A Theory for Control-Flow Graph Exploration

Stephan Arlt[1], Philipp Rümmer[2], and Martin Schäf[1]

[1] United Nations University, IIST, Macau S.A.R., China.
[2] Uppsala University, Sweden.

Abstract. Detection of infeasible code has recently been identified as a scalable and automated technique to locate likely defects in software programs. Given the (acyclic) control-flow graph of a procedure, infeasible code detection depends on an exhaustive search for feasible paths through the graph. A number of encodings of control-flow graphs into logic (understood by theorem provers) have been proposed in the past for this application. In this paper, we compare the performance of these different encodings in terms of runtime and the number of queries processed by the prover. We present a theory of acyclic control-flow as an alternative method of handling control-flow graphs. Such a theory can be built into theorem provers by means of theory plug-ins. Our experiments show that such native handling of control-flow can lead to significant performance gains, compared to previous encodings.

1 Introduction

Recently, attempts are being made to use static verification to prove the presence of *infeasible code* [7, 3, 14]. Infeasible code refers to statements that cannot occur on any feasible (and complete) control-flow path in a program. Infeasible code detection can be used to detect common coding mistakes like unreachable code, insufficient error handling, or redundant checks whether pointers are well-defined (see [3] for further examples). The benefit of using static verification to prove the presence of bugs instead of their absence is that it can be implemented in a modular and scalable way with a very low rate of false warnings. If a proof exists that a certain statement cannot be executed, most likely this indicates a coding mistake (not necessarily a bug), whereas, if the proof fails, infeasible code detection simply remains silent. That is, while infeasible code detection can miss occurrences of infeasible code (i.e., false negatives), it hardly causes false alarms. Another benefit is that infeasible code detection can be implemented in a modular fashion: if a statement does not have a feasible execution in its containing procedure (regardless of the calling context), then it will not have an execution in its containing program as a whole. Thus, infeasible code detection can be implemented in a modular way on isolated code snippets without risking false alarms (but possibly false negatives).

In order to show that a code fragment (basic block) within a given program is feasible, an execution trace has to be found that contains the fragment. For each code fragment, a formula is constructed whose satisfiability implies the existence of an execution for the considered code fragment. This formula is sent to a theorem prover, which either proves that the fragment in fact has no execution (the formula is unsatisfiable), or computes a model that witnesses the existence of an execution. The construction of execution traces is the main bottleneck of algorithms for checking feasibility.

We present a tighter integration of feasibility checking with the search procedure executed by theorem provers, by defining a *theory of control flow* that is natively implemented and integrated into the prover in the form of a *theory plug-in*. With the help of the theory, we implement a query-optimal algorithm for feasibility checking, similar in spirit to the procedure presented in [3]. The use of a theory plug-in eliminates the need for helper variables to implement the query-optimal algorithm, and generally enables more efficient control-flow exploration; in experiments on real-world Java applications, our new algorithm is more than one order of magnitude faster than the one in [3], and significantly faster than other (encoding-based) algorithms for infeasible code detection that we compared to. We believe that our results make a convincing case for native implementation of decision procedures reasoning about program structure, with implications also for other forms of static analysis, including verification of safety properties and white-box methods to generate test cases.

The *contributions* of this paper are: (i) the definition of a theory of acyclic control flow, and an efficient implementation with the help of a theory plug-in; (ii) a query-optimal algorithm for feasibility checking; and (iii) an experimental evaluation on a set of large Java applications. Our implementation and benchmarks are publicly available.[1]

Related Work. Different approaches have been presented to identify code that does not occur on feasible executions within a given program, such as [6, 8, 14, 3]. In this paper we focus on static verification based approaches to detect infeasible code and on their strategies to explore all paths in a program. In [14] so called *wedges* are identified as a suitable subset of statements that need to be check. In [7] Boolean helper variables are used to render all executions that do not pass a location infeasible. Both approaches are worst-case optimal but neither proposes a strategy to explore the program efficiently.

In [3], integer-typed helper variables are used to enable queries that check for the existence of a feasible path which covers at least n previously uncovered statements. With that, a query-optimal algorithm is possible. We have reimplemented this approach for our experiments.

