Counterexample Analysis in Abstract Regular Tree Model Checking of Complex Dynamic Data Structures

Lukáš Holík[1] and Adam Rogalewicz[1,2]

[1] FIT BUT, Božetěchova 2, CZ-61266, Brno, e-mail: holik@fit.vutbr.cz
[2] VERIMAG, CNRS, 2 av. de Vignate, F-38610 Gières, e-mail:rogalewi@imag.fr

Abstract. We focus in details on the use of abstract regular tree model checking (ARTMC) within a successful method for verification of programs with dynamic data structures. The method is based on a use of a set of transducers to describe the behaviour of the verified system. But the ARTMC method was originally developed for systems of one transducer only and its generalization to several ones was supposed to be straightforward. Although this straightforward generalization (used in a prototype implementation) works well in a number of examples, the counterexample analysis is in general unreliable and can cause infinite looping of the tool as we demonstrate on a simple example containing an erroneous pointer manipulation. Therefore we propose a new algorithm used for a counterexample analysis and we prove its correctness.

1 Introduction

Use of the pointers and dynamic data structures is a common technique allowing one to effectively handle unbounded data structures using a finite number of pointer variables. But on the other hand the code of such programs is hard to understand and error prone. The detection of errors can be really hard, because the data structure itself is hidden behind pointer manipulations. Therefore methods for formal verification of such programs are quite welcome. Various approaches differing in their principles, degree of automation, generality, and scalability have been proposed (e.g. [5, 7, 6, 9]). The method which we concentrate on in this work is based on *abstract regular tree model checking* [1, 2].

The aim of the method [2] is to handle non-recursive C programs (with variables over finite data domains) manipulating dynamic linked data structures with possibly *several* next pointer selectors. The properties considered are basic consistency of pointer manipulations (no *null pointer assignments*, etc.). Further undesirable behaviour of the verified programs (such as an introduction of undesirable sharing, cycles, etc.) may be detected via *testers written in C* and attached to the verified procedures.

The principles of the abstract regular tree model checking (ARTMC), basis of the discussed method, are fully described in [1]. In regular tree model checking, configurations of a system being examined are encoded as trees over a suitable ranked alphabet. Then possibly infinite sets of such configurations are naturally

captured by finite tree automata. Transitions of a system are encoded as tree transducers, or by more general regularity preserving relations (see e.g. [4]). One can use one transducer for the whole behaviour of the system, or several ones—e.g. one for each program line. Subsequently, one computes the set of all configurations reachable from an initial set by repeatedly applying the tree transducers on the set of the so-far reached configurations. In order to make the method terminate as often as possible and to fight the state explosion problem, various kinds of automatically refinable abstractions over automata are used in ARTMC. The whole computation is then based on the CEGAR [3] principle. If a counterexample is found, one needs to check whether it is a real counterexample, or just an error behaviour introduced by the used abstraction. The computation is stopped in the first case and the error trace is provided. In the second case, one needs to refine the abstraction and restart the computation.

The CEGAR loop within the context of ARTMC was originally studied for the case of one transducer. The generalization to a set of transducers was supposed to be straightforward. A set of reachable configurations is computed for each program location, and the computation is done by applying the transducers in depth-first search (DFS) order according to the control flow of the program. Then the discovered counterexample is propagated backwards by the transitions on the path leading from the actual position to the root of a DFS tree. If a nonempty intersection with the initial sets is found, then the counterexample is a real one. The problem is that the path in the DFS tree leading to the counterexample can contain some loops in the control flow of the program, but the real execution path leading to the counterexample may skip some of these loops. Then, some of the backward transitions inside such loops may not be defined and one obtains an empty intersection with the set of initial configuration.

In spite of this fact, this generalization can work on a quite lot of examples, as was proved by experiments with various programs in [2]. However, it is problematic because the detection of spurious counterexample is not reliable—a real counterexample can be claimed as a spurious one and the CEGAR loop is not stopped. Note that the correctness of the answer cannot be affected.

