Credit: CAV14

A new Decision Procedure Logic Solver on:

- Words of any length fed into <u>word equations</u>

  (example of word equation: integer expressions $\{e_i\}^n{}_{i=1}$, e_i is an equation for words

  vs $\{tr_i\}^n{}_{i=1}$)
- Limitations on words' length
- Regex of the word

  Decidability of Cav14 Decision Procedure Logic was open as of 2014.

  A problem is **Decidable** if we can always construct a corresponding **algorithm** that can answer the problem correctly.

1. Cav14 Decision Procedure Logic is sound for general logic
2. Cav14 Decision Procedure Logic is original in its implementation
3. Cav14 Decision Procedure Logic is tailored towards:
    - hefty word fragments
      * no assumption about the input string's length
    - restrictenning word equations
4. Cav14 Decision Procedure Logic has been integrated to a CEGAR based model checker (uses horn clauses) which is referred to in this summary as Cav14 Decision Procedure Logic Solver
5. Gave14 Decision Procedure Logic implemented through a CEGAR based model checker (ie CAV14 Decision Procedure Logic Solver) automatically establishes the correctness of several programs that are beyond the reach of existing methods.

1 Introduction

Satisfiability Modulo Theories (SMT) problem:

decision problem for ***logical formulas***
with respect to
combinations of ***background theories*** expressed in
classical first-order logic.

Model checking tools are limited by the data types that can be handled by the underlying SMT solver.

For the string data type,
a satisfying Decision Procedure Logic was missing
before CAV14 Decision Procedure Logic.

String data types are necessary in:

- conventional programming (examples C, Pascal, COBOL, FORTRAN)
- scripting languages

Conventional Programming VS ML Programming:

The approach of conventional programming is to feed the computer with a set of instructions for a defined set of scenarios. After that computer will utilize its computing capabilities to help human process the data faster and in an efficient fashion

Examples of where Decision Procedure Logic Solvers are needed for strings:

1- A program allowing users to choose a username and a password may require the password to be of a minimal length, to be different from the username, and to be free from invalid characters.

2- Reasoning about such constraints is crucial when verifying that database and web applications are free from SQL injections and other security vulnerabilities.

Existing Decision Procedure Logic Solvers for programs manipulating string variables and their length are:

- Either unsound
  Some solvers unsound since they assume max word length
- Or not expressive enough
  Some solvers not expressive enough as they do not handle
    - word equations
    - or length constraints
    - or membership predicates.

  Such solvers are mostly aimed at performing symbolic executions,
  i.e., establishing feasibility of paths in a program.
- Or lack the ability to provide counterexamples
  Some solvers might perform sound over-approximation without detecting counterexamples
  when software's verification fails at a certain point.

Cav14 Decision Procedure Logic Solver's decision procedure is tailored towards model checkers
It's used in prototype model checkers for automatic program correctness verification of several examples.

Cav14 Decision Procedure Logic establishes satisfiability of:

(i)

$(a \cdot u = v \cdot b)$ or $(a \cdot u \neq v \cdot b)$ where:

- $a, b \in$ Chars
- $u, v \in$ String Vars   arbitrary length

(ii)

length constraints such as $(|u| = |v| + 1)$,

where $|u|$ length of the word held in variable u

(iii)

Predicate u representing membership in a regular expression , e.g. $u \in c.(a+b)^*$

Cav14 Decision Procedure Logic's analysis is not trivial as it needs to capture subtle interactions between different types of predicates.

Example 1:
These formulae are unsatisfiable
(unsatisfiable: no possible assignment of words to u and v that makes the conjunctions evaluate to true)

$$\phi_1 = (a \cdot u = v \cdot b) \wedge (|u| = |v| + 1)$$

$$\phi_2 = (a \cdot u = v \cdot b) \wedge u \in c.(a+b)^*$$

To capture the unsatisfiability in Example 1, Cav14 Decision Procedure Logic's analysis needs to establish relations between predicates.

Example 2) in ϕ_1
(a · u = v · b) => |u| = |v|,
=> (|u| = |v| /\ |u| = |v| + 1) (truth value is false)

Cav14 Decision Procedure Logic Solver has been integrated to verify properties of implementations of common string manipulating functions such as Hamming and Levenshtein distances.

