Electronic Notes in Theoretical Computer Science 89 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 12 pages

Comparing Abstraction Refinement Algorithms

Dennis Dams

Bell Labs, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974.
dennis@research.bell-labs.com

Abstract

We present a generic algorithm that provides a unifying scheme for the comparison of abstraction refinement algorithms. It is centered around the notion of *refinement cue* which generalizes counterexamples. It is demonstrated how the essential features of several refinement algorithms can be captured as instances.

We argue that the generic algorithm does not limit the completeness of instances, and show that the proposed generalization of counterexamples is necessary for completeness — thus addressing a shortcoming of more limited notions of counterexample-guided refinement.

1 Introduction

In order to use model checking [7] for the analysis of implementations, one needs to extract models that are small enough to allow for exhaustive exploration yet contain sufficient detail to be able to demonstrate the properties of interest. Automation of this model extraction process is needed if model checking is to be used more widely in software debugging. One approach to this is *abstraction refinement*, which starts with a coarse initial model (abstraction) of the system, iteratively refining it until it contains sufficient detail. Examples of this approach can e.g. be found in [13,3,20,10,6,18,1]. Recently, there have been various proposals to use *counterexamples* to drive the refinement process (among others, [2,14]).

Appropriate abstractions of infinite state programs are not computable in general: it would imply that the program verification problem is decidable (see e.g. [11]). On the other hand, it has been shown [21,12,17] that for any given program and (temporal logic) correctness property, there *exists* an appropriate finitary abstraction. Hence, there will always exist better (semi-)algorithms [1]. However, the comparison of proposals is currently difficult, due to the lack of a "common denominator" of such algorithms and established criteria for

[1] We will sometimes use the word algorithm in cases where *semi-algorithm* is meant.

comparison. One comparison is carried out in [1], however, the criterion used in that paper is tailored towards a particular algorithm.

In this paper, we suggest a generic abstraction refinement algorithm that facilitates reasoning about many concrete instances. Its parameters capture aspects that are common to such algorithms but that vary in their details. In particular, we introduce the notion of a *refinement cue*, that generalizes the notion of counterexample. Indeed, it allows to frame several algorithms that are not counterexample-driven. The generic algorithm repeats the following two steps: (1) the identification of a finite number of refinement cues, followed by (2) a refinement of the current abstraction based on each of the cues. The second step is formalized by the notion of a *refinement function*. These steps are iterated until the abstraction is sufficiently fine to prove the given property of interest, or until it is clear that the property does not hold. We illustrate the generality of this framework by rephrasing the essential aspects of several proposed algorithms in terms of it.

More than a blueprint of what an abstraction refinement algorithm is, our generic scheme offers a framework for reasoning about its instances. One issue is *completeness*: how often does an algorithm succeed (terminate)? We argue that the generic algorithm does not limit the completeness of instances, and show that the proposed generalization of counterexamples is necessary for completeness — thus addressing a shortcoming of more limited notions of counterexample-guided refinement.

2 Preliminaries

A *program* P is a finite automaton over an alphabet A of *actions*, representing the program's control-flow graph. Its states are called *(control) locations*. An action $a \in A$ represents a program instruction; it is a guarded command specifying a conditional update of P's *data state* ($\in D$), i.e. a is a partial function from D to D. We assume w.l.o.g. that every action a has a unique source and destination location in the control-flow graph, and denote these by $\mathit{srcloc}(a)$ and $\mathit{destloc}(a)$ resp. The semantics of P, denoted $\|P\|$, is the transition system whose states ($\in \Sigma$) are pairs (ℓ, d) of a control location ℓ and a data valuation $d \in D$, and in which there is a transition from (ℓ, d) to (ℓ', d') iff there is an action a with $\mathit{srcloc}(a) = \ell$, $\mathit{destloc}(a) = \ell'$, and $a(d) = d'$; in such a case we write $(\ell, d) \xrightarrow{a} (\ell', d')$. (ℓ, d) is initial in $\|P\|$ iff ℓ is initial in the control-flow graph P; we assume there is a unique initial state. The *pre-image* operator is defined by $\mathit{pre}_a(S) = \{s' \in \Sigma \mid \exists_{s \in S}\ s' \xrightarrow{a} s\}$.

