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.

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 $\Rightarrow$ harder analysis.
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

Background and Preliminaries (Exposition)
- Specifying Graph Languages using Type Graphs
- Retracts and Cores

Core Computation via SAT/SMT Encodings (Rising Action)
- Retract Morphism Properties
- Core Computation Encodings

CoReS (Peripety)
- Tool Demo
- Runtime Results

Final Remarks (Dénouement)
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 $T$:

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

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

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

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 \( T \):

\[
\mathcal{L}(T) = \{ G \mid \text{there exists a morphism } \varphi: G \to T \}
\]

**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

\[ \mathcal{L}(C \parallel A, B, C) \]

\[ \mathcal{L}(A, C) \]
Minimization

\[ \mathcal{L}(C \rightarrow C) = (A \rightarrow B, B \rightarrow B) \]
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 $\varphi: T \rightarrow T'$ 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 $\varphi: T \to T'$ is called a retract of $T$.

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

Let $T$ be a graph and $\text{core}(T)$ be its core.

Closure under rewriting

$L(T)$ is closed under application of $\rho$ if and only if

$$\forall t_L \exists t_R \text{core}(T)$$

\[\begin{array}{ccc}
\rho & \rho \\
L & I & R \\
\forall t_L & & \exists t_R
\end{array}\]
Invariant Checking

Let $T$ be a graph and $\text{core}(T)$ be its core.

**Closure under rewriting**

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

$$
\rho
\begin{array}{c}
\quad
L
\quad
I
\quad
R
\end{array}
\quad
\begin{array}{c}
\forall t_L
\quad
\exists t_R
\end{array}
\quad
\begin{array}{c}
\quad
\text{core}(T)
\end{array}
$$

**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 \( \triangle \) is equivalent to checking 3-colourability.

\[
G \text{ is 3-colourable} \iff \text{core}(G \cup \triangle) = \triangle
\]
The Problem

Core computation is \textbf{NP-hard}!

\textbf{Reason:} Checking whether there exists a morphism into $\triangledown$ is equivalent to checking $3$-colourability.

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

\textbf{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 \( \triangle \) is equivalent to checking 3-colourability.

\[
G \text{ is 3-colourable } \iff \text{core}(G \cup \triangle) = \triangle
\]

**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 \( \varphi' : G \to G \) with \( \varphi'|_H = id_H \) where \( H = \text{img}(\varphi') \)?
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.

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)
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

Example

\[(\text{declare-const } x \text{ Int}) \land (\text{declare-const } y \text{ Int}) \land (\text{assert } (= (\neg x y) (+ x (\neg y) 1))) \land (\text{check-sat})]\]
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 \( \mapsto \) 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 \( \Rightarrow \) prefix notation.

Example

\[
\begin{align*}
(\text{declare-const } x \text{ Int}) & \quad | \quad x, y \in \text{Int} \\
(\text{declare-const } y \text{ Int}) \\
(\text{assert } (= (\neg x y) (+ x (\neg y) 1))) & \quad | \quad x - y = x - y + 1 \\
(\text{check-sat})
\end{align*}
\]
Core Computation in a Nutshell

Input Graph
Core Computation in a Nutshell

Input Graph

Retract Morphism
Problem Reduction

SAT/SMT Encoding
Core Computation in a Nutshell

Input Graph

Retract Morphism
Problem Reduction

SAT/SMT
Encoding

Input

SAT/SMT
Solver

Satisfiable?

Retract
Morphism

Parse
Model

Retract
Image

Set
Core Computation in a Nutshell

Input Graph

Retract Morphism
Problem Reduction

SAT/SMT Encoding

Input

Satisfiable?

Output

SAT/SMT Solver

Retract
Morphism
Parse
Model
Retract
Image
Set
Core
Core Computation in a Nutshell

Input Graph

Retract Morphism
Problem Reduction

SAT/SMT
Encoding

Input

Satisfiable?

Output

SAT/SMT
Solver

Retract
Morphism
Parse
Model
Retract
Image
Set
Core
Core Computation in a Nutshell

Input Graph

Retract Morphism

Parse Model

Satisfiable?

SAT/SMT Solver

Input

Output

Retract Morphism

Problem Reduction

SAT/SMT Encoding

Input

Satisfiable?

.Parse Model

Retract Morphism

Input Graph

SAT/SMT Encoding

Input

Output

Retract Morphism

Problem Reduction

SAT/SMT Solver
Core Computation in a Nutshell

Input Graph

Retract Morphism

Image

Retract

SAT/SMT Encoding

Parse Model

Satisfiable?

SAT/SMT Solver

Retract Morphism Problem Reduction

Input

Output
Core Computation in a Nutshell

- Input Graph
- SAT/SMT Encoding
- Retract Morphism
- Problem Reduction
- SAT/SMT Solver
- Input
  - Satisfiable?
  - Output
- Retract
  - Image
  - Parse Model
  - Set
  - Retract Morphism
  - Problem Reduction

Retract Morphism
Core Computation in a Nutshell

Input Graph

- Set
- Retract Morphism
- Problem Reduction

Retract

- Image
- Parse Model
- Satisfiable?

Satisfiable?

- Output
- SAT/SMT Solver

SAT/SMT Encoding

Input

Retract Morphism

Input

Output
Core Computation in a Nutshell

Input Graph

Retract Morphism

Set

Retract