Predicates required for verification can be provided manually
For automation: a constraint-based interpolation procedure for regular word constraints.

Related Work;
Makanin's decision procedure for (a · u = or ≠ v · b) where u and v are variables of arbitrary lengths
The decidability problem is already open for $|u| = |v|$

Cav14 Decision Procedure Logic Solver's Advantages:

- Cav14 Decision Procedure Logic adds regular languages' predicates to word equations and length constraints.

  ($predicate\ u \in c.(a+b)^*$)
  means that decidability is still an open problem (why?)
  because unlimited instances of Char combination $c.(a+b)^*$ *available ??*

- In [10] decidability is determined, but where:

No string variables can appear in the right hand sides of the equality predicates
(this severely restricts the expressiveness of the logic).

- In [26], the authors augment the Z3 [7] SMT solver in order to handle word equations with length constraints. However, they do not support regular membership predicates.

- Can establish program correctness for programs with loops

2 A Simple Example

Program 1 **Fig 1**

```
//    Pre  =  (true)
String s = ' ';
//    P1  =  (s ∈ ε)
while(*)          // P2  loop invariant
{
      //    P2 = (s = u.v  ∧  u ∈ a*  ∧  v ∈ b*  ∧  |u| = |v|
      s = 'a' + s + 'b';
}
//    P3 = P2
assert(   !s.contains('ba')  &&   (s.length() % 2) == 0   );
//    Post = P3
```

Correctness requirements for programs like Program 1:

(i) handling strings of arbitrary lengths

(ii) the ability to express combinations of constraints,
like that the words denoted by the variables belong to regular expressions,
that their lengths obey arithmetic inequalities,
or that the words themselves are solutions to word equations

(iii) the ability for a decision procedure
to precisely capture the subtle interaction between
the different kinds of involved constraints

In Program 1:
- String s is initialized with the empty word
- While loop runs an arbitrary number of times
- At each iteration of the while loop,
the instruction s= 'a' + s + 'b'

appends letter ’a’ at the beginning of variable s
appends letter ’b’ at the end of variable s

- After the while loop, Program 1 asserts:
  String s does not have the word ’ba’ as a substring (!s.contains(’ba’))
  String s length is even [(s.length() % 2) == 0]

String s does not imply a maximal length

Any verification procedure that requires an a-priori fixed bound on the length of the string variables

- is necessarily unsound
- due to unsoundness it will fail to establish correctness.

Moreover, establishing correctness requires the ability to express and to reason about predicates such as those mentioned in the comments of the code in

Cav14 Decision Procedure Logic allows to precisely capture:

- the word equations,
- membership predicates
- and length constraints required for validating the assertion (never violated)

Fig 2 Verification Conditions $vc_1 \ldots vc_6$

$vc_1:\quad post\,(Pre,\ s\ =\ "")\ \Rightarrow\ P_1$

corresponding Hoare Triple: // `Pre = (true)` // P_1 `= (s` $\in$ `ε)`

{ true } s := " " { post ($true,\ s\ =\ ""$) } implies P_1 `= (s` $\in$ `ε)`

$vc_2:\quad P_1\ \Rightarrow\ P_2$ P_2 = { `s = u.v` $\wedge$ `u` $\in$ `a*` $\wedge$ `v` $\in$ `b*` $\wedge$ `|u| = |v|` }

P_1 = `(s` $\in$ `ε)`

$vc_3:\quad post\,(\ P_2\ ,\ \ s\ =\ "a"\ .\ s\ .\ "b"\ )\ \Rightarrow\ P_2$

{ `s = u.v` $\wedge$ `u` $\in$ `a*` $\wedge$ `v` $\in$ `b*` $\wedge$ `|u| = |v|` }

$s\ :=\ "a"\ .\ s\ .\ "b"$

{ post ...}

implies P_2

$vc_4:\quad P_2\ \Rightarrow\ P_3$ P_3 = P_2

$vc_5:\quad post\,(P_3,\ \ assume(\ s.contains("ba")\ \ \|\ \ (\ s.length()\ \%\ 2\ ==\ 0\ )\ )\ )\ \Rightarrow\ false$

{ `s = u.v` $\wedge$ `u` $\in$ `a*` $\wedge$ `v` $\in$ `b*` $\wedge$ `|u| = |v|` }

