ICGT 2018: CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers

Barbara König Maxime Nederkorn **Dennis Nolte**

University of Duisburg-Essen

25.06.2018

Motivation

Aim

Analyse the behaviour and verify the correctness of dynamically evolving systems.

Motivation

Aim

Analyse the behaviour and verify the correctness of dynamically evolving systems.

Graph transformation systems are well suited to model:

- Concurrent systems
- Infinite state spaces
- Dynamic creation and deletion of objects
- Variable topologies
- ...

Motivation

Aim

Analyse the behaviour and verify the correctness of dynamically evolving systems.

Graph transformation systems are well suited to model:

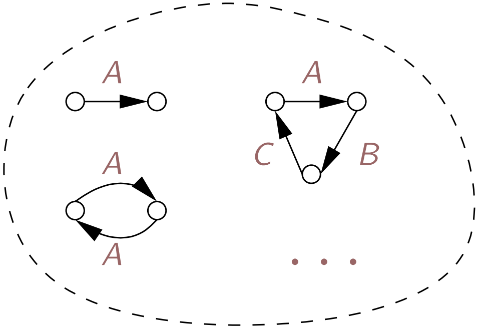
- Concurrent systems
- Infinite state spaces
- Dynamic creation and deletion of objects
- Variable topologies
- ...

Trade-off: More complex modeling language $\rightsquigarrow$ harder analysis.

Overview

In this Talk

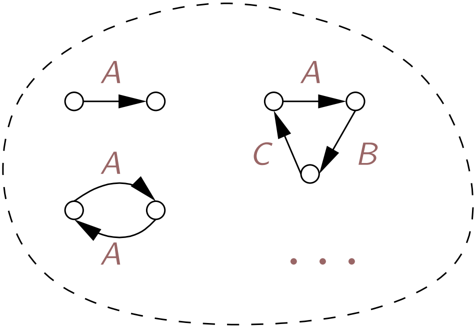
Specify (possibly infinite) sets of graphs by finite graphs and compute their corresponding minimal representation.



Overview

In this Talk

Specify (possibly infinite) sets of graphs by finite graphs and compute their corresponding minimal representation.



Solving a *subtask* from our predecessor paper (ICGT 2017)

Contents

Part I

Background and Preliminaries

The Basic Framework of Type Graphs

We started by studying type graphs as a specification language.

Type Graph Language

Given a graph T, the language of T consists of all graphs that can be mapped homomorphically into T:

$$\mathcal{L}(T) = \{G \mid \text{there exists a morphism } \varphi\colon G \to T\}$$

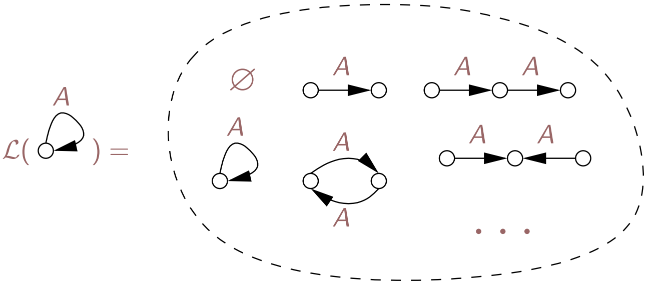
The Basic Framework of Type Graphs

We started by studying type graphs as a specification language.

Type Graph Language

Given a graph T, the language of T consists of all graphs that can be mapped homomorphically into T:

$$\mathcal{L}(T) = \{G \mid \text{there exists a morphism } \varphi\colon G \to T\}$$



The Basic Framework of Type Graphs

We started by studying type graphs as a specification language.

Type Graph Language

Given a graph T, the language of T consists of all graphs that can be mapped homomorphically into T:

$$\mathcal{L}(T) = \{G \mid \text{there exists a morphism } \varphi\colon G \to T\}$$

Why study Type Graphs?

- They are simple.
- Other formalisms are based on type graphs (e.g., abstract graphs that use type graphs with additional annotations)
- Refine/Extend this basic formalism and analyse the properties.

The Basic Framework of Type Graphs

We started by studying type graphs as a specification language.

Type Graph Language

Given a graph T, the language of T consists of all graphs that can be mapped homomorphically into T:

$$\mathcal{L}(T) = \{G \mid \text{there exists a morphism } \varphi\colon G \to T\}$$

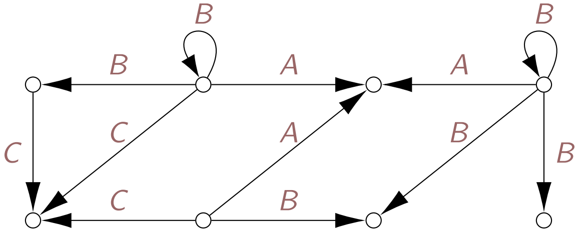
Why study Type Graphs?

- They are simple.
- Other formalisms are based on type graphs (e.g., abstract graphs that use type graphs with additional annotations)
- Refine/Extend this basic formalism and analyse the properties.

