Norn: an SMT solver for string constraints

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Lukáš Holík[3], Ahmed Rezine[4], Philipp Rümmer[1], and Jari Stenman[1]

[1] Department of Information Technology, Uppsala University, Sweden
[2] Institute of Information Science, Academia Sinica, Taiwan
[3] Faculty of Information Technology, Brno University of Technology, Czech Republic
[4] Department of Computer and Information Science, Linköping University, Sweden

Abstract. We present version 1.0 of the Norn SMT solver for string constraints. Norn is a solver for an expressive constraint language, including word equations, length constraints, and regular membership queries. As a feature distinguishing Norn from other SMT solvers, Norn is a decision procedure under the assumption of a set of acyclicity conditions on word constraints, without any restrictions on the use of regular membership.

1 Introduction

We introduce version 1.0 of the Norn SMT solver. Norn targets an expressive constraint language that includes word equations, length constraints, and regular membership queries. Norn is based on the calculus introduced in [1]. This version adopts several improvements on the original version, which allow it to efficiently establish or refute the satisfiability of benchmarks that are out of the reach of existing state of the art solvers.

Norn aims to establish satisfiability of constraints written as Boolean combinations of: (i) word equations such as equalities $(a \cdot u = v \cdot b)$ or disequalities $(a \cdot u \neq v \cdot b)$, where a, b are letters and u, v are string variables denoting words of arbitrary lengths, (ii) length constraints such as $(|u| = |v| + 1)$, where $|u|$ refers to the length of the word denoted by string variable u, and (iii) predicates representing membership in regular expressions, e.g., $u \in c \cdot (a+b)^*$. The analysis is not trivial as it needs to capture subtle interactions between different types of predicates. The general decidability problem is still open. We guarantee termination of our procedure in case the considered initial constraints are *acyclic*. Acyclicity is a syntactic condition and it ensures that no variable appears more than once in word (dis)equalies during the analysis. This defines a fragment that is rich enough to capture all the practical examples we have encountered.

This version of the Norn solver follows a DPLL(T) architecture in order to turn the calculus introduced in [1] into an effective proof procedure, and introduces optimizations that are key to its current efficiency: an improved approach to handling disequalities, and a better strategy for splitting equalities compared to [1]. Norn accepts SMT-LIB scripts as input, both in the format proposed in [2] and in the CVC4 dialect [5], and can handle the combination of string constraints and linear integer arithmetic. In addition, Norn contains a fixed-point engine for processing recursive programs in the form of Horn constraints, which are expressed as SMT-LIB scripts with uninterpreted predicates; the algorithm for solving such Horn constraints was introduced in [8, 1].

Related work. Over the last years, several SMT solvers for strings and related logics have been introduced, starting from a number of tools that handled strings by means of a translation to bit-vectors [4, 9, 10], thus assuming an a priori fixed upper bound on the length of the possible words. More recently, DPLL(T)-based string solvers started to enter the stage, lifting the restriction to strings of bounded length; this generation of solvers includes Z3-str [12], CVC4 [5], and S3 [11], which are all compared to Norn in Sect. 4. Most of those solvers are more restrictive than Norn in their support for language constraints. In our experience, such restrictions are particularly problematic for software model checking, where regular membership constraints offer an elegant and powerful way of expressing and synthesising program invariants.

2 Logic and Calculus

Our constraint language includes word equations, membership queries in regular languages and length and arithmetic inequalities. We assume a finite alphabet Σ and write Σ^* to mean the set of finite words over Σ. We work with a set U of string variables denoting words in Σ^* and write $\mathbb{Z}$ for the set of integer numbers.

Constraints. We let variables u, v range over U. We write $|u|$ to mean the length of the word denoted by u, k to mean an integer in $\mathbb{Z}$, c to mean a letter in Σ and w to mean a word in Σ^*. Syntax of the constraints is given by:

$$\begin{array}{rcll}
\phi & ::= & \phi \wedge \phi \,\big|\, \neg\phi \,\big|\, \varphi & \text{constraints} \\
\varphi & ::= & t = t \,\big|\, e \leq e \,\big|\, t \in \mathcal{R} & \text{atomic predicates} \\
t & ::= & \varepsilon \,\big|\, c \,\big|\, u \,\big|\, t \cdot t & \text{terms} \\
\mathcal{R} & ::= & \emptyset \,\big|\, \varepsilon \,\big|\, c \,\big|\, w \,\big|\, \mathcal{R} \cdot \mathcal{R} \,\big|\, \mathcal{R} + \mathcal{R} \,\big|\, \mathcal{R} \cap \mathcal{R} \,\big|\, \mathcal{R}^C \,\big|\, \mathcal{R}^* & \text{regular expressions} \\
e & ::= & k \,\big|\, |t| \,\big|\, k * e \,\big|\, e + e & \text{integer expressions}
\end{array}$$