$assume(\ s.contains("ba")\ \ \|\ \ (\ s.length()\ \%\ 2\ ==\ 0\ )\ )$

{ post ... }

implies $false$

$$vc_6 : \quad post\,(P_3, \quad assume(\ \ !s.contains("ba") \quad \&\& \quad (\ s.length()\ \%\ 2\ == \ 0\ )\ )\ ) \quad \Rightarrow \quad Post$$

{ s = u.v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v| }

$assume(\ !\ s.contains("ba") \quad || \quad (\ s.length()\ \%\ 2\ == \ 0\ )\ )$

{ post ... }

implies $Post$

Example: the loop invariant P2 states that:

(i) variable s denotes a <u>finite</u> word Ws of arbitrary length,

(ii) Ws equals the concatenation of two words Wu and Wv (s = v.u)

(iii) Wu $\in$ a^* and Wu $\in$ b^*

(iv) | Wu | = | Wv |

vc_5 is valid if the following is unsatisfiable:

$$\neg\ vc_5$$

$$=$$

$$(\quad s\ =\ u.v \quad \wedge \quad u\ \in\ a* \quad \wedge \quad v\ \in\ b* \quad \wedge \quad |u|\ =\ |v| \quad)$$

$$\wedge$$

$$(\ s\ = s_1 . b\ .\ a\ .\ s_2 \quad \vee \quad \neg\ (\ |s|\ =\ 2n)\ )$$

To prove this, Cav14 Decision Procedure Logic Solver generates the two proof obligations

$\neg\ vc_{51}\ :\ (\ s\ =\ u.v \quad \wedge \quad u\ \in\ a* \quad \wedge \quad v\ \in\ b* \quad \wedge \quad |u|\ =\ |v| \quad \wedge \quad s\ = s_1 . b\ .\ a\ .\ s_2\ )$

$\neg\ vc_{52}\ :\ (\ s\ =\ u.v \quad \wedge \quad u\ \in\ a* \quad \wedge \quad v\ \in\ b* \quad \wedge \quad |u|\ =\ |v| \quad \wedge \quad \neg\ (\ |s|\ =\ 2n) \quad)$

$\neg\ vc_5\ =\ \neg\ vc_{51}\ \vee\ \neg\ vc_{52}$

To determine if $\neg\ vc_5$ is satisfiable

Cav14 Decision Procedure Logic Solver symbolically matches

all the possible ways in which $v\ .\ u$ would match $s\ = s_1 . b\ .\ a\ .\ s_2$

For instance, $u\ = s_1 . b \quad \wedge \quad v\ =\ a\ .\ s_2$ is one possible matching.

No guarantee that the sequence of generated matchings will terminate.

But the sequence terminates for a realistically applicable fragment of the logic.

The matching $u = s_1 . b \quad \Lambda \quad v = a . s_2$ is shown to be unsatisfiable

(because of the membership predicate $v \in b^*$)

Cav14 Decision Procedure Logic Solver automatically proves that ¬ v51 is not satisfiable
(through checking all possible matchings for $v . u$ and $s = s_1 . b . a . s_2$)
Proving $\neg \; vc_{51}$ is not satisfiable, Cav14 Decision Procedure Logic Solver moves on to determining whether $\neg \; vc_{52}$ can be shown to be satisfiable.

$\neg \; vc_{52}$ implies that $|u| = |v| \quad \Lambda \quad |u| + |v| = 2n$ must be satisfiable
Cav14 Decision Procedure Logic Solver proves it wrong using: linear arithmetic

Hence $\neg \, vc_5$ is unsatisfiable and vc_5 is valid.

Establishing validity for an arbitrary Verification Condition vc_i is about processing all the relations possibly existing in-between the initial set of predicates corresponding to the Verification Condition vc_i

For instance, proving unsatisfiability of verification condition vc_5
through construction of $\neg \; vc_{51}$ and $\neg \; vc_{52}$

$\neg \; vc_{51}$: $(\; s = u.v \quad \Lambda \quad u \in a* \quad \Lambda \quad v \in b* \quad \Lambda \quad |u| = |v| \quad \Lambda \quad s = s_1 . b . a . s_2 \;)$