Correctness properties of P are expressed as formulas in some temporal logic which is interpreted over $\|P\|$. We assume that the reader is familiar with the general concepts of temporal logic in the context of program verification. Here, we restrict attention to *safety properties* in a *linear-time* logic such as LTL [16]. This ensures that (i) truth of formulas is *preserved* (from the more abstract to the more concrete side) under simulation relations, and (ii) falsehood can be demonstrated by a linear counterexample. The finite set of atomic propositions over which formulas are built is denoted *atoms*; they are predicates over Σ. We use predicates of the form $(@\ell, p)$ where ℓ is a control location and p a predicate over D; such a predicate is true precisely for those states (m, d) where $m = \ell$ and $p(d)$ is true.

A *(predicate) abstraction* Abs is a set of such predicates. Abs is called *finitary* when it has finite cardinality. Every predicate q partitions Σ into those states for which q is true and those for which q is false; the partitioning induced by a set $Q \subseteq \mathit{Abs}$ of predicates is the coarsest that refines each of the partitionings caused by any single $q \in Q$. In other words, in every class C of the partitioning, every $q \in Q$ is either true for all states in C or false for all states in C; and for every two different classes, there exists some $q \in Q$ that is (uniformly) true in one class and false in the other. Abs then determines an *abstract transition system* (denoted $\alpha_{\mathit{Abs}}(\|P\|)$) whose *abstract states* are the classes of the partitioning induced by Abs, and where there is an *abstract transition* between classes C_1 and C_2 iff there exist $s_1 \in C_1$ and $s_2 \in C_2$, such that in $\|P\|$ we have $s_1 \xrightarrow{a} s_2$ for some action a; in such a case we write $C_1 \xrightarrow{a} C_2$. Furthermore, the valuation of an atom q in an abstract state C is defined by $C \models q$ iff $\forall_{s \in C}\ s \models q$. A temporal logic formula can then be evaluated over an abstract transition system.

Depending on the possible shapes of predicates, the questions whether a predicate holds in an abstract state and whether there is an abstract transition between two abstract states may be undecidable. We assume that there are *oracles* for these questions. Oracles are functions that solve problems that are typically undecidable. In order to compare abstraction refinement (semi-)algorithms, we formalize them in a framework in which certain elements are fixed while others can be varied in a controlled way. If we were to fix undecidable subquestions by particular approximation methods, this could blur the comparison because those methods' performances might depend on other parameters which are varying. The use of oracles as "black box subroutines" avoids this.

A *refinement* of an abstraction Abs is a superset of Abs, inducing a finer partitioning of Σ. Such a refinement results in an abstract transition system that possibly satisfies more correctness properties, and certainly not fewer. An abstraction refinement algorithm takes a program P and a correctness property φ as input. Being a semi-algorithm, it may not terminate. If it does terminate, then it returns an abstraction Abs that is *fine for* φ, i.e. either $\alpha_{\mathit{Abs}}(\|P\|) \models \varphi$ (thus demonstrating, by preservation, that $\|P\| \models \varphi$), or $\alpha_{\mathit{Abs}}(\|P\|) \not\models \varphi$ while there exists no refinement $\mathit{Abs}' \supseteq \mathit{Abs}$ for which $\alpha_{\mathit{Abs}'}(\|P\|) \models \varphi$ (which implies that $\|P\| \not\models \varphi$)[2]. Typically, such algorithms are based on successively refining intermediate abstractions (where the initial abstraction can be seen as $\emptyset$). Thus, a major challenge in designing good

[2] Note that we do not pose any requirements on how easy it should be to extract a concrete counterexample (on the level of $\|P\|$) from any counterexample on the level of $\alpha_{\mathit{Abs}}(\|P\|)$.

refinement algorithms is to identify a refinement, given a program P and an abstraction Abs that is too coarse, i.e. not fine for φ.

3 A Generic Abstraction Refinement Algorithm

In many abstraction refinement algorithms that have been proposed in the literature, the identification of new predicates for abstraction refinement can be viewed as a two-step process:

(i) *Collect information on which to base the computation of new predicates.* A typical example is to construct a counterexample from a failed model checking run, which will then guide the refinement.