A constraint is said to be *linear* if no variable appears more than once in any of its (dis)equalities. We write w_t to mean a word denoted by a term t. Semantics of the constraints are straightforward [1].

Calculus. Given a constraint ϕ in our logic, we build a proof tree rooted at ϕ by repeatedly applying inference rules. We assume here, without loss of generality, that ϕ is given in Disjunctive Normal Form. An inference rule is of the form:

$$\frac{B_1\ B_2\ \ldots\ B_n}{A}\ \textsc{Name}_{cond}$$

$\textsc{Name}$ is the name of the rule, *cond* is a side condition on A for the application of the rule, $B_1\ B_2\ \ldots\ B_n$ are the premises, and A is the conclusion. Premises and conclusions are constraints. Each application consumes a conclusion and produces premises. In our calculus, if one of the produced premises turns out to be satisfiable, then ϕ is also satisfiable. If none of the produced premises is satisfiable, then ϕ is unsatisfiable. The inference rules are introduced in [1]. The repeated application of the rules starting from a constraint ϕ is guaranteed to terminate (i.e., giving a decision procedure) in case ϕ is *acyclic*. Intuitively, *acyclicity* is a syntactic condition on the occurences of variables. This condition ensures all (dis)equalities are linear, whether in ϕ or after the application of some inference rule. We describe one rule. Other rules are introduced in [1].

Rule EQ-VAR eliminates variable u from the equality $u \cdot t_1 = t_2 \wedge \phi$. The equality is satisfied if a word w_u coincides with the prefix of a word w_{t_2}. We assume $u \cdot t_1 = t_2 \wedge \phi$ is linear (see [1] for the general case). There are two sets of premises. The first set corresponds to all the cases where w_u coincides with a word w_{t_3} where t_2 is the concatenation $t_3 \cdot t_4$. The second set represents all situations where w_{t_3} is a prefix of w_u which is a prefix of $w_{t_3 \cdot v}$ with t_2 being written as the concatenation $t_3 \cdot v \cdot t_4$.

$$\frac{\begin{array}{c}\{t_1 = t_4 \wedge \phi[u/t_3] \mid t_2 = t_3 \cdot t_4\} \cup \\ \{t_1 = v_2 \cdot t_4 \wedge \phi[u/t_3 \cdot v_1][v/v_1 \cdot v_2] \mid t_2 = t_3 \cdot v \cdot t_4\}\end{array}}{u \cdot t_1 = t_2 \wedge \phi} \ \textsc{Eq-Var}\big(u \cdot t_1 = t_2 \text{ is linear}\big)$$

3 A DPLL(T)-Style Proof Procedure for Strings

We follow the classical DPLL(T)-architecture [7] to turn the calculus from the previous section into an effective proof procedure. For a given (quantifier-free) formula in our logic, first a Boolean skeleton is computed, abstracting every atom to a Boolean variable. A SAT-solver is then used to check satisfiability of the Boolean skeleton, producing (in the positive case) an implicant of the skeleton; the implicant is subsequently translated back to a conjunction of string literals, and checked for satisfiability in the string logic.

Our theory solver for checking conjunctions of string literals implements the rules of Sect. 2 and Sect. 3.1, and handles all necessary splitting internally, i.e., without involving the SAT-solver. In our experience (which is consistent with observations in other domains, e.g., [3]), this approach makes it easier to integrate splitting heuristics, and often shows better performance in practice. In particular, our approach to split equalities is model-based and exploits information extracted from arithmetic constraints in order to prune the search space; the method is explained in Sect. 3.2.