In this article, we focus on the use of more transducers to describe the behaviour of the verified system. We in details describe, how sets of all reachable configurations are computed and how counterexamples are analyzed. As we already mentioned, the simple counterexample analysis used in the original implementation can declare a real counterexample as a spurious one. Then, instead of reporting the counterexample, the CEGAR loop is reiterated, and the verification process may not terminate. We show this problem on a simple example containing an incorrect pointer manipulation. Then, we propose a fixed algorithm for handling counterexamples and we prove its correctness.

2 Applying ARTMC to Dynamic Data Structures

In this section, we shortly describe the use of ARTMC technique for the programs manipulating dynamic data structures originally proposed in [2]. Then we show the problem in counterexample analysis.

In general, configurations of pointer manipulating programs are unrestricted directed graphs (often called shape graphs). In order to be able to apply ARTMC, one needs an efficient encoding of their configurations into trees and tree automata. Such an encoding was proposed in [2]. Trees are used to encode a *tree skeleton* of a shape graph. The edges of the shape graph that are not directly encoded in the tree skeleton are then represented by *routing expressions* over the tree skeleton—i.e., regular expressions over directions in a tree (as, e.g., left up, right down, etc.) and the kind of nodes that can be visited on the way. Both the tree skeletons and the routing expressions are discovered automatically.

Then it was showed, how all considered *pointer-manipulating statements* of the C programming language (without pointer arithmetic, recursion, and with finite-domain non-pointer data) may be automatically translated to (i) tree transducers manipulating the proposed tree skeletons, and (ii) an algorithm for updating the routing expressions. The translation is proposed in such a way that it allows to use ARTMC technique to compute the overapproximations of the reachable sets. To simplify the following text, let us suppose that one transition of the program is described by one tree transducer only—the updates of routing expressions are not taken in account.

This is exactly the case, where several transducers are used in the context of ARTMC. As was already mentioned, an overapproximation of reachable configurations is computed for each program line separately starting from an initial set of shape graphs (corresponding to the initial line of the program). The depth-first strategy is used according to the program control flow graph when iterating the transducers corresponding to the particular program lines (for more details see Section 3). Sets of so-far computed reachable configurations are continuously examined for an nonempty intersection with the set of error configurations.

Conterexample Analysis. As was already mentioned, the use of several transducers within the context of ARTMC may bring some problems with the counterexample analysis, because this analysis is not so straight-forward as was originally supposed. The problem can cause that a real counter-example is declared as a spurious one, and the computation is not stopped. Moreover, in the case of spurious counterexample, one does not necessary correctly localize the source of the overapproximation leading to an error behaviour, and the abstraction refinement could not be correct. Let us now demonstrate the problem on an example depicted in Figure 1. This procedure changes the value stored in the last node od a singly-linked list (pointed by pointer *top*). In the case of empty list ($top = NULL$), an undefined pointer exception arise at the line 5.

Let us take all possible singly-linked lists (including the empty one) as an initial set of configuration. Due to the depth-first strategy, the forward procedure tries first to compute fixpoint of the loop $2, 3, 4$ (using ARTMC to accelerate), and then it continues by the step 5. Let us now suppose that the fixpoint is computed after two iterations of the loop. Then an *error trace* leading to a counterexample is the following: $1, 2, 3, 4, 2, 3, 4, 2, 5$. But the path leading to the counterexample in the control flow graph of the program is $1, 2, 5$.

The counterexample is a real one, if one can find a configuration from the initial set, from which the error configuration is reachable. To find such a configuration, backward execution, starting from the counterexample computed on the line 5[3] is used. The problem is that the backwards statement 4 is not defined on the encountered error configuration—empty list ($top == NULL$) can never be result of line 4. Therefore after the backward execution of the last three steps of the error trace $(5, 2, 4)$, one obtains an empty set, and a nonempty intersection with initial configurations is not find.

```
1: y = top;
2: while(y! = NULL) do
3:     end = y;
4:     y = y → next;
   done
5: end → data = 7;
```

Fig. 1. An example with incorrect pointer manipulations

To correct this problem, we need to analyze also error traces, which are obtained from the encountered error trace by skipping some loops. (In our example, we need to analyze trace $1, 2, 5$.) In Section 3, we present an efficient backward algorithms, which allows us to correctly decide spuriousness of counterexamples.