(ii) *Compute the new predicates from that information.* E.g. this step may involve computation of pre- or post-images of current predicates, or application of widening operators.

We formalize the "information on which to base the computation of new predicates" by the concept of a *refinement cue*, or *cue* for short. Being a generalization of counterexamples, cues play a central role in our comparison of, and reasoning about, refinement algorithms. A cue is a regular language over pairs (a, p) in which a is an action from A and p a predicate. Intuitively, an element of a cue, i.e. a word $(a_1, p_1) \cdots (a_n, p_n)$, suggests the computation of new predicates by viewing the sequences $a_1 \cdots a_n$ of actions as predicate transformers and applying them to the p_i in some way. For example, in typical counterexample-guided refinement approaches, every cue consists of a (single) sequence denoting a counterexample in an abstract transition system. In approaches that do not depend on counterexamples, cues may be sets of single pairs (i.e. languages of "one-letter words"); examples will be given in the next section. Note that a cue can always be finitely represented, e.g. explicitly (enumerating the words) if it is a finite language, and by a finite-state automaton or regular expression otherwise.

Our generic algorithm depends on two functions, whose definitions are the parameters to the algorithm. One, called a *cue selection function* and denoted *selcue*, takes a program, a property, and an abstraction, and returns a refinement cue. The other function is called *refinement function*, denoted *refine*. It takes a refinement cue and returns a set of predicates. Thus, the refinement function abstracts away from the particular way in which new predicates are computed from a cue. For example, a cue may consist of infinitely many words. Computing pre-images over the corresponding sequences of actions might result in an infinite refinement, whence a more "clever" technique might be used, e.g. based on widening functions. The generic algorithm is presented in Figure 1.

```
input: program $P$, property $\varphi$;
$npreds := \emptyset$; $Abs := \emptyset$;
repeat
    $Abs := Abs \cup npreds$;
    $cue := selcue(P, \varphi, Abs)$;
    $npreds := refine(cue)$
until $npreds = \emptyset$;
output: $Abs$
```

Fig. 1. Generic Abstraction Refinement Algorithm

3.1 Completeness

Formulating a generic algorithm is a balancing act. On the one hand, it should fix those elements that are common to all concrete algorithms that it intends to capture, so that the amount of instantiation needed to obtain each of those algorithms is minimal — this allows for better comparisons. In the next section we demonstrate how several abstraction refinement algorithms from the literature are readily expressed as instances of our generic algorithm.

On the other hand it should not fix too much or it might exclude too many algorithms. The following observation pleads for the genericness of our algorithm by showing that it poses no limitations on the "degree of completeness" of its instantiations. An abstraction refinement algorithm is said to *succeed* on an input (of a program and a formula) if it terminates on that input; otherwise it *fails*. We call algorithm A_1 *more complete* [3] than A_2 if, for every input on which A_2 succeeds, A_1 succeeds as well. An abstraction refinement algorithm is *complete* if it succeeds on every input. Clearly, if *selcue* and *refine* are required to be computable functions, then the generic algorithm has no complete instances. However, if we are allowed to utilize oracles in defining these functions, then a complete [4] instance exists. The reason is that the statement of the verification problem, $P \models \varphi$, can be encoded in terms of a refinement cue. This is done by rephrasing this statement as the emptiness problem of the program obtained as the synchronous product $P \times T_{\neg\varphi}$ where $T_{\neg\varphi}$ is the tableau automaton for $\neg\varphi$ [15,9]. As φ is a safety property, this product is a finite-state automaton, representing a regular language L. Its alphabet consists of actions from P combined with atoms from φ. We let $selcue(P, \varphi, Abs)$ be the refinement cue L, hence this cue contains all the information needed to determine whether $P \models \varphi$. Now suppose that we have an oracle that can tell us whether $P \models \varphi$. The function *refine*, when applied to L, first calls this oracle. If the oracle responds that $P \not\models \varphi$, then *refine* returns any abstraction (e.g. $\emptyset$), since it is certain that no abstraction *Abs* exists such

[3] Note that the word "more" is to be interpreted as "greater than or equal".