In [4] a different encoding of the weakest liberal precondition is proposed that also encodes the backward reachability of statements. With this encoding, counterexamples from the theorem prover can be used to identify feasible control-flow paths in a program. Based on this, they present a covering algorithm that uses enabling clauses. They argue that allowing to prover to find a covering strategy is more efficient than forcing it towards a particular strategy. We will also compare with this algorithm in our experiments.

[1] `http://www.joogie.org`

2 A Theory of Acyclic Control-Flow

2.1 Programs and Acyclic Control-Flow Graphs

Throughout this paper, we consider programs written in a simple unstructured language. The language can be seen as a simplified version of Boogie [10]:

$$\begin{aligned} \mathit{Program} &::= \mathit{Block}^* \\ \mathit{Block} &::= \mathit{label}: \ \mathit{Stmt}^* \ \textbf{goto}\, \mathit{label}^* \textbf{;} \\ \mathit{Stmt} &::= \mathit{VarId} := \mathit{Expr}\textbf{;} \ \mid \ \textbf{assume}\, \mathit{Expr}\textbf{;} \end{aligned}$$

The semantics of programs is as usual. We focus on the case of *loop-free* (or *acyclic*) programs, and refer to related work for sound approaches to compute, for an arbitrary program P, a loop-free program $P^{\#}$ that over-approximates the feasible executions of P (e.g., [7, 5]). We also assume that programs are upfront transformed to *passive form* [2], which means that assignments are replaced with fresh program variables and **assume** statements. Without loss of generality, it can further be assumed that every block in a program only consists of a single **assume** statement.

Definition 1 (Acyclic Control-Flow Graph). *An* acyclic control-flow graph *(ACFG) is defined by a tuple (B, E, b_e, b_x), where B is a set of propositional variables representing basic blocks, $E \subseteq B^2$ is an acyclic edge relation, and $\{b_e, b_x\} \subseteq B$ are two nodes such that every node in B is reachable from b_e, and b_x is reachable from every node in B.*

Every passive loop-free program can be represented by an ACFG (B, E, b_e, b_x), together with a function $S : B \to \mathit{For}$ that maps every block variable $b \in B$ (representing a block $l_b :$ **assume** ϕ_b**; goto** $\ldots$) to the formula $S(b) = \phi_b$.

Definition 2 (Feasible block). *Suppose (B, E, b_e, b_x) is an ACFG, and S a labelling of the blocks as above. A block represented by block variable $b \in B$ is called* feasible *iff there is a set $P = \{b_1, b_2, \ldots, b_n\} \subseteq B$ of nodes such that $b \in P$, $b_1 = b_e$, $b_n = b_x$, for all $i \in \{1, \ldots, n-1\}$ it is the case that $(b_i, b_{i+1}) \in E$, and $\bigwedge_{i=1}^{n} S(b_i)$ is satisfiable.*

In order to systematically discover feasible blocks, we consider the models of the formula $\mathit{WLP} \wedge b_e$, where:

$$\mathit{WLP} = \bigwedge_{b \in B} \Big(b \implies S(b) \wedge \mathit{SuccConj}(b) \Big) \qquad (1)$$

$$\mathit{SuccConj}(b) = \begin{cases} \bigvee_{(b,b') \in E} b' & \text{if } b \neq b_x \\ \mathit{true} & \text{otherwise} \end{cases}$$

It is easy to see that a block $b \in B$ is feasible iff $\mathit{WLP} \wedge b_e$ has a model that maps b to *true*, and in which the subset $B' \subseteq B$ of block variables that is mapped to *true* is minimal [1]. Hence, a theorem prover can be used to enumerate feasible blocks by repeatedly computing models of the conjunction $WLP \wedge b_e$. While this encoding is correct and practical, for the application of infeasible code detection it can be observed that it tends to provide insufficient guidance to a solver. In particular, when forcing the prover to follow a particular cover strategy (like in [3]), we claim that it is beneficial to assist the prover by providing domain-specific knowledge about the structure of the control-flow graph to be explored; otherwise, the implemented cover strategy can cause a slowdown rather than a speedup, as the prover has to do significantly more internal backtracking [4] in order to accommodate the cover strategy. We present a way of integrating CFG exploration more deeply into the search algorithm used by solvers.

