A Devil's Advocate against Termination of Direct Recursion

Thom Frühwirth
University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

```
Rule Transformation Tool now available (use "Devil" options):
http://pmx.informatik.uni-ulm.de/chr/translator/index.php
```

Abstract

A devil's advocate is one who argues against a claim, not as a committed opponent but in order to determine the validity of the claim. We are interested in a devil's advocate that argues against termination of a program. He does so by producing a maleficent program that can cause the non-termination of the original program. By inspecting and running the malicious program, one may gain insight into the potential reasons for non-termination and produce counterexamples for termination.

We introduce our method in the concurrent programming language Constraint Handling Rules (CHR). Like in other declarative languages, non-termination occurs through unbounded recursion. Given a self-recursive rule, we automatically generate one or more devil's rules from it. The construction of the devil's rules is straightforward and involves no guessing. The devil's rules can be simple. For example, they are non-recursive for rules with single recursion.

We show that the devil's rules are maximally vicious in the following sense: For any program that contains the self-recursive rule and for any infinite computation through that rule in that program, there is a corresponding infinite computation with the recursive rule and the devil's rules alone. In that case, the malicious rules serve as a finite witness for non-termination. On the other hand, if the devil's rules do not exhibit an infinite computation, the recursive rule is unconditionally terminating. We also identify cases where the static analysis of the devil's rule decides termination or non-termination of the recursive rule.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program verification; I.2.2 [*Automatic Programming*]: Program transformation; D.2.4 [*Software/Program Verification*]: Formal methods; D.3.2 [*Language Classifications*]: Constraint and logic languages; D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages

General Terms Algorithms, Verification

Keywords Non-Termination, Termination, Program Transformation, Constraint Reasoning, Constraint Handling Rules

1. Introduction

It is well known that termination is undecidable for Turing-complete programming languages. Thus, there is a long tradition in research on analysis methods to tame the problem by semi-automatic or approximative approaches. Here we turn the problem around - we look at the dual problem of non-termination. We present a devil's advocate algorithm that argues against termination. It produces a maleficent program that can cause the non-termination of the original program. The devil's program may be simpler than the orginal one, for example, it can be non-recursive and thus terminating. The malicious program can form an alternative basis for dynamic and static termination analysis. When it is run, it can be quite useful in debugging, providing counterexamples for termination. We can also derive conditions for both termination and non-termination, as we will show in this paper.

We introduce our devil's advocate method in the programming language Constraint Handling Rules (CHR). CHR is a practical concurrent, declarative constraint-based language and versatile computational formalism at the same time. This allows us to operate on a high level of abstraction, namely first-order predicate logic. In this paper, we consider direct recursion. We think that this setting is best for introducing and demonstrating our novel approach to (non-)termination analysis. At the same time, we are confident that our devil's advocate technique carries over to other types of recursion and in general to traditional programming languages with while-loops as well.

The following program for determining if a number is even will serve as a running example in our paper. It also serves as a first overview of the characteristic features of the CHR language.

Example 1. In CHR, we use a first-order logic syntax, but variable names start with upper-case letters, while function and predicate symbols start with lower-case letters. Predicates will be called constraints. The built-in binary infix constraint symbol = stands for syntactical equality. For the sake of this example, numbers are expressed in successor notation. The user-defined unary constraint *even* can be implemented by two rules

$$even(X) \Leftrightarrow X{=}0 \mid true.$$
$$even(X) \Leftrightarrow X{=}s(Y) \mid Y{=}s(Z) \wedge even(Z).$$

The first rule says that X is even if it equals the number 0. The recursive rule says that X is even, if it is the successor of some number Y, and then the predecessor of this number Z is even. $X{=}s(Y)$ is a guard, a precondition for the applicability of the rule. It serves as a test, while $Y{=}s(Z)$ in the right-hand-side of the rule asserts an equality.

In logical languages like CHR, variables cannot be overwritten, but they can be without value (unbound). For example, if X is $s(s(A))$, then it will satisfy the guard, and Y will be $s(A)$. If X is unbound, then the guard does not hold. If the variable X later becomes (partially) bound in a syntactic equality, the computation of *even* may resume.

CHR is a committed-choice language, i.e. there is no backtracking in the rule applications. Computations in CHR are sequences of rule applications starting with a query and ending in an answer. To the query *even(0)* the first rule applies, the answer is *true*. The query

PPDP '15, July 14–16, 2015, Siena, Italy.

ACM 978-1-4503-3516-4/15/07…$15.00.
http://dx.doi.org/10.1145/2790449.2790518

even(N) delays, since no rule is applicable. The answer is the query itself. To the query *even(s(N))* the recursive rule is applicable once, the (conditional) answer is $N{=}s(N') \wedge even(N')$.

For the recursive rule of *even*, the devil's advocate just constructs the non-recursive propagation rule

$$even(X) \Rightarrow X{=}s(Y).$$

(In general it may not be that simple.) In a propagation rule, the left-hand-side constraint is not removed when the rule is applied. So this devil's rule observes occurrences of *even(X)* and maliciously adds $X{=}s(Y)$. This will trigger another application of the recursive rule (or lead to an inconsistency if X is 0). Thus there exist programs in which the recursive rule does not terminate.

Moreover, as we will show, any non-termination of *even* in any program is in essence characterized by the behavior of its devil's rule. Conversely, the devil's rule rather bluntly tells us that termination is ensured if eventually there is a recursive goal $even(X)$, where the variable X is different from $s(Y)$, including the case where X is unbound.

In this minimalistic example, the recursive rule alone suffices to produce non-termination. The query $even(N) \wedge even(s(N))$ will not terminate. Applying the recursive rule to $even(s(N))$ leads to $even(N) \wedge N{=}s(N') \wedge even(N')$. Since $N{=}s(N')$, the rule can now be applied to $even(N)$ and so on ad infinitum.

Related Work. While there is a vast literature on proving termination (one may start with [6] and with [17] for CHR), proving non-termination has only recently come to the attention of program analysis research.

Non-termination research can be found for term rewriting systems [10, 13], logic programming languages like Prolog [14, 20] and XSB [12], constraint logic programming languages (CLP) [15] and imperative languages like Java [4, 11, 16, 19].

In [20] non-termination of Prolog is proven by statically checking for loops in a finite abstract computation tree derived from moded queries. Similarily, [12] studies the problem of non-termination in tabled logic engines with subgoal abstraction, such as XSB. The algorithms proposed analyse forest logging traces and output sequences of tabled subgoal calls that are the likely causes of non-terminating cycles.

The papers [15, 16] give a criterion for detecting non-terminating atomic queries with respect to binary CLP rules. The approach is based on abstracting states by so-called filters and proving a recurrence. The recurrence criterion is similar to the one in [11].

For Java, the approach is often to translate into declarative languages and formalisms, e.g. into term rewriting systems [4], into logical formulae [19] and into CLP [16]. However, these translations are abstractions and in general this results in a loss of accuracy. The method in [11] for imperative languages such as Java is incomplete because the loop must be periodic. So it cannot deal with nested loops.

Overall, most research on non-termination can be seen as being based on the approach that is explained in [11]. It is a combination of dynamic and static analysis. One searches for counterexamples for termination. First, one dynamically enumerates a certain class of candidate execution paths (computations) until a state is re-entered. This has the drawback of combinatorial explosion in the number of paths. Then the candidate paths are checked if they contain a loop, i.e. a syntactic cycle.

The check amounts to proving the existence of a so-called recurrence set of states (transition invariants on states that are visited infinitely often). This problem is formulated as a constraint satisfaction problem and it is equivalent to the one for invariant generation.

As a reviewer of this paper has pointed out, the recent paper [5] also builds on recurrence sets, but avoids the need for periodicity in the non-terminating computation. This work is done in the context of a simple imperative while-loop language and the utilization of tools for proving safety properties. The approach performs an underapproximation search of the program to synthesize a reachable non-terminating loop, i.e. to produce an abstraction of the program with assumptions that lead to non-termination. The algorithms are quite different from our straightforward construction, but the underlying insights into the problem of non-termination seem similar. In particular, this approach, like ours, also produces a (kind of abstracted) program as a witness for non-termination. A further investigation of the relationship between the methods seems warranted.

Our Devil's Advocate Method. To the best of our knowledge our technique of constructing a malicious program by a devil's advocate algorithm is novel. The construction of the devil's rules is a straightforward program transformation. Our approach works well in a concurrent setting. Our proposed methodology is fully static, it avoids the exploration of all possible paths in a program's execution. Furthermore, there is no need for guessing transition invariants, because they are readily encoded in the malicious program. Our approach covers both periodic, aperiodic and nested loops (in the form of direct recursion), because there is no need to detect syntactic cycles.

The constructed devil's rules immediately give rise to a non-terminating computation, if there exists one at all. This maximally vicious computation is the abstraction of all non-terminating computations, but at the same time it is an executable program. This is made possible by the use of constraints as abstraction mechanism.

Our approach differs from most existing ones in that we explore non-terminating executions that are essential in that they are independent of the program context. The devil's rules exactly characterize what such a context in essence has to do in order to cause non-termination. Such a maximally vicious program is more general and more concise than characterizing the typically infinite set of queries that would lead to (non-)termination. The the execution of the devil's rules may even correspond to query of infinite size while having a compact finite representation.

