Efficient E-matching for SMT Solvers

Leonardo de Moura,
Nikolaj Bjørner
Microsoft Research, Redmond

The Z3tting

- Z3 is an inference engine tailored towards formulas arising from program verification tools (Boogie/Spec#).
  - Large formulas
  - Integer arithmetic + other theories
  - Mostly universally quantified axioms
- Contributions:
  - E-matching code trees for efficient matching over **congruence classes**.
  - Inverted path indices for efficient **incremental** matching.

SMT solving using DPLL(QT)

- Review:
  - Γ: Context of asserted literals, initially $\Gamma = \emptyset$
  - C: list(conjunction) of clauses
- Combined with theories in DPLL(T)
  - Subsets of Γ are propagated to theories.
  - Γ = { a = f(a), a $\neq$ f(f(a)) } unsat by Th(Equality).
  - Th(Equality) maintains *E-graph* (congruence closure)
    - Nodes are sets of terms appearing in C
    - Each set is congruence class of equalities asserted by Γ
    - E-graph({ a = f(a), a $\neq$ f(f(a)), b = c }) = {{a, f(a), ff(a)}, {b, c}}
    - class(a) = {a, f(a), ff(a)},
    - find(a) = find(f(a)) = a

Instantiating Quantifiers

But how to find *t* during instantiation?

$(\forall x.\varphi(x) \rightarrow \varphi(t))$

Approach:

1. Extract patterns from quantified formulas:

   $\forall$x,i,v. { select(store(x,i,v),i) } . select(store(x,i,v),i) = v

2. E-match: Search E-graph of Γ for terms matching patterns.

3. Add axioms for the matches that were found.

The E-matching problem

Input: A set of ground equations *E* a ground term *t* and a term *p,* where *p* possibly contains variables.

Output: The set of substitutions θ modulo *E* over the variables in *p,* such that

$$E \models t = \theta(p)$$

The E-matching challenge

- E-matching is in theory NP-hard
- The real challenge is finding new matches
  - **Incrementally** during a backtracking search
  - In a **large** database of patterns, many **sharing** substantial structure

Abstract E-matching

$$match(x,t,S) = \{\beta \cup \{x \mapsto t\} \mid \beta \in S, x \notin dom(\beta)\} \; \cup$$
$$\{\beta \mid \beta \in S, find(\beta(x)) = find(t)\}$$
$$match(c,t,S) = S \; if \; c \in class(t)$$
$$match(c,t,S) = \emptyset \; if \; c \notin class(t)$$
$$match(f(p_1,\ldots,p_n),t,S) = \bigcup_{f(t_1,\ldots,t_n) \in class(t)} match(p_n,t_n,\ldots,match(p_1,t_1,S))$$

A more efficient approach

- Match is invoked for every pattern in database.
- To avoid common work:
  - Compile set of patterns into instructions.
    - By partial evaluation of naïve algorithm
  - Instruction sequences share common sub-terms.
  - Substitutions are stored in registers, backtracking just updates the registers.

E-matching code-trees

- Pattern $f(x_1, g(x_1, a), h(x_2), b)$:

Pc	Instructions
pc0	**init**(f, pc1)
pc1	**check**(4, b, pc2)
Pc2	**bind**(2, g, 5, pc3)
Pc3	**compare**(1, 5, pc4)
Pc4	**check**(6, a, pc5)
Pc5	**bind**(3, h, 7, pc6)
Pc6	**yield**(1,7)

Instruction	f(h(a),g(h(c),a),h(c), b)
init(f)	reg[1] ← h(a), reg[2] ←g(h(c),a), reg[3] ← h(c), reg[4] ← b 👍
check(4, b)	reg[4] = b 👍
bind(2, g, 5)	reg[5] ← h(c), reg[6] ← a 👍
compare(1, 5)	h(a) = reg[1] ≠ reg[5] = h(c) 👎