3 Forward and Backward Algorithm

The main CEGAR loop is represented by algorithm in fig. 2. The computation of reachable sets is represented by the function *GoForward*. To analyze the counterexamples, the function *GoBackward* is used. We present two variants of the function *GoBackward* in this paper—the old problematic one and the repaired one for which we prove correctness.

A program is modeled as a tuple of lines, where each line contains a specification of its actions—tuples (τ, l), where τ is a tree transducer performing an action on our tree memory encoding, and l is an identificator of a successor location in the control flow. There can be more than one action from a single line—e.g. *then* and *else* branch in the case of an *if* statement.

Definition 1. *A* program *is tuple* $P = P(1), ..., P(n_P)$ *where* $P(i)$ *is a set of tuples* (τ, l) *such that* $0 \leq l \leq n_p$ *is number of the next line and* τ *is a transducer.*

Given a program P, a set *Init* of all initial configurations of memory and a set of error configurations *Bad*, our task is to check, whether it is possible to get an error configuration starting the program with a configuration from *Init*.

Definition 2. *A* verification sequence *is finite sequence* $s = s(0), \ldots s(n_s)$ *where each* $s(i)$ *is a couple consisting of a tree automaton representation of a set of configurations of the program memory* $conf_s(i)$ *and an integer* $line_s(i)$.

Given verification sequence s*, we denote* s_i *the* i*-th suffix* $s(i), \ldots, s(n_s)$ *of* s*. The word* $trace_s = line_s(0), \ldots, line_s(n_s)$ *from* $\mathbb{N}_0^\star$ *is called a* trace of s*. Let* w, v *be two words of alphabet* Σ*. We say that* v *is* contraction of w *iff* $w = v$ *or if* $w = x.y.a.u.a.z$ *and* v *is also contraction of* $y.a.z$*. Let* s, s' *be verification sequences. Then* s *is contraction/prefix/suffix of* s' *iff* $trace_s$ *is contraction/prefix/suffix of* $trace_{s'}$*. Given program* P*, verification sequence* s *is* sequence of P *iff for all* $0 < i \leq n_s$ $conf_s(i) = \tau(conf_s(i-1))$ *where* $(\tau, line_s(i)) \in P(line_s(i-1))$.

[3] actually, it is only one configuration with $top == NULL$, y and *end* are undefined

The set of all sequences of the program describes entire behaviour of the program. The contraction of a sequence intuitively means the sequence, where one skips some of the loops of the original sequence. To check correctness of the program, we are systematically going through all possible sequences of the program using DFS strategy. As the number of such sequences is in general infinite, we would never terminate. Therefere we are not going through the all sequences, but through some kind of their overapproximations.

Definition 3. *Given an abstraction function α and a program P, verification sequence m is* multisequence of P *iff for all* $0 < i \leq n_m$, $conf_m(i) = \alpha\big[\tau(conf_m(i-1)) \cup \bigcup_{\{j \in \mathbb{N} | j<i \wedge line_m(j)=line_m(i)\}} conf_m(j)\big]$ *where* $(\tau, line_m(i)) \in P(line_m(i-1))$.

Given a sequence s of P we say that s is included in *m iff $trace_s$ is contraction of $trace_m$ and $\forall i \leq n_s.\ \exists j \leq n_m.\ conf_s(i) \subseteq conf_m(j) \wedge line_s(i) = line_m(j)$ and $line_s(n_s) = line_m(n_m)$.*

We denote $Starts_m(i, c)$ the set of all sequences s of P included in m such that $line_s(0) = line_m(i), conf_s(0) = conf_m(i), conf_s(n_s) \subseteq c$, and we denote $Incl_m(i, c)$ the set of all sequences included in m_i leading to c, i.e. the set $\bigcup_{i \leq j \leq n_m} Starts_m(j, c)$.