Starting from a conjunction $\phi = (\phi_= \wedge \phi_{\neq} \wedge \phi_{\in} \wedge \phi_a)$ of literals (which is here split into equalities $\phi_=$, disequalities $\phi_{\neq}$, membership constraints $\phi_{\in}$, and arithmetic constraints ϕ_a) the theory solver performs depth-first exploration until either a proof branch is found that cannot be closed (and constitutes a model), or all branches have been closed and discharged. If all branches could be closed, information about the string literals involved in showing unsatisfiability is propagated back to the SAT-solver as a blocking clause.

Rules are applied to $\phi = (\phi_= \wedge \phi_{\neq} \wedge \phi_{\in} \wedge \phi_a)$ in the following order: (1) Satisfiability of ϕ_a (in Presburger arithmetic) is checked, (2) Compound disequalities in $\phi_{\neq}$ are eliminated (Sect. 3.1), (3) Equalities in $\phi_=$ with complex left-hand side are split (Sect. 3.2), (4) Membership constraints in $\phi_{\in}$ with complex term are split, and (5) Satisfiability of all remaining membership literals and arithmetic constraints is checked using automata algorithms.

3.1 Efficient Handling of Disequalities

To handle disequalities, we proceed differently than the method presented in [1]. For each disequality of the form $t \neq t'$, the rule DISEQ-SPLIT produces only two premises. The first premise corresponds to the case where the words w_t and $w_{t'}$ have different length. The second case is when w_t and $w_{t'}$ have the same length but contain different letters $c \neq c'$ after a common prefix. Rather than constructing a premise for each pair of different letters (as it is done in [1]), we introduce two special variables μ and μ' (called *witness variables*) such that the letters c and c' correspond to the words denoted by μ and μ'. Therefore, the length of these witness variables should be always equal to one and this fact is added to the arithmetic constraints. Furthermore, we add a disequality $\mu \neq \mu'$ between these two witness variables in order to denote that c should be different from c'. Assuming fresh variables u, v and v', we rewrite $t \neq t'$ as two equalities $t = u \cdot \mu \cdot v$ and $t' = u \cdot \mu' \cdot v'$. Finally, w.l.o.g, we restrict the inference rules such that witness variables can only be substituted by other witness variables.

$$\frac{\begin{array}{c}\{|t| \neq |t'| \wedge \phi\} \cup \\ \{|v| = |v'| \wedge t = u \cdot \mu \cdot v \wedge t' = u \cdot \mu' \cdot v' \wedge |\mu| = 1 \wedge |\mu'| = 1 \wedge \mu \neq \mu' \wedge \phi\}\end{array}}{t \neq t' \wedge \phi} \text{ DISEQ-SPLIT}$$

The new Rule REG-WITNESS can only be applied to a witness variable μ in a certain case. For a formula ϕ, we define the condition $\Theta(\phi,\mu)$ which means that μ appears in ϕ only in disequalities. The Rule REG-WITNESS replaces all the membership predicates $\{\mu \in R_i\}_{i=1}^{n}$ with an arithmetic constraint $\mathsf{Unicode}(R_1,R_2,\ldots,R_n,\mu)$. This constraint uses a fresh variable μ_{uni} such that the set of possible lengths of the word denoted by μ_{uni} represents the set of Unicode characters belonging to the intersection of all regular expressions $\{R_i\}_{i=1}^{n}$. In order to do so, we first construct a finite state automaton representing the intersection of $\{R_i\}_{i=1}^{n}$. Furthermore, we restrict our automaton to accept only words of size exactly one (since μ is a witness variable). The obtained automaton is then determined. Notice that the determined automaton has only transitions from the initial state to the final one. Each transition of this automaton are labelled by a Unicode character interval as specified by the automata library [6] we are using. (Observe that each letter in our alphabet is represented by its unique Unicode character.) Then, for each transition labeled by an interval of the form $\{\text{min}, \ldots, \text{max}\}$, we associate an arithmetic constraint of the form $min \leq |\mu_{uni}| \leq max$. Finally, our arithmetic constraint $\mathsf{Unicode}(R_1,R_2,\ldots,R_n,\mu)$ will be the disjunction of all associated arithmetic constraints to all the transitions of the automaton. In the case that the intersection is empty, we set $\mathsf{Unicode}(R_1,R_2,\ldots,R_n,\mu)$ to `ff`.