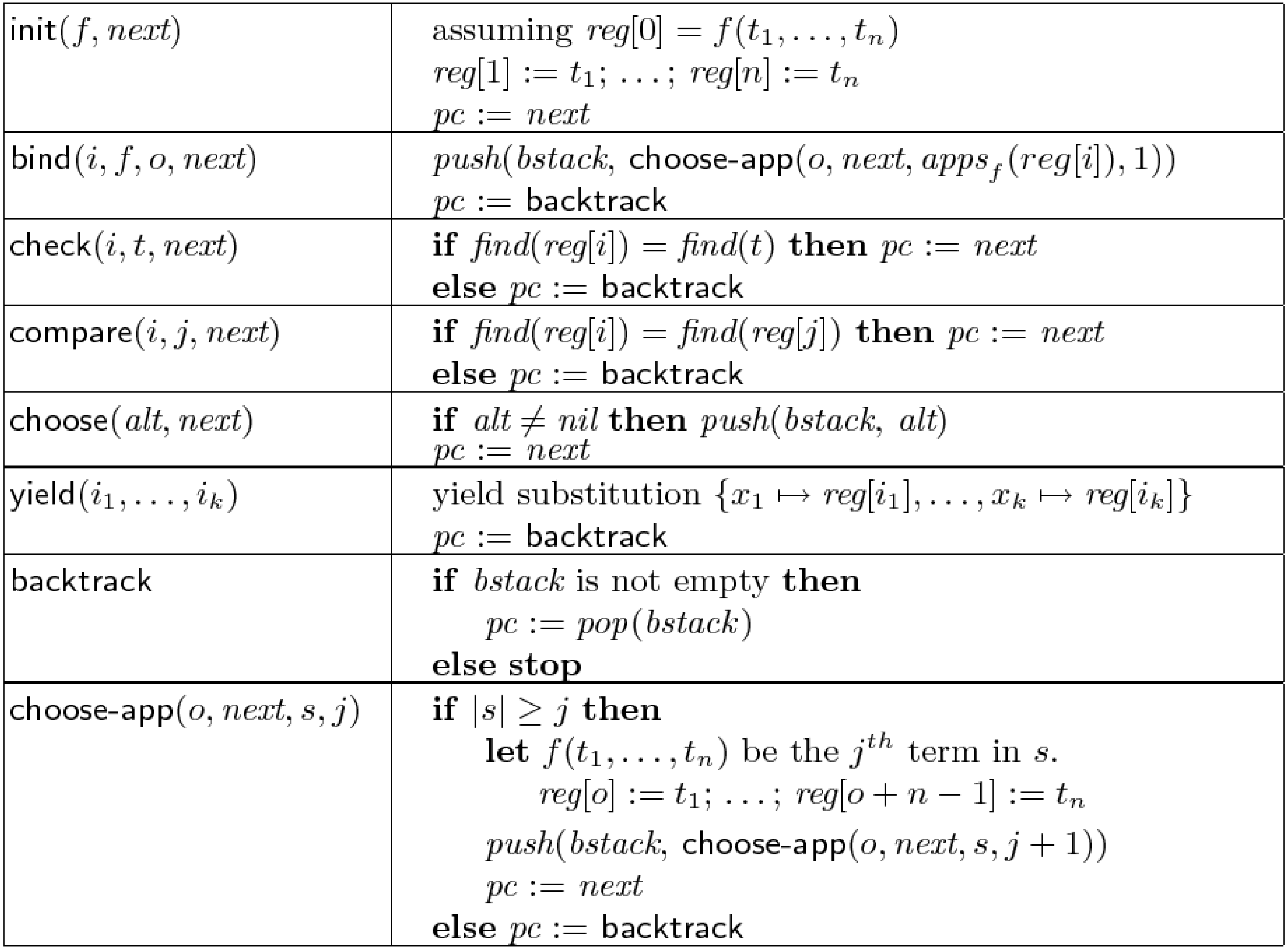
E-matching code-trees

- Pattern $f(x_1, g(x_1, a), h(x_2), b)$:

Pc	Instructions
pc0	**init**(f, pc1)
pc1	**check**(4, b, pc2)
Pc2	**bind**(2, g, 5, pc3)
Pc3	**compare**(1, 5, pc4)
Pc4	**check**(6, a, pc5)
Pc5	**bind**(3, h, 7, pc6)
Pc6	**yield**(1,7)

Instruction	f(h(a),g(h(a),a),h(c), b)
init(f)	reg[1] ← h(a), reg[2] ←g(h(a),a), reg[3] ← h(c), reg[4] ← b 👍
check(4, b)	reg[4] = b 👍
bind(2, g, 5)	reg[5] ← h(a), reg[6] ← a 👍
compare(1, 5)	h(a) = reg[1] =reg[5] = h(a) 👍
check(6, a)	a = reg[6] = a 👍
bind(3, h, 7)	reg[7] ← c 👍
yield(1,7)	$X_1 \to h(a)$, $X_2 \to c$ 👍

The E-matching abstract machine



init$(f, next)$	assuming $reg[0] = f(t_1, \ldots, t_n)$ $reg[1] := t_1; \ldots; reg[n] := t_n$ $pc := next$
bind$(i, f, o, next)$	$push(bstack,$ choose-app$(o, next, apps_f(reg[i]), 1))$ $pc :=$ backtrack
check$(i, t, next)$	**if** $find(reg[i]) = find(t)$ **then** $pc := next$ **else** $pc :=$ backtrack
compare$(i, j, next)$	**if** $find(reg[i]) = find(reg[j])$ **then** $pc := next$ **else** $pc :=$ backtrack
choose$(alt, next)$	**if** $alt \neq nil$ **then** $push(bstack, alt)$ $pc := next$
yield$(i_1, \ldots, i_k)$	yield substitution $\{x_1 \mapsto reg[i_1], \ldots, x_k \mapsto reg[i_k]\}$ $pc :=$ backtrack
backtrack	**if** $bstack$ is not empty **then** $pc := pop(bstack)$ **else stop**

choose-app$(o, next, s, j)$	**if** $\lvert s \rvert \geq j$ **then**
	    let $f(t_1, \ldots, t_n)$ be the j^{th} term in s.
	        $reg[o] := t_1; \ldots; reg[o+n-1] := t_n$
	    $push(bstack,$ choose-app$(o, next, s, j+1))$
	    $pc := next$
	else $pc :=$ backtrack

Additional instructions

- Forward pruning
  - Prune exponential search early on
    - f(g(x,y), h(x,z)) – first check that t_1 = g(…) and t_2 = h(…) when matching f(t_1, t_2)
- Multi-patterns
  - Continue
  - Join = continue + compare

Incremental matching

5 = select(b, 2) $E_1 = \{ \{5, select(b,2)\} , \{b\} \}$

c = store(a, 2, 4) $E_2 = E_1 \cup \{ \{c, store(a,2,4) \}$

b = c $E_3 = \{ \{b, c, store(a,2,4)\},$
$\{5, select(b,2)\} \}$

$E_3 \models 5 = select(b,2) = select(store(a,2,4),2)$

Observation: pattern select(store(x, i, v), i) gets enabled when *child* of select is merged with term labeled by store.