2.2 Acyclic Control-Flow Graphs as a Theory

We want to force a prover to discover paths with many previously uncovered nodes first, and define a particular notion of control-flow path for this purpose:

Definition 3 (k-C-Path). *Suppose (B, E, b_e, b_x) is an ACFG, $C \subseteq B$ is a set of nodes, and $k \in \mathbb{N}$ is an integer. A* k-C-path *is a set $P = \{b_1, b_2, \ldots, b_n\} \subseteq B$ of nodes such that $b_1 = b_e$, $b_n = b_x$, for all $i \in \{1, \ldots, n-1\}$ it is the case that $(b_i, b_{i+1}) \in E$, and $|P \cap C| \geq k$.*

Since the nodes B of an ACFG are defined to be propositional variables, we can regard an ACFG (together with a set C and an integer k) as a logical theory that restricts the interpretation of B to k-C-paths:

Definition 4 (ACFG theory). *The* ACFG theory *over (B, E, b_e, b_x), a set $C \subseteq B$, and $k \in \mathbb{N}$ is defined by the following axiom:*

$$\bigvee_{\substack{P \subseteq B \\ \text{a } k\text{-}C\text{-path}}} \Big(\bigwedge P \wedge \neg \bigvee (B \setminus P) \Big) \tag{2}$$

The ACFG theory can be used to discover feasible blocks in a program, by choosing $C \subseteq B$ as the set of blocks in a program that still have to be covered, and k as some constant determining how many blocks are supposed to be covered simultaneously. Every model of the formula $WLP \wedge b_e$ that also satisfies (2) (i.e., every *model modulo the ACFG theory, with parameters k and C*) represents a feasible k-C-path through the given program.

Clearly, axiom (2) will in general be of exponential size (in the size of the underlying ACFG), and is therefore not a practical way to implement the theory. The next section discusses how an efficient implementation, in the context of a DPLL-based solver, can be achieved by combining a set of smaller logical axioms with a tailor-made constraint propagator.

2.3 Native Implementation of ACFG Theories

We implement a decision procedure for an ACFG theory in two parts:

1. a set of *axioms* that ensure that at least one path from b_e to b_x is selected in every accepted model;
2. a *propagator* (or *theory solver*) that ensures that at most one path is selected in a model, and that this path is a k-C-path.

In the rest of the section, we assume that an ACFG (B, E, b_e, b_x), a set $C \subseteq B$, and a threshold $k \in \mathbb{N}$ have been fixed.

Axioms. The required axioms are simple implications in forward direction, starting at the initial node of the ACFG:

$$\Big\{b_e\Big\} \cup \Big\{\Big(b \implies \bigvee \mathit{Succ}(b)\Big) \mid b \in B \setminus \{b_x\}\Big\} \tag{3}$$

where $\mathit{Succ}(b) = \{b' \mid (b, b') \in E\}$ is the set of direct successors of $b \in B$.

In order to satisfy (3), a prover has to assign *true* to variable b_e, and whenever some block variable b is selected (assigned *true*), also one of the successors of b needs to be selected. Consequently, the axioms (3) are satisfied by interpretations of the variables B in which at least one path from b_e to b_x is selected; it is left to the heuristics of the prover which path to pick. However, satisfying interpretations might select multiple paths simultaneously, and they might also contain paths that do not start in the initial node b_e (but end in b_x). Selected paths might moreover not be k-C-paths.

In our context, it can be observed that the axioms (3) are implied by the formula $\mathit{WLP} \wedge b_e$ constructed in Section 2.1. This means that a search for models of $\mathit{WLP} \wedge b_e$ (as done in Section 2.4 below) will automatically satisfy also (3), and it is not necessary to explicitly assert (3) as well.

Propagator. DPLL-style solvers [11] construct models of a given formula (usually modulo a set of background theories) by step-wise extension of partial interpretations, with backtracking being carried out whenever conflicts occur (dead ends in the search space are reached). In the context of an ACFG (B, E, b_e, b_x), this means that at any point during DPLL search there is a subset $B^+ \subseteq B$ of variables that have been assumed to be *true*, and a subset $B^- \subseteq B \setminus B^+$ of variables that have been assumed to be *false*. Other variables $B \setminus (B^+ \cup B^-)$ are unassigned. This means that the search has narrowed down the set of considered k-C-paths to those paths $P \subseteq B$ with $B^+ \subseteq P$ and $B^- \cap P = \emptyset$.