$$\frac{\mathsf{Unicode}(\mathcal{R}_1 \cap \ldots \cap \mathcal{R}_m, u) \wedge \phi}{\mu \in \mathcal{R}_1 \wedge \ldots \wedge u \in \mathcal{R}_m \wedge \phi} \text{ REG-WITNESS}_{(\Theta(\phi,\mu))}$$

Finally, the Rule DISEQ-WITNESS replaces a disequality of the form $\mu \neq \mu'$ by the arithmetic constraint $|\mu_{uni}| \neq |\mu'_{uni}|$.

$$\frac{|\mu_{uni}| \neq |\mu'_{uni}| \wedge \phi}{\mu \neq \mu' \wedge \phi} \text{ DISEQ-WITNESS}_{(\Theta(\phi,\mu))}$$

3.2 Length-Guided Splitting of Equalities

The original calculus rule for handling complex equalities is EQ-VAR, which systematically enumerates the different ways of matching up left-hand and right-hand side terms. For a practical proof procedure, naive use of this rule is sub-optimal in two respects: the number of cases to be considered grows quickly (in the worst case, exponentially in the number of equalities); and the rule does not provide any guidance on the order in which the cases should be considered, which can have dramatic impact on the performance for satisfiable problems. We found that both aspects can be improved by eagerly taking arithmetic constraints on the length of strings into account.

To present the approach, we assume that conjunctions $\phi = (\phi_= \wedge \phi_{\neq} \wedge \phi_{\in} \wedge \phi_a)$ are continuously saturated by propagating length information from $\phi_=$ to ϕ_a: for every equality $s = t$, a corresponding length equality $|s| = |t|$ is added, compound expressions $|s \cdot t|$ are rewritten to $|s| + |t|$, and the length $|w|$ of concrete words $w \in \Sigma^*$ is evaluated. In addition, for every variable v an inequality $|v| \geq 0$ is generated. Similar propagation is possible for membership constraints in $\phi_{\in}$.

Prior to splitting equalities from $\phi_=$, it is then possible to check the satisfiability of arithmetic constraints ϕ_a (using any solver for Presburger arithmetic), and compute a satisfying assignment β. This assignment defines the length $val_\beta(|v|)$ of all string variables v, and thus uniquely determines how the right-hand side of an equality $u \cdot t_1 = t_2$ should be split into a prefix corresponding to u, and a suffix corresponding to t_1. We obtain the following modified splitting rule, which has the side condition that $u \cdot t_1 = t_2 \cdot v \cdot t_3$ is linear, and that a satisfying assignment β of ϕ_a exists such that $val_\beta(|t_2|) \leq val_\beta(|u|) \leq val_\beta(|t_2 \cdot v|)$:

$$\dfrac{\begin{array}{l}\big((t_1 = v_2 \cdot t_3 \wedge \phi_a \wedge \phi)[u/t_2 \cdot v_1]\big)[v/v_1 \cdot v_2] \\ u \cdot t_1 = t_2 \cdot v \cdot t_3 \wedge (|u| < |t_2| \wedge \phi_a) \wedge \phi \\ u \cdot t_1 = t_2 \cdot v \cdot t_3 \wedge (|t_1| < |t_3| \wedge \phi_a) \wedge \phi\end{array}}{u \cdot t_1 = t_2 \cdot v \cdot t_3 \wedge \phi_a \wedge \phi}\ \textsc{Len-Eq-Split}$$

A similar rule is introduced to cover the situation that the right-hand side has to be split between two concrete letters, i.e., in case we have $val_\beta(|u|) = val_\beta(|t_2|)$ and $val_\beta(|t_1|) = val_\beta(|t_3|)$ for an equation $u \cdot t_1 = t_2 \cdot t_3$.

4 Implementation and Experiments

We compare the new version of Norn[5] to other solvers on two sets of benchmarks. First, we use the well-known set of Kaluza benchmarks, which were translated to SMT-LIB by the authors of CVC4 [5]. These benchmarks contain constraints generated by a Javascript analysis tool, and are mainly equational, with relatively little use of regular expressions. Results are given in Table 1, and show that currently Z3-str [12] performs best for this kind of benchmarks; however, Norn can solve 27 benchmarks that no other tool can handle (Table 2). S3 [11] produced internal errors on a larger number of the Kaluza benchmarks, and sometimes results that were contradictory with the other

[5] Tool and benchmarks are available on http://user.it.uu.se/%7Ejarst116/norn/

Table 1. Experimental results. All experiments were done on an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8. Runtime was limited to 240s (wall clock time), and heap space to 1.5GB. CEGAR benchmarks downsized from UTF16 when necessary.