Inverted path indices

Index all patterns with f(…g(…)…) sub-term, that *may* become enabled when

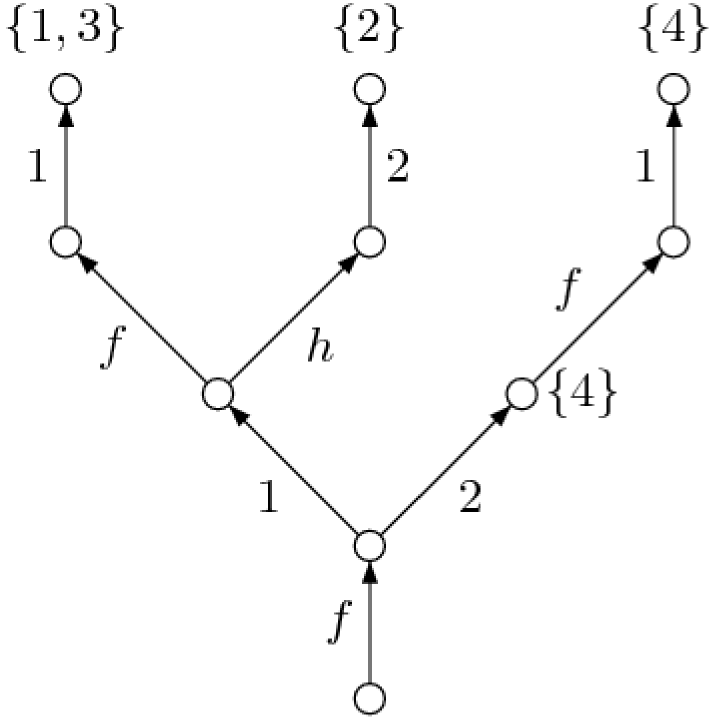
merge(n_1, n_2) where

$\exists$ parent p_1 of n_1 . Label(p_1) = f(…n_1…)

$\exists$ sibling m_2 of n_2 . Label(m_2) = g(…)

Pattern id	pattern	Path to g under f	Inverted paths
p1	f(f(g(x) ,a),x)	p1 → g: f.1.f.1	(f,g): f.1.f.1 → p1
p2	h(c, f(g(y) , x))	p2 → g: h.2.f.1	(f,g): f.1.h.2 → p2
p3	f(f(g(x) ,b),y)	p3 → g: f.1.f.1	(f,g): f.1.f.1 → p3
p4	f(f(a, g(x)), g(y))	p4 → g: f.1.f.2, f.2	(f,g): f.2.f.1 → p4, (f,g): f.2→ p4