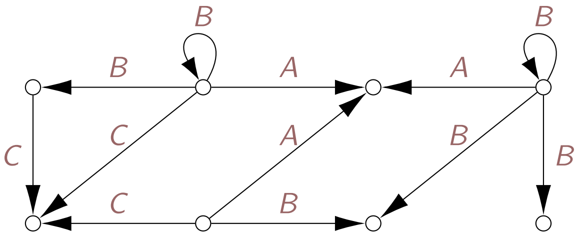
Today's aim:

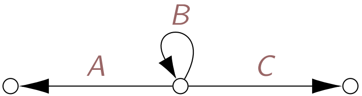
Efficiently minimize the type graph without changing its language.

Minimization

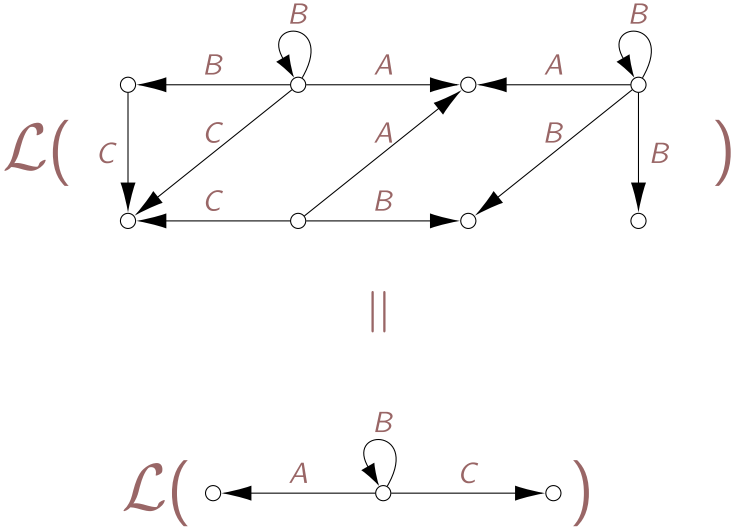


Minimization

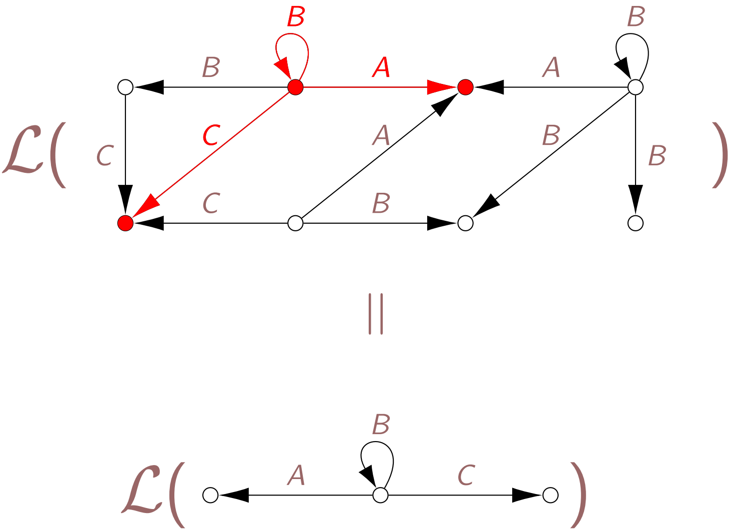




Minimization



Minimization



Minimization

Among all type graphs that generate the same language (equivalence class of the homomorphism preorder), one is a subgraph of all the others. This graph is called the core.

Minimization

Among all type graphs that generate the same language (equivalence class of the homomorphism preorder), one is a subgraph of all the others. This graph is called the core.

Retracts and Core Graphs

A subgraph T' of a graph T for which there exists a morphism $\varphi\colon T \to T'$ is called a retract of T.

If a graph has no proper retracts itself, it is called core graph. (Nešetřil, Tardif).

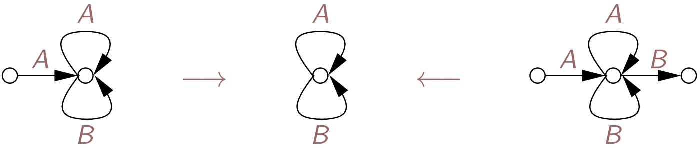
Minimization

Among all type graphs that generate the same language (equivalence class of the homomorphism preorder), one is a subgraph of all the others. This graph is called the core.

Retracts and Core Graphs

A subgraph T' of a graph T for which there exists a morphism $\varphi\colon T \to T'$ is called a retract of T.

If a graph has no proper retracts itself, it is called core graph. (Nešetřil, Tardif).



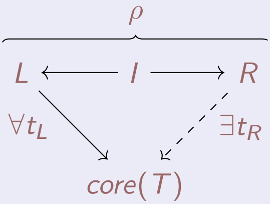
Core

Invariant Checking

Let T be a graph and $core(T)$ be its core.

Closure under rewriting

$\mathcal{L}(T)$ is closed under application of $\rho \iff$

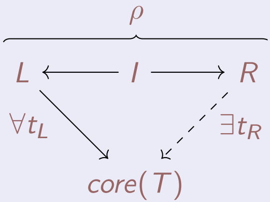


Invariant Checking

Let T be a graph and $core(T)$ be its core.

Closure under rewriting

$\mathcal{L}(T)$ is closed under application of $\rho \iff$



Question: How can we efficiently compute the core graph?

Part II

Core Computation via SAT/SMT Encodings

The Problem

Core computation is NP-hard!