Fig. 2. ARTMC main loop

Input: set of initial configurations *Init*, set of bad configurations *Bad*, initial abstraction function α, program P, array *Configs* of configurations.
Output: verified or error trace found
Data: *direction*, *oldestBad*,

```
1 forall config ∈ Configs do config ← ∅;
2 repeat
3     oldestBad ← ∅;
4     direction ← forward;
5     if GoForward(0,Init)≠ ∅ then return error trace found;
6     α ← RefineAbstraction(α, oldestBad);
7 until oldestBad ≠ ∅ ;
8 return verified
```

The computation of the reachable sets based on DFS is described in Fig. 2. The array *Configs* is used to store so-far computed configurations—the set *Configs*$[i]$ contains configurations reachable on the line i. On the beginning, we set *Configs*$[0] = Init$ and $\forall i \neq 0 : Configs[0] = \emptyset$. At the end, the set *Configs*$[i]$ contains an overapproximation of the configurations reachable on the line i. The root of our DFS-tree contains couple $(Init, line\ 0)$. Each configuration set we reach is unioned with the corresponding set *Configs*$[i]$—the configurations discovered before. Then the branch is propagated from abstraction of this union. A branch will be ended, if nothing new is added to the corresponding set *Configs*$[i]$.

At the end, the sets stored in the array *Configs* are overapproximations of all reachable configurations. Therefore an empty intersection with the set *Bad* guarantees the correctness of the program. In the case of nonempty intersection, we have to check, whether this counterexample is caused by the used abstraction. The multisequence of the branch includes a set of sequences of the program, but

Fig. 3. function GoForward(thisRowNo,newConfiguration)

```
1  if newConfiguration ⊆ Configs[thisRowNo] then
2      return ∅;
3  if newConfiguration ∩ Bad = ∅ then
4      direction ← backward;
5      return newConfiguration ∩ Bad
6  else
7      Configs[thisRowNo] ← α(Configs[thisRowNo] ∪ newConfiguration);
8      forall (τ, nextRowNo) ∈ P(thisRowNo) do
9          nextConfiguration ← τ(Configs[thisRowNo]);
10         badFromTop ←GoForward(nextRowNo, nextConfiguration);
11         if direction = backward then
12             return GoBackward(badFromTop,τ,thisRowNo)
13     return ∅
```

not all of them starts by the $(c, 0), c \subseteq Init$, some of them are started higher in the stack due to the abstraction. Thus we want to find the deepest place in the stack, where such a sequence leading to Bad starts. If the deepest place is the bottom of the stack , then the counterexample is real. If not, we can use the information about its exact location and shape to refine the abstraction.

Fig. 4. function GoBackward(τ,badFromTop,thisRowNo)

```
/* previous problematic version */
1  badHere ← τ⁻¹(badFromTop) ∩ Configs[thisRowNo];
2  if badHere ≠ ∅ then oldestBad ← badHere;
3  return badHere;
```

Fig. 5. function GoBackward(τ,badFromTop,thisRowNo,nextRowNo)

```
/* repaired version */
Data: BadOnLines : static array initialized as array of ∅ of length n_P + 1
1  BadOnLines[nextRowNo] ← BadOnLines[nextRowNo] ∪ badFromTop;
2  badHere ← τ⁻¹(BadOnLines[nextRowNo]) ∩ Configs[thisRowNo];
3  if badHere ≠ ∅ then  oldestBad ← badHere;
4  return badHere;
```

Correctness of the counterexample analysis. Here we show basic correctness properties of the repaired version. The case when a fixpoint is reached and there is no intersection with the set Bad is transparent—the program conforms to the given properties. Let us concern on the case of nonempty intersection with the set Bad. We are able to give a warranty that (i) if the intersection with Bad is caused by a real bug in the program, then we discover it and stop the CEGAR loop, and (ii) in the case of a spurious counterexample, we are able to refine the abstraction in such a way that the encountered spurious counterexample is excluded in the following iterations of the CEGAR loop[4]. In order to express and show this properties properly we introduce the following notation:

[4] This warranty is provided for the predicate based and finite height abstraction [1].