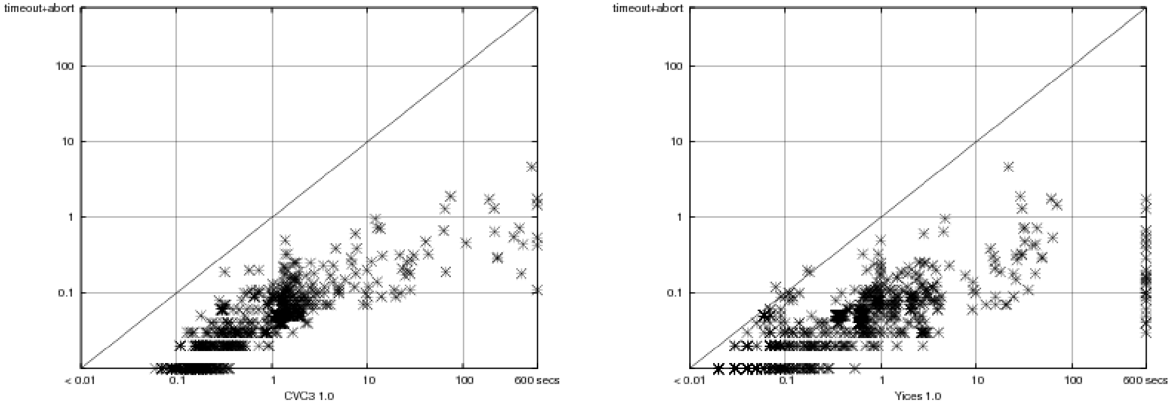
Inverted path index



When to apply E-matching

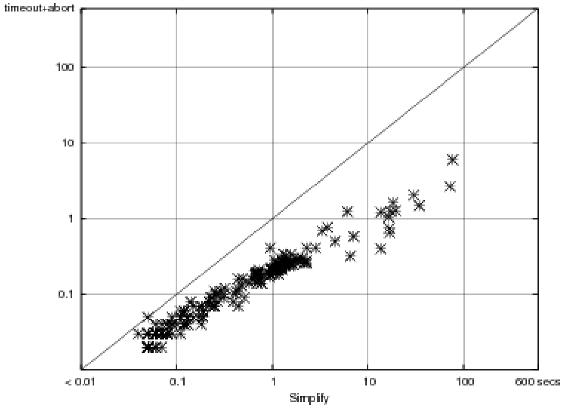
- **Lazy Instantiation**:
  - Have SAT core assign all Boolean variables.
  - Then find new quantifier instantiations.
  - Useful if most instantiations are useless and explode the search space.
- **Eager Instantiation**:
  - Find new quantifier instantiations whenever new terms are created and new equalities are asserted.
  - Useful if instantiations help pruning the search space.
- **Hybrid**:
  - Uses scoring on useful quantifiers to promote/demote instantiation time.