[4] Of course, completeness is relative to the unavoidable "Gödelian" incompleteness of the assertion language used to express predicates.

that $\alpha_{Abs}(\|P\|) \models \varphi$ (because that would imply $P \models \varphi$ by preservation). If the oracle says that $P \models \varphi$, then *refine* uses another oracle, namely one that gives the proper invariant needed to demonstrate emptiness of the semantics $\|P \times T_{\neg\varphi}\|$ of the product automaton. Note that $\|P \times T_{\neg\varphi}\|$ may be an infinite transition system and constructing the invariant is undecidable in general. That such an oracle nevertheless exists follows from the completeness results about finitary abstractions in [21,12]. Note that *selcue* is in fact computable — all "oracle power" goes into *refine*, and the oracles are so powerful that no iteration is needed in this case.

This argument sets our proposal apart from the oracle-guided widening method of [1]. The purpose of that paper is to show that a particular real (implementable) algorithm is equally complete as their oracle-guided widening method. Being tailored towards that algorithm, the widening method is not general enough to capture other abstraction refinement algorithms. As observed in [19], that particular widening method would have to be generalized and made into a parameter of the method.

4 Comparing Abstraction Refinement Algorithms

Oracles are useful in reasoning about algorithms that attempt to solve undecidable problems. When it comes to implementing abstraction refinement algorithms, we must do without them however. In practice, the power of oracles is usually substituted by an iterative approach in which heuristics are used to find ever better abstractions. In this section we rephrase some such real algorithms as instances of our framework.

Throughout the sequel, we assume that $selcue(P, \varphi, Abs)$ always first performs a verification (e.g. a model checking run) in order to determine whether $\alpha_{Abs}(\|P\|) \models \varphi$. If so, it returns the empty cue, causing the algorithm to terminate (we assume that $refine(\emptyset) = \emptyset$). Otherwise, it calls another function (with the same arguments), $selcue_1$. Different possibilities for this function are considered below. The refinement functions are given by specifying their effect on a single word; the effect on a set of such words is then obtained by a standard lifting to sets.

Recall that *atoms* is the finite set of atomic propositions of the temporal logic. $atoms(P)$ denotes the set of atoms that occur in the guards or the initial condition of P, $atoms(\varphi)$ the atoms that occur in φ. $actions(P)$ denotes the set of actions that occur in program P.

4.1 Non-guided refinement

The first abstraction refinement algorithm that we consider, from [18], is "non-guided": the selection of cues is not based on a counterexample produced by a model checking run. Cues are formed by taking all pairs of a single action from the program with a predicate from the current abstraction. The programs considered in [18] are assumed to have a single control location ℓ_0, so all control flow needs to be encoded in variables.

$$selcue_1(P, \varphi, Abs) = \begin{cases} \{(a, (@\ell_0, p)) \mid a \in actions(P), p \in atoms(P) \cup atoms(\varphi)\} & \text{if } Abs = \emptyset \\ \{(a, q) \mid a \in actions(P), q \in Abs\} & \text{otherwise} \end{cases}$$

Predicates are quantifier-free formulas from a first-order logic. The refinement function computes the *syntactic weakest precondition* relative to an action a. This is done by performing substitution, possibly followed by quantifier elimination and other rewriting steps; we denote this process by $swlp_a$. The atoms that occur in the resulting formula are then used to refine the abstraction.

$$refine((a, (@\ell_0, p))) = \{(@\ell_0, p') \mid p' \in atoms(swlp_a(p))\}$$

In [18], this strategy is shown to be complete for the class of inputs P, φ in which P has a finite reachable bisimulation quotient (w.r.t. $atoms(\varphi) \cup atoms(P)$), and $P \models \varphi$.

If the control flow can be made more explicit in the form of a non-trivial control-flow graph, then its structure can be used to limit the set of cues, by tying predicates more closely to control locations. Function $selcue_1$ would be as above but it would limit the set of cues $(a, (@\ell, p))$ it returns to those for which $destloc(a) = \ell$. This approach is taken in [4]. Similar non-guided abstraction refinement algorithms have been presented (in the context of finite-state programs) in [3,5].