Let us look at the procedural stack in the moment of computation, when the intersection with *Bad* was returned by `GoForward`. We index recursive calls of `GoForward` (and also nested calls of `GoBackward`) according to their depth in the stack by indexes from 0 to *top*. We index the values of variables in the moment of returning from $\texttt{GoForward}_i$ by i and verification sequence in the stack $m = (\mathit{Configs}(\mathsf{thisRowNo}_0), \mathsf{thisRowNo}_0), \ldots, (\mathit{Configs}(\mathsf{thisRowNo}_{top}), \mathsf{thisRowNo}_{top})$ is obviously a multisequence of P corresponding to the last branch of the depth-first traversal. We denote deep_m the position in m, where for the first time was created (by the used abstraction) a configuration leading to the encountered spurious counterexample, i.e. $\mathit{deep}_m = \min\{0 \leq i \leq n_m \mid \mathit{Starts}_m(i, \mathit{Bad}) \neq \emptyset\}$.

1. If the counterexample is real and there are sequences of the program included in m which starts by $(c, 0), c \subseteq \mathit{Init}$ and leads to *Bad*, then union of such c will be returned by the $\texttt{GoForward}_0$.
2. If there is no sequence included in m starting by $(c, 0), c \subseteq \mathit{Init}$, leading to *Bad*, then $\texttt{GoForward}_0$ returns $\emptyset$ and $\mathit{oldestBad}_0$ contains the starting configurations of sequences included in m, starting in deep_m, leading to *Bad*.

At the point 1, we declare the counterexample as a real one and the set c as a set of *dangerous* initial configurations. At the point 2, we refine abstraction α to abstraction β using information about the set $\mathit{Deep} = \bigcup_{s \in \mathit{Starts}_m(\mathit{deep}_m, \mathit{Bad})} \mathit{conf}_s(0)$ contained in $\mathit{oldestBad}_0$. The set *Deep* is a set of configurations, which were introduced to $\mathit{conf}_m(\mathit{deep}_m)$ by abstraction α—it is the source leading finally to the spurious counterexample. As is proved in [1] for the predicate based abstraction, the knowledge of *Deep* allows us to guarantee that $\mathit{Deep} \not\subseteq \beta(c) \Leftrightarrow \mathit{Deep} \subseteq \alpha(c)$, where c is any set of configurations.

From the previous, we are able to state that (1) if we are iterating through some sequence starting in initial configuration leading to bad configuration, we detect it, and (2) on the other hand, although ARTMC method is from its nature not guaranteed to terminate, the progress is guaranteed. A loop, where we are still going through the same misleading sequences finding still the same bad configuration sets (as in the example from Section 2) is not possible.

Both of the properties 1. and 2. depends on correct detection of the set $\bigcup_{s \in \mathit{Starts}_m(\mathit{deep}_m, \mathit{Bad})} \mathit{conf}_s(0)$. This is to be showed in the rest of the section.

During the backward run, we are returning from recursion and examining sequences included in m leading to *Bad* in a direction from their endpoints down to their starting points. Our aim is for each i to detect the set $\bigcup_{s \in \mathit{Starts}_m(i, \mathit{Bad})} \mathit{conf}_s(0)$. This set is computed on line 2 of $\texttt{GoForward}_i$. For that purpose we maintain the array *BadOnLines*, where for each $0 \leq l \leq n_P$ the cell $\mathit{BadOnLines}_i[l]$ contains union of the initial configuration sets of sequences leading to *Bad*, starting at line l from $i+1$-th position of the stack or higher, i.e. the set $\bigcup_{s \in \mathit{Incl}_m(i+1, \mathit{Bad}) \wedge \mathit{line}_s(0) = l} \mathit{conf}_s(0)$. Thus if also $\mathit{BadOnLines}_i[\mathit{line}_m(i+1)]$ is computed correctly (Lemma 1), then the value of $\tau_i^{-1}(\mathit{BadOnLines}_i[\mathit{line}_m(i+1)]) \cap \mathit{Configs}_i[\mathit{line}_m(i)]$ equals $\bigcup_{s \in \mathit{Starts}_m(i, \mathit{Bad})} \mathit{conf}_s(0)$ and thus even $\bigcup_{s \in \mathit{Starts}_m(\mathit{deep}_m, \mathit{Bad})} \mathit{conf}_s(0)$ will be computed correctly (Lemma 2).

