Z3-Noodler 1.3: Shepherding Decision Procedures for Strings with Model Generation

or How to Select Appropriate Decision Procedures
and Generate Models in Z3-Noodler

David Chocholatý, Vojtěch Havlena, Lukáš Holík, Jan Hranička, Ondřej Lengál, and Juraj Síč

Brno University of Technology, Czech Republic

SMT String constraint solving

- Checking **satisfiability** of formulae with **string variables** and operations

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^* a^+ (b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\text{contains}(u, \text{replace}(z, b, c)) \land \ldots}_{\text{more complex operations}}$$

SMT String constraint solving

- Checking **satisfiability** of formulae with **string variables** and operations

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^* a^+ (b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\mathsf{contains}(u, \mathsf{replace}(z, b, c)) \land \dots}_{\text{more complex operations}}$$

- **Motivation**: **large and complex real-world programs** need security guarantees
  - **analysis** of string manipulating programs (**vulnerabilities** of web applications)

```
let x = y.substring(1, y.length - 1);
let z = y.concat(x);
assert(x === z);
```

$x_0 = \mathit{substr}(y, 1, |y| - 1) \land$
$z_0 = y \cdot x_0 \land$
$x_0 \neq z_0$

  - Amazon web services: cloud **access control policies** [Rungta-CAV'22]

```
action: deactivate,
resource: (a1, a2),
condition: {StringLike, s3:prefix, home*}
```

$A = \text{"}\mathit{deactivate}\text{"} \land$
$(R = \text{"}\mathit{a1}\text{"} \lor R = \text{"}\mathit{a2}\text{"}) \land$
$\mathit{prefix} \in \mathit{home}^*$

  - **verification** of cockpit systems (Boeing), etc.

SMT String constraint solving

- Checking **satisfiability** of formulae with **string variables** and operations

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^* a^+ (b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\mathsf{contains}(u, \mathsf{replace}(z, b, c)) \land \ldots}_{\text{more complex operations}}$$

- **Motivation**: **large and complex real-world programs** need security guarantees
  - **analysis** of string manipulating programs (**vulnerabilities** of web applications)

```
let x = y.substring(1, y.length - 1);
let z = y.concat(x);
assert(x === z);
```

$x_0 = substr(y, 1, |y| - 1) \land$
$z_0 = y \cdot x_0 \land$
$x_0 \neq z_0$

  - Amazon web services: cloud **access control policies** [Rungta-CAV'22]

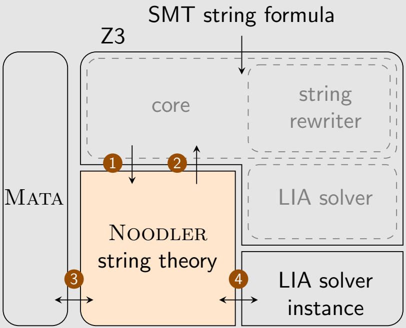
```
action: deactivate,
resource: (a1, a2),
condition: {StringLike, s3:prefix, home*}
```

$A = \text{"}deactivate\text{"} \land$
$(R = \text{"}a1\text{"} \lor R = \text{"}a2\text{"}) \land$
$prefix \in home^*$

  - **verification** of cockpit systems (Boeing), etc.
  - $\rightsquigarrow$ **efficient and expressive** SMT string solvers are **needed**
  - improving **efficiency**, but also **expressiveness** (on the **edge of decidability**)

Z3-Noodler: SMT string solver

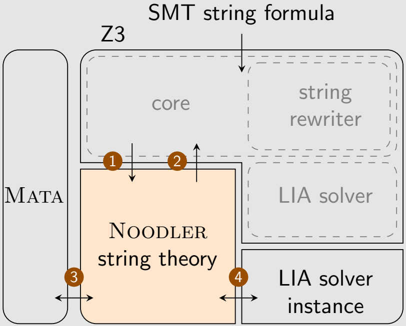
- Based on SMT solver **Z3**
  - formula **parsed** by Z3 and handled by **DPPL(T)**-based framework
  - Z3-Noodler replaces Z3's **string theory solver**
  - modified **string rewriter** (simplifications)
  - uses **Z3's linear arithmetic** (LIA) theory solver



[1] Chocholatý, D. et al. Mata: A Fast and Simple Finite Automata Library. In: TACAS'24

Z3-Noodler: SMT string solver

- Based on SMT solver **Z3**
  - formula **parsed** by Z3 and handled by **DPPL($\mathcal{T}$)**-based framework
  - Z3-Noodler replaces Z3's **string theory solver**
  - modified **string rewriter** (simplifications)
  - uses **Z3's linear arithmetic** (LIA) theory solver
- Uses **Nondeterministic finite automata** (NFAs)
  - Uses **Mata**[1] automata library for efficient handling of finite automata and operations
  - **Explicit alphabets** sufficient



The fastest string solver: **winner** of SMT-COMP'24 string division

[1] Chocholatý, D. et al. Mata: A Fast and Simple Finite Automata Library. In: TACAS'24

Our previous work

$$\underbrace{x = yz \land y \neq u}_{\textit{(dis)equations}} \land \overbrace{x \in (ab)^*a^+(b|c)}^{\textit{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\textit{length constraints}} \land \underbrace{\mathsf{contains}(u, \mathsf{replace}(z, b, c)) \land \ldots}_{\textit{more complex operations}}$$

Our previous work

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^*a^+(b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\text{contains}(u, \text{replace}(z, b, c)) \land \dots}_{\text{more complex operations}}$$

FM'23

- **tight integration** of equations with regular constraints
- works with **languages of variables** encoded as NFAs
- **refining** the languages of variables
- algorithm **stabilization** (**noodlification**)

Our previous work

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^* a^+ (b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\mathsf{contains}(u, \mathsf{replace}(z, b, c)) \land \ldots}_{\text{(some) more complex operations}}$$

FM'23

- **tight integration** of equations with regular constraints
- works with **languages of variables** encoded as NFAs
- **refining** the languages of variables
- algorithm **stabilization** (**noodlification**)

OOPSLA'23

- combines FM'23 with **Align&Split**
- **linear-integer arithmetic** (LIA) encoding
- complete for **chain-free** fragment
- complex operations **reduced to simpler** ones (regular, length constraints, and equations)

Our previous work

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^* a^+ (b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\text{contains}(u, \text{replace}(z, b, c)) \land \ldots}_{\text{(some) more complex operations}}$$

TACAS'24: tool paper for Z3-Noodler v1.0

Our previous work

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^* a^+ (b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\text{contains}(u, \text{replace}(z, b, c)) \land \dots}_{\text{(some) more complex operations}}$$

TACAS'24: tool paper for Z3-Noodler v1.0

SAT'24

- Extends OOPSLA'23 procedure with handling **string-integer conversions**
  - **`to_int`/`from_int`** - string to/from integer:
    `to_int('0324')` $= 324$ `to_int('34a')` $= -1$ `from_int(134)` $=$ `'134'`
  - **`to_code`/`from_code`** - char to/from (Unicode) code point:
    `to_code('0')` $= 48$ `from_code(97)` $=$ `'a'` `to_code('ab')` $= -1$
- encoding conversions into **LIA formulae**
  - $\mathcal{L} \land \varphi_{\text{len}} \land \varphi_{\text{conv}}$ is **satisfiable**, or find a **different** solution

This work

- Earlier work: General fast decision procedure **stabilization**
- **Improve further by combining with specialized decision procedures** for specific (theory) fragments or constraints

This work

- Earlier work: General fast decision procedure **stabilization**
- **Improve further by combining with specialized decision procedures** for specific (theory) fragments or constraints