The exemplary criterions for (non-)termination that we will derive for the devil's rules are similar to those that can be found in the literature. In principle, a transition invariant is defined that is a sufficient condition for the property at hand. However, in contrast to other research, we do not have to search for invariants or guess them by a heuristic or a process of abstraction steps. Our invariants can be readily derived from the devil's rules, that in turn are built from the culprit recursive rules.

Outline of the Paper. In the Preliminaries we introduce syntax and semantics of Constraint Handling Rules (CHR). In Section 3, we define the construction of malicious rules from directly recursive simplification rules. They give rise to maximally vicious computations, that never terminate successfully. We show that each non-terminating computation of the recursive rule contains a vicious computation. In Section 4, we look at static analysis of the devil's rules. We propose sufficient conditions for termination and non-termination of vicious computations. In Section 5, we address the simplification of devil's rules and give some more extended examples for our devil's advocate approach, before we conclude the paper. Readers who want a quick overview of the devil's advocate method can skip the proofs.

2. Preliminaries

In this section we give an overview of syntax and semantics for Constraint Handling Rules (CHR) [9]. We assume basic familiarity with first-order predicate logic and state transition systems.

Simplify

If $(r:\ H \Leftrightarrow C \mid B)$ is a disjoint variant of a rule in P
and $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then $(H' \wedge G) \mapsto_r (B \wedge G \wedge H{=}H' \wedge C)$

Propagate

If $(r:\ H \Rightarrow C \mid B)$ is a disjoint variant of a rule in P
and $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then $(H' \wedge G) \mapsto_r (H' \wedge B \wedge G \wedge H{=}H' \wedge C)$

Simpagate

If $(r: H_1 \setminus H_2 \Rightarrow C \mid B)$ is a disjoint variant of a rule in P
and $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H_1{\wedge}H_2){=}(H_1'{\wedge}H_2') \wedge C))$
then $(H_1' \wedge H_2' \wedge G) \mapsto_r$
$(H_1' \wedge B \wedge G \wedge (H_1{\wedge}H_2){=}(H_1'{\wedge}H_2') \wedge C)$

Figure 1. Transitions of Constraint Handling Rules

Abstract Syntax of CHR. Constraints are distinguished predicates of first-order predicate logic. We use two disjoint sets of predicate symbols (or: constraint names) for two different kinds of constraints: *built-in (or: pre-defined) constraints* which are handled by a given constraint solver, and *user-defined (or: CHR) constraints* which are defined by the rules in a CHR program. A *CHR program* is a finite set of rules. There are three kinds of rules:

Simplification rule: $r:\ H \Leftrightarrow C \mid B,$
Propagation rule: $r:\ H \Rightarrow C \mid B,$
Simpagation rule: $r:\ H_1 \setminus H_2 \Leftrightarrow C \mid B,$

where *r:* is an optional, unique identifier of a rule, the *head* denoted by H, H_1 and H_2 is a non-empty conjunction of user-defined constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal (or: query)* is a conjunction of built-in and CHR constraints.

Conjuncts can be permuted since conjunction is associative and commutative. We will, however consider conjunction not to be idempotent, since we allow for duplicates, i.e. multiple occurrences of user-defined constraints. The empty conjunction is denoted by the built-in constraint $true$, which is the neutral element of the conjunction operator $\wedge$. A trivial guard expression "`true |`" can be omitted from a rule.

When it is convenient, we allow for *generalized simpagation rules*. In such rules, either H_1 or H_2 may be empty. If H_1 is empty, we may write the rule as a simplification rule. If H_2 is empty, we may write it as a propagation rule.

Abstract Operational Semantics of CHR. The operational semantics of CHR is given by the state transition system in Fig. 1. In the figure, all single upper-case letters except P are meta-variables that stand for goals. Let P be a CHR program. Let the variables in a disjoint variant of a rule be denoted by $\bar{x}$. Let CT be a complete and decidable constraint theory for the built-in constraints, including the trivial $true$ and $false$ as well as syntactical equivalence $=$. For a goal G, the notation G_{bi} denotes the built-in constraints of G and G_{ud} denotes the user-defined constraints of G.

A *disjoint (or: fresh) variant* of an expression is obtained by uniformly replacing its variables by different, new (fresh) variables. A *variable renaming* is a bijective function over variables.

Starting with a given initial state (or: query), CHR rules are applied exhaustively, until a fixed-point is reached. A rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is logically implied by the built-in constraints in the goal. An expression of the form $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$ is called *applicability condition*. Any one of the applicable rules can be applied in a transition, and the application cannot be undone, it is committed-choice.

A simplification rule $H \Leftrightarrow C \mid B$ that is applied *removes* the user-defined constraints matching H and replaces them by B provided the guard C holds. A propagation rule $H \Rightarrow C \mid B$ instead *keeps* H and *adds* B. A simpagation rule $H_1 \setminus H_2 \Leftrightarrow C \mid B$ keeps H_1, removes H_2 and adds B. If new constraints arrive, rule applications are restarted.

States are goals. In a transition (or: computation step) $S \mapsto_r T$, S is called *source state* and T is called *target state*. A *computation* of a goal G in a program P is a connected sequence $S_i \mapsto S_{i+1}$ beginning with the initial state $S_0 = G$ and ending in a final state (or: answer) or the sequence is non-terminating (or: diverging). The length of a computation is the number of its computation steps. The notation $\mapsto_P{}^n$ denotes a finite computation of length n where rules from P have been applied. Given a computation starting with S_0 in which a state S_i with $(0 \leq i)$ occurs, then the computation up to S_i is a *prefix* of the computation.

Note that built-in constraints in a computation are accumulated, i.e. added but never removed, while user-defined constraints can be added as well as removed.

In the transitions of the abstract semantics as given, there are two sources of *trivial non-termination*. For simplicity, we have not made their avoidance explicit in the transitions of the abstract semantics. (Concurrency is also not made explicit in the semantics given.)

First, if the built-in constraints G_{bi} in a state are inconsistent (or: unsatisfiable), any rule could be applied to it, since the applicability condition trivially holds since the premise of the logical implication is false. We call such a state *failed*. Non-termination due to failed states is avoided by requiring G_{bi} to be consistent when a rule is applied. In other words, any state with inconsistent built-in constraints is a (failed) final state.

Second, a propagation rule could be applied again and again, since it does not remove any constraints and thus its applicability condition always continues to hold after the rule has been applied (due to CHR's monotonicity). This non-termination is avoided by applying a propagation rule at most once to the same user-defined constraints. Note that syntactically identical user-defined constraints are not necessarily the same, since we allow for duplicates. In implementations, each user-defined constraint has a unique identifier, and only constraints with the same identifier are considered to be the same for this purpose.

3. Devil's Advocate against Termination of Direct Recursion

Our devil's advocate algorithm constructs one or more malicious rules from a given recursive rule. The idea behind these devil's rules can be explained as follows: A devil's rule observes the computation. When it sees constraints that could come from the body of the recursive rule, it suspects the recursive rule has just been applied. It then maliciously adds the constraints necessary to trigger another recursive step by making the recursive rule applicable.

We therefore first prove that this interplay between the recursive rule and the devil's rule can only lead to non-termination or a failed state. We call such a computation maximally vicious. This is because the devil's rules capture the essence of any non-termination of the recursive rule, no matter in which program. Even if the devil's rule is not present, every infinite computation through the recursive rule will remove and add some constraints in exactly the same way as the devil's rule would do.

We therefore prove a second claim, namely that any non-terminating computation without the devil's rule contains the max-

imally vicious computation with the devil's rule. Therefore, if the devil's rules do not exhibit an infinite computation, the recursive rule is unconditionally terminating.

3.1 Direct Recursion, Devil's Advocate and Devil's Rules

In this paper, we consider direct recursion expressed by simplification rules. From them, our devil's advocate algorithm will construct devil's rules.

Definition 1. A CHR rule is *direct recursive (or: self-recursive)* if the head and the body of the rule have common predicate symbols. A constraint is direct recursive if its predicate symbol occurs in the head and in the body of a rule.

An overlap is a conjunction built from two goals, where one or more constraints from different goals are equated pairwise.

Definition 2. Given two conjunctions of constraints A of the form $A_1 \wedge A_2$ and B of the form $B_1 \wedge B_2$, where A_2 and B_2 are non-empty conjunctions. An *overlap $A \diamond B$ at the common constraints A_2 and B_2* is a conjunction of the form $A_1 \wedge A_2 \wedge B_1 \wedge A_2{=}B_2$. The goal $A_2 \wedge A_2{=}B_2$ is called the *common part* of the overlap.

Note an overlap is only possible if the two goals have common predicate symbols. If there are more than two such constraints, there are several overlaps.

Now the rules constructed by the devil's advocate come into play.

Definition 3. Given a self-recursive simplification rule r of the form

$$r : H \Leftrightarrow C \mid B_{bi} \wedge B_{ud},$$

where B_{bi} denotes the built-in constraints and B_{ud} denotes the user-defined constraints comprising the rule body B. Let r' be a disjoint variant of the rule r of the form

$$r' : H' \Leftrightarrow C' \mid B'_{bi} \wedge B'_{ud}.$$

For each overlap $(B_{ud} \diamond H')$ at the common constraints $O_{B_{ud}}$ and $O_{H'}$, we generate a devil's rule.