4.2 Counterexample-guided refinement

In counterexample-guided approaches, counterexamples that are produced by the model checking runs are used as cues for refinement. A counterexample is a sequence $C_0, a_1, C_1, \ldots, C_{n-1}, a_n, C_n$ of abstract states C_i (these are equivalence classes of concrete states) and actions a_i of P such that C_0 is the initial abstract state, $C_i \xrightarrow{a_{i+1}} C_{i+1}$ for all i, and φ is not satisfied along the path $C_0, C_1, \ldots, C_n$. Given an abstract transition system S for P, let $mc(S, \varphi)$ be a model checking procedure that returns the empty set if $S \models \varphi$ and a singleton with a counterexample otherwise.

$$\begin{aligned} selcue_1(P, \varphi, Abs) = \{&(a_1, (@destloc(a_1), C_1)) \cdots (a_n, (@destloc(a_n), C_n)) \mid \\ &C_0, a_1, C_1, \ldots, C_{n-1}, a_n, C_n \in mc(\alpha_{Abs}(\|P\|), \varphi)\} \end{aligned}$$

Refinement triggered by such a cue seeks to weed out the counterexample by splitting one of its states. One example is the following inductive definition of *refine*, which captures the essence of the algorithm CouAnal in [14] (ϵ denotes the empty sequence of pairs, $s \cdot p$ the sequence s with pair p appended at its end, and s^- the sequence s with its last pair removed):

$$refine(\epsilon) = \emptyset$$

$$refine(s \cdot (a_i, (@destloc(a_i), C))) = \quad \text{(when } i \geq 1\text{)}$$
$$\begin{cases} (@destloc(a_i), C) & \text{if } C_{i-1} \cap pre_{a_i}(C) = \emptyset \\ refine(s^- \cdot (a_{i-1}, (@destloc(a_{i-1}), C_{i-1} \cap pre_{a_i}(C)))) & \text{otherwise} \end{cases}$$

Note that this functions performs a *backwards* computation along the counterexample, using pre-image operators. The algorithm SplitPATH from [2] performs a similar refinement in the *forward* direction.

4.3 Considering loops in counterexamples

The refinement algorithm above could be called *single-counterexample-guided*, as it refines w.r.t. one counterexample at a time, introducing a predicate that weeds out the counterexample. If a counterexample corresponds to a loop in the program, then it may be that the next counterexample found is similar to the one that just has been removed, corresponding to a path that takes one more iteration through the loop. In such a case, a single-counterexample-guided algorithm could go on forever, removing one by one the infinitely many counterexamples that are generated by the loop in the control-flow graph. This phenomenon is addressed in [14]. Before discussing the solution proposed, we repeat the example of [14]. Consider the (infinite state) program depicted in

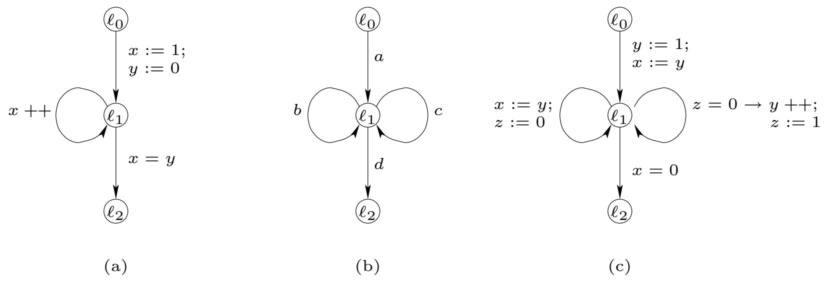

Fig. 2. Counterexamples with loops

Figure 2(a), and suppose we want to check the property that location ℓ_2 is not reachable. The single-counterexample-guided algorithm described above will introduce the predicates $x = y$, $x = y - 1$, $x = y - 2$, ... at ℓ_1 (we omit the details), aimed at weeding out the different counterexamples that correspond to successive unfoldings of the loop. Clearly, this approach will not succeed in establishing the property.