- An **interface for selecting appropriate** decision procedures
  - **pure regular constraints** (regexes as NFAs)
  - **quadratic equations** (**Nielsen transformation**)
  - **lengths** for **block acyclic constraints**

This work

- Earlier work: General fast decision procedure **stabilization**
- **Improve further by combining with specialized decision procedures** for specific (theory) fragments or constraints

- An **interface for selecting appropriate** decision procedures
  - **pure regular constraints** (regexes as NFAs)
  - **quadratic equations** (**Nielsen transformation**)
  - **lengths** for **block acyclic constraints**

- **Model generation** for all decision procedures
  - for stabilization
  - for the specialized decision procedures

Pure regular constraints: General regular constraints

$$\bigwedge_{1 \leq i \leq n} x \in \mathcal{S}_i \wedge \bigwedge_{1 \leq i \leq m} x \notin \mathcal{R}_i \qquad P = \bigcap_{1 \leq i \leq n} \mathrm{aut}(\mathcal{S}_i) \qquad U = \bigcup_{1 \leq i \leq m} \mathrm{aut}(\mathcal{R}_i)$$

- **Problem**: **Expensive complement computation** (determinization) for negations

Pure regular constraints: General regular constraints

$$\bigwedge_{1\leq i\leq n} x \in \mathcal{S}_i \wedge \bigwedge_{1\leq i\leq m} x \notin \mathcal{R}_i \qquad P = \bigcap_{1\leq i\leq n} \mathrm{aut}(\mathcal{S}_i) \qquad U = \bigcup_{1\leq i\leq m} \mathrm{aut}(\mathcal{R}_i)$$

- **Problem**: **Expensive complement computation** (determinization) for negations
- **Solution**: **Postpone the construction of the complement**, **construct lazily**
- Solved by automata-/**Regex-based** reasoning
- **Expensive emptiness check**: the difference of P and U ($P \cap U^{\complement} = \emptyset$)
- **Instead**: Simple **inclusion checking**: $L(P) \subseteq L(U)$ does *not* hold
  - **antichain-based algorithms**: perform well on real-world problems

Pure regular constraints: Single regular constraint

- Analyze **regexes** ($x \in \mathcal{R}, x \notin \mathcal{R}$) to **extract properties as bool flags**
- **Propagate flags** (e, u, ℓ) through operations:
  - $e \in \mathbb{B}_3$: the regex includes the empty word
  - $u \in \mathbb{B}_3$: the regex is universal
  - $\ell \in \mathbb{N} \cup \{\text{undef}\}$: the minimum length of a word recognized by the regex

$$R_1 : (e_1, u_1, \ell_1) \qquad R_2 : (e_2, u_2, \ell_2) \qquad \texttt{re.++}(R_1, R_2)$$

$$(e_1 \wedge e_2, u, \ell_1 + \ell_2), \ell_1 + \ell_2 > 0 \rightsquigarrow u = \bot, \text{otherwise } u = \text{undef}$$

Pure regular constraints: Single regular constraint

- Analyze **regexes** ($x \in \mathcal{R}, x \notin \mathcal{R}$) to **extract properties as bool flags**
- **Propagate flags** (e, u, ℓ) through operations:
  - $e \in \mathbb{B}_3$: the regex includes the empty word
  - $u \in \mathbb{B}_3$: the regex is universal
  - $\ell \in \mathbb{N} \cup \{\text{undef}\}$: the minimum length of a word recognized by the regex

$$R_1 : (e_1, u_1, \ell_1) \qquad R_2 : (e_2, u_2, \ell_2) \qquad \texttt{re.++}(R_1, R_2)$$

$$(e_1 \wedge e_2, u, \ell_1 + \ell_2), \ell_1 + \ell_2 > 0 \rightsquigarrow u = \bot, \text{otherwise } u = \text{undef}$$

- Completely **avoid the NFA construction by reasoning about the flags**
- undef: only when flags are insufficient $\rightsquigarrow$ construct NFAs

Pure regular constraints: Model generation

- **General regular constraints**:
  - Simple regexes: **direct generation from regexes**
  - Automata construction: **Depth-First-Search** through NFAs in found solutions
    Lazy construction of $P \cap U^{\mathsf{C}}$ (exit on first accepted word)

Pure regular constraints: Model generation

- **General regular constraints**:
  - Simple regexes: **direct generation from regexes**
  - Automata construction: **Depth-First-Search** through NFAs in found solutions

    Lazy construction of $P \cap U^{\mathsf{C}}$ (exit on first accepted word)

- **Single regular constraint**:
  - Positive regex and no complex operations (intersection, complement, or difference): **direct generation from the regex**
  - Otherwise: Automata construction

Quadratic equations: Nielsen transformation

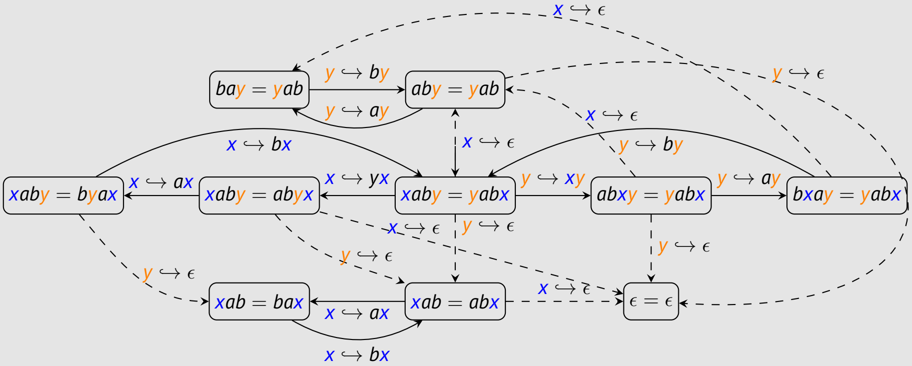
- Quadratic: each variable has at most two occurrences in a conjunction of equations
- Create a **Nielsen graph** (finite for a quadratic system of equations)
    - Node: set of equations, Nielsen tranformation metarules:

$$(x \hookrightarrow \alpha x) : \frac{\mathcal{E}' \uplus \{xu = \alpha v\}}{\mathsf{trim}(\mathcal{E}[x/\alpha x])} \mathcal{E} = \mathcal{E}' \uplus \{xu = \alpha v\} \qquad (x \hookrightarrow \epsilon) : \frac{\mathcal{E}' \uplus \{xu = v\}}{\mathsf{trim}(\mathcal{E}[x/\epsilon])} \mathcal{E} = \mathcal{E}' \uplus \{xu = v\}$$

Quadratic equations: Nielsen transformation

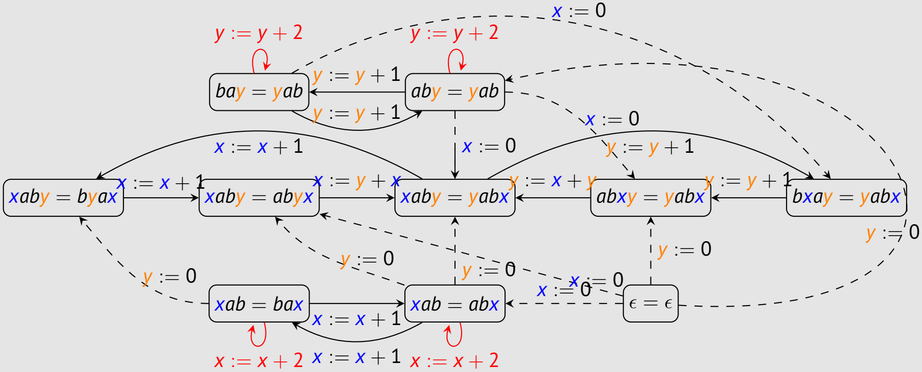
- Quadratic: each variable has at most two occurrences in a conjunction of equations
- Create a **Nielsen graph** (finite for a quadratic system of equations)
  - Node: set of equations, Nielsen tranformation metarules:

$$(x \hookrightarrow \alpha x) : \frac{\mathcal{E}' \uplus \{xu = \alpha v\}}{\mathrm{trim}(\mathcal{E}[x/\alpha x])}\ \mathcal{E} = \mathcal{E}' \uplus \{xu = \alpha v\} \qquad (x \hookrightarrow \epsilon) : \frac{\mathcal{E}' \uplus \{xu = v\}}{\mathrm{trim}(\mathcal{E}[x/\epsilon])}\ \mathcal{E} = \mathcal{E}' \uplus \{xu = v\}$$