The Problem

Core computation is NP-hard!

Reason: Checking whether there exists a morphism into  is equivalent to checking 3-colourability.

$$G \text{ is 3-colourable} \iff \mathit{core}(G \uplus$$  $$) =$$ 

The Problem

Core computation is NP-hard!

Reason: Checking whether there exists a morphism into  is equivalent to checking 3-colourability.

G is 3-colourable $\iff$ $core(G \uplus$  $) =$ 

Question: Given a graph G, does G contain a retract H?

The Problem

Core computation is NP-hard!

Reason: Checking whether there exists a morphism into  is equivalent to checking 3-colourability.

G is 3-colourable $\iff$ $core(G \uplus$  $) =$ 

Question: Given a graph G, does G contain a retract H?

Retract Morphism Problem

Given a graph G. Does there exist a non-surjective endomorphism $\varphi' \colon G \to G$ with $\varphi'|_H = id_H$ where $H = img(\varphi')$?

SMT Solver

Satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic.

SMT Solver

Satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic.

SMT solvers are useful for

- Verification
- Correctness proofs of programs
- Software testing based on symbolic execution

SMT Solver

Satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic.

SMT solvers are useful for

- Verification
- Correctness proofs of programs
- Software testing based on symbolic execution

We are using the SMT-LIB2 standard $\rightsquigarrow$ prefix notation.

SMT Solver

Satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic.

SMT solvers are useful for

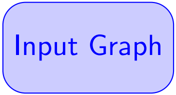
- Verification
- Correctness proofs of programs
- Software testing based on symbolic execution

We are using the SMT-LIB2 standard $\rightsquigarrow$ prefix notation.

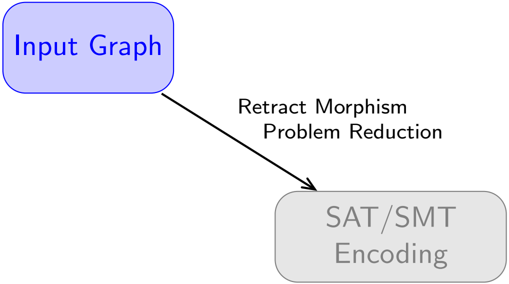
Example

```
(declare-const x Int)                      | x, y ∈ Int
(declare-const y Int)
(assert (= (- x y) (+ x (- y) 1)))         | x − y = x − y + 1
(check-sat)
```