Note that a non-guided algorithm might "accidentally" find a predicate that does help to establish the property, as the result of considering some other precondition. For example, if we add another location, ℓ_3, to the program, and an edge from ℓ_1 to ℓ_3 with action $x \leq y$, then $x \leq y$ would be found as a predicate of interest at ℓ_1; at the same time, it would rule out all counterexamples around the loop, in "one fell swoop". This shows that the single-counterexample-guided algorithm cannot be more complete than the non-guided algorithm.

In [14], a step towards more complete algorithms is made by considering all counterexamples around the loop at once, and using an *acceleration* of the precondition calculations along it. When a segment L of the counterexample corresponds to a loop, then the union is computed of all iterated pre-image sets along L ("pre_L^*") as part of the refinement. Even in cases where single pre-image sets are computable, such an acceleration function may be undecidable as it may involve infinitely many pre-image computations — it is like constructing a loop invariant. In other words, an oracle is needed, or, for practical purposes, an approximation method. The resulting algorithm is called AccCouAnal. A similar approach is proposed in [2], but since it considers finite state programs, no acceleration is needed.

4.4 A shortcoming

The approaches to abstraction refinement discussed above propose several notions of refinement cue (using single actions, single counterexamples, multiple counterexamples caused by loops), in an attempt to arrive at more complete refinement algorithms. Each of the *selcue* functions discussed above is computable (up to the computation of abstract transitions that is needed in model checking — for this we assume that an oracle is available, see Section 2). The *refine* functions are not, and when viewing them as oracles, we see that stronger oracles are needed as we move towards more sophisticated notions of refinement cue.

Let's assume that we have an oracle that meets the demands of algorithm AccCouAnal of [14], i.e. it can essentially compute pre-images over loops. Is the resulting algorithm complete? The answer is no. On an intuitive level, the explanation is that although all unfoldings of all loops in the counterexample are considered (and this can be done, as noted in [14], for all possible partitionings of the counterexample into loops), the *interaction among different loops* is not considered. In other words, only a *collection of loops* is considered where in fact one would need to reason about a *strongly connected component*. As an example, consider the program of Figure 2(b), where a, b, c, and d are actions. Suppose that during the course of abstraction refinement the counterexample abd occurs. Since b sits on a loop, a refinement would be computed based on all of the counterexamples in ab^*d. Using this refinement, we might find as the next counterexample $abcbd$. Taking into account all different partitionings into loops, a refinement would be computed weeding out all counterexamples of any one of the forms $ab^*c^*b^*d$, $a(bc)^*b^*d$, $ab^*(cb)^*d$, and $a(bcb)^*d$. Continuing with the refinement thus found, the next counterexample might be $abbcbbcbbd$, which is not included in any of the above forms. Indeed, if we continue to remove sequences according to this scheme, there will always remain sequences that have not been considered. What is missing is a refinement based on $a(b^*c^*)^*d$.

A more concrete example is the program in Figure 2(c), with the property of interest being unreachability of ℓ_2.

5 Discussion

The observations above are a strong indication that the generalization that is suggested by the concept of refinement cues is essential if we are not to limit the "degree of completeness" in the search for better refinement algorithms. The discussion in the previous section showed that, regardless of the power of the oracle-based refinement function used, even dealing with multiple loop-generated counterexamples at the same time may not result in a complete algorithm. In refinement cues, these notions are generalized to regular languages. The observation, in Section 3, that a complete algorithm exists within the framework of our generic algorithm, implies that this generalization suffices. It suggests novel approaches to refinement, guided by *counterexample automata*. A counterexample automaton is a representation of *all* counterexamples to a property. For example, a refinement function taking a cue consisting of such an automaton could compute iterated weakest preconditions over the structure of the automaton, identifying new relevant predicates associated with the automaton's locations (which correspond to program locations).

An interesting question is whether such a counterexample automaton can be decomposed into (finitely many) smaller automata so as to allow for a modular approach. In a way, counterexample automata can be viewed as mini-programs that are "error slices". The smaller they can be, the easier it will be for a human user to understand the cause of the counterexamples. In the end, such interaction remains necessary as the general problem we are dealing with is undecidable. One option for decomposing the task is to consider the counterexample automaton for one error at a time. An error could be defined as a final (accepting) state of the product of the program's abstracted state-transition system (relative to some current abstraction) with the tableau automaton of the negated property. The counterexample automaton for one such error would then be the sub-automaton of this product consisting of all reachable states from which the error can be reached.