The following lemma shows that the values of *BadOnLines* are always computed correctly in the sense of the preceding paragraph.

Lemma 1. *For the Algorithm in fig. 2 is valid the following: For all* $0 < i < n_m$ *and for each* $0 \leq l \leq n_P$, $BadOnLines_{i-1}[l] = \bigcup_{s \in Incl_m(i,Bad) \wedge line_s(0)=l} conf_s(0)$.

Lemma 1 allows us to prove the Lemma 2, which implies the two discussed correctness properties of the counterexample analysis.

Lemma 2. *At the end of backward computation, oldestBad is union of all* $conf_s(0)$ *such that s is deepest sequence included in m leading to Bad, i.e.* $\bigcup_{s \in Starts_m(deep_m, Bad)} conf_s(0) = oldestBad_0$.

To show the correctness properties we need the $\supseteq$-part of Lemma 1 only. The point of the $\subseteq$-part is that as the $oldestBad_0$ contains the smallest correct set, abstraction will not be refined more than necessary. It can be seen as a benefit, because the more general is abstraction, the faster is convergence to the fixpoint.

4 Conclusion

We discussed the use of more than one transducer to describe behaviour of the examined system in the context of ARTMC. We have shown on a simple example that the original counterexample analysis used in the tool from [2] is not reliable—it can claim a real counterexample as a spurious one and the algorithm may never terminate. As a part of this paper, we have provided a correct counterexample analysis and we have proven its correctness.

As a part of our future work, we plan to investigate a possibility of use of nondeterministic automata to avoid the expensive determinisation step and to investigate a possibility to use footprint analysis [8] in the context of ARTMC.

Acknowledgement. This work was supported by the Czech Grant Agency project 102/05/H050, and by the Czech-French project Barrande 2-06-27.

References

1. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006.
2. A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, Springer, 2006.
3. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV'00*, *LNCS* 1855. Springer, 2000.
4. P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. In *Proc. of ATVA'07* (to appear).
5. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001.
6. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.
7. S. Sagiv and T.W. Reps and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic TOPLAS, 24(3), 2002.
8. C. Calcagno and D. Distefano and P. O'Hearn and H. Yang Footprint Analysis: A Shape Analysis That Discovers Preconditions *Proc. of SAS'07*. Springer, 2007.
9. D. Distefano and Josh Berdine and Byron Cook and P.W. O'Hearn Automatic Termination Proofs for Programs with Shape-shifting Heaps In *Proc. of CAV'06*. Springer, 2006.

A Proofs

Observations: There are several facts to be noted before we go over to the proof, all of them can be easily seen from the definitions and from the algorithm. Let c be a configuration set and $0 \leq i \leq n_m, i \leq j \leq n_m$.

1. Let $line_m(i) = line_m(j)$. Then $conf_m(i) \subseteq conf_m(j)$.
2. $Incl_m(i, c) \subseteq Incl_m(j, c)$
3. $Starts_m(i, c) \subseteq \{s \in Incl_m(i, c) \mid line_s(0) = line_m(i)\}$
4. $s \in Starts_m(i, c) \implies s_1 \in \{s' \in Incl_m(i + 1, c) \mid line_{s'}(0) = line_m(i + 1)\}$

A.1 Proof of Lemma 1

Proof. We will prove the lemma in the following two steps:

$\supseteq$-part:

$$\forall 0 < i < n_m, 0 \leq l \leq n_P.\ BadOnLines_{i-1}[l] \supseteq \bigcup_{s \in Incl_m(i,Bad) \wedge line_s(0)=l} conf_s(0)$$

$\subseteq$-part:

$$\forall 0 < i < n_m, 0 \leq l \leq n_P.\ BadOnLines_{i-1}[l] \subseteq \bigcup_{s \in Incl_m(i,Bad) \wedge line_s(0)=l} conf_s(0)$$

$\supseteq$-part: By induction on i.