Quadratic equations: Counter abstraction system

- Derived from the Nielsen graph [LIN-LMCS'21]
  - heuristic for **handling lengths in Nielsen transformation**
- Infinitely many runs $\rightsquigarrow$ heuristic: selecting runs with self-loops
- **Self-loop saturation**

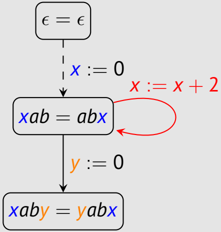


Quadratic equations: Derivation of a LIA formula

$xaby = yabx \land \texttt{len}(x) \geq 50$

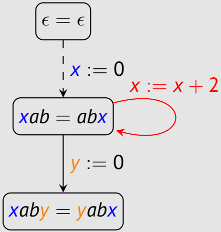
Quadratic equations: Derivation of a LIA formula

$$xaby = yabx \wedge \mathtt{len}(x) \geq 50$$



Quadratic equations: Derivation of a LIA formula

$$xaby = yabx \wedge \mathtt{len}(x) \geq 50$$

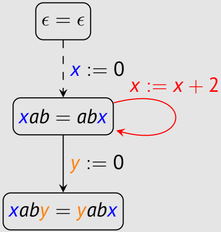


- NFA with counter updates on edges
- **under-approximation**: selected runs into LIA formulae
- Still often enough for **unsat**
- Fresh counter variables for each step

$$\begin{aligned}\varphi(x,y) \Leftrightarrow\ & x_0 = 0 \wedge y_0 = 0 \wedge \\ & x_1 = 0 \wedge y_1 = y_0 \wedge \\ & x_2 = x_1 + 2k \wedge y_2 = y_1 \wedge \\ & y_3 = 0 \wedge x_3 = x_2 \wedge \\ & x = x_3 \wedge y = y_3\end{aligned}$$

Quadratic equations: Derivation of a LIA formula

$$xaby = yabx \land \mathtt{len}(x) \geq 50$$



- NFA with counter updates on edges
- **under-approximation**: selected runs into LIA formulae
- Still often enough for **unsat**
- Fresh counter variables for each step

$$\begin{aligned}\varphi(x,y) \Leftrightarrow\ & x_0 = 0 \land y_0 = 0 \land \\ & x_1 = 0 \land y_1 = y_0 \land \\ & x_2 = x_1 + 2k \land y_2 = y_1 \land \\ & y_3 = 0 \land x_3 = x_2 \land \\ & x = x_3 \land y = y_3\end{aligned}$$

Is $\varphi(\mathtt{len}(x), \mathtt{len}(y)) \land \mathtt{len}(x) \geq 50$ satisfiable?

Quadratic equations: Model generation

- **From counter abstraction system from runs**
- Remember Nielsen rules for the counter updates, and number of times each self-loop was taken
- Model constructed by **following a run with applied rules**

Length-based decision procedure

$$x = abyc \land x = zw \land x = uddc \land y = vad \land y = as$$

- Large systems (many equations, unrestricted variables and literals)
- **Symbolically encode all possible alignments of literals (their positions) into LIA formulae**
- Solving string formula converted into solving LIA formula

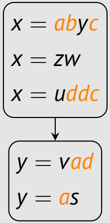
Length-based decision procedure

$$x = abyc \land x = zw \land x = uddc \land y = vad \land y = as$$

- Large systems (many equations, unrestricted variables and literals)
- **Symbolically encode all possible alignments of literals (their positions) into LIA formulae**
- Solving string formula converted into solving LIA formula

$$\bigwedge_{1 \leq i \leq n} x = R_i$$

- ***Equational blocks* of a variable**
- ***Block string constraint***: a conjunction of equational blocks $\rightsquigarrow$ *block graph*
- ***Block-acyclic* string constraint**: acyclic block graph
- Block-acyclic string constraints **extended with length constraints**



Length-based decision procedure: Alignments to LIA formula

$$x = abyc \land x = zw \land x = uddc \land y = vad \land y = as$$

Length-based decision procedure: Alignments to LIA formula

$$x = abyc \land x = zw \land x = uddc \land y = vad \land y = as$$

$x = abyc$

$x = zw$

$x = uddc$

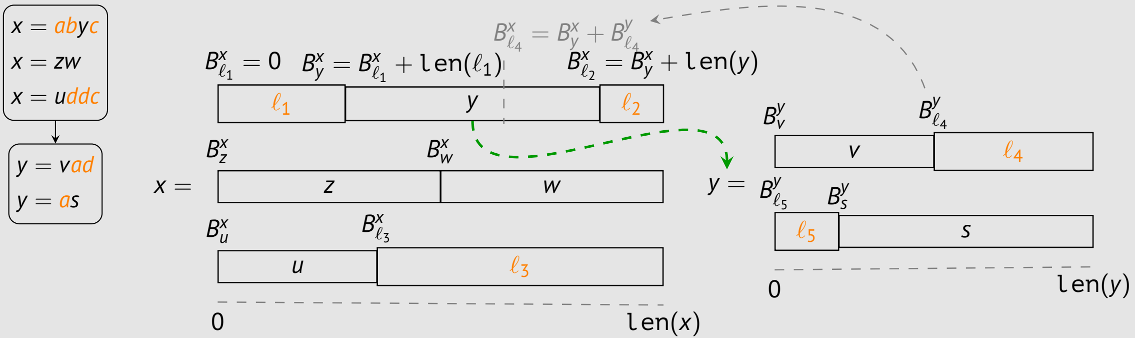
↓

$y = vad$

$y = as$

Length-based decision procedure: Alignments to LIA formula

$$x = abyc \land x = zw \land x = uddc \land y = vad \land y = as$$



Length-based decision procedure: Blocks with cycles

- Extension to **blocks with cycles using under-approximation**
- Shared non-block variables between two blocks with a cycle

$$x = ay\mathbf{z} \ \wedge \ x = ab \ \wedge \ y = b\mathbf{z}$$

Length-based decision procedure: Model generation

- Model for each variable derived from the **positions of the literals**
- **Iteratively filling in an empty skeleton** for each variable with the corresponding string literals

Stabilization: Model generation

- **Recursive construction** of models for variables
- **Language assignments** for variables
- **Restrict** to found lengths

Experimental evaluation

- **SMT-LIB benchmarks**, split into 3 categories:
  - **Regex** (mainly regular and length constraints): **AutomatArk**, **Denghang**, **Redos**, **StringFuzz**, **Sygus-qgen**
  - **Equations** (mostly word equations and length constraints with some small number of more complex constraints): **Kaluza**, **Kepler**, **Norn**, **Omark**, **Slent**, **Slog**, **Webapp**, **Woorpje**
  - **Predicates-small** (complex predicates): **FullStrInt**, **LeetCode**, **PyEx**, **StrSmallRw**, **Transducer+**