Abstraction refinement for liveness

Taking automata as counterexamples also provides the right starting point for considering abstraction refinement in the context of liveness properties. Counterexamples to liveness properties are by definition infinite sequences, and, when these are generated by a finite abstraction, then they must involve loops. Such a refinement will necessarily involve the identification of additional *ranking functions*, as opposed to additional predicates for the case of safety properties.

Acknowledgement

Kedar Namjoshi and the anonymous referees are thanked for their suggestions.

References

[1] Ball, T., A. Podelski and S. K. Rajamani, *Relative completeness of abstraction refinement for software model checking*, in: Katoen and Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference*, LNCS **2280** (2002), pp. 158–172.

[2] Clarke, E., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Counterexample-guided abstraction refinement*, in: Emerson and Sistla [8], pp. 154–169.

[3] Dams, D., R. Gerth and O. Grumberg, *Generation of reduced models for checking fragments of CTL*, in: C. Courcoubetis, editor, *Computer Aided Verification*, number 697 in LNCS (1993), pp. 479–490.

[4] Dams, D. and K. S. Namjoshi, *Shape analysis through predicate abstraction and model checking*, in: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, number 2575 in LNCS (2003), pp. 310–323.

[5] Dams, D. R., "Abstract Interpretation and Partition Refinement for Model Checking," Ph.D. thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands (1996).

[6] Das, S., D. L. Dill and S. Park, *Experience with predicate abstraction*, in: *Computer Aided Verification*, 1999, pp. 160–171.

[7] E.M. Clarke, J., O. Grumberg and D. Peled, "Model Checking," MIT Press, 2000.

[8] Emerson, E. A. and A. P. Sistla, editors, "Computer Aided Verification," Springer, Berlin, 2000.

[9] Gerth, R., D. Peled, M. Y. Vardi and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, in: *Protocol Specification Testing and Verification* (1995), pp. 3–18.

[10] Graf, S. and H. Saidi, *Construction of abstract state graphs with PVS*, in: O. Grumberg, editor, *Computer Aided Verification*, number 1254 in LNCS (1997), pp. 72–83.

[11] Henzinger, T., R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in: *POPL*, 2002, pp. 58–70.

[12] Kesten, Y. and A. Pnueli, *Verification by augmented finitary abstraction*, Information and Computation **163** (2000), pp. 203–243.

[13] Kurshan, R. P., "Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach," Princeton Series in Computer Science, Princeton University Press, Princeton, NJ, 1994.

[14] Lakhnech, Y., S. Bensalem, S. Berezin and S. Owre, *Incremental verification by abstraction*, in: T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference*, LNCS **2031** (2001), pp. 98–112.

[15] Lichtenstein, O. and A. Pnueli, *Checking that finite state concurrent programs satisfy their linear specification*, in: *Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGACT/SIGPLAN, 1985, pp. 97–107.

[16] Manna, Z. and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems: Specification," Springer-Verlag, New York, 1992.

[17] Namjoshi, K., *Abstraction for branching time properties*, in: W. A. Hunt and F. Somenzi, editors, *Computer Aided Verification*, number 2725 in LNCS (2003), pp. 288–300.

[18] Namjoshi, K. S. and R. P. Kurshan, *Syntactic program transformations for automatic abstraction*, in: Emerson and Sistla [8], pp. 435–449.

[19] Podelski, A., *Software model checking with abstraction refinement* (2002), invited talk at the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI).

[20] Sipma, H. B., T. E. Uribe and Z. Manna, *Deductive model checking*, in: R. Alur and T. A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification*, number 1102 in LNCS (1996), pp. 208–219.

[21] Uribe, T., "Abstraction-based Deductive-Algorithmic Verification of Reactive Systems," Ph.D. thesis, Computer Science Department, Stanford University (1998), technical report STAN-CS-TR-99-1618.