A *devil's rule d for the rule r* is a generalised simpagation rule of the form

$$d : O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \mid C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}$$

where $B_{ud}{=}(O_{B_{ud}} \wedge R_{B_{ud}})$ and $H'{=}(O_{H'} \wedge R_{H'})$.

Note that the effect of the devil's rule is to manipulate exactly those constraints that occur in its recursive rule. In particular, it in effect replaces all user-defined body constraints of the recursion by the head constraints that are needed for the next recursive step.

Example 2. We continue with Example 1 and its rule

$$even(X) \Leftrightarrow X{=}s(Y) \mid Y{=}s(Z) \wedge even(Z).$$

The rule for *even* is direct single recursive, so there is only one overlap and the resulting single devil's rule is not recursive. Also, the goal $R_{B_{ud}}$ is empty, thus we can write the generated devil's rule as a propagation rule

$$even(Z) \Rightarrow X{=}s(Y) \wedge Y{=}s(Z) \mid X'{=}s(Y') \wedge even(Z){=}even(X').$$

In Section 5 we will simplify this rule into $even(X) \Rightarrow X{=}s(Y)$.

Example 3. Consider a rule scheme for tree traversal of the form

$$traverse(node(L,V,R)) \Leftrightarrow C \mid B \wedge traverse(L) \wedge traverse(R).$$

It yields two devil's rules that are variants of each other

$$\begin{aligned} &traverse(L) \setminus traverse(R) \Leftrightarrow C \wedge B \mid \\ &\quad C' \wedge (traverse(L){=}traverse(node(L',V',R'))), \\ &traverse(R) \setminus traverse(L) \Leftrightarrow C \wedge B \mid \\ &\quad C' \wedge (traverse(R){=}traverse(node(L',V',R'))). \end{aligned}$$

3.2 Maximally Vicious Computations

We now prove that the devil's rules will cause infinite computations or failed states when these devil's rules and their recursive rule are applied alternatingly. For the proof we need the following lemmata.

Lemma 1. Given goal C consisting of built-in constraints only and a goal H consisting of user-defined constraints only. Let the pairs (H, C) with variables x and (H', C') with variables y be disjoint variants. Then the applicability condition

$$CT \models \forall \bar{x}(C \rightarrow \exists \bar{y}(H'{=}H \wedge C'))$$

trivially holds.

Proof. Since H and H' are disjoint variants, the syntactic equality $H'{=}H$ is satisfiable. It implies a variable renaming between the variables in x and y that occur in H and H', respectively. We apply this variable renaming, replacing variables in y in the applicability condition by the corresponding variables in x. This can only affect the consequent $(H'{=}H \wedge C')$ of the condition, where the constraints with the variables y occur.

In particular, the variable renaming will turn the equality $H'{=}H$ into $H{=}H$. Since this trivially holds, we can remove the equality. Moreover, the variable renaming will replace variables in C' by their corresponding variables in C. Since C and C' are disjoint variants, the variables from H will occur in the same positions in both expressions. Thus the two expressions will still be variants after the variable replacement.

This means that the premise and conclusion of the resulting implication can be written as $\forall \bar{w}, \bar{x}'(C[\bar{x}, \bar{x}'] \rightarrow \exists \bar{y}' C'[\bar{x}, \bar{y}'])$, where $\bar{x} = \bar{w}$, $\bar{x}'$ and $\bar{x}'$ are the variables from C and $\bar{y}'$ are the variables from C' (and thus y) that have not been replaced. This implication is logically equivalent to $\forall \bar{x}(\exists \bar{x}' C[\bar{x}, \bar{x}'] \rightarrow \exists \bar{y}' C'[\bar{x}, \bar{y}'])$, which is a tautology in first order predicate logic. □

Lemma 2. Given a self-recursive rule r of the form

$$H_* \Leftrightarrow C_* \mid B_{*bi} \wedge B_{*ud},$$

and its devil's rule d of the form

$$O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \mid C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}.$$

Note that $B_{ud}{=}(O_{B_{ud}} \wedge R_{B_{ud}})$ and $H'{=}(O_{B_{ud}} \wedge R_{H'})$ since $O_{B_{ud}}{=}O_{H'}$.

Then according to the abstract semantics of CHR, any transition with r and then d has the form

$$\begin{aligned} &(H \wedge G) \mapsto_r \\ &(B_{*bi} \wedge B_{*ud} \wedge G \wedge H_*{=}H \wedge C_*) \mapsto_d \\ &((C' \wedge O_{B_{ud}}{=}O_{H'} \wedge H') \wedge \\ &\quad B_{*bi} \wedge G \wedge H_*{=}H \wedge C_* \wedge (B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})) \end{aligned}$$

with $CT \models \forall(G_{bi} \rightarrow \exists \bar{x}(H_*{=}H \wedge C_*))$, where $\bar{x}$ are the variables of a disjoint variant of the rule r, and with $CT \models \forall((B_{*bi} \wedge G_{bi} \wedge H_*{=}H \wedge C_*) \rightarrow \exists \bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}))$, where $\bar{y}$ are the variables of a disjoint variant of the devil's rule d.

We will refer to the above transitions as *transition scheme*.

Definition 4. A *maximally vicious computation* of the self-recursive simplification rule r and its devil's rules D is of the form

$$(S'_0{=}(H_r \wedge C_r))\ S'_0 \mapsto_r T'_0 \ldots\ S'_i \mapsto_r T'_i \mapsto_D S'_{i+1} \ldots (i \geq 0),$$

where the pair (H_r, C_r) is a disjoint variant of the head and guard of the rule r.

In a maximally vicious computation, the only user-defined constraints contained in the states come from the recursive rule.

Lemma 3. Given a maximally vicious computation of a recursive rule r with head H and body $B_{bi} \wedge B_{ud}$ and one of its devil's rules d.

Then the states S_i contain as the only user-defined constraints a disjoint variant of the head H, and the states T_i contain as the only user-defined constraints a disjoint variant of the body B_{ud}.

Proof. By Definition 4, the first state S_0 contains the user-defined constraints H_r, which are a disjoint variant of the head of the rule r. Consider the transition scheme of Lemma 2. For our inductive proof assume that H is a disjoint variant of the head of the rule r and that G only contains built-in constraints, i.e. $G = G_{bi}$. An application of rule r replaces user-defined constraints H by B_{*ud}. An application of a devil's rule d of r replaces user-defined constraints B_{*ud} by H', which is a disjoint variant of the head of rule r by Definition 3. □

Theorem 1. *All maximally vicious computations of a self-recursive simplification rule r and its devil's rules D are either non-terminating or end in a failed state.*

Proof. We prove the claim by induction over the computation steps. The base case consists of showing that for any devil's rule d in D there exists a computation

$$(H_r \wedge C_r) = S'_0 \mapsto_r T'_0 \mapsto_d S'_1,$$

or a prefix of this computation ending in a failed state. The induction step consists of two cases that together prove the claim:

1. If the self-recursive rule r has been applied in a computation and the resulting state is not failed, a devil's rule d from D associated with r is applicable in the next computation step, i.e.
$$(i \geq 0) \wedge (S'_i \mapsto_r T'_i) \wedge \exists(T'_{ibi}) \rightarrow (T'_i \mapsto_D S'_{i+1}).$$
2. If a devil's rule d from D has been applied in a computation and the resulting state is not failed, the self-recursive rule r associated with it is applicable in the next computation step, i.e.
$$(i \geq 0) \wedge (T'_i \mapsto_D S'_{i+1}) \wedge \exists(S'_{i+1bi}) \rightarrow (S'_{i+1} \mapsto_r T_{i+1}).$$

Base Case. According to the abstract semantics of CHR and Lemma 2, when we apply the direct recursive rule r of the form

$$r : H_* \Leftrightarrow C_* \mid B_{*bi} \wedge B_{*ud},$$

to the state $S'_0 = (H_r \wedge C_r)$, the transition is

$$(H_r \wedge C_r) \mapsto_r (B_{*bi} \wedge B_{*ud} \wedge C_r \wedge H_*{=}H_r \wedge C_*)$$

with $CT \models \forall(C_r \rightarrow \exists\bar{x}(H_*{=}H_r \wedge C_*))$, where $\bar{x}$ are the variables of the rule r. By construction according to Definition 3, the pairs (H_r, C_r) and (H_*, C_*) are disjoint variants. Thus we can apply Lemma 1 to show that this applicability condition trivially holds.

If the target state is failed, then we are done with the proof of this case. Otherwise we apply to B_{*ud} in the target state a devil's rule d of the rule r of the form

$$d : O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \mid C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}.$$

This yields the transition

$$(B_{*bi} \wedge B_{*ud} \wedge C_r \wedge H_*{=}H_r \wedge C_*) \mapsto_d$$
$$(O_{B_{ud}} \wedge (C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}) \wedge B_{*bi} \wedge C_r \wedge H_*{=}H_r \wedge C_* \wedge (B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}))$$

with $CT \models \forall(G'_{bi} \rightarrow \exists\bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}))$, where $\bar{y}$ are the variables of the devil's rule d. The built-in constraints G'_{bi} of the source state are $(B_{*bi} \wedge C_r \wedge H_*{=}H_r \wedge C_*)$, and thus the following applicability condition must hold

$$CT \models \forall((B_{*bi} \wedge C_r \wedge H_*{=}H_r \wedge C_*) \rightarrow \exists\bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})).$$

To show that this condition holds, it suffices to show that

$$CT \models \forall((C_* \wedge B_{*bi}) \rightarrow \exists\bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})).$$

By Definition 3, the tuples (B_{ud}, B_{bi}, C) and (B_{*ud}, B_{*bi}, C_*) are disjoint variants. Thus we can apply Lemma 1 to show that this applicability condition trivially holds.