- Timeout: 120 s, memory limit: 8 GiB
- **Significantly faster** than other solvers

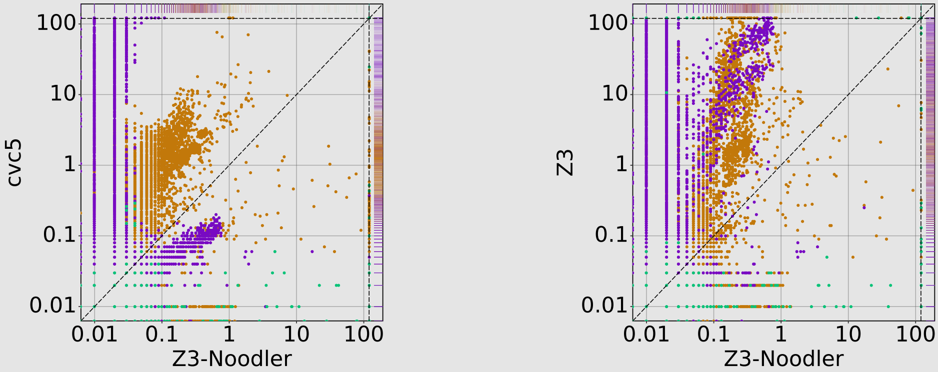
Experimental evaluation: Procedures comparison

	number of calls	Regex proc.		Nielsen transf.		Length-based		Stabillization-based	
		called	solved	called	solved	called	solved	called	solved
Sygus-qgen	747	100%	100%	0%	0%	0%	0%	0%	0%
Denghang	999	0.10%	0.10%	0%	0%	96.10%	96.10%	3.80%	3.80%
AutomatArk	20,062	99.97%	99.97%	0%	0%	0.02%	0.02%	0.01%	0.01%
StringFuzz	9,941	46.45%	46.45%	0%	0%	27.98%	27.96%	25.58%	25.58%
Redos	2,952	70.02%	70.02%	0%	0%	11.21%	11.21%	18.77%	18.77%
Full **Regex**	34,701	79.21%	79.21%	0%	0%	11.75%	11.74%	9.04%	9.04%
LeetCode	874	1.37%	1.37%	0%	0%	59.27%	16.70%	81.92%	81.92%
StrSmallRw	6,327	0%	0%	0%	0%	4.85%	3.75%	96.25%	96.25%
PyEx	26,045	0.10%	0.10%	0%	0%	0.08%	0.08%	99.82%	99.82%
FullStrInt	9,003	0.04%	0.04%	0%	0%	0.26%	0.26%	99.70%	99.70%
Transducer+	0	-		-		-		-	
Full **Predicates-small**	42,249	0.10%	0.10%	0%	0%	2.06%	1.01%	98.89%	98.89%
Norn	918	11.76%	11.76%	0%	0%	6.86%	6.86%	81.37%	81.37%
Slog	1,565	25.37%	25.37%	0%	0%	0.13%	0.13%	74.50%	74.50%
Slent	1,489	0.40%	0.40%	0%	0%	35.19%	30.09%	69.51%	69.51%
Omark	9	0%	0%	11.11%	11.11%	11.11%	0%	88.89%	88.89%
Kepler	579	0%	0%	99.83%	99.83%	0%	0%	0%	0%
Woorpje	478	0.84%	0.84%	43.10%	42.47%	30.96%	27.20%	20.50%	20.50%
Webapp	381	0.52%	0.52%	0%	0%	2.36%	0.26%	99.21%	99.21%
Kaluza	11,222	35.31%	35.31%	0%	0%	63.45%	61.78%	2.91%	2.91%
Full **Equations**	16,641	26.92%	26.92%	4.72%	4.70%	47.27%	45.53%	22.59%	22.59%
All	93,591	34.20%	34.20%	0.84%	0.84%	13.69%	12.91%	52.01%	52.01%

Experimental evaluation: Generating models

	Regex (32,242)		**Equations** (25,727)		**Predicates-small** (45,436)		All (103,405)	
	solved	time	solved	time	solved	time	solved	time
Z3-Noodler	32,232	3,688	25,301	1,147	45,035	6,353	**102,568**	**11,118**
Z3-Noodler$^{\mathcal{M}}$	32,228	4,010	25,299	1,456	45,035	7,321	**102,562**	**12,787**
cvc5	29,290	59,705	25,214	2,529	45,337	11,627	99,841	73,861
cvc5$^{\mathcal{M}}$	29,287	59,892	25,214	2,756	45,337	12,220	99,838	74,868
Z3	29,075	51,379	24,569	3,240	44,101	74,094	97,745	128,712
Z3$^{\mathcal{M}}$	29,064	51,830	24,571	4,013	44,096	74,708	97,731	130,551

Experimental evaluation: Comparison with other solvers



Times are in seconds, axes are logarithmic, timeouts on side dashed lines (120 s)

● **Regex**, ● **Equations**, and ● **Predicates-small**.

Conclusion

- **Combination of decision procedures**
- **Specialized decision procedures**
  (**regular and length constraints**, **quadratic equations**)
- **Model generation**
- **Z3-Noodler**:
  `https://github.com/VeriFIT/z3-noodler`
- **The fastest string solver**

Conclusion

- **Combination of decision procedures**
- **Specialized decision procedures** (**regular and length constraints**, **quadratic equations**)
- **Model generation**
- **Z3-Noodler**: `https://github.com/VeriFIT/z3-noodler`
- **The fastest string solver**

- **Future work**:
  - using **transducers** for **`replace_all`** operations (*nearly done*)
  - better handling of **negated `contains`**
  - application of Z3-Noodler on the **analysis of the security of web applications**

Noodlification (FM'23) on an example

$$xyx = zu \land ww = xa \land u \in (baba)^*a \land z \in a(ba)^* \land x \in \Sigma^* \land y \in \Sigma^* \land w \in \Sigma^*$$

- $\Sigma = \{a, b\}$

Noodlification (FM'23) on an example

$$xyx = zu \land ww = xa \land u \in (baba)^*a \land z \in a(ba)^* \land x \in \Sigma^* \land y \in \Sigma^* \land w \in \Sigma^*$$

- $\Sigma = \{a, b\}$
- Regular constraints are **collected** in a **language assignment** represented by **automata**

$$Lang = \{u \mapsto (baba)^*a, z \mapsto a(ba)^*, x \mapsto \Sigma^*, y \mapsto \Sigma^*, w \mapsto \Sigma^*\}$$

Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa$	$u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad w \mapsto \Sigma^*$

- $\Sigma = \{a, b\}$
- Regular constraints are **collected** in a **language assignment** represented by **automata**

$$Lang = \{u \mapsto (baba)^*a, z \mapsto a(ba)^*, x \mapsto \Sigma^*, y \mapsto \Sigma^*, w \mapsto \Sigma^*\}$$

Noodlification (FM'23) on an example

$xyx = zu$ $\quad ww = xa$	$u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad w \mapsto \Sigma^*$

- $\Sigma = \{a, b\}$
- Regular constraints are **collected** in a **language assignment** represented by **automata**

$$Lang = \{u \mapsto (baba)^*a, z \mapsto a(ba)^*, x \mapsto \Sigma^*, y \mapsto \Sigma^*, w \mapsto \Sigma^*\}$$