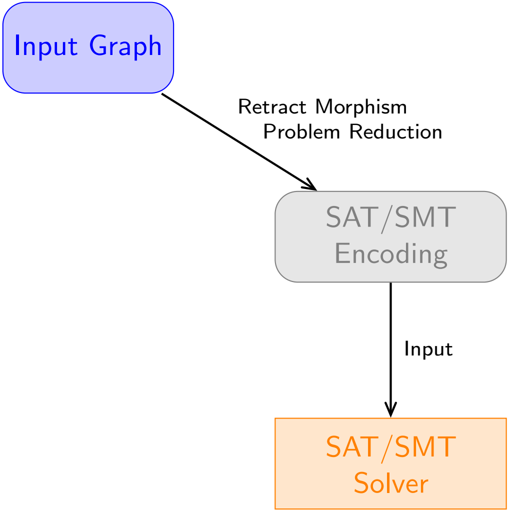
Core Computation in a Nutshell



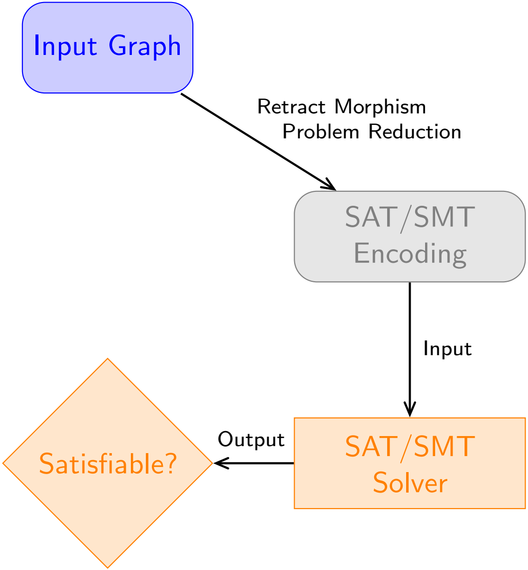
Core Computation in a Nutshell



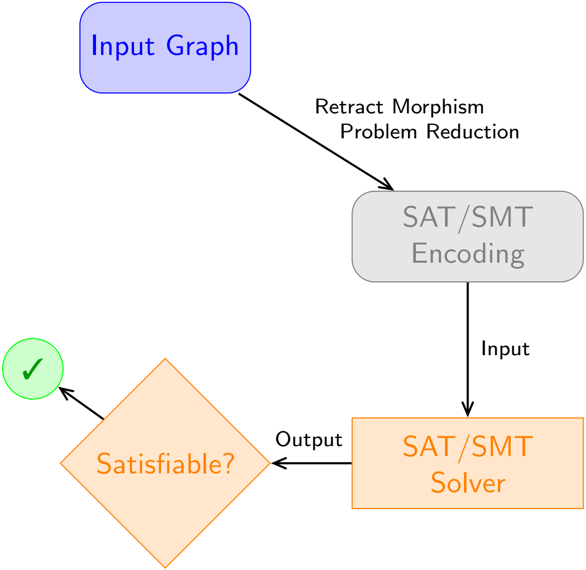
Core Computation in a Nutshell



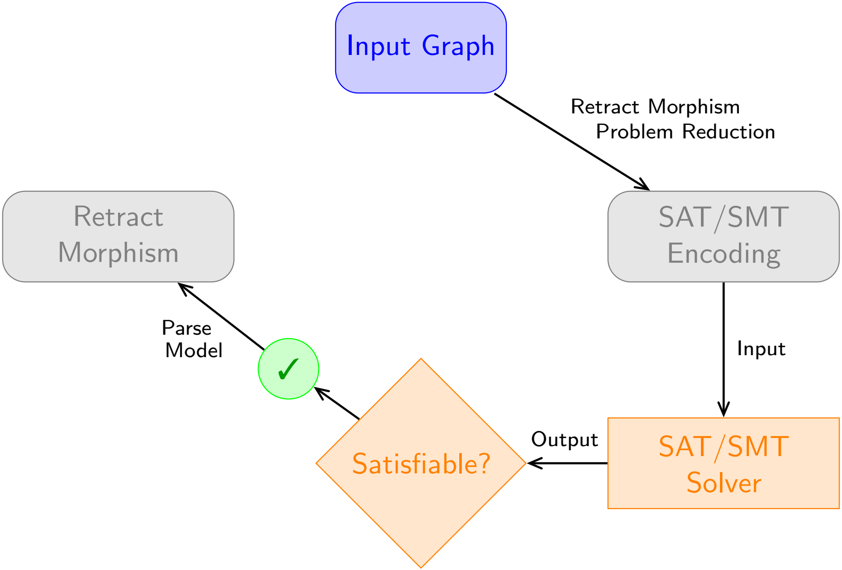
Core Computation in a Nutshell



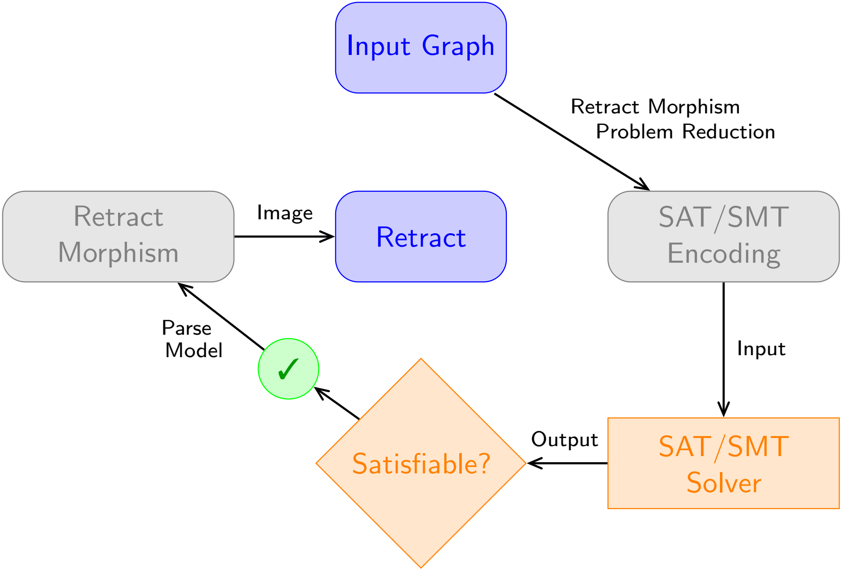
Core Computation in a Nutshell



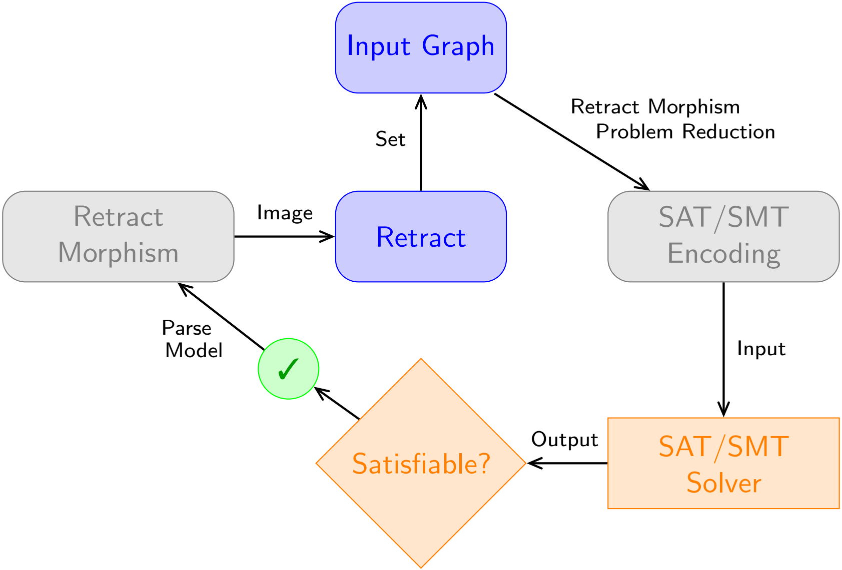
Core Computation in a Nutshell



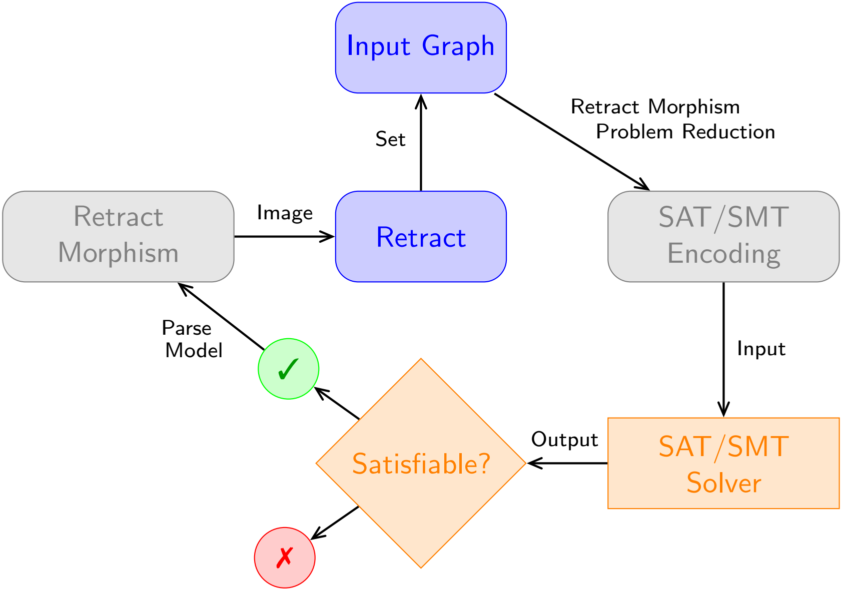
Core Computation in a Nutshell



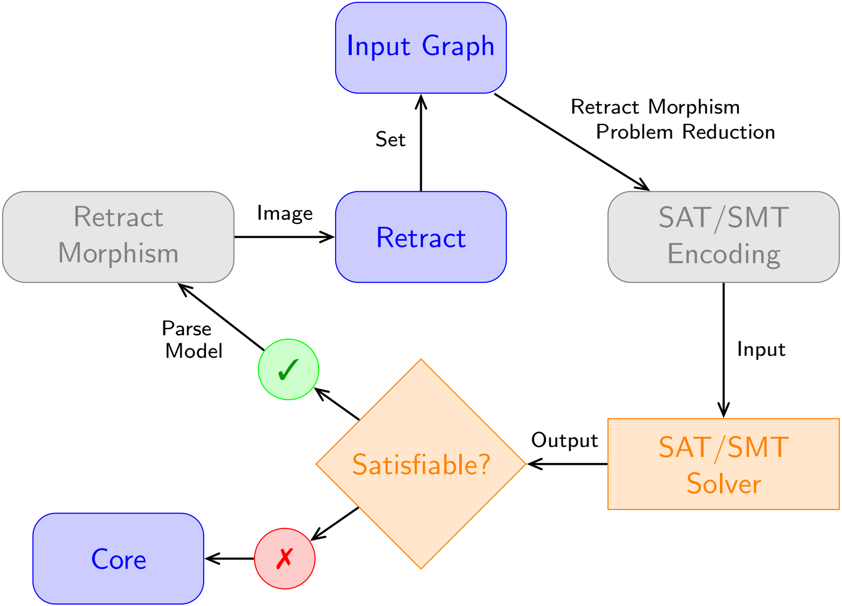
Core Computation in a Nutshell



Core Computation in a Nutshell



Core Computation in a Nutshell



Retract Morphism Properties

For an input graph $G = (V, E, src, tgt, lab)$, the encoding of φ needs to satisfy the following three conditions:

Retract Morphism Properties

For an input graph $G = (V, E, src, tgt, lab)$, the encoding of φ needs to satisfy the following three conditions:

1) Graph morphism property:
The morphism φ needs to be structure preserving, i.e.

$$src(\varphi_E(e)) = \varphi_V(src(e)) \quad tgt(\varphi_E(e)) = \varphi_V(tgt(e)) \quad lab(\varphi_E(e)) = lab(e)$$

Retract Morphism Properties

For an input graph $G = (V, E, src, tgt, lab)$, the encoding of φ needs to satisfy the following three conditions:

1) Graph morphism property:
The morphism φ needs to be structure preserving, i.e.

$$src(\varphi_E(e)) = \varphi_V(src(e)) \quad tgt(\varphi_E(e)) = \varphi_V(tgt(e)) \quad lab(\varphi_E(e)) = lab(e)$$

2) Subgraph property:
The morphism φ needs to be a non-surjective endomorphism, i.e.

$$dom(\varphi) = cod(\varphi) \qquad \exists v \in V : v \notin img(\varphi)$$

Retract Morphism Properties

For an input graph $G = (V, E, src, tgt, lab)$, the encoding of φ needs to satisfy the following three conditions:

1) Graph morphism property:
The morphism φ needs to be structure preserving, i.e.

$$src(\varphi_E(e)) = \varphi_V(src(e)) \quad tgt(\varphi_E(e)) = \varphi_V(tgt(e)) \quad lab(\varphi_E(e)) = lab(e)$$