Induction Step Case 1. According to the abstract semantics of CHR and Lemma 2, when we apply the direct recursive rule r of the form

$$r : H_* \Leftrightarrow C_* \mid B_{*bi} \wedge B_{*ud},$$

the transition is

$$(H \wedge G) \mapsto_r (B_{*bi} \wedge B_{*ud} \wedge G \wedge H_*{=}H \wedge C_*)$$

with $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H_*{=}H \wedge C_*))$, where $\bar{x}$ are the variables of the rule r and G_{bi} are the built-in constraints in G.

If the target state is failed, then we are done with the proof of this case. Otherwise we apply to B_{*ud} in the target state a devil's rule d of the rule r of the form

$$d : O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \mid C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}.$$

This yields the transition

$$(B_{*bi} \wedge B_{*ud} \wedge G \wedge H_*{=}H \wedge C_*) \mapsto_d$$
$$(O_{B_{ud}} \wedge (C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}) \wedge B_{*bi} \wedge G \wedge H_*{=}H \wedge C_* \wedge (B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}))$$

with $CT \models \forall(G'_{bi} \rightarrow \exists\bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}))$, where $\bar{y}$ are the variables of the devil's rule d. The built-in constraints G'_{bi} of the source state are $(B_{*bi} \wedge G_{bi} \wedge H_*{=}H \wedge C_*)$, and thus the following applicability condition must hold

$$CT \models \forall((B_{*bi} \wedge G_{bi} \wedge H_*{=}H \wedge C_*) \rightarrow \exists\bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})).$$

To show that this condition holds, it suffices to show that

$$CT \models \forall((C_* \wedge B_{*bi}) \rightarrow \exists\bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})).$$

By Definition 3, the tuples (B_{ud}, B_{bi}, C) and (B_{*ud}, B_{*bi}, C_*) are disjoint variants. Thus we can apply Lemma 1 to show that this applicability condition trivially holds.

Induction Step Case 2. In a similar way we now prove the second claim.

Any devil's rule d for the rule r is of the form

$$d : O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \mid C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}$$

It yields the transition

$$(H \wedge G) \mapsto_d$$
$$(O_{B_{ud}} \wedge (C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}) \wedge G \wedge (B_{ud}{=}H \wedge C \wedge B_{bi}))$$

with $CT \models \forall(G_{bi} \rightarrow \exists\bar{y}(B_{ud}{=}H \wedge C \wedge B_{bi}))$, where $\bar{y}$ are the variables of the devil's rule d, and G_{bi} are the built-in constraints in G.

If the target state is failed, then we are done with the proof of this case. Otherwise we apply to the target state the direct recursive rule r of the form

$$r : H_* \Leftrightarrow C_* \mid B_{*bi} \wedge B_{*ud}.$$

Since $O_{B_{ud}}{=}O_{H'}$, we can replace $O_{B_{ud}}$ by $O_{H'}$ in the target state. But then we can replace $O_{H'} \wedge R_{H'}$ by H' and apply rule r to it. The resulting transition is:

$$((C' \wedge O_{B_{ud}}{=}O_{H'} \wedge H') \wedge G \wedge (B_{ud}{=}H \wedge C \wedge B_{bi})) \mapsto_r$$

$$((B_{*bi} \wedge B_{*ud}) \wedge (C' \wedge O_{B_{ud}}{=}O_{H'}) \wedge \\ G \wedge (B_{ud}{=}H \wedge C \wedge B_{bi}) \wedge (H_*{=}H' \wedge C_*))$$

provided the following applicability condition holds

$$CT \models \forall(((C' \wedge O_{B_{ud}}{=}O_{H'}) \wedge \\ G_{bi} \wedge (B_{ud}{=}H \wedge C \wedge B_{bi})) \rightarrow \exists\bar{x}(H_*{=}H' \wedge C_*)),$$

where $\bar{x}$ are the variables of the rule r.

It suffices show that $\forall(C' \rightarrow \exists\bar{x}(H_*{=}H' \wedge C_*))$. The tuples (H_*, C_*) and (H', C') are disjoint variants. Thus by Lemma 1 it is a tautology. □

3.3 Characterizing Non-Terminating Computations

The next theorem shows that any non-terminating computation through a recursive rule in any program contains a maximally vicious computation of that recursive rule and its devil's rules. So if there is no non-terminating maximally vicious computation, then the recursive rule must be always terminating, no matter in which program it occurs.

We need the following two lemmata from [2].

Lemma 4. A computation can be repeated in a state where implied (or: redundant) built-in constraints have been removed. Let $CT \models \forall\ (D \rightarrow C)$.

$$\text{If } (H \wedge C \wedge D \wedge G) \mapsto^* S \text{ then } (H \wedge D \wedge G) \mapsto^* S.$$

The next lemma states an important monotonicity property of CHR.

Lemma 5. *(CHR monotonicity)* A computation can be repeated in any larger context, i.e. with states in which built-in and user-defined constraints have been added.

$$\text{If } G \mapsto^* G' \text{ then } (G \wedge H) \mapsto^* (G' \wedge H).$$

The following definition gives a necessary, sufficient, and decidable criterion for equivalence of states [18].

Definition 5. Given two states $S_1 = (S_{1bi} \wedge S_{1ud})$ and $S_2 = (S_{2bi} \wedge S_{2ud})$. Then the two states are *equivalent*, written $S_1 \equiv S_2$, if and only if

$$CT \models \forall(S_{1bi} \rightarrow \exists\bar{y}((S_{1ud} = S_{2ud}) \wedge S_{2bi})) \wedge \\ \forall(S_{2bi} \rightarrow \exists\bar{x}((S_{1ud} = S_{2ud}) \wedge S_{1bi}))$$

with $\bar{x}$ those variables that only occur in S_1 and $\bar{y}$ those variables that only occur in S_2.

Note that this notion (operational) equivalence is stricter than logical equivalence since it rules out idempotence of conjunction, i.e. it considers multiple occurrences of user-defined constraints to be different.

The overlap makes sure that the recursive rule is applied in a directly recursive way, common constraints of the overlap are denoted by O_i.

Definition 6. A *non-terminating computation through a direct recursive rule* r

$$H_* \Leftrightarrow C_* \mid B_{*bi} \wedge B_{*ud},$$

in a program P is of the form

$$S \mapsto_P^n S_0 \ldots\ S_i \mapsto_r T_i \mapsto_P^{n_i} S_{i+1} \ldots\ \ (i \geq 0,\ n, n_i \geq 0),$$

where there is an overlap at common recursive constraints O_i of r with $T_i = (T_{Ri} \wedge O_i)$ and $S_{i+1} = (S_{Ri+1} \wedge O_i)$ where O_i had been added by r in the transition $S_i \mapsto_r T_i$ and O_i is to be removed by r in the transition $S_{i+1} \mapsto_r T_{i+1}$.

Theorem 2. *Given a self-recursive rule* r *of the form*

$$H_* \Leftrightarrow C_* \mid B_{*bi} \wedge B_{*ud},$$

and its devil's rules d *in* D *of the form*

$$O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \mid C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}.$$

Given a non-terminating computation through r *in* P,

$$S \mapsto_P^n S_0 \ldots\ S_i \mapsto_r T_i \mapsto_P^{n_i} S_{i+1} \ldots\ \ (i \geq 0,\ n, n_i \geq 0).$$

Then there is a corresponding maximally vicious computation with r *and its devil's rules* D

$$(S'_0 = (H_r{\wedge}C_r))\ S'_0 \mapsto_r T'_0 \ldots\ S'_i \mapsto_r T'_i \mapsto_D S'_{i+1} \ldots\ \ (i \geq 0),$$

where there exist constraints Gi *such that*

$$S'_i \wedge G'_i \equiv S_i \text{ and } T'_i \wedge G'_i \equiv T_i,$$

and there exist overlaps at the common constraints O_i *that occur in the states*

$$T'_i \equiv (T_{Ri} \wedge O_i) \text{ and } S'_{i+1} \equiv (S_{Ri+1} \wedge O_i).$$

Proof. We prove the claim by induction over the transitions in the computation. The base case is to prove that $S'_0 \wedge G'_0 \equiv S_0$ and $T'_0 \wedge G'_0 \equiv T_0$ and the induction step means to prove $S'_i \wedge G'_i \equiv S_i$ and $T'_i \wedge G'_i \equiv T_i$.

Base Case 1. There exists a goal G'_0 such that $(S'_0 \wedge G'_0) = S_0$.