Given such a partial interpretation (B^+, B^-), a tailor-made propagator can infer further information, and thus decide the value of further variables in the remaining set $B \setminus (B^+ \cup B^-)$:

1. most importantly, the propagator can check whether there is at all a k-C-path $P \subseteq B$ with $B^+ \subseteq P$ and $B^- \cap P = \emptyset$. If this is not the case, the assignment (B^+, B^-) is inconsistent, and search has to backtrack.
2. it can be checked whether there are *inevitable nodes*

$$I = \bigcap \{P \subseteq B \mid P \text{ a } k\text{-}C\text{-path with } B^+ \subseteq P,\ B^- \cap P = \emptyset\}$$

that have to visited by every k-C-path that is consistent with the chosen assignment (B^+, B^-). Variables in I can immediately be set to *true*.

3. it can be checked whether there are *unreachable nodes*

$$U = \quad B \setminus \bigcup \{P \subseteq B \mid P \text{ a } k\text{-}C\text{-path with } B^+ \subseteq P,\ B^- \cap P = \emptyset\}$$

   that are not visited by any k-C-path consistent with the assignment (B^+, B^-). Variables in U can immediately be set to *false*.

Note that the first kind of inference ensures that only satisfying assignments with at most one path are accepted, and only in case the path is a k-C-path. In combination with the axioms (3), this implies that only models representing single k-C-paths are produced. The second and third kind of propagation provide input for further *Boolean constraint propagation,* and ensure that a theorem prover does not spend time exploring parts of the ACFG in which no k-C-paths can exist; in particular, a prover can immediately ignore any implication $b \implies S(b) \wedge \mathit{SuccConj}(b)$ (from (1)) with $b \in U$, and can immediately process the succedent $S(b) \wedge \mathit{SuccConj}(b)$ in case $b \in I$. In comparison with a direct encoding into logic (as in [3]), this provides a degree of look-ahead that can significantly speed up search.

All three types of inference can be performed in linear time in the size of the ACFG (B, E, b_e, b_x) by means of simple dynamic programming.

For our experiments, we implemented an ACFG constraint propagator in form of a *theory plug-in* that is loaded into the Princess theorem prover [13] and initialised with the ACFG (B, E, b_e, b_x), the set $C \subseteq B$, and the threshold $k \in \mathbb{N}$. The theory plug-in monitors the variable assignments made during search, and if possible adds inferred information (about inconsistency of the assignment (B^+, B^-), or the value of the variables in I and U) to the state of the search.

2.4 Infeasibility Checking with ACFG Theories

With the propagator from above, we can now implement a greedy path-cover algorithm for an ACFG on top of an incremental prover, following the idea of [3]: our algorithm *InfCode*, as shown in Algorithm 1, takes a loop-free program P and the associated ACFG (B, E, b_e, b_x), and then repeatedly computes models of the formula $\mathit{WLP} \wedge b_e$ representing terminating executions of P. Such models are constructed modulo the ACFG theory for (B, E, b_e, b_x), with the set C initially set to the set of all nodes B, and k to a sufficiently large number (e.g., the length of a path from b_e to b_x); only models are accepted that represent k-C-paths.

We repeatedly check for the existence of ACFG-models of $\mathit{WLP} \wedge b_e$ using the helper function `checkSat` (line 7). If a model exists, i.e., `checkSat` returns SAT, we remove all variables b_i from C that were assigned *true* (line 9); such variables represent blocks on the found k-C-path. We then re-initialize the theory plug-in with the new reduced set C (line 10), and search for further models.

If no k-C-path exists, i.e., `checkSat` returns $UNSAT$, we restart the search for models with $k \leftarrow \lceil k/2 \rceil$ (line 15 and 16). The algorithm terminates if our set C becomes empty and thus all nodes have been covered (line 4), or we do

Algorithm 1: *InfCode*: an algorithm to detect infeasible code.

```
Input: Passive loop-free program $P$ with ACFG $(B, E, entry, exit)$
Output: $C$: The set of block variables that do not have feasible executions.
begin
    $k \leftarrow$ average path length ;
    $C \leftarrow B$ ;
    assert($WLP \wedge b_e$);
    restartModelSearch($k, C$);
    while $C \neq \{\}$ do
        $R \leftarrow$ checkSat ;
        if $R = SAT$ then
            $C \leftarrow \{b_i \in C \mid b_i$ is assigned false in model$\}$ ;
            reinitPlugin($C$);
        else
            if $k = 1$ then
                return $C$
            endif
            $k \leftarrow \lceil k/2 \rceil$ ;
            restartModelSearch($k, C$);
        endif
    endw
end
```

not find any k-C-path for $k = 1$ (line 10). In that case, all remaining nodes in C cannot occur on a feasible path. A proof is given in [3].