2) Subgraph property:
The morphism φ needs to be a non-surjective endomorphism, i.e.

$$dom(\varphi) = cod(\varphi) \qquad \exists v \in V : v \notin img(\varphi)$$

3) Retract property:
The morphism φ restricted on its image is an identity morphism, i.e.

$$\varphi|_{img(\varphi)} = id_{img(\varphi)}$$

SMT-LIB2 Encoding of Retract Morphism Properties

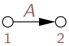
Initialize the components of the input $G = (V, E, src, tgt, lab)$:

(declare-datatypes () ((V v1 ... vN)))	$\mid (V = \{v_1, \ldots, v_n\})$
(declare-datatypes () ((E e1 ... eM)))	$\mid (E = \{e_1, \ldots, e_m\})$
(declare-datatypes () ((L A ...)))	$\mid (\Lambda = \{A, \ldots\})$
(declare-fun src (E) V)	$\mid src\colon E \to V$
(declare-fun tgt (E) V)	$\mid tgt\colon E \to V$
(declare-fun lab (E) L)	$\mid lab\colon E \to \lambda$

SMT-LIB2 Encoding of Retract Morphism Properties

Initialize the components of the input $G = (V, E, src, tgt, lab)$:

(declare-datatypes () ((V v1 ... vN)))	$\mid (V = \{v_1, \ldots, v_n\})$
(declare-datatypes () ((E e1 ... eM)))	$\mid (E = \{e_1, \ldots, e_m\})$
(declare-datatypes () ((L A ...)))	$\mid (\Lambda = \{A, \ldots\})$
(declare-fun src (E) V)	$\mid src\colon E \to V$
(declare-fun tgt (E) V)	$\mid tgt\colon E \to V$
(declare-fun lab (E) L)	$\mid lab\colon E \to \lambda$

For instance the graph 

can be encoded in the following way:

(assert (= (src e1) v1))	$\mid src(e_1) = v_1$
(assert (= (tgt e1) v2))	$\mid tgt(e_1) = v_2$
(assert (= (lab e1) A))	$\mid lab(e_1) = A$

SMT-LIB2 Encoding of Retract Morphism Properties

Next, we specify the constraints for the morphism $\varphi\colon G \to G$:

1) Graph morphism property

(declare-fun vphi (V) V)	$\mid \varphi_V\colon V \to V$
(declare-fun ephi (E) E)	$\mid \varphi_E\colon E \to E$
(assert (forall ((e E)) (= (src (ephi e)) (vphi (src e)))))	$\mid src(\varphi_E(e)) = \varphi_V(src(e))$
(assert (forall ((e E)) (= (tgt (ephi e)) (vphi (tgt e)))))	$\mid tgt(\varphi_E(e)) = \varphi_V(tgt(e))$
(assert (forall ((e E)) (= (lab (ephi e)) (lab e))))	$\mid lab(\varphi_E(e)) = lab(e)$

SMT-LIB2 Encoding of Retract Morphism Properties

Next, we specify the constraints for the morphism $\varphi\colon G \to G$:

1) Graph morphism property

(declare-fun vphi (V) V)	$\mid \varphi_V\colon V \to V$
(declare-fun ephi (E) E)	$\mid \varphi_E\colon E \to E$
(assert (forall ((e E)) (= (src (ephi e)) (vphi (src e)))))	$\mid src(\varphi_E(e)) = \varphi_V(src(e))$
(assert (forall ((e E)) (= (tgt (ephi e)) (vphi (tgt e)))))	$\mid tgt(\varphi_E(e)) = \varphi_V(tgt(e))$
(assert (forall ((e E)) (= (lab (ephi e)) (lab e))))	$\mid lab(\varphi_E(e)) = lab(e)$

2) Subgraph property

(assert (exists ((v1 V)) not(exists ((v2 V)) (= v1 (vphi v2)))))	$\mid \exists v_1 \in V \neg \exists v_2 \in V : v_1 = \varphi_V(v_2)$

SMT-LIB2 Encoding of Retract Morphism Properties

We need to specify that the retract property $\varphi|_{img(\varphi)} = id_{img(\varphi)}$ holds. We rephrase this requirement in the following way:

$$\forall x \in G\Big((\exists y \in G \ (\varphi(y) = x)) \implies \varphi(x) = x \Big)$$

Every element in the image of φ is part of the retract and therefore always has to be mapped to itself.

SMT-LIB2 Encoding of Retract Morphism Properties

We need to specify that the retract property $\varphi|_{img(\varphi)} = id_{img(\varphi)}$ holds. We rephrase this requirement in the following way:

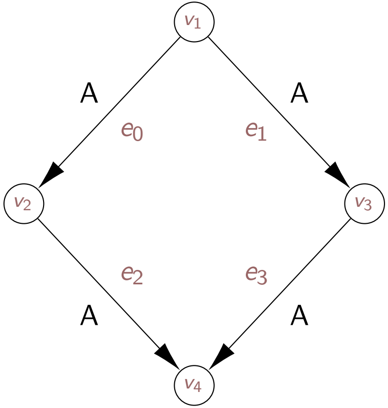
$$\forall x \in G\Big((\exists y \in G\ (\varphi(y) = x)) \implies \varphi(x) = x\Big)$$

Every element in the image of φ is part of the retract and therefore always has to be mapped to itself.