$\neg \; vc_{52}$: $(\; s = u.v \quad \Lambda \quad u \in a* \quad \Lambda \quad v \in b* \quad \Lambda \quad |u| = |v| \quad \Lambda \quad \neg \; (|s| = 2n) \;)$

which relay on separating vulnerability points in the logic of vc_5 into the two seperate predicates $s = s_1 . b . a . s_2$ and $\neg \; (|s| = 2n)$

3 Defining the String Logic $E_{e,r,l}$

- We assume a finite alphabet Σ
- We assume Σ^* (the set of finite words over Σ)
- We have set $U = \{$ String s | s $\in \Sigma^* \}$
- We assume Z set of integer numbers

Let	$E_{e,r,l}$, Logic:
- Strings u, v $\in$ U	e: equations for words
- Integer k $\in Z$	r: regex
- Character c $\in \Sigma$	l: length inequalities
- Word $\boldsymbol{w} \in \Sigma^*$	

Syntax of $E_{e,r,l}$, Logic :

$$
\begin{array}{llll}
\phi & ::= & \phi \wedge \phi \mid \neg\phi \mid \varphi_e \mid \varphi_l \mid \varphi_r & \text{formulae} \\
\varphi_e & ::= & tr = tr \mid tr \neq tr & \text{(dis)equalities} \\
\varphi_l & ::= & e \leq e & \text{arithmetic inequalities} \\
\varphi_r & ::= & tr \in \mathcal{R} & \text{membership predicates} \\
tr & ::= & \epsilon \mid c \mid u \mid tr \cdot tr & \text{terms} \\
\mathcal{R} & ::= & \emptyset \mid \epsilon \mid c \mid w \mid \mathcal{R}\cdot\mathcal{R} \mid \mathcal{R}+\mathcal{R} \mid \mathcal{R}\cap\mathcal{R} \mid \mathcal{R}^C \mid \mathcal{R}^* & \text{regular expressions} \\
e & ::= & k \mid |tr| \mid k * e \mid e + e & \text{integer expressions}
\end{array}
$$

- variables $\{u_i\}^n_{i=1}$, $u_i \in U$, U = { String s | s $\in \sum^*$ } , $\sum^*$ finite words on alphabet $\sum$

- terms $\{tr_i\}^n_{i=1}$, tr := ε , c, u , $tr.tr$, terms are predicates $\varphi_r := tr \in \mathcal{R}$ ($\mathcal{R}$ is regex)

- integer expressions $\{e_i\}^n_{i=1}$, e_i is an equation for words

write $\phi[u_1/tr_1]\ldots[u_n/tr_n]$ (resp. $\phi[|u_1|/e_1]\ldots[|u_n|/e_n]$)

⇑ formula obtained ⇑ Replacing $|u_i|$ by expression e_i

by syntactically

substituting

in each occurrence

of u_i by term tr_i

Such a substitution is said to be well-defined if no variable u_i appears in tr_i

Or if no variable $|u_i|$ appears in e_i

The set of word variables appearing in a term:

Vars(ε) = $\varnothing$

Vars(c) = $\varnothing$

Vars(u) = { u }

Vars($tr_1 . tr_2$) = Vars(tr_1) $\cup$ Vars(tr_2)

Semantics.

The semantics of $E_{e,r,l}$ described using mapping η

Mapping η is called interpretation

Mapping η assigns a word $\boldsymbol{w} \in \sum^*$ to String s $\in$ U

$\Rightarrow \quad \eta(\boldsymbol{w}) = s$

Extending η to terms:

$\eta(\varepsilon) = \varepsilon$

$\eta(c) = c$

$\eta(tr_1 . tr_2) = \eta(tr_1) . \eta(tr_2)$

Regex $\mathcal{R}$

Language of regex $\mathcal{R}$, $\mathcal{L}(\mathcal{R})$

Given interpretation η we define mapping β_η

β_η associates Z to integer expressions as follows:

$\beta_\eta(k) = k,$

$\beta_\eta(|u|) = |\eta(u)|$

$\beta_\eta(|tr|) = |\eta(tr)|$

$\beta_\eta(k * e) = k * \beta_\eta(e)$

$\beta_\eta(e_1 + e_2) = \beta_\eta(e_1) + \beta(e_2)$

Semantics continued:

Formula in $E_{e,r,l}$ then evaluated to true and false as { ff , tt } as follows:

$$
\begin{array}{rcll}
val_\eta(\phi_1 \wedge \phi_2) & = & tt & \text{iff} \quad val_\eta(\phi_1) = tt \text{ and } val_\eta(\phi_2) = tt \\
val_\eta(\neg\phi_1) & = & tt & \text{iff} \quad val_\eta(\phi_1) = ff \\
val_\eta(tr \in \mathcal{R}) & = & tt & \text{iff} \quad \eta(tr) \in \mathcal{L}(\mathcal{R}) \\
val_\eta(tr_1 = tr_2) & = & tt & \text{iff} \quad \eta(tr_1) = \eta(tr_2) \\
val_\eta(tr_1 \neq tr_2) & = & tt & \text{iff} \quad \neg(\eta(tr_1) = \eta(tr_2)) \\
val_\eta(e_1 \leq e_2) & = & tt & \text{iff} \quad \beta_\eta(e_1) \leq \beta_\eta(e_2)
\end{array}
$$

Formula φ satisfiable if $\exists\ \eta$ s.t. $val_\eta(\varphi) = tt$

Formula φ unsatisfiable if $\forall\ \eta$ s.t. $val_\eta(\varphi) \neq tt$

Some Reminders Before Part 4, Inference Rules

An Inference Rule (یک قاعده استنتاج):

قاعده استنتاج قاعده‌ای است که با دریافت دسته‌ای از مقدمات به عنوان ورودی, نتیجه (یا نتایجی) را بازمی‌گرداند

ممکن نیست که مقدمات درست باشند اما نتیجه غلط باشد

Disjunctive Normal Form (DNF):

- $A \land B$
- A
- $(A \land B) \lor C$
- $(A \land \neg B \land \neg C) \lor (\neg D \land E \land F)$

Without Loss of Generality: in mathematics and logic, the term is used before an assumption in a proof which narrows the premise to some special case

Not in Disjunctive Normal Form (DNF):

- $\neg(A \lor B)$
- $A \lor (B \land (C \lor D))$

تمام فرمول‌های منطقی را می‌توان به شکلDNF تبدیل کرد
تبدیل یک فرمول به DNF شامل استفاده از معادله‌های منطقی مانند حذف دو منفی، قوانین دمورگان و قانون توزیع است
در برخی موارد تبدیل به DNF می‌تواند به انفجار نمایی از فرمول (Exponential Explosion of the Formula) منجر شود
مثال: فرمول منطقی زیر n جمله DNF ان 2^n جمله دارد (when distribution law applied to make the DNF form)

$$(X_1 \lor Y_1) \land (X_2 \lor Y_2) \land \ldots \land (X_n \lor Y_n)$$

در زیر یک گرامر از فرم نرمال DNF وجود دارد

- *disjunct* → *conjunct*
- *disjunct* → *disjunct* ∨ *conjunct*
- *conjunct* → *literal*
- *conjunct* → (*conjunct* ∧ *literal*)
- *literal* → *variable*
- *literal* → ¬*variable*

هر تابع بولی را می‌توان با یک و تنها یک DNF ، یکی از دو شکل متعارف، نشان داد که در ان یک متغیر هر متغیری می‌تواند باشد

فرم‌های نرمال اصلی

۱: فرم Principal Disjunctive Normal Form (PDNF)

فرمی که فقط از فصل عباراتی تشکیل شده که فقط شامل Conjunction (and) هستند

$(p \land q) \lor (\sim p \land q)$

۲: فرم Principal Conjunctive Normal Form (PCNF)

فرمی که فقط از عطف عباراتی تشکیل شده که فقط شامل Disjunction (or) هستند

$(p \lor q) \land (\sim p \lor q)$

Examples of Inference Rules

Rule of Inference	Name
$\dfrac{P \vee Q \quad \neg P}{\therefore Q}$	Disjunctive Syllogism
$\dfrac{P \to Q \quad Q \to R}{\therefore P \to R}$	Hypothetical Syllogism
$\dfrac{(P \to Q) \wedge (R \to S) \quad P \vee R}{\therefore Q \vee S}$	Constructive Dilemma
$\dfrac{(P \to Q) \wedge (R \to S) \quad \neg Q \vee \neg S}{\therefore \neg P \vee \neg R}$	Destructive Dilemma