For $i = n_m$, the $\supseteq$-part apparently holds. Suppose that the $\supseteq$-part holds for $i+1$ and let s be any sequence from $Incl_m(i, Bad)$. Our goal is to show that $\supseteq$-part holds also for i, for which $conf_s(0) \in BadOnLines_{i-1}[line_s(0)]$ is sufficient to show.

If s is included in m_{i+1}, then the lemma still holds for s from the induction hypothesis.

If s is not included in m_{i+1}, then it is in $Starts_m(i, Bad)$, thus $line_m(i) = line_s(0) = l_0$, Let k be the least such index that $s_1 \in Starts_m(k, Bad)$. Then $s \in Starts_m(k-1, Bad)$ (obviously $k-1 \geq i$). When processing $\texttt{GoBackward}_{k-2}$, we did $BadOnLines_{k-2}[l_0] = BadOnLines_{k-1}[l_0] \cup \mathsf{badFromTop}_{k-2}$. The value of $\mathsf{badFromTop}_{k-2}$ returned by $\texttt{GoBackward}_{k-1}$ equals $\tau_{k-1}^{-1}(BadOnLines_{k-1}[l_1]) \cap conf_m(k)$ $(\tau_{k-1} = \tau_i)$. If $BadOnLines_{k-1}[l_1] \supseteq conf_s(1)$, then $\mathsf{badFromTop}_{k-2}$ contains $conf_s(0)$, which we want to show. As $k \geq i + 1$, we have it from the induction hypothesis.

$\subseteq$-part: By induction on i:

For $i = n_m$, it is obvious. Suppose that the $\supseteq$-part holds for $i + 1$ and let also $inc = BadOnLines_{i-1}[line_m(i)] \setminus BadOnLines_i[line_m(i)]$. Our goal is to show that $\subseteq$-part holds also for i, for which it is sufficient to show that $inc \subseteq \bigcup_{s \in Starts_m(i,Bad)} conf_s(0)$:

Obviously $inc \subseteq \mathsf{badFromTop}_{i-1}$, which equals $\tau_i^{-1}(BadOnLines_i[line_m(i+1)]) \cap conf_m(i)$. Now, because $BadOnLines_i[line_m(i + 1)]$ fulfills the $\subseteq$-part (from the

induction hypothesis), we get that $\mathsf{badFromTop}_{i-1}$ has to be a subset of the set $\tau_i^{-1}\big(\bigcup_{s \in Incl_m(i+1,Bad) \wedge line_s(0)=line_m(i+1) \wedge conf_s(0) \subseteq conf_m(i)} conf_s(0)\big)$ which equals $\tau_i^{-1}\big(\bigcup_{s \in Starts_m(i+1,Bad)} conf_s(0)\big)$. Therefore $\mathsf{badFromTop}_{i-1}$ is also subset of the set $\bigcup_{s \in Starts_m(i,Bad)} conf_s(0)$ and the $\subseteq$-part holds for i.

□

A.2 Proof of Lemma 2

Proof. During the backward run we go downward through the stack until we reach its bottom. From the previous lemma it follows (and we can use argumentation similar to the one from the $\subseteq$-part of proof of Lemma 1) that $\mathsf{badHere}_i$ always equals $\bigcup_{s \in Starts_m(i,Bad)} conf_s(0)$. If $\mathsf{badHere}_i$ is not empty, then its value is attached to $oldestBad_i$. Therefore at the end of backward run, there is the value of the last nonempty $\mathsf{badHere}$ which equals to the last nonempty set $\bigcup_{s \in Starts_m(j,Bad)} conf_s(0)$, which is exactly $\bigcup_{s \in Starts_m(deep_m,Bad)} conf_s(0)$.