- Use equations to **refine *Lang***, starting with $\mathbf{xyx = zu}$
- For any solution (assignment v) string $s = \nu(x) \cdot \nu(y) \cdot \nu(x) = \nu(z) \cdot \nu(u)$ satisfies:

$$s \in \overbrace{\Sigma^*}^{x}\ \overbrace{\Sigma^*}^{y}\ \overbrace{\Sigma^*}^{x} \overset{=}{\cap} \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u}$$

- Use right side to **refine** languages of variables x, y on the left side by **noodlification**

Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa \quad u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad w \mapsto \Sigma^*$

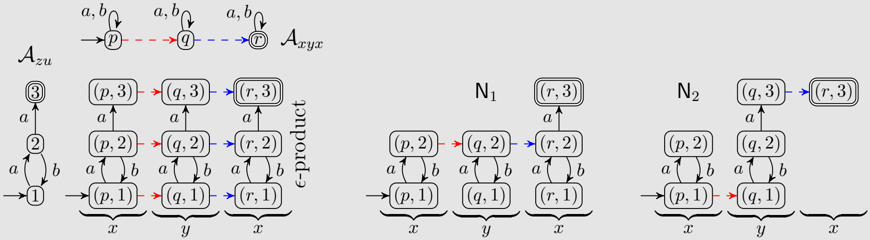
- Use right side to **refine** languages of variables x, y on the left side by **noodlification**

Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa \quad u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad w \mapsto \Sigma^*$

- Use right side to **refine** languages of variables x, y on the left side by **noodlification**
- Leads to two noodles:

$N_1 : \overbrace{\Sigma^*}^{x} \overbrace{\Sigma^*}^{y} \overbrace{\Sigma^*}^{x} \overset{=}{\cap} \overbrace{a(ba)^*}^{z} \overbrace{(baba)^*a}^{u} \qquad N_2 : \overbrace{\Sigma^*}^{x} \overbrace{\Sigma^*}^{y} \overbrace{\Sigma^*}^{x} \overset{=}{\cap} \overbrace{a(ba)^*}^{z} \overbrace{(baba)^*a}^{u}$

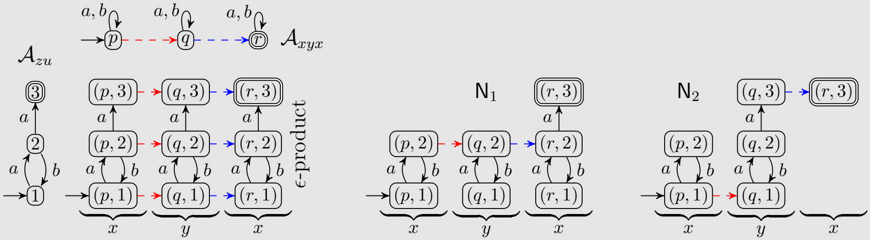


Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa \quad u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad w \mapsto \Sigma^*$

- Use right side to **refine** languages of variables x, y on the left side by **noodlification**
- Leads to two noodles:

$$N_1 : \overbrace{a}^{x}\ \overbrace{(ba)^*}^{y}\ \overbrace{a}^{x} \overset{=}{\cap} \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u} \qquad N_2 : \overbrace{\Sigma^*}^{x}\ \overbrace{\Sigma^*}^{y}\ \overbrace{\Sigma^*}^{x} \overset{=}{\cap} \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u}$$

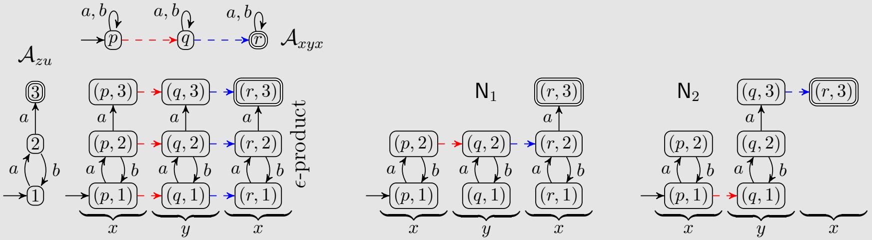


Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa$ | $u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad w \mapsto \Sigma^*$

- Use right side to **refine** languages of variables x, y on the left side by **noodlification**
- Leads to two noodles:

$$N_1 : \overbrace{a}^{x}\ \overbrace{(ba)^*}^{y}\ \overbrace{a}^{x} \ \overset{=}{\cap}\ \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u} \qquad N_2 : \overbrace{\epsilon}^{x}\ \overbrace{a(ba)^*a}^{y}\ \overbrace{\epsilon}^{x} \ \overset{=}{\cap}\ \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u}$$

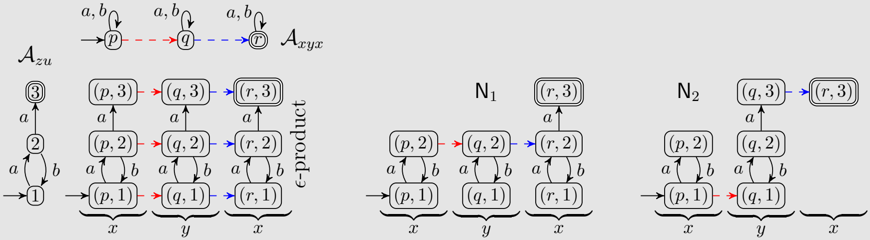


Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa \quad u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto a \quad y \mapsto (ba)^* \quad w \mapsto \Sigma^*$

- Use right side to **refine** languages of variables x, y on the left side by **noodlification**
- Leads to two noodles:

$$N_1 : \overbrace{a}^{x}\ \overbrace{(ba)^*}^{y}\ \overbrace{a}^{x} \cap \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u} \qquad N_2 : \overbrace{\epsilon}^{x}\ \overbrace{a(ba)^*a}^{y}\ \overbrace{\epsilon}^{x} \cap \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u}$$



Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa$	$u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto a \quad y \mapsto (ba)^* \quad w \mapsto \Sigma^*$

- Refine further with $ww = xa$:

$$\overbrace{\Sigma^*}^{w}\overbrace{\Sigma^*}^{w} \overset{=}{\cap} \overbrace{a}^{x}\overset{a}{a}.$$

Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa$	$u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto a \quad y \mapsto (ba)^* \quad w \mapsto a$

- Refine further with $ww = xa$:

$$\overbrace{a}^{w}\ \overbrace{a}^{w} \overset{=}{\cap} \overbrace{a}^{x}\ \overset{a}{a}.$$

Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa$ | $u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto a \quad y \mapsto (ba)^* \quad w \mapsto a$

- Refine further with $ww = xa$:

$$\overbrace{a}^{w}\ \overbrace{a}^{w} \overset{=}{\cap} \overbrace{a}^{x}\ \overset{a}{a}.$$

- Languages in equations now **match**:

$$\overbrace{a}^{x}\ \overbrace{(ba)^*}^{y}\ \overbrace{a}^{x} \overset{=}{=} \overbrace{a(ba)^*}^{z}\ \overbrace{(baba)^*a}^{u} \quad \text{and} \quad \overbrace{a}^{w}\ \overbrace{a}^{w} \overset{=}{=} \overbrace{a}^{x}\ \overset{a}{a}.$$

Noodlification (FM'23) on an example

$xyx = zu \quad ww = xa \mid u \mapsto (baba)^*a \quad z \mapsto a(ba)^* \quad x \mapsto a \quad y \mapsto (ba)^* \quad w \mapsto a$

- Refine further with ***ww* = *xa***:

$$\overbrace{a}^{w}\overbrace{a}^{w} \overset{=}{\cap} \overbrace{a}^{x}\overset{a}{a}.$$