Rule of Inference	Name
$\dfrac{P}{\therefore P \lor Q}$	Addition
$\dfrac{\begin{array}{c}P \\ Q\end{array}}{\therefore P \land Q}$	Conjunction
$\dfrac{P \land Q}{\therefore P}$	Simplification
$\dfrac{\begin{array}{c}P \to Q \\ P\end{array}}{\therefore Q}$	Modus Ponens
$\dfrac{\begin{array}{c}P \to Q \\ \neg Q\end{array}}{\therefore \neg P}$	Modus Tollens

4 Inference Rules

Without loss of generality, the inference rule is in Disjunctive Normal Form:

$$\text{NAME}: \frac{B_1 \; B_2 \; \dots \; B_n}{A} \; cond$$

- Name: name of the rule
- cond: side condition on conclusion A for the application of the rule
- B_1 , B_2 , $\dots$, B_n are called premises
- A is the conclusion of the rule
- (The side condition cond is omitted when it is tt)
- The premises B_1 , B_2 , $\dots$, B_n and conclusion A are formulae in $E_{e,r,l}$
- Each application:
  - consumes a conclusion A
  - produces the set of premises B_1 , B_2 , $\dots$, B_n
- Inference Rule soundness:
  conclusion A satisfiable $\rightarrow$ one of the premises B_i satisfiable
- Inference Rule local completeness:
  one of the premises B_i satisfiable $\rightarrow$ conclusion A satisfiable

Satisfiability of φ :

If:

- all inference rules NAME_1 ... NAME_m are locally complete

- φ satisfiable or one of the produced premises B_1 , B_2 , $\dots$, B_n satisfiable

then: φ is also satisfiable

If:

- all inference rules NAME_1 ... NAME_m are sound

- None of the produced premises B_1 , B_2 , $\dots$, B_n satisfiable

then: φ is unsatisfiable

We organize the inference rules in four groups:

NAME_1: Removing Dis-equalities

NAME_2: Simplifying Equalities

NAME_3: Removing Membership Predicates

NAME_4: Propagating Term Lengths

Lemma 1. The inference rules of this section are sound and locally complete.

Having 3 rules:

- Rule Eq
- Rule Not-Eq
- Rule Diseq-Split

$$\textsc{Not-Eq}: \frac{*}{tr \neq tr \wedge \phi} \qquad\qquad \textsc{Eq}: \frac{\phi}{tr = tr \wedge \phi}$$

$$\textsc{Diseq-Split}: \frac{\textsc{Split}_{\textsc{Diseq-Split}} \cup \textsc{Split}'_{\textsc{Diseq-Split}} \cup \textsc{Split}''_{\textsc{Diseq-Split}}}{tr \neq tr' \wedge \phi}$$

Using

- Rule Not-Eq
- Rule Diseq-Split

To eliminate dis-equalities.

Rule Not-Eq,

$$\textsc{Not-Eq} : \frac{*}{tr \neq tr \wedge \phi}$$

Rule Not-Eq establishes that $tr \neq tr \wedge \varphi$ is not satisfiable without φ as a premise

In Diseq-Split,

$$\textsc{Diseq-Split} : \frac{\textsc{Split}_{\textsc{Diseq-Split}} \cup \textsc{Split}'_{\textsc{Diseq-Split}} \cup \textsc{Split}''_{\textsc{Diseq-Split}}}{tr \neq tr' \wedge \phi}$$

Eliminating dis-equalities involving arbitrary terms,

only a set of premises in the union of SPLIT SPLIT' SPLIT'' will remain:

alphabet $\sum$ is finite $\rightarrow$ possible to replace any dis-equality with a finite set of equalities

assume a formula $tr \neq tr' \wedge \varphi$ in $E_{e,r,l}$

Disequality $tr \neq tr'$ holds $\leftrightarrow$ the words w_{tr} , $w_{tr'}$ denoted by the terms tr , tr' are different

- This corresponds to one of three cases

Read onto the next page!!

Assume three fresh variables $u, v, v' \in U$ (reminder $U \subseteq \sum^*$)

In the first case,

after a common prefix w_u (meaning $w_u \subseteq w_{tr}$ and $w_u \subseteq w_{tr}'$) ,

the words w_{tr} , w_{tr}' will contain different letters $c \in w_{tr}$ and $c' \in w_{tr}'$ s.t. $c \neq c'$