Experimental evaluation

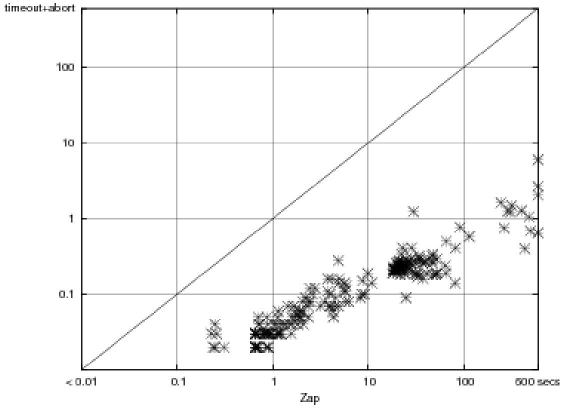


(a) Z3 vs. CVC3 1.0 (b) Z3 vs. Yices 1.0

Fig. 8. SMT-LIB Benchmarks

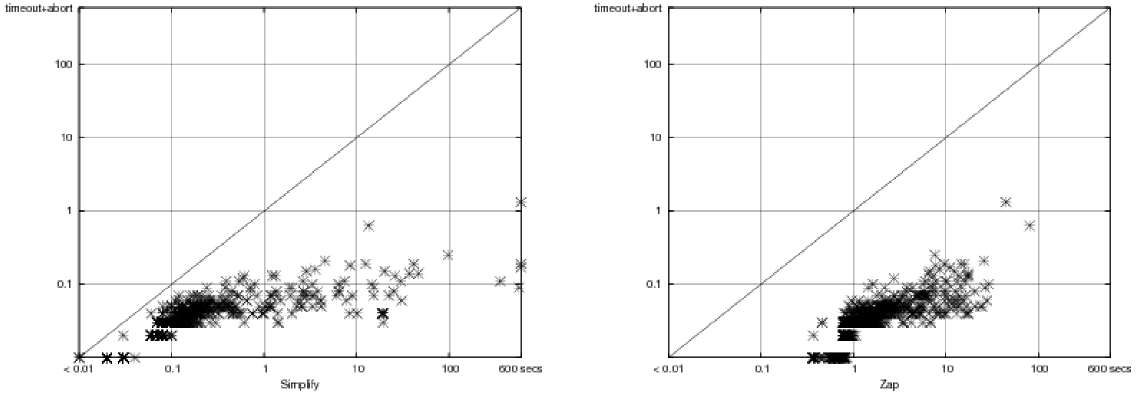


(a) Z3 vs. Simplify



(b) Z3 vs. Zap 2.0

Experimental evaluation



(a) Z3 vs. Simplify (b) Z3 vs. Zap 2.0

Fig. 10. Boogie Benchmarks

	ESC/Java		Boogie		S-expr Simplifier	
	# valid	time	# valid	time	# valid	time
Simplify	2331	499.03	903	1851.29	18	10985.80
Zap	2222	6297.04	901	2612.64	22	777.78
Z3 (*lazy*)	2331	212.81	907	157.2	32	2904.27
Z3 (*lazy wo. code trees*)	2331	224.14	907	240.44	28	2369.00
Z3 (*eager wo. inc.*)	2331	1495.07	907	229.2	10	2410.52
Z3 (*eager mod-time*)	2331	85.1	907	39.79	32	1341.38
Z3 (*eager wo. code trees*)	2331	48.28	907	26.85	32	654.62
Z3 (*default*)	**2331**	**45.22**	**907**	**18.47**	**32**	**194.54**

E-matching limitations

E-matching needs ground (seed) terms.
It fails to prove simple properties when ground (seed) terms are not available.

Example:

$(\forall x \,.\, f(x) \leq 0) \land (\forall x \,.\, f(x) > 0)$

Matching loops:

$(\forall x \,.\, f(x) = g(f(x))) \land (\forall x \,.\, g(x) = f(g(x)))$

- Inefficiency and/or non-termination.

- Some solvers have support for detecting matching loops based on instantiation chain length.

- Our technology for inferring patterns is *weak*. Strong reliance on (Spec#/Boogie) compiler or theory supplied patterns.

Future work

- Model checking.
- Superposition calculus + SMT.
- Decidable fragments.

Conclusions

- Matching-time significantly reduced when using E-matching **code trees** and inverted path indices.
- **Inverted path indices**: Pay for what you use, not for what you might.
- Lazy vs. Eager depends on quality of patterns.

Related work

DPLL(T):	Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: CAV 04.
Simplify:	Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. (2005)
	Nelson, G.: Techniques for program verification. (1981)
Verifun:	Flanagan, C., Joshi, R., Saxe, J.B.: An explicating theorem prover for quantified formulas. (2004)
ESC-Java:	Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI. (2002)
Boogie:	DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. MSR-TR (2005)
Spec#	Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS 2004.
E-matching:	Kozen, D.: Complexity of finitely presented algebras. In: STOC. (1977) 164–177
Theories:	Slagle, J.R.: Automatic theorem proving with built-in theories including equality, partial ordering, and sets. J. ACM (1972)
Theories:	Stickel, M.E.: Automated deduction by theory resolution. J. Autom. Reasoning (1985)
Theories;	Baalen, J.V., Roach, S.: Using decision procedures to accelerate domain-specific deductive synthesis systems. (1999)
Theories:	Waldmann, U., Prevosto, V.: SPASS+T. In: ESCoR. (2006) 18–33
SMT:	Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: On a rewriting approach to satisfiability procedures: Extension, combination of theories and an experimental appraisalFroCos. (2005)
SMT:	Leino, K.R.M., Musuvathi, M., Ou, X.: A two-tier technique for supporting quantifiers in a lazily proof-explicating tp TACAS05
SMT:	Barrett, C., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker. In: CAV ’04. LNCS 3114 (2004)
SMT:	Moskal, M., Lopusza´nski, J.: Fast quantifier reasoning with lazy proof explication.
SMT:	Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV’06.
SMT	Ball, T., Lahiri, S.K., Musuvathi, M.: Zap: Automated theorem proving for software analysis. In: LPAR 2005.
Indexing:	A¨ıt-Kaci, H.: Warren’s abstract machine: a tutorial reconstruction. MIT Press, Cambridge, MA, USA (1991)
Indexing:	Voronkov, A.: The anatomy of vampire implementing bottom-up procedures with code trees. JAR (1995)
Indexing:	Riazanov, A., A.Voronkov: Vampire 1.1 (system description). In: IJCAR ’01.
Indexing:	Graf, P., Meyer, C.: Advanced indexing operations on substitution trees. In CADE 1996
Indexing:	Ganzinger, H., Nieuwenhuis, R., Nivela, P.: Context trees. IJCAR’01

.