- Languages in equations now **match**:

$$\overbrace{a}^{x}\overbrace{(ba)^*}^{y}\overbrace{a}^{x} \overset{=}{=} \overbrace{a(ba)^*}^{z}\overbrace{(baba)^*a}^{u} \quad \text{and} \quad \overbrace{a}^{w}\overbrace{a}^{w} \overset{=}{=} \overbrace{a}^{x}\overset{a}{a}.$$

- *Lang* is a **stable solution** (we prove this is enough to decide it is SAT)

OOPSLA'23

$$\underbrace{x = yz \land y \neq u}_{\text{(dis)equations}} \land \overbrace{x \in (ab)^*a^+(b|c)}^{\text{regular constraints}} \land \overbrace{|x| = 2|u| + 1}^{\text{length constraints}} \land \underbrace{\text{contains}(u, \text{replace}(z, b, c)) \land \dots}_{\text{(some) more complex operations}}$$

- FM'23 can handle **equations** and **regular constraints** (at least **chain-free fragment**)
- How to handle more **complex operations** and **disequations**?
  - $\rightsquigarrow$ reduced (at least partially) to simpler constraints
- How to handle **lengths**?
  - create linear-integer arithmetic (LIA) formula **encoding possible lengths of words** in each language in *Lang*
  - stable solution *Lang* does not keep **dependencies** between lengths of vars
    - $\rightsquigarrow$ we use noodlification combined with **Align&Split** algorithm [Abdulla-CAV'14]

OOPSLA'23 on example: $xx = w \wedge z = xy \wedge w \in a^+ \wedge |z| = 2|w| - |x|$

$xx = w \quad z = xy$	$w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	

↓ Align&Split $(xx = w)$

$x = v_1 \quad w = v_1v_1 \quad z = xy$	$v_1 \mapsto a^+ \quad w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	

↓ Subst $(x = v_1)$

$w = v_1v_1 \quad z = v_1y$	$v_1 \mapsto a^+ \quad w \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	$x \mapsto v_1$

↓ Subst $(w = v_1v_1)$

$z = v_1y$	$v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	$x \mapsto v_1 \quad w \mapsto v_1v_1$

↓ Subst $(z = v_1y)$

	$v_1 \mapsto a^+ \quad y \mapsto \Sigma^*$	$x \mapsto v_1 \quad w \mapsto v_1v_1 \quad z \mapsto v_1y$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

$\mathbf{xx = w} \quad z = xy \;|\; \mathbf{w \mapsto a^+} \quad \mathbf{x \mapsto \Sigma^*} \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|$

↓ **Align&Split** $(\mathbf{xx = w})$

$\mathbf{x = v_1} \quad \mathbf{w = v_1 v_1} \quad z = xy \;|\; \mathbf{v_1 \mapsto a^+} \quad w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|$

↓ Subst $(x = v_1)$

$w = v_1 v_1 \quad z = v_1 y \;|\; v_1 \mapsto a^+ \quad w \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\; x \mapsto v_1$

↓ Subst $(w = v_1 v_1)$

$z = v_1 y \;|\; v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\; x \mapsto v_1 \quad w \mapsto v_1 v_1$

↓ Subst $(z = v_1 y)$

$\;|\; v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \;|\; x \mapsto v_1 \quad w \mapsto v_1 v_1 \quad z \mapsto v_1 y$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

$xx = w \quad z = xy \;|\; w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|$

↓ Align&Split $(xx = w)$

$x = v_1 \quad w = v_1v_1 \quad z = xy \;|\; v_1 \mapsto a^+ \quad w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|$

↓ **Subst $(x = v_1)$**

$w = v_1v_1 \quad z = v_1y \;|\; v_1 \mapsto a^+ \quad w \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\; x \mapsto v_1$

↓ Subst $(w = v_1v_1)$

$z = v_1y \;|\; v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\; x \mapsto v_1 \quad w \mapsto v_1v_1$

↓ Subst $(z = v_1y)$

$\;|\; v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \;|\; x \mapsto v_1 \quad w \mapsto v_1v_1 \quad z \mapsto v_1y$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

$xx = w \quad z = xy \;|\; w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\;$

↓ Align&Split $(xx = w)$

$x = v_1 \quad w = v_1v_1 \quad z = xy \;|\; v_1 \mapsto a^+ \quad w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\;$

↓ Subst $(x = v_1)$

$\mathbf{w = v_1v_1} \quad z = v_1y \;|\; \mathbf{v_1 \mapsto a^+} \quad \mathbf{w \mapsto a^+} \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\; x \mapsto v_1$

↓ **Subst $(w = v_1v_1)$**

$z = v_1y \;|\; v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \;|\; x \mapsto v_1 \quad \mathbf{w \mapsto v_1v_1}$

↓ Subst $(z = v_1y)$

$\;|\; v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \;|\; x \mapsto v_1 \quad w \mapsto v_1v_1 \quad z \mapsto v_1y$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

$xx = w \quad z = xy$	$w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	

↓ Align&Split $(xx = w)$

$x = v_1 \quad w = v_1v_1 \quad z = xy$	$v_1 \mapsto a^+ \quad w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	

↓ Subst $(x = v_1)$

$w = v_1v_1 \quad z = v_1y$	$v_1 \mapsto a^+ \quad w \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	$x \mapsto v_1$

↓ Subst $(w = v_1v_1)$

$z = v_1y$	$v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^*$	$x \mapsto v_1 \quad w \mapsto v_1v_1$

↓ **Subst $(z = v_1y)$**

	$v_1 \mapsto a^+ \quad y \mapsto \Sigma^*$	$x \mapsto v_1 \quad w \mapsto v_1v_1 \quad z \mapsto v_1y$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

$xx = w \quad z = xy \quad \| \quad w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \quad \| \quad$

↓ Align&Split $(xx = w)$

$x = v_1 \quad w = v_1v_1 \quad z = xy \quad \| \quad v_1 \mapsto a^+ \quad w \mapsto a^+ \quad x \mapsto \Sigma^* \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \quad \| \quad$

↓ Subst $(x = v_1)$

$w = v_1v_1 \quad z = v_1y \quad \| \quad v_1 \mapsto a^+ \quad w \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \quad \| \quad x \mapsto v_1$

↓ Subst $(w = v_1v_1)$

$z = v_1y \quad \| \quad v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \quad z \mapsto \Sigma^* \quad \| \quad x \mapsto v_1 \quad w \mapsto v_1v_1$

↓ Subst $(z = v_1y)$

$\quad \| \quad v_1 \mapsto a^+ \quad y \mapsto \Sigma^* \quad \| \quad x \mapsto v_1 \quad w \mapsto v_1v_1 \quad z \mapsto v_1y$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

	$v_1 \mapsto a^+ \quad y \mapsto \Sigma^*$	$x \mapsto v_1 \quad w \mapsto v_1v_1 \quad z \mapsto v_1y$

- stable solution $(Lang, \sigma)$:
  - **language assignment** $Lang$: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1, w \mapsto v_1v_1, z \mapsto v_1y$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

	$v_1 \mapsto a^+$	$y \mapsto \Sigma^*$	$x \mapsto v_1$	$w \mapsto v_1v_1$	$z \mapsto v_1y$

- stable solution **$(Lang, \sigma)$**:
  - **language assignment** *Lang*: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1, w \mapsto v_1v_1, z \mapsto v_1y$
- **LIA formula** encoding **possible lengths** of variables:

$$\varphi_{\text{len}} \overset{\text{def.}}{\Leftrightarrow} \quad \land \quad \land \quad \land \quad \land$$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

	$v_1 \mapsto a^+$ $y \mapsto \Sigma^*$	$x \mapsto v_1$ $w \mapsto v_1 v_1$ $z \mapsto v_1 y$

