameterisation particularly paid off when the property of interest turned out to be independent from certain branches in the system, and the costs for concretising these branches via classical refinement were high. In fact, such cases are very common for systems with many *if*-, *while*-, and similar operations. We also observed verification tasks (primarily where the system only exhibited very few branches, or where the property was dependent on most of the branches) that did not profit from the application of parameterisation rules. Here verification under the new approach was slower but did not fail, since parameterisation only increases the number of checks per iteration, but not the size of the abstraction (spotlight processes and predicates). Thus, so far it is a good strategy to apply the pure three-valued approach first and in case of failure the approach with parameterisation subsequently. Nevertheless, with our intended heuristic approach, we aim at directly discovering the best possible combination of refinement and parameterisation for each verification task. A more extensive experimental evaluation of such an enhanced approach is also planned as future work.

6 Related Work

Or research is situated in the field of model checking temporal logic properties on partial system models. The idea of evaluating temporal logic formulae on three-valued Kripke structures was initially proposed in [3] and is now established under the name *three-valued model checking* (3MC). Our new concept *parameterised three-valued model checking* (PMC) is an extension of 3MC. In our approach, unknown parts of the modelled system cannot only be represented by the constant $\bot$, but also by expressions over boolean parameters. The evaluation of temporal logic formulae is then performed for each possible parameter instantiation. The idea of considering possible instantiations resp. concretisations of a partial model is adopted from *generalised model checking* (GMC) [7]. In contrast to the concretisations in GMC, our instantiations only affect parameters

[3] A detailed description of these systems can be found in [14].

but do not concern the constant $\perp$. Moreover, our instantiations are always of the same size as the partial model, whereas the concretisations in GMC can be exponentially larger. Neither 3MC nor GMC offer a concept for drawing connections between unknown parts. While 3MC and GMC are general concepts for the verification of partial models, our approach is application-oriented and takes advantage from the consideration of the system structure when applying the parameterisation rules within our automated verification procedure.

Another work related to ours is that of Herbstritt et al. [9] who combine three-valued logic and quantified boolean parameters for representing unspecified parts of a hardware model with different precision. Their technique is geared towards equivalence checking of circuits. In contrast to our approach, [9] do not introduce a concept for establishing connections between parameters in the model. Moreover, the decision for modelling an unspecified part via the third truth value $\perp$ or via a boolean parameter has to be done by hand and not based on automatable rules. [9] encode their hardware verification tasks as bounded model checking problems that can be efficiently solved via SAT/QBF-solvers. The definition of such encodings for our parameterised three-valued model checking is another interesting direction for future research. A similar approach to the verification of hardware circuits, but in the context of BDD-based symbolic model checking was introduced in [11]. Their method supports the verification of full CTL properties based on models with a flexible representation of unknowns. This approach necessitates the manual selection of the type of modelling unknown parts. Establishing logical relations between parameters is not possible here.

7 Conclusion

We developed a concept for modelling unknown parts of an abstract software system with different types of approximation: In our parameterised three-valued Kripke structures the loss of information about a predicate or a transition can be either represented by the constant $\perp$ or by an expression over boolean parameters. A novel feature of our modelling approach is that it allows for establishing logical connections between *unknown* parameters, like equality or complementarity – and thus, to preserve more details under abstraction that can be crucial for the success and efficiency of verification. We introduced temporal logic model checking for parameterised three-valued Kripke structures and showed that this method is sound if the models are constructed with regard to parameterisation rules that we defined. These rules take the branching structure and the program code of the modelled system into account and arrange the connections between parameters in the model. We then presented an automatic verification procedure based on iterative abstraction refinement and parameterisation. For several verification tasks, particularly for verifying systems with many conditional branches, our new approach with parameterisation can significantly outperform verification based on classical modelling techniques that are not capable of characterising connections between unknown parts. We are convinced that our concept for parameterisation can be easily and effectively adapted to other types of systems and verification tasks, which we intend to investigate in our future research.

References

1. Alfaro, L., Roy, P.: Solving games via three-valued abstraction refinement. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007, LNCS, vol. 4703, pp. 74–89. Springer-Verlag Berlin Heidelberg (2007)
2. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: ACM SIGPLAN 2001. pp. 203–213. PLDI '01, ACM, New York, NY, USA (2001)
3. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. pp. 274–287. LNCS, Springer-Verlag Berlin Heidelberg, London, UK (1999)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000, LNCS, vol. 1855, pp. 154–169. Springer-Verlag Berlin Heidelberg (2000)
5. Easterbrook, S.M., Chechik, M., Devereux, B., Gurfinkel, A., Lai, A.Y.C., Petrovykh, V., Tafliovich, A., Thompson-Walsh, C.: χChek: A model checker for multi-valued reasoning. In: ICSE 2003. pp. 804–805 (2003)
6. Fitting, M.: Kleene's three valued logics and their children. Fundamenta Informaticae 20(1-3), 113–131 (Mar 1994)
7. Godefroid, P., Piterman, N.: LTL generalized model checking revisited. In: Jones, N.D., Mueller-Olm, M. (eds.) VMCAI 2009, LNCS, vol. 5403, pp. 89–104. Springer Berlin Heidelberg (2009)
8. Grumberg, O.: 2-valued and 3-valued abstraction-refinement in model checking. In: Logics and Languages for Reliability and Security, pp. 105–128. IOS Press, Incorporated (2010)
9. Herbstritt, M., Becker, B.: On combining 01X-logic and QBF. In: Moreno Diaz, R., Pichler, F., Quesada Arencibia, A. (eds.) Comp. Aided Systems Theory - EUROCAST 2007, LNCS, vol. 4739, pp. 531–538. Springer Berlin Heidelberg (2007)
10. Katoen, J.P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for probabilistic systems. Logic and Algebraic Programming 81(4), 356 – 389 (2012)
11. Nopper, T., Scholl, C.: Symbolic model checking for incomplete designs with flexible modeling of unknowns. IEEE Trans. Computers 62(6), 1234–1254 (2013)
12. Schrieb, J., Wehrheim, H., Wonisch, D.: Three-valued spotlight abstractions. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009: Formal Methods, LNCS, vol. 5850, pp. 106–122. Springer-Verlag Berlin Heidelberg (2009)
13. Shoham, S., Grumberg, O.: 3-valued abstraction: More precision at less cost. Information and Computation 206(11), 1313 – 1333 (2008)
14. Timm, N.: Three-Valued Abstraction and Heuristic-Guided Refinement for Verifying Concurrent Systems. Phd thesis, University of Paderborn (2013)
15. Timm, N.: Spotlight abstraction with shade clustering – automatic verification of parameterised systems. In: 8th International Symposium on Theoretical Aspects of Software Engineering, IEEE Computer Society (to appear) (2014)
16. Timm, N., Wehrheim, H.: On symmetries and spotlights – verifying parameterised systems. In: Dong, J., Zhu, H. (eds.) ICFEM 2010, LNCS, vol. 6447, pp. 534–548. Springer, Heidelberg (2010)
17. Timm, N., Wehrheim, H., Czech, M.: Heuristic-guided abstraction refinement for concurrent systems. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012, LNCS, vol. 7635, pp. 348–363. Springer Berlin Heidelberg (2012)
18. Wei, O., Gurfinkel, A., Chechik, M.: On the consistency, expressiveness, and precision of partial modeling formalisms. Information and Computation 209(1), 20 – 47 (2011)