Now let C be a so-called *effectual* subset of B [3]. That is, $C \subseteq B$ is called effectual if it is a minimal set of block variables such that a set of feasible paths that covers all elements in C also covers all elements in B.

Theorem 1. *Given an ACFG (B, E, b_e, b_x) with an unknown set of feasible paths. Let $C \subseteq B$ be an effectual subset of B, and $N(C)$ be the maximum number of elements in C that can occur together on one control-flow path. If K is the size of the smallest set of feasible paths that covers all coverable elements in C, then Algorithm 1 performs at most $O(K \cdot \log(N(C)))$ queries.*

A proof is given in [3]. As shown in [9] and [12] this is a query-optimal solution for the case that the set of feasible paths is unknown. However, the algorithm queries a theorem prover to check for a feasible path with certain properties. This is the most expensive part of the whole algorithm; in previous experiments, which implemented the algorithm with the help of a purely logical encoding and auxiliary variables, this led to the observation that the query-optimal algorithm is in reality slower than theoretically sub-optimal solutions [4]. We hypothesise that the implementation in form of a theory plug-in alleviates this bottleneck; to check if our intuition holds, we compare our algorithm with other approaches on several large-scale programs in the following section.

Program	*LOC*	*# methods*	*# inf code*
Open eCard	456,220	15,654	26
ArgoUML	156,294	9,981	28
FreeMind	53,737	5,613	10
Joogie	11,401	973	0
Rachota	11,037	1,279	1
TerpWord	6,842	360	3

Table 1. Name, size, and detected infeasible code for the six AUTs in our experiments.

3 Experiments

We have implemented our approach in Joogie, `http://www.joogie.org`. The Joogie tool takes a Java program as input and computes a loop-free abstraction of this program that can be translated into first-order logic (modulo the theory of arrays, and linear integer arithmetic). Joogie then generates feasibility checks, using four encoding schemes outlined in the next paragraph, and sends the resulting constraints to the theorem prover Princess [13]. For details on this translation and the inserted run-time assertions we refer to [3].

Experimental setup. For the loop-free program provided by Joogie, we compare four ways of detecting infeasible code. **(A)** the method presented in Section 2, implemented using a native theory plug-in, **(B)** an approach that uses enabling clauses to cover at least one new block in each iteration [4], **(C)** an approach that uses blocking clauses to never cover the same path twice [4], and **(D)** an approach that is similar to ours, but uses the solver as a black box and asserts linear inequalities to implement a query-optimal algorithm [3].

Note that all four approaches to detect infeasible code are complete for loop-free Boogie programs. Hence the detection rate is the same for all approaches (and only limited by the abstraction performed by Joogie), and we are only interested in the computation time of each approach.

We evaluate our approach on six open-source applications (AUTs): Open-eCard, a software to support the German eID, a CASE tool called ArgoUML, the mind-mapping tool FreeMind, the time-keeping software Rachota, the word processor TerpWord, and Joogie itself. Joogie applied each infeasible code detection algorithm to each procedure of an AUT individually. That is, Joogie does not perform inter-procedural analysis. Calls to procedures are replaced by non-deterministic assignments to all variables modified by the callee instead. For each procedure, we stop the time spent in the theorem prover process. If the theorem prover takes more than 30 seconds to analyze one procedure, we kill the process with a timeout and continue with the next procedure. All experiments are run on a workstation with 3 GHz CPU, 8 GB RAM, and 640 GB HDD.