- stable solution $(Lang, \sigma)$:
  - **language assignment** $Lang$: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1, w \mapsto v_1 v_1, z \mapsto v_1 y$
- **LIA formula** encoding **possible lengths** of variables:

$$\varphi_{\text{len}} \overset{\text{def.}}{\Leftrightarrow} |v_1| \geq 1 \land \qquad \land \qquad \land \qquad \land$$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

	$v_1 \mapsto a^+$ $y \mapsto \Sigma^*$	$x \mapsto v_1$ $w \mapsto v_1v_1$ $z \mapsto v_1y$

- stable solution **$(Lang, \sigma)$**:
  - **language assignment** *Lang*: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1, w \mapsto v_1v_1, z \mapsto v_1y$
- **LIA formula** encoding **possible lengths** of variables:

$$\varphi_{\text{len}} \overset{\text{def.}}{\Leftrightarrow} |v_1| \geq 1 \land |y| \geq 0 \land \qquad \land \qquad \land$$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

	$v_1 \mapsto a^+ \quad y \mapsto \Sigma^*$	$x \mapsto v_1 \quad w \mapsto v_1 v_1 \quad z \mapsto v_1 y$

- stable solution **$(Lang, \sigma)$**:
  - **language assignment** *Lang*: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1$, $w \mapsto v_1 v_1, z \mapsto v_1 y$
- **LIA formula** encoding **possible lengths** of variables:

$$\varphi_{\text{len}} \overset{\text{def.}}{\Leftrightarrow} |v_1| \geq 1 \land |y| \geq 0 \land |x| = |v_1| \land \qquad \land$$

OOPSLA'23 on example: $xx = w \wedge z = xy \wedge w \in a^+ \wedge |z| = 2|w| - |x|$

	$v_1 \mapsto a^+$	$y \mapsto \Sigma^*$	$x \mapsto v_1$	$w \mapsto v_1 v_1$	$z \mapsto v_1 y$

- stable solution **(*Lang*, σ)**:
  - **language assignment** *Lang*: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1, \boldsymbol{w \mapsto v_1 v_1}, z \mapsto v_1 y$
- **LIA formula** encoding **possible lengths** of variables:

$$\varphi_{\text{len}} \overset{\text{def.}}{\Leftrightarrow} |v_1| \geq 1 \wedge |y| \geq 0 \wedge |x| = |v_1| \wedge |w| = |v_1| + |v_1| \wedge$$

OOPSLA'23 on example: $xx = w \land z = xy \land w \in a^+ \land |z| = 2|w| - |x|$

	$v_1 \mapsto a^+$	$y \mapsto \Sigma^*$	$x \mapsto v_1$	$w \mapsto v_1 v_1$	$z \mapsto v_1 y$

- stable solution **(*Lang*, σ)**:
  - **language assignment** *Lang*: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1, w \mapsto v_1 v_1, \boldsymbol{z \mapsto v_1 y}$
- **LIA formula** encoding **possible lengths** of variables:

$$\varphi_{\text{len}} \overset{\text{def.}}{\Leftrightarrow} |v_1| \geq 1 \land |y| \geq 0 \land |x| = |v_1| \land |w| = |v_1| + |v_1| \land |z| = |v_1| + |y|$$

OOPSLA'23 on example: $xx = w \wedge z = xy \wedge w \in a^+ \wedge |z| = 2|w| - |x|$

	$v_1 \mapsto a^+$ $\quad y \mapsto \Sigma^*$	$x \mapsto v_1$ $\quad w \mapsto v_1v_1$ $\quad z \mapsto v_1y$

- stable solution $(Lang, \sigma)$:
  - **language assignment** $Lang$: $v_1 \mapsto a^+, y \mapsto \Sigma^*$
  - **substitution map** σ: $x \mapsto v_1, w \mapsto v_1v_1, z \mapsto v_1y$
- **LIA formula** encoding **possible lengths** of variables:

$$\varphi_{\mathrm{len}} \overset{\mathrm{def.}}{\Leftrightarrow} |v_1| \geq 1 \wedge |y| \geq 0 \wedge |x| = |v_1| \wedge |w| = |v_1| + |v_1| \wedge |z| = |v_1| + |y|$$

- ask LIA solver if $|z| = 2|w| - |x| \wedge \varphi_{\mathrm{len}}$ is satisfiable
  - it is, we have model $|v_1| = |x| = 1, |w| = |y| = 2, |z| = 3$
  - we can choose any word from $Lang(v_1)$ and $Lang(y)$ with correct lengths:
    $v_1 = a$ and $y = bc$
  - models for $x, w,$ and z are computed using the substitution map σ:
    $x = v_1 = a, w = v_1v_1 = aa,$ and $z = v_1y = abc$

How to combine OOPSLA'23 with conversions?

- What we **have**:
  - stable **solution** $(Lang, \sigma)$
  - the **LIA part** of the initial formula $\mathcal{L}$
  - formula φ_{len} **encoding** possible **lengths** of variables
  - set of **conversion constraints** $\mathcal{C} = \{k = \texttt{to\_int}(x), y = \texttt{from\_code}(l), \ldots\}$
- How about **encoding** conversions into **LIA formula** too?
  - each conversion constraint $c \in \mathcal{C}$ encoded into **LIA formula** φ_c
  - $\varphi_{\text{conv}} \overset{\text{def.}}{\Leftrightarrow} \bigwedge_{c \in \mathcal{C}} \varphi_c$
  - if $\mathcal{L} \wedge \varphi_{\text{len}} \wedge \varphi_{\text{conv}}$ is **satisfiable**, we have a **solution**
  - otherwise find **different** stable solution (if possible)

Handling $k = \texttt{to\_int}(x)$

- Semantics:
  - for a valid x (it contains only digits), k is the number represented by x
  - for an invalid x (it contains some non-digit), $k = -1$
- For stable solution $(Lang, \sigma)$ we have two distinct cases:
  - x is mapped to some language L_x in language assignment $Lang$
  - x is substituted by $x_1 \cdots x_n$ in substitution map σ

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in Lang$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying `to_int` on some word from L_x
- Generally possible only with **non-linear** arithmetic

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in Lang$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying `to_int` on some word from L_x
- Generally possible only with **non-linear** arithmetic
  $\rightsquigarrow$ stronger restriction: L_x is **finite** (can be mitigated with **underapproximations**)

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in \mathit{Lang}$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying `to_int` on some word from L_x
- Generally possible only with **non-linear** arithmetic
  $\rightsquigarrow$ stronger restriction: L_x is **finite** (can be mitigated with **underapproximations**)
- We can iterate over all words:

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} \bigvee_{w \in L_x} \left(\texttt{to\_int}(x) = \texttt{to\_int}(w)\right)$$

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in \mathit{Lang}$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying to_int on some word from L_x
- Generally possible only with **non-linear** arithmetic
  $\rightsquigarrow$ stronger restriction: L_x is **finite** (can be mitigated with **underapproximations**)
- We can iterate over all words:

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} \bigvee_{w \in L_x} \left(\texttt{to\_int}(x) = \texttt{to\_int}(w)\right)$$

- Problems:

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in Lang$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying to_int on some word from L_x
- Generally possible only with **non-linear** arithmetic
  $\rightsquigarrow$ stronger restriction: L_x is **finite** (can be mitigated with **underapproximations**)
- We can iterate over all words:

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} \bigvee_{w \in L_x} (\texttt{to\_int}(x) = \texttt{to\_int}(w))$$

- Problems:
  1. the **correspondence** between the length of x and the value of $\texttt{to\_int}(x)$