3) Retract property

```
(assert (forall ((v1 V)) (=> (exists ((v2 V)) (= v1 (vphi v2))) (= v1 (vphi v1)))))
(assert (forall ((e1 E)) (=> (exists ((e2 E)) (= e1 (ephi e2))) (= e1 (ephi e1)))))
```

Example Graph



SAT Encoding of Retract Morphism Properties

The SAT encoding is more tedious to achieve.

SAT Encoding of Retract Morphism Properties

The SAT encoding is more tedious to achieve.

Remove parallel edges from the type graph in a preprocessing step
$\rightsquigarrow$ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

SAT Encoding of Retract Morphism Properties

The SAT encoding is more tedious to achieve.

Remove parallel edges from the type graph in a preprocessing step
$\rightsquigarrow$ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

Our set of atomic propositions $\mathcal{A}$ has size $|\mathcal{A}| = |V \times V|$.

For a pair of nodes $(x, y) \in V \times V$ we use `Ax-y` with

$$\mathcal{A} \ni \texttt{A}x\texttt{-}y \equiv \texttt{true} \text{ iff } \varphi_V(x) = y \text{ holds.}$$

SAT Encoding of Retract Morphism Properties

The SAT encoding is more tedious to achieve.

Remove parallel edges from the type graph in a preprocessing step $\rightsquigarrow$ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

Our set of atomic propositions $\mathcal{A}$ has size $|\mathcal{A}| = |V \times V|$.

For a pair of nodes $(x, y) \in V \times V$ we use Ax-y with

$$\mathcal{A} \ni \texttt{A}x\text{-}y \equiv \texttt{true} \text{ iff } \varphi_V(x) = y \text{ holds.}$$

The node mapping must be a function.

SAT Encoding of Retract Morphism Properties

The SAT encoding is more tedious to achieve.

Remove parallel edges from the type graph in a preprocessing step
$\rightsquigarrow$ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

Our set of atomic propositions $\mathcal{A}$ has size $|\mathcal{A}| = |V \times V|$.

For a pair of nodes $(x, y) \in V \times V$ we use `Ax-y` with

$$\mathcal{A} \ni \texttt{A}x\texttt{-}y \equiv \texttt{true} \text{ iff } \varphi_V(x) = y \text{ holds.}$$

The node mapping must be a function.

Additional requirement

$$\bigwedge_{x \in V} \bigvee_{y \in V} \Big(\texttt{A}x\texttt{-}y \wedge \big(\bigwedge_{z \in V \setminus \{y\}} \neg \texttt{A}x\texttt{-}z \big) \Big) \quad | \; \forall x \exists! y \; \varphi_V(x) = y$$

SAT Encoding of Retract Morphism Properties

1) Graph morphism property

$$\bigwedge_{e \in E} \bigvee_{e' \in E_{lab(e)}} \Big((\mathtt{A}src(e)\text{-}src(e')) \wedge (\mathtt{A}tgt(e)\text{-}tgt(e')) \Big)$$

2) Subgraph property

$$\bigvee_{x \in V} \Big(\bigwedge_{y \in V} \neg \mathtt{A}y\text{-}x \Big) \qquad | \exists x \forall y \ \varphi(y) \neq x$$

3) Retract property

$$\bigwedge_{x \in V} \Big(\big(\bigvee_{y \in V} \mathtt{A}y\text{-}x \big) \Rightarrow \mathtt{A}x\text{-}x \Big) \qquad | \ \varphi_{|H} = id_H$$

SAT Encoding of Retract Morphism Properties

1) Graph morphism property

$$\bigwedge_{e \in E} \bigvee_{e' \in E_{lab(e)}} \Big((\mathtt{A}src(e)\text{-}src(e')) \wedge (\mathtt{A}tgt(e)\text{-}tgt(e')) \Big)$$

2) Subgraph property

$$\bigvee_{x \in V} \Big(\bigwedge_{y \in V} \neg \mathtt{A}y\text{-}x \Big) \qquad | \exists x \forall y \; \varphi(y) \neq x$$

3) Retract property

$$\bigwedge_{x \in V} \Big(\big(\bigvee_{y \in V} \mathtt{A}y\text{-}x \big) \Rightarrow \mathtt{A}x\text{-}x \Big) \qquad | \; \varphi_{|H} = id_H$$

The derivation of the formulas above is given in our paper.

Part III

CoReS

(Computation of Retracts encoded SAT/SMT)

Experiments

The encodings were tested on 125 random graphs consisting of

- a fixed number of nodes $|V|$.
- a fixed number of available edge labels $|\Lambda|$.
- a fixed probability ρ for an edge to exist.