Let $S_0 = (H \wedge G)$. According to the transition scheme in Lemma 2, we know that $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H_*{=}H \wedge C_*))$, where $\bar{x}$ are the variables of the rule r. As defined by the CHR semantics, $(H \wedge G)$ does not contain variables from $\bar{x}$. Let $S'_0 = (H'_* \wedge C'_*)$, such that $(H'_* \wedge C'_*)$ does not contain variables from $(H{\wedge}G)$ and $(H_*{\wedge}C_*)$. Let G'_0 be $(H'_*{=}H{\wedge}G)$, then $(S'_0{\wedge}G'_0) = ((H'_* \wedge C'_*) \wedge (H'_*{=}H \wedge G))$.

Since $H'_*{=}H$, we can replace the first conjunct H'_* by H. Since $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H_*{=}H \wedge C_*))$, and since $(H'_* \wedge C'_*)$ and $(H_* \wedge C_*)$ are disjoint variants, by Lemma 1 it also holds that $CT \models \forall(G_{bi} \rightarrow \exists\bar{z}(H'_*{=}H \wedge C'_*))$, where z are the variables of $(H'_* \wedge C'_*)$. Therefore, $(H'_*{=}H \wedge C'_*)$ is redundant and can be removed according to Lemma 4. Hence $(S'_0 \wedge G'_0) = (H \wedge G) = S_0$.

Base Case 2. Given $G'_0 = (H'_*{=}H \wedge G)$ from Base Case 1, we proceed to prove $(T'_0 \wedge G'_0) = T_0$.

W.l.o.g. we apply to S'_0 and S_0 the same disjoint variant of rule r, namely $H_* \Leftrightarrow C_* \mid B_{*bi} \wedge B_{*ud}$, to reach T'_0 and T_0, respectively.

Then we have that $T_0 = (B_{*bi} \wedge B_{*ud} \wedge G \wedge H_*{=}H \wedge C_*)$ and $(T'_0{\wedge}G'_0) = ((B_{*bi}{\wedge}B_{*ud}{\wedge}C'_*{\wedge}H_*{=}H'_*{\wedge}C_*){\wedge}(H'_*{=}H{\wedge}G))$. In the state $(T'_0 \wedge G'_0)$ we can replace $H_*{=}H'_*$ by $H_*{=}H$, since the state also contains $H'_*{=}H$. Now the states differ only in that $(H'_*{=}H{\wedge}C'_*)$ additionally occurs in $(T'_0{\wedge}G'_0)$. Analogously to the reasoning for the Base Case 1, we can show that this conjunction is redundant. So these states are indeed equivalent.

Induction Step Case 1. We prove that there exists a G'_i such that $S'_i \wedge G'_i \equiv S_i$.

According to the transition scheme in Lemma 2 we know that any transition with r and then some d in any computation has the form

$$(H \wedge G) \mapsto_r \\ (B_{*bi} \wedge B_{*ud} \wedge G \wedge H_*{=}H \wedge C_*) \mapsto_d \\ ((C' \wedge O_{B_{ud}}{=}O_{H'} \wedge H') \wedge \\ B_{*bi} \wedge G \wedge H_*{=}H \wedge C_* \wedge (B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}))$$

with $CT \models \forall(G_{bi} \rightarrow \exists\bar{x}(H_*{=}H \wedge C_*))$, where $\bar{x}$ are the variables of a disjoint variant of the rule r, and with $CT \models \forall((B_{*bi} \wedge G_{bi} \wedge H_*{=}H \wedge C_*) \rightarrow \exists\bar{y}(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}))$, where $\bar{y}$ are the variables of a disjoint variant of the devil's rule d.

In a maximally vicious computation, by Lemma 3 we know that G above does not contain user-defined constraints, i.e. $G = G_{bi}$.

W.l.o.g. let S'_i be the last state of the above transitions.

Furthermore, any transition with r and then P in any computation has the form

$$(H \wedge G'_{i-1}) \mapsto_r$$
$$(B_{*bi} \wedge B_{*ud} \wedge G'_{i-1} \wedge H_*{=}H \wedge C_*) \mapsto_P^{n_i}$$
$$(B_{*bi} \wedge G''_i \wedge H_*{=}H \wedge C_*).$$

W.l.o.g. let S_i be $(B_{*bi} \wedge G''_i \wedge H_*{=}H \wedge C_*)$. Since r is applicable to the target state S_i, G''_i must contain user-defined constraints H'_i such that

$$CT \models \forall((B_{*bi} \wedge G''_{ibi} \wedge H_*{=}H \wedge C_*) \rightarrow \exists\bar{x}(H_{i*}{=}H'_i \wedge C_{i*})),$$

where $\bar{x}_i$ are the variables of a disjoint variant of rule r with head H_{i*} and guard C_{i*}.

By Definition 3 and Definition 6, H'_i must overlap with B_{*ud}. This overlap at the common constraints $O_{H'_i}$ in H'_i of state S_i has its correspondence in the overlap at the common constraints $O_{B_{*ud}}$ in B_{*ud} of state T_i. This means it must hold that $CT \models \forall(G''_{ibi} \rightarrow \exists(O_{B_{*ud}}{=}O_{H'_i}))$.

The previous state T'_{i-1} contains G_{bi}. Since T'_{i-1} is contained in T_{i-1}, it must be (implied) there as well. According to the CHR semantics, built-in constraints are accumulated during a computation. Thus it must hold that $CT \models \forall(G''_{ibi} \rightarrow \exists G_{bi})$.

In the previous state T'_{i-1} it holds that

$$CT \models \forall((B_{*bi}{\wedge}G_{bi}{\wedge}H_*{=}H{\wedge}C_*) \rightarrow \exists\bar{z}(B_{ud}{=}B_{*ud}{\wedge}C{\wedge}B_{bi})),$$

where $\bar{z}$ are the variables of $(B_{ud} \wedge C \wedge B_{bi})$. Since T'_{i-1} is contained in T_{i-1}, the implication must also hold there. Since built-in constraints are accumulated, the implication must also hold in the next state S_i.

Putting these observation all together, we now know the state S_i is of the form

$$((H'_i \wedge (H_{i*}{=}H'_i \wedge C_{i*}) \wedge O_{B_{*ud}}{=}O_{H'_i}) \wedge G_{bi} \wedge B_{*bi} \wedge G'''_i \wedge H_*{=}H \wedge C_* \wedge (B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})),$$

where G'''_i are the remaining constraints from G''_i. Since $B_{ud}{=}B_{*ud}$, we can write $O_{B_{*ud}}{=}O_{H'_i}$ as $O_{B_{ud}}{=}O_{H'_i}$. Since (H_{i*}, C_{i*}) and (H', C') are disjoint variants, we can apply Lemma 1 to show that $CT \models \forall(C_{i*} \rightarrow \exists(H'{=}H_{i*} \wedge C'))$. Finally, let $G'_i = G'''_i$, then $S'_i \wedge G'_i \equiv S_i$.

Induction Step Case 2. Finally, we prove that $(T'_i \wedge G'_i) = T_i$.

We know that $(S'_i \wedge G'_i) = S_i$. Since $S'_i \mapsto_r T'_i$, by CHR monotonicity (Lemma 5) we have that $S'_i \wedge G'_i \mapsto_r T'_i \wedge G'_i$. Thus $T_i = (T'_i \wedge G'_i)$. □

Example 4. We continue with Example 2, its recursive rule and its simplified devil's rule

$$r : even(X) \Leftrightarrow X{=}s(Y) \,|\, Y{=}s(Z) \wedge even(Z)$$
$$d : even(X) \Rightarrow X{=}s(Y).$$

For readability, we will simplify states w.r.t. equivalence $\equiv$ and underline the goals to which a rule is applied.

The maximally vicious computation is

$$\underline{even(U) \wedge U{=}s(V)} \mapsto_r$$
$$U{=}s(V) \wedge V{=}s(Z') \wedge \underline{even(Z')} \mapsto_d$$
$$U{=}s(V) \wedge V{=}s(Z') \wedge \underline{even(Z') \wedge Z'{=}s(Y'')} \mapsto_r \ldots$$

The non-terminating computation for the goal $even(N) \wedge even(M) \wedge M{=}s(N)$ is

$$\mathrm{even(N)} \wedge \underline{even(M) \wedge M{=}s(N)} \mapsto_r$$
$$\underline{\mathrm{even(N)} \wedge M{=}s(N) \wedge N{=}s(N')} \wedge even(N') \mapsto_r$$
$$N{=}s(N') \wedge \underline{N'{=}s(N'')} \wedge \mathrm{even(N'')} \wedge M{=}s(N) \wedge \underline{even(N')} \mapsto_r ..$$

This computation contains the maximally vicious computation when we rename the variables appropriately. Note that the second application of the rule r to the other, *even(N)* constraint is considered as the arbitrary transition sequence between recursive steps using rules from a given program P. The actual infinite computation differs from the maximally vicious computation only in an additional *even* constraint (that we have set in standard font style).

4. Static Termination and Non-Termination Analysis with Devil's Rules

We identify cases where the static analysis of the devil's rules decides termination or non-termination of the recursive rule in *any* program as exhibited by a maximally vicious computation. These conditions are just meant to be indicative of the potential of our devil's advocate approach, they are a starting point in the search for interesting conditions. The first lemma gives a necessary condition for non-termination. The negation of the condition thus gives a sufficient condition for termination. The second lemma gives a sufficient condition for non-termination. It implies the first condition, but it is not a sufficient and necessary condition.