Table 1 shows the details of the AUTs including the infeasible code that is found by Joogie. Even though we picked stable releases of each AUT, we could detect infeasible code in all of them besides Joogie. Most of the infeasible code

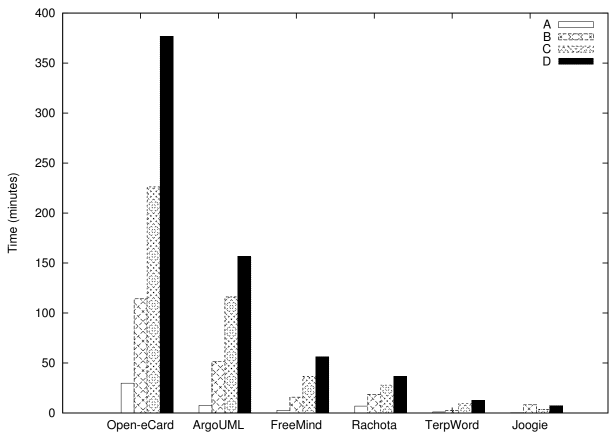

Fig. 1. Total time needed by the four considered algorithms on the six AUTs.

found is unreachable, some is caused by `null` checks of objects that have already been accessed (which is actually unreachable), and few cases are reported due to explicit contradictions (e.g., dereferencing a pointer known to be `null`).

Results. Figure 1 shows our experimental results. Our algorithm **A** computed a total 133,330 queries in 48.26 minutes. Algorithm **B** used 130,059 queries and 206.20 minutes, algorithm **C** 250,566 queries and 424.11 minutes, and algorithm **D** 132,976 queries and 646.21 minutes. The experiments show that our approach is significantly faster on all AUTs than existing approaches. The greedy algorithms **A** and **D** require a similar amount of queries. However, algorithm **D** is significantly slower. This is because **D** forces the prover to restrict it's search to a particular subset of paths by adding linear inequalities which can only be understood by the prover when using the corresponding theory. The algorithms **A**, **B**, **D** require a similar number of queries. Algorithm **C** used significantly more time than **A** and **B**. Apparently, the theorem prover tries to change as few values as possible in each iteration, thus blocking clauses might cause the prover to explore all possible control-flow paths. However, we can see that not restricting the solver results in very fast queries (the smallest time per query).

Threats to validity. We report the expected threats to validity: our AUTs do not represent a statistically significant sample. However, they are real programs, not tailored towards the experiments, and of reasonable size. Another threat is that theorem provers other than Princess might produce different results.

4 Conclusion

We have presented a new algorithm to detect infeasible code. In contrast to previous work, our algorithm deeply integrates with the used theorem prover. Not treating the theorem prover as a block box not only allows us to avoid additional instrumentation variables, it also enables the theorem prover to search for feasible control-flow paths significantly faster than in previous work. Since the time needed to process individual procedures is comparable to the time required for compilation (in particular type-checking), and since no false alarms are produced, we believe that infeasible code detection can be integrated in an IDE in a non-obtrusive way. With this integration, we will be able to detect infeasible code even before a program is executed. This is also the context in which we expect most occurrences of infeasible code, and a setting in which a large audience of users can be reached, opening a back door to provide programmers with a smooth learning curve towards the use of formal methods.

Acknowledgements. This work is supported by the grant COLAB of the Macao Science and Technology Development Fund, and by Vetenskapsrådet (VR).

References

1. Stephan Arlt, Zhiming Liu, and Martin Schäf. Reconstructing paths for reachable code. In *ICFEM*, 2013. To appear.
2. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT SEN*, 31, September 2005.
3. Cristiano Bertolini, Martin Schäf, and Pascal Schweitzer. Infeasible code detection. In *VSTTE*, 2012.
4. Jürgen Christ, Jochen Hoenicke, and Martin Schäf. Towards bounded infeasible code detection. *CoRR*, abs/1205.6527, 2012.
5. Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *SAS*, 2011.
6. Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, 2001.
7. Jochen Hoenicke, K. Rustan Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. Doomed program points. *FMSD*, 2010.
8. David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA*, 2004.
9. David S. Johnson. Approximation algorithms for combinatorial problems. volume 9, 1974.
10. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, pages 312–327, 2010.
11. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6), 2006.
12. Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *STOC*, 1997.
13. Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, 2008.
14. Aaron Tomb and Cormac Flanagan. Detecting inconsistencies via universal reachability analysis. In *ISSTA*.