SAT (Limboole) vs SMT (Z3)

		$\rho \cdot \lvert V\rvert \cdot \lvert\Lambda\rvert$									
		0.5		0.8		1.0		1.2		1.5	
$\lvert V\rvert$	$\lvert\Lambda\rvert$	SAT	SMT	SAT	SMT	SAT	SMT	SAT	SMT	SAT	SMT
16	1	.075	.116	.078	.344	.078	.733	.071	1.17	.070	3.01
	2	.067	.155	.096	.463	.080	1.12	.079	2.11	.078	4.21
	3	.063	.172	.100	.548	.074	1.14	.071	2.02	.073	4.09
32	1	.301	.620	.306	4.58	.396	12.4	.424	27.4	.500	67.5
	2	.389	1.08	.407	7.27	.415	14.9	.447	37.6	.450	121
	3	.322	1.52	.383	5.27	.365	19.3	.391	40.3	.382	110

Final Remarks

Contribution:

- Investigation of encodings for core computations:
  Analysis and encoding of needed properties in SAT/SMT.
- Benchmarks:
  Trade-off between readability and performance.

Tool support:

- CoReS:
  Automatically compute core graphs via SAT/SMT encodings.

Features:

- GUI mode for visualized core computations.
- Integrable and executable standalone command line interface.
- User-manual and source code (Python) available on GitHub:
  `https://github.com/mnederkorn/CoReS`

Thank You

for your attention

Part IV

Additional Material

Invariant checking

Closure under Rewriting

Question: Given T and a (DPO) GTS rule $r = (L \leftarrow I \rightarrow R)$.
Does $Post_{\{r\}}(\mathcal{L}(T)) \subseteq \mathcal{L}(T)$ hold?

Invariant checking

Closure under Rewriting

Question: Given T and a (DPO) GTS rule $r = (L \leftarrow I \rightarrow R)$.
Does $Post_{\{r\}}(\mathcal{L}(T)) \subseteq \mathcal{L}(T)$ hold?

$Post_{\{r\}}(\mathcal{L}(T))$ can not be computed...

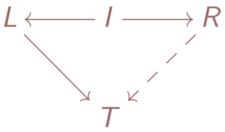
Invariant checking

Closure under Rewriting

Question: Given T and a (DPO) GTS rule $r = (L \leftarrow I \rightarrow R)$.
Does $Post_{\{r\}}(\mathcal{L}(T)) \subseteq \mathcal{L}(T)$ hold?

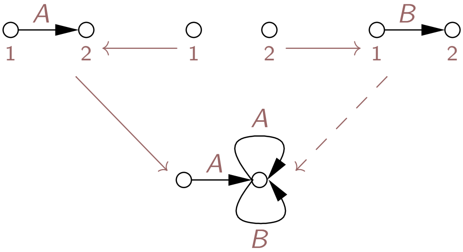
$Post_{\{r\}}(\mathcal{L}(T))$ can not be computed...

Sufficient condition: Check whether for each morphism $L \rightarrow T$ there exists a morphism $R \rightarrow T$ such that the diagram below commutes. This implies closure under rewriting.



The missing piece

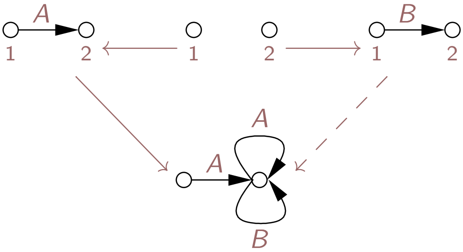
This is **not** an if-and-only-if condition. Counterexample:



However, the type graph represents *all* graphs with A- and B-labelled edges and is hence closed under rewriting.

The missing piece

This is not an if-and-only-if condition. Counterexample:



However, the type graph represents *all* graphs with A- and B-labelled edges and is hence closed under rewriting.

Solution: We obtain an if-and-only-if condition if we require that the type graph T is a core!

Experiments

Additional SAT runtimes

		$\rho \cdot \lvert V \rvert \cdot \lvert \Lambda \rvert$										
$\lvert V \rvert$	$\lvert \Lambda \rvert$	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5
	1	.462	.595	.309	.333	.351	.359	.388	.476	.371	.589	.354
24	2	.337	.356	.548	.587	1.29	.623	.685	.685	.511	.739	.497
	3	.410	.401	1.00	.460	.456	.871	.450	.490	1.60	.615	.574
	1	.619	.828	.901	1.17	1.11	.85	.973	1.29	.986	1.01	1.53
32	2	.683	.809	.792	.988	1.03	1.27	1.04	1.13	1.23	1.22	1.23
	3	1.13	1.01	.821	.819	1.16	.937	1.10	1.05	1.87	1.27	1.20
	1	2.39	2.62	3.27	3.15	4.45	5.18	5.34	7.18	5.01	5.93	6.24
48	2	1.83	1.83	3.23	3.68	3.97	3.98	4.75	5.47	4.98	5.02	5.37
	3	2.35	2.57	3.06	3.25	3.59	3.94	3.88	4.17	4.28	5.33	4.96
	1	6.63	8.65	12.0	12.7	19.4	21.9	21.2	26.2	22.5	22.1	26.0
64	2	4.04	5.91	6.73	10.9	10.3	14.9	15.2	15.2	15.4	15.7	18.4
	3	4.53	5.60	7.22	8.96	9.02	11.0	10.6	12.0	12.7	11.9	12.1
	1	37.5	49.8	92.8	125	123	165	140	163	193	152	194
96	2	28.6	49.9	59.7	85.5	98.9	102	107	115	127	111	116
	3	23.7	36.7	50.4	60.6	52.0	51.8	48.8	52.6	49.0	44.0	46.6