4.1 A Termination Condition

Lemma 6. Given a recursive rule r and a devil's rule d for r of the form

$$d : O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \,|\, C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}$$

Then the computation

$$(H_r \wedge C_r) = S'_0 \mapsto_r T'_0 \mapsto_d S'_1$$

or any of its prefixes does not end in a failed state, if and only if

$$C \wedge B_{bi} \wedge C' \wedge O_{B_{ud}}{=}O_{H'}$$

is consistent.

Proof. From Lemma 2 and the proof of Theorem 1 we can see that the built-in constraints of the three states in the computation are

$$S'_{0bi} = (C_r)$$
$$T'_{0bi} = ((B_{*bi}) \wedge C_r \wedge (H_*{=}H_r \wedge C_*))$$
$$S'_{1bi} = ((C' \wedge O_{B_{ud}}{=}O_{H'}) \wedge B_{*bi} \wedge C_r \wedge H_*{=}H_r \wedge C_* \wedge (B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})),$$

where constraints in brackets are newly added in a state. We also know that the tuples (H_r, C_r) and (H_*, C_*) and (H', C') as well as (B_{ud}, B_{bi}, C) and (B_{*ud}, B_{*bi}, C_*) are disjoint variants and that the constraints $(H_*{=}H_r \wedge C_*)$ and $(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})$ are implied by the states S'_{0bi} and T'_{0bi}, respectively.

According to the semantics of CHR, built-in constraints in a computation are accumulated, i.e. added but never removed. To prove the claim, it therefore suffices to consider the built-in constraints of the last state S'_{1bi}. We have to show that

$$C \wedge B_{bi} \wedge C' \wedge O_{B_{ud}}{=}O_{H'} \leftrightarrow \exists\bar{x}(C' \wedge O_{B_{ud}}{=}O_{H'} \wedge B_{*bi} \wedge C_r \wedge H_*{=}H_r \wedge C_* \wedge B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi}),$$

where $\bar{x}$ consist of all variables that do not occur in the left-hand side of the logical equivalence. Since the left-hand side constraints occur in the right hand side, we just have to show that the left-hand side implies the right-hand side. As in the the proof of Theorem 1, we can apply Lemma 1 to show that remaining constraints on the right-hand side $B_{*bi}{\wedge}C_r{\wedge}H_*{=}H_r{\wedge}C_*{\wedge}B_{ud}{=}B_{*ud}$ are implied

by the left-hand side, since they are disjoint variants as given above. □

Note that the lemma does not say that any computation with the recursive rule alone will fail. But it will fail with a transition of the devil's rule.

Example 5. We continue with Example 4 and its correct devil's rule

$$even(Z) \Rightarrow X{=}s(Y) \wedge Y{=}s(Z) \mid X'{=}s(Y') \wedge even(Z){=}even(X').$$

The conjunction to check can be simplified into

$$X{=}s(Y) \wedge Y{=}s(Z) \wedge Z{=}s(Y') \wedge X'{=}Z$$

and is clearly satisfiable. Thus non-termination is not ruled out.

Example 6. Consider the direct recursive rule and its devil's rule

$$c(0) \Leftrightarrow c(s(X)).$$

$$c(s(X)) \Rightarrow c(s(X)){=}c(0).$$

Since $s(X){=}0$ is unsatisfiable, the recursive rule must always terminate. Actually, the recursive goal $c(s(X))$ will delay.

4.2 A Non-Termination Condition

Lemma 7. Given a recursive rule r and a devil's rule d for r of the form

$$d : O_{B_{ud}} \setminus R_{B_{ud}} \Leftrightarrow C \wedge B_{bi} \mid C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}$$

and let (H, C, B_{bi}) and (H', C', B'_{bi}) be disjoint variants derived from the recursive rule r. Then the maximally vicious computation

$$(S'_0 = (H_r \wedge C_r))\ S'_0 \mapsto_r T'_0 \ldots\ S'_i \mapsto_r T'_i \mapsto_d S'_{i+1} \ldots\ (i \geq 0)$$

is non-terminating, if

$$(NT)\ \exists(C \wedge B_{bi}) \wedge \forall((C \wedge B_{bi}) \rightarrow \exists(C' \wedge O_{B_{ud}}{=}O_{H'} \wedge B'_{bi})).$$

Proof. We prove by induction over the transitions analgous to Theorem 1. We show that if a state in the computation is not failed and condition *NT* holds, then the next state is not failed as well. In the induction step we distinguish between applications of rule r and rule d.

Base Case. For the base case, we consider the prefix of the computation

$$(H_r \wedge C_r) = S'_0 \mapsto_r T'_0 \mapsto_d S'_1.$$

Condition *NT* implies by the laws of first-order predicate logic

$$\exists(C \wedge B_{bi} \wedge C' \wedge O_{B_{ud}}{=}O_{H'}).$$

From Lemma 6 we know that this conjunction implies that the prefix of the computation has no failed states.

Induction Step Case 1. The transition for rule r yields the condition

$$(i \geq 0) \wedge NT \wedge \exists(S'_i) \wedge (S'_i \mapsto_r T'_i) \rightarrow \exists(T'_{ibi}).$$

According to Lemma 2 and Theorem 1, this transition is of the form

$$((C' \wedge O_{B_{ud}}{=}O_{H'} \wedge H') \wedge G \wedge (B_{ud}{=}H \wedge C \wedge B_{bi})) \mapsto_r$$
$$((B_{*bi} \wedge B_{*ud}) \wedge (C' \wedge O_{B_{ud}}{=}O_{H'}) \wedge$$
$$G \wedge (B_{ud}{=}H \wedge C \wedge B_{bi}) \wedge (H_*{=}H' \wedge C_*)).$$

In the target state, the built-in constraints $(C' \wedge O_{B_{ud}}{=}O_{H'}) \wedge G_{bi} \wedge (B_{ud}{=}H \wedge C \wedge B_{bi})$ are satisfiable, because they already occured in the source state. The new constraints $(H_*{=}H' \wedge C_*)$ are satisfiable, because they are implied by (C') in the source state (due to the applicability condition that must hold for the transition). They have been added by the rule application together with B_{*bi}.

The constraints $(C \wedge B_{bi})$ of the source state must be satisfiable and by condition *NT* we have that

$$\forall(C \wedge B_{bi} \rightarrow \exists(C' \wedge O_{B_{ud}}{=}O_{H'} \wedge B'_{bi})).$$

The tuples (H', C', B'_{bi}) and (H_*, C_*, B_{*bi}) are disjoint variants. Therefore $(C_* \wedge B_{*bi})$ must be consistent as well, and thus the built-in constraints of the target state are all satisfiable.

Induction Step Case 2. The transition for a devil's rule d yields the condition

$$(i \geq 0) \wedge NT \wedge \exists(T'_i) \wedge (T'_i \mapsto_d S'_{i+1}) \rightarrow \exists(S'_{i+1bi}).$$

According to Lemma 2 and Theorem 1, this transition is of the form

$$(B_{*bi} \wedge B_{*ud} \wedge G \wedge H_*{=}H \wedge C_*) \mapsto_d$$
$$(O_{B_{ud}} \wedge (C' \wedge O_{B_{ud}}{=}O_{H'} \wedge R_{H'}) \wedge$$
$$B_{*bi} \wedge G \wedge H_*{=}H \wedge C_* \wedge (B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})).$$

In the target state, the constraints $(B_{*bi} \wedge B_{*ud} \wedge G_{bi} \wedge H_*{=}H \wedge C_*)$ are satisfiable, because they occur in the source state. The new constraints $(B_{ud}{=}B_{*ud} \wedge C \wedge B_{bi})$ are satisfiable, because they are implied by $(C_* \wedge B_{*bi})$ in the source state (due to the applicability condition that must hold for the transition). They have been added by the rule application together with $(C' \wedge O_{B_{ud}}{=}O_{H'})$. By condition *NT* we have that

$$\forall(C \wedge B_{bi} \rightarrow \exists(C' \wedge O_{B_{ud}}{=}O_{H'} \wedge B'_{bi})).$$

Therefore $(C' \wedge O_{B_{ud}}{=}O_{H'})$ must be consistent as well. Thus the built-in constraints of the target state are all satisfiable. □

If condition *NT* holds for a devil's rule, its maximally vicious computation is non-terminating. Thus there may be other non-terminating computations for the recursive rule. Conversely, if the maximally vicious computation terminates in a failed state, the condition cannot hold. On the other hand, if the condition does not hold, we cannot draw any conclusion about non-termination from it. We rather have to look at the maximally vicious computation for further insight about the termination behavior.

Example 7. Let *odd* and *prime* be built-in constraints. Consider the following recursive rule and its devil's rule

$$c(X) \Leftrightarrow odd(X) \mid c(s(s(X))),$$

$$c(s(s(X))) \Rightarrow odd(X) \mid (odd(X') \wedge c(s(s(X)))){=}c(X')).$$

Condition *NT* amounts to

$$\exists odd(X) \wedge \forall(odd(X) \rightarrow (odd(X') \wedge c(s(s(X)))){=}c(X'))).$$

Since the successor of the successor of an odd number is always odd, the condition holds. Actually, the recursive rule on its own is non-terminating for odd numbers.