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in Lang$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying `to_int` on some word from L_x
- Generally possible only with **non-linear** arithmetic
  - $\rightsquigarrow$ stronger restriction: L_x is **finite** (can be mitigated with **underapproximations**)
- We can iterate over all words:

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} \bigvee_{w \in L_x} \left(\texttt{to\_int}(x) = \texttt{to\_int}(w) \land |x| = |w|\right)$$

- Problems:
  1. the **correspondence** between the length of x and the value of $\texttt{to\_int}(x)$
     - $\rightsquigarrow$ **relate** words with the corresponding length

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in Lang$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying to_int on some word from L_x
- Generally possible only with **non-linear** arithmetic
  $\rightsquigarrow$ stronger restriction: L_x is **finite** (can be mitigated with **underapproximations**)
- We can iterate over all words:

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} \bigvee_{w \in L_x} \left(\texttt{to\_int}(x) = \texttt{to\_int}(w) \land |x| = |w|\right)$$

- Problems:
  1. the **correspondence** between the length of x and the value of $\texttt{to\_int}(x)$
     $\rightsquigarrow$ **relate** words with the corresponding length
  2. can easily **blow-up**

Handling $k = \texttt{to\_int}(x)$ when x is in the language assignment

- Assume that $x \mapsto L_x \in Lang$
- LIA formula $\varphi_{k=\texttt{to\_int}(x)}$ should encode that k is the result of applying `to_int` on some word from L_x
- Generally possible only with **non-linear** arithmetic
  $\rightsquigarrow$ stronger restriction: L_x is **finite** (can be mitigated with **underapproximations**)
- We can iterate over all words:

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} \bigvee_{w \in L_x} \left(\texttt{to\_int}(x) = \texttt{to\_int}(w) \land |x| = |w|\right)$$

- Problems:
  1. the **correspondence** between the length of x and the value of $\texttt{to\_int}(x)$
     $\rightsquigarrow$ **relate** words with the corresponding length
  2. can easily **blow-up**
     $\rightsquigarrow$ encode **intervals** of words instead of single words

Intervals on an example

- Let $L_x = [0\text{-}7] \cup [2\text{-}5][0\text{-}9] \cup [3\text{-}6][0\text{-}9][0\text{-}9]$
- We create the following formula:

$$\varphi_{k=\mathtt{to\_int}(x)} \stackrel{\text{def.}}{\Leftrightarrow}$$

Intervals on an example

- Let $L_x =$ **[0-7]** $\cup$ [2-5][0-9] $\cup$ [3-6][0-9][0-9]
- We create the following formula:

$$\varphi_{k=\mathtt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} (0 \leq \mathtt{to\_int}(x) \leq 7 \wedge |x| = 1)$$

Intervals on an example

- Let $L_x = [0\text{-}7] \cup \mathbf{[2\text{-}5][0\text{-}9]} \cup [3\text{-}6][0\text{-}9][0\text{-}9]$
- We create the following formula:

$$\varphi_{k=\mathtt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} (0 \leq \mathtt{to\_int}(x) \leq 7 \land |x| = 1) \\ \lor (20 \leq \mathtt{to\_int}(x) \leq 59 \land |x| = 2)$$

Intervals on an example

- Let $L_x = [0\text{-}7] \cup [2\text{-}5][0\text{-}9] \cup$ **[3-6][0-9][0-9]**
- We create the following formula:

$$\begin{aligned}\varphi_{k=\mathtt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} & (0 \leq \mathtt{to\_int}(x) \leq 7 \wedge |x| = 1) \\ & \vee (20 \leq \mathtt{to\_int}(x) \leq 59 \wedge |x| = 2) \\ & \vee (300 \leq \mathtt{to\_int}(x) \leq 699 \wedge |x| = 3)\end{aligned}$$

Intervals on an example

- Let $L_x = [0\text{-}7] \cup [2\text{-}5][0\text{-}9] \cup [3\text{-}6][0\text{-}9][0\text{-}9]$
- We create the following formula:

$$\begin{aligned}\varphi_{k=\mathtt{to\_int}(x)} \overset{\text{def.}}{\Leftrightarrow} & (0 \leq \mathtt{to\_int}(x) \leq 7 \wedge |x| = 1) \\ & \vee (20 \leq \mathtt{to\_int}(x) \leq 59 \wedge |x| = 2) \\ & \vee (300 \leq \mathtt{to\_int}(x) \leq 699 \wedge |x| = 3)\end{aligned}$$

- Easily implementable on automata level
- Handling invalid cases makes it a bit more complicated

Handling $k = \texttt{to\_int}(x)$ when x is in the substitution map

- Assume that $x \mapsto x_1 \cdots x_n \in \sigma$
- In stable solution, each x_i is mapped to some L_{x_i} in the language assignment *Lang*
- We can create LIA formulas encoding each $\texttt{to\_int}(x_i)$ using the interval method
- For each $(l_1, \ldots, l_n)$ with l_i some possible length of x_i we create

$$\texttt{to\_int}(x) = \sum_{1 \leq i \leq n} \left(\texttt{to\_int}(x_i) \cdot 10^{\ell_{i+1} + \cdots + \ell_n} \right) \wedge \bigwedge_{1 \leq i \leq n} \left(|x_i| = \ell_i \right)$$

- $\varphi_{k=\texttt{to\_int}(x)}$ is defined as a disjunction of these equations
- Again, invalid cases make it more complicated

Handling $k = \texttt{to\_code}(x)$

- Semantics:
  - for a valid x (a char), k is the code points of x
  - for an invalid x (not a char), $k = -1$
- **Valid** part is always **finite**
  - **no problem** with **infinite** languages
  - we can iterate over all **characters**:

$$\varphi_{k=\texttt{to\_code}(x)} \overset{\text{def.}}{\Leftrightarrow} \bigvee_{a \in L_x \cap \Sigma} \texttt{to\_code}(x) = \texttt{to\_code}(a) \land |x| = 1$$

- Still problem with a **blow-up** (Σ is large)
  - set Σ_e of explicitly **used** symbols in formula is usually **small**
  - introduce a **special symbol** δ representing all **unused symbols**
  - work with a **much smaller** alphabet $\Sigma = \Sigma_e \cup \{\delta\}$
  - **special handling** of δ
- Needs to also encode the **correspondence** between $\texttt{to\_code}(x)$ and $\texttt{to\_int}(x)$

Handling `from_int/from_code`

- Very **similar** to `to_int/from_code`
- Instead of constraining the result, we want to constrain the argument
- We can use nearly the **same encoding**
- Slight **difference** in handling **invalid** cases

Handling word disequations trough to_code

- In OOPSLA'23 we showed how to handle **arbitrary disequation** $s \neq t$:

$$\varphi_{s \neq t} \overset{\text{def.}}{\Leftrightarrow} |s| \neq |t| \vee \left(s = x_1 a_1 y_1 \wedge t = x_2 a_2 y_2 \wedge |x_1| = |x_2| \wedge a_1 \in \Sigma \wedge a_2 \in \Sigma \wedge \overbrace{a_1 \neq a_2}^{\text{dist}(a_1, a_2)} \right)$$

- Convoluted LIA formula $\text{dist}(a_1, a_2)$ computed after getting stable solution
- Important: this encoding has **no impact** on chain-free fragment
- Problem: encoding of $\text{dist}(a_1, a_2)$ is **incompatible** with conversions
- Solution:

$$\text{dist}(a_1, a_2) \overset{\text{def.}}{\Leftrightarrow} \texttt{to\_code}(a_1) \neq \texttt{to\_code}(a_2)$$

- Still **no impact** on chain-free fragment