		Norn 1.0	**CVC4 1.4**	**Z3-str 1.0.0**	**S3**
Kaluza	(sat)	33 072	33 727	34 770	30 925
	(unsat)	11 595	11 625	11 799	11 408
	(unknown)	2 617	1 932	715	3 081
	(crash)	0	0	0	1 870
CEGAR	(sat)	712	268	–	307
	(unsat)	315	112	–	530
	(unknown)	0	340	–	158
	(crash/OOM)	0	307	–	32

Table 2. Complementarity of the results: number of problems for which one tool can show sat/unsat, whereas another tool times out or crashes. For instance, Norn can prove satisfiability of 480 Kaluza benchmarks on which CVC4 times out.

		Norn		**CVC4**		**Z3-str**		**S3**	
		sat	unsat	sat	unsat	sat	unsat	sat	unsat
Norn	(Kaluza)	–	–	+1 135	+57	+1 698	+231	+64	+125
	(CEGAR)	–	–	0	0	–	–	0	0
CVC4	(Kaluza)	+480	+27	–	–	+1 043	+174	+1	0
	(CEGAR)	+436	+211	–	–	–	–	+134	+396
Z3-str	(Kaluza)	0	+27	0	0	–	–	0	0
	(CEGAR)	–	–	–	–	–	–	–	–
S3	(Kaluza)	+2 184	+339	+2 708	+312	+3 750	+486	–	–
	(CEGAR)	+134	+56	+51	+22	–	–	–	–

solvers: for 95 problems, S3 claimed unsat, whereas Z3-str and CVC4 reported sat. For 27 of those, also Norn gave the answer sat. No contradictions were observed between CVC4, Z3-str, and Norn.

As a second set of benchmarks, we considered queries generated during CEGAR-based verification of string-processing programs [1]; those queries are quite small, but make heavy use of regular expressions and operators like the Kleene star. Norn could solve all of the benchmarks. Comparison with Z3-str was not possible, since the solver does not support regular expressions. CVC4 and S3 both showed timeouts, ran out of memory, or crashed on a large number of the benchmarks. For 10 problems, CVC4 claimed unsat, but Norn sat; for 1 problem, CVC4 reported sat, but Norn unsat. After manual inspection of those examples, we concluded that Norn was giving the correct answers; the CVC4 behaviour might be related to fact that we were forced to use the options `--strings-exp` to enable experimental string functionality, required for negated membership predicates. More drastically, S3 and Norn gave contradicting answers in altogether 413 cases, with manual inspection again indicating that the answer by Norn was correct. We plan to clarify all issues with the authors of CVC4 and S3 (they could not be resolved before paper submission).

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 150–166. Springer (2014)
2. Bjorner, N., Ganesh, V., Michel, R., Veanes, M.: Smt-lib sequences and regular expressions. In: Fontaine, P., Goel, A. (eds.) SMT 2012. EPiC Series, vol. 20, pp. 77–87. EasyChair (2013)
3. Griggio, A.: A practical approach to satisfiability modulo linear integer arithmetic. JSAT 8(1/2), 1–27 (2012)
4. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A Solver for String Constraints. In: ISTA. pp. 105–116. ACM (2009)
5. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 646–662. Springer (2014)
6. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010), `http://www.brics.dk/automaton/`
7. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)
8. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for Horn-clause verification. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 347–363. Springer (2013)
9. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A Symbolic Execution Framework for JavaScript. In: IEEE Symposium on Security and Privacy. pp. 513–528. IEEE Computer Society (2010)
10. Saxena, P., Hanna, S., Poosankam, P., Song, D.: FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In: NDSS. The Internet Society (2010)
11. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Ahn, G., Yung, M., Li, N. (eds.) CCS. pp. 1232–1243. ACM (2014)
12. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A Z3-based string solver for web application analysis. In: Meyer, B., Baresi, L., Mezini, M. (eds.) ESEC/FSE. pp. 114–124. ACM (2013)