Now consider a variation of the above rule

$$c(X) \Leftrightarrow prime(X) \mid c(s(s(X))).$$

Condition *NT* amounts to

$$\exists prime(X) \wedge \forall(prime(X) \rightarrow (prime(X') \wedge c(s(s(X)))){=}c(X'))).$$

Since the successor of the successor of a prime number may not be prime, the condition does not hold. Thus the status of non-termination is undecided. Actually, the recursive rule always terminates after at most two recursive steps: one of every three sequential odd numbers is a multiple of three, and hence not prime. Hence the maximally vicious computation always ends in a failed state.

5. Examples - Putting It All Together

Devil's Rule Simplification. In practice, we will simplify the built-in constraints in the devil's rules by replacing them with logically

equivalent ones. We do so taking into account that variables not occurring in the head of a rule are implicitly existentially quantified according to the CHR semantics. Moreover, if built-in constraints of the body are implied by the guard, we can remove them if other constraints in the body are not affected.

We can then distinguish two extreme cases:

1. If the built-in constraints in the devil's rule are inconsistent, we can replace the body of the rule by the built-in constraint $false$. We have a case for Lemma 6. So the recursive rule is unconditionally terminating if all its devil's rules simplify in this way.
2. If the simplification of a devil's rule yields a satisfiable guard and a body built-in constraint equivalent to $true$, then Lemma 7 may apply. Part of the condition NT of the lemma already holds in that case. It remains to check if the body built-in constraint of the recursive rule is implied in the context of the condition. This trivially holds if there are no such body constraints.

5.1 Even Numbers

We continue with Example 5. Recall the correct rule for *even* and its devil's rule

$$even(X) \Leftrightarrow X{=}s(Y) \mid Y{=}s(Z) \wedge even(Z).$$

$even(Z){\Rightarrow}X{=}s(Y){\wedge}Y{=}s(Z)|X'{=}s(Y'){\wedge}even(Z){=}even(X')$. The devil's rule can be simplified into $even(Z) \Rightarrow Z{=}s(Y')$.

Lemma 6 does not apply. Lemma 7 yields the condition *NT* (with the equality $even(Z){=}even(X')$ simplified away for convenience)

$$\exists(X{=}s(Y) \wedge Y{=}s(Z)) \wedge \forall((X{=}s(Y) \wedge Y{=}s(Z)) \rightarrow \exists(Z{=}s(Y') \wedge Y'{=}s(Z'))).$$

Since the predecessor of the predecessor of an even natural number does not exist for the number 0, it is not always even. Thus the condition does not hold. But it easy to see from the recursive rule and its devil's rule that the maximally vicious computation does not terminate. Actually, the recursive rule on its own may not terminate as we have shown in the introduction.

Termination. Next we consider two examples for the application of Lemma 6 for termination. Consider an erroneous version of the rule for *even* with a typo (highlighted by bold type font)

$$even(X) \Leftrightarrow X{=}s(Y) \mid Y{=}s(\mathbf{X}) \wedge even(Z).$$

This leads to a devil's rule with the unsatisfiable guard $X{=}s(Y) \wedge Y{=}s(X)$. After rule simplification we arrive at

$$even(Z) \Rightarrow false \mid false.$$

Since the body of the simplified devil's rule is $false$, the recursive rule will always terminate. It will actually lead to a failed state, when it is applicable, since its guard and body built-in constraints are in contradiction.

Now consider another typo and the resulting simplified devil's rule

$$even(X) \Leftrightarrow X{=}s(Y) \mid Y{=}s(\mathbf{z}) \wedge even(\mathbf{z}).$$
$$even(z) \Rightarrow false.$$

The recursive rule will thus always terminate, but not necessarily in a failed state. Once the recursive rule is applied, it cannot be applied again to the recursive goal $even(z)$, since the argument z does not satisfy the guard that demands a term of the form $s(Y)$.

Non-termination. Now we look at examples for application of Lemma 7 for non-termination. Consider the following rule with a typo and its devil's rule

$$even(X) \Leftrightarrow X{=}s(Y) \mid Y{=}s(Z) \wedge even(\mathbf{X}).$$

$even(X){\Rightarrow}X{=}s(Y){\wedge}Y{=}s(Z)|X'{=}s(Y'){\wedge}even(X){=}even(X')$. After simplification we arrive at

$$even(X) \Rightarrow X{=}s(Y) \wedge Y{=}s(Z) \mid true.$$

The implication in condition *NT* corresponds to

$$\forall((X{=}s(Y) \wedge Y{=}s(Z)) \rightarrow \exists(X{=}s(Y') \wedge Y'{=}s(Z'))).$$

Actually, the recursive rule on its own is already always non-terminating.

Now consider another erroneous rule and the resulting simplified devil's rule

$$even(X) \Leftrightarrow X{=}s(Y) \mid Y{=}s(\mathbf{z}) \wedge even(\mathbf{Y}).$$
$$even(Y) \Rightarrow Y{=}s(z) \mid true.$$

The implication in condition *NT* corresponds to

$$\forall((X{=}s(Y) \wedge Y{=}s(z)) \rightarrow \exists(Y{=}s(Y') \wedge Y'{=}s(z))).$$

The condition does not hold, since $Y{=}s(z)$ and $Y{=}s(Y') \wedge Y'{=}s(z)$ are in contradiction. So $true$ in the body of a simplified devil's rule does not necessarily mean non-termination, the maximally vicious computation may end in a failed state. Actually, here any computation where the recursive rule is applied will lead to a failed state.

5.2 Minimum

We compute the minimum of a multiset of numbers n_i, and $\leq$ a non-strict total order over an infinite domain of numbers, given as a computation of the query $min(n_1), min(n_2), ..., min(n_k)$ with the recursive rule

$$min(N) \wedge min(M) \Leftrightarrow N \leq M \mid min(N).$$

The rule takes two min candidates and removes the one with the larger value. It keeps going until only one, the smallest value, remains as single min constraint.

There are two overlaps at the recursive constraint *min* in the rule. The resulting devil's rules and then their simplified versions are

$$min(N) \Rightarrow N \leq M \mid N' \leq M' \wedge min(N){=}min(N') \wedge min(M')$$
$$min(N) \Rightarrow N \leq M \mid N' \leq M' \wedge min(N){=}min(M') \wedge min(N')$$

$$min(N) \Rightarrow N \leq M' \wedge min(M')$$
$$min(N) \Rightarrow N' \leq N \wedge min(N')$$

These are propagation rules that either add a smaller or larger min constraint. According to Lemma 7, the condition NT amounts to

$$\exists(N \leq M) \wedge \forall((N \leq M) \rightarrow \exists(N \leq M'))$$
$$\exists(N \leq M) \wedge \forall((N \leq M) \rightarrow \exists(N' \leq N))$$

Since both conditions hold, all maximally vicious computations are indeed non-terminating, no matter which of the two devil's rules are used. Actually, the recursive rule on its own does not terminate if we keep adding min constraints. Otherwise, it terminates, since every rule application removes one min constraint.

5.3 Exchange Sort

We can sort an array by keeping exchanging values at positions that are in the wrong order. Given an array as a conjunction of constraints representing array elements $a(Index, Value)$, i.e. $a(1, A_1) \wedge \ldots \wedge a(n, A_n)$, and a strict total order $<$ over the integers, the following recursive rule sorts in this way

$$a(I, V) \wedge a(J, W) \Leftrightarrow I{>}J \wedge V{<}W \mid a(I, W) \wedge a(J, V).$$

In a sorted array, it holds for each pair $a(I,V), a(J,W)$ where $I{>}J$ that $V \geq W$. The rule ensures that this indeed will hold for every such pair by exchanging the values if necessary.

There are two full overlaps, the resulting devil's rules and their simplified versions are

$$a(I,W) \wedge a(J,V) \Rightarrow I{>}J \wedge V{<}W \mid I'{>}J' \wedge V'{<}W' \wedge (a(I,W) \wedge a(J,V)){=}(a(I',V') \wedge a(J',W'))$$
$$a(I,W) \wedge a(J,V) \Rightarrow I{>}J \wedge V{<}W \mid I'{>}J' \wedge V'{<}W' \wedge (a(I,W) \wedge a(J,V)){=}(a(J',W') \wedge a(I',V'))$$

$$a(I,W) \wedge a(J,V) \Rightarrow I{>}J \wedge V{<}W \mid \mathit{false}$$
$$a(I,W) \wedge a(J,V) \Rightarrow I{>}J \wedge V{<}W \mid \mathit{false}$$

To the devil's rules of the full overlaps Lemma 6 applies. Indeed, the recursive rule terminates for any two array constraints. It cannot be applied a second time to the same pair of constraints.

There are four more partial overlaps between one array constraint from the head and one from the body of the recursive rule, yielding four devil's rules and their simplifications

$$a(I,W) \setminus a(J,V) \Leftrightarrow I{>}J \wedge V{<}W \mid I'{>}J' \wedge V'{<}W' \wedge a(I,W){=}a(I',V') \wedge a(J',W')$$
$$a(I,W) \setminus a(J,V) \Leftrightarrow I{>}J \wedge V{<}W \mid I'{>}J' \wedge V'{<}W' \wedge a(I,W){=}a(J',W') \wedge a(I',V')$$
$$a(J,V) \setminus a(I,W) \Leftrightarrow I{>}J \wedge V{<}W \mid I'{>}J' \wedge V'{<}W' \wedge a(J,V){=}a(I',V') \wedge a(J',W')$$
$$a(J,V) \setminus a(I,W) \Leftrightarrow I{>}J \wedge V{<}W \mid I'{>}J' \wedge V'{<}W' \wedge a(J,V){=}a(J',W') \wedge a(I',V')$$

$$a(I,W) \setminus a(J,V) \Leftrightarrow I{>}J \wedge V{<}W \mid I{>}J' \wedge W{<}W' \wedge a(J',W')$$
$$a(I,W) \setminus a(J,V) \Leftrightarrow I{>}J \wedge V{<}W \mid I'{>}I \wedge V'{<}W \wedge a(I',V')$$
$$a(J,V) \setminus a(I,W) \Leftrightarrow I{>}J \wedge V{<}W \mid J{>}J' \wedge V{<}W' \wedge a(J',W')$$
$$a(J,V) \setminus a(I,W) \Leftrightarrow I{>}J \wedge V{<}W \mid I'{>}J \wedge V'{<}V \wedge a(I',V')$$

The condition NT of Lemma 7 is satisfied for these four devil's rules, for example consider

$$\exists(I{>}J \wedge V{<}W) \wedge \forall((I{>}J \wedge V{<}W) \rightarrow \exists(I{>}J' \wedge W{<}W')).$$

To cause non-termination, the devil's rules replace an array constraint by another one. For every pair of array elements that has been ordered, this replacement results in an unordered pair. Therefore the array sort rule may not terminate if the array is updated during sorting.

We next consider an erroneous version of array sort, where the guard condition $I{>}J$ is missing, and its two full overlaps.

$$a(I,V) \wedge a(J,W) \Leftrightarrow V{<}W \mid a(I,W) \wedge a(J,V)$$

$$a(I,W) \wedge a(J,V) \Rightarrow V{<}W \mid \mathit{false}$$
$$a(I,W) \wedge a(J,V) \Rightarrow V{<}W \mid \mathit{true}$$

The second full overlap now produces a different simplified devil's rule. It has the body true. Moreover, the recursive rule does not have any built-in constraints in its body. Therefore Lemma 7 holds. Actually, the guard condition $V{<}W$ applies to any pair of array constraints with different values. So the rule will keep exchanging values in such two array constraints forever.

6. Conclusions

In this paper we have introduced a novel approach to non-termination analysis, exemplified for the programming language Constraint Handling Rules (CHR). It is based on the notion of a devil's advocate that produces a malicious program that causes non-termination of the given program, if it is possible at all. We have introduced the devil's advocate method using direct recursive simplifications rules of CHR. From them, the devil's advocate constructs so-called devil's rules in a simple manner. If a recursive rule and its devil's rules are applied alternatingly, the result is a maximally vicious computation. It is either infinite or ends in a failed state, as we have proven. The latter means that no infinite computation is possible with the recursive rule, independent of the program in which it appears. Otherwise, every non-terminating computation of the recursive rule in any program contains the maximally vicious computation, as we have proven, too.

The resulting devil's rules are often simple, e.g. non-recursive for single-recursion, and thus can be more easily inspected and analysed than their recursive counterparts. Also, the maximally vicious computation can be performed as an instructive help for the programmer during debugging, since it exhibits the essence of non-termination.

The devil's advocate approach can be characterized as follows: It is concerned with universal (non-)termination, while most other work deals with termination in the given context of a specific program. In the latter, it is important to find out which non-terminating computations are unreachable, and this is a necessary complication that comes at a considerable cost. It leads to combinatorial explosion and requires guessing of suitable abstractions. The search for feasible execution paths that is typical for most research on non-termination is a dynamic analysis technique, while the construction of the devil's rule is straightforward and a static analysis technique.

We have also introduced preliminary sufficient conditions for termination and non-termination that are directly derived from the devil's rules. At the moment, this compares favorable with other approaches, where the search for suitable invariants and recurrences involves indeterminism, heuristics and approximation techniques. It should be noted that the conditions are quite similar. Currently it is not clear if the simplicity is due to universal termination that we are interested in or if it will vanish once our conditions become more tight.

We think the main appeal of our approach lies in the particulary simple construction of the malicious program, providing a finite witness for (non-)termination. Indeed, as we have shown, the malicious program can form an alternative basis for dynamic and static termination analysis.

Last but not least, our approach works well in a concurrent distributed language setting. Universal termination is an important issue there, since a malicious program produced by a devil's advocate could be introduced into the distributed environment to cause harm. A concrete example would be denial-of-service attacks.

Future Work. We consider this paper as a starting point. Many directions for future work are possible. First of all, we clearly should extend the applicability of the devil's rule construction to recursive simpagation and propagation rules as well as to mutual recursion. Secondly, we would like to improve the results on static analysis with additional conditions for termination and non-termination. We suspect there is a close relationship with existing approaches concerning that aspects of our work. We also think that the presentation of the proofs could be made more accessible if an operational semantics more adequate for this kind of analysis can be found. Recent CHR semantics such as [3] could provide a starting point.

In the context of CHR, ranking functions have been used to prove termination and complexity bounds for bounded goals [7, 8]. Do devil's advocate rules respect such rankings? Can we derive boundedness conditions from them? A classic analysis result for CHR is a decidable, sufficient and necessary condition for confluence of terminating programs [1]. There are also conditions for confluence of non-terminating programs. Do devil's rules respect confluence? We also think that the restriction to confluent programs

may provide for additional conditions concerning static analysis of (non-)termination.

Our devil's advocate method should be applied to other programming languages and paradigms. Logic programming languages such as Prolog and concurrent constraint languages should be especially suitable, since they are predecessors of CHR. In Prolog there exist successful tools for termination analysis, which could provide an environment for fruitful comparisons with our approach.

For non-declarative languages we are confident that the advantages of our new technique carry over to this setting, where loops dominate over recursion as a language construct. A concrete starting point for this line of investigation might be to explore the relationship with the work [5], as mentioned in the introduction.

We can regard the devil's rules as a finite, concise and compact representation of queries of finite and infinite size. It still could be worthwhile to derive (some of) these queries as counter-examples from the malicious program. This could be useful for the user, but also foster comparison with other approaches in the field. Similarily, the given program context could be taken into account to see if infinite computations according to the devil's rule are possible at all.

In the end, our devil's advocate method, once fully understood and explored, might work best when combined with existing approaches, hopefully combining their advantages and leveling out their disadvantages.

Acknowledgments

We would like to thank the anonymous referees that provided us with detailed comments and suggestions for improvements of this paper.

References

[1] S. Abdennadher and T. Frühwirth. On completion of Constraint Handling Rules. In M. J. Maher and J.-F. Puget, editors, *CP '98*, volume 1520 of *LNCS*, pages 25–39. Springer, Oct. 1998. ISBN 3-540-65224-8.

[2] S. Abdennadher and T. Frühwirth. Operational equivalence of constraint handling rules. In *5th International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713. Springer, 1999.

[3] H. Betz, F. Raiser, and T. Frühwirth. A complete and terminating execution model for Constraint Handling Rules. volume 10(4–6) of *TPLP*, pages 597–610. Cambridge University Press, July 2010. .

[4] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and nullpointerexceptions for Java bytecode. In *Formal Verification of Object-Oriented Software*, pages 123–141. Springer, 2012.

[5] H.-Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. OHearn. Proving nontermination via safety. In *TACAS 2014 - Tools and Algorithms for the Construction and Analysis of Systems*, pages 156–171. Springer, 2014.

[6] B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[7] T. Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In A. D. Pierro and H. Wiklicky, editors, *QAPL '01: Proc. First Intl. Workshop on Quantitative Aspects of Programming Languages*, volume 59(3) of *ENTCS*. Elsevier, 2002.

[8] T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *KR '02: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning*, pages 547–557. Morgan Kaufmann, Apr. 2002.

[9] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009. ISBN 9780521877763. URL `http://www.constraint-handling-rules.org`.

[10] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. *Frontiers of Combining Systems*, pages 216–231, 2005.

[11] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. *ACM Sigplan Notices*, 43(1):147–158, 2008.

[12] S. Liang and M. Kifer. A practical analysis of non-termination in large logic programs. *Theory and Practice of Logic Programming*, 13(4-5): 705–719, 2013.

[13] É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403(2):307–327, 2008.

[14] E. Payet and F. Mesnard. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):256–289, 2006.

[15] É. Payet and F. Mesnard. A non-termination criterion for binary constraint logic programs. *Theory and Practice of Logic Programming*, 9 (02):145–164, 2009.

[16] É. Payet, F. Mesnard, and F. Spoto. Non-termination analysis of Java bytecode, CoRR abs/1401.5292, 2014.

[17] P. Pilozzi and D. De Schreye. Termination analysis of CHR revisited. In *Logic Programming*, pages 501–515. Springer, 2008.

[18] F. Raiser, H. Betz, and T. Frühwirth. Equivalence of CHR states revisited. In F. Raiser and J. Sneyers, editors, *6th International Workshop on Constraint Handling Rules (CHR)*, pages 34–48, 2009.

[19] H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In *Tests and Proofs*, pages 154–170. Springer, 2008.

[20] D. Voets and D. Schreye. A new approach to non-termination analysis of logic programs. In *Proceedings of the 25th International Conference on Logic Programming*, pages 220–234. Springer-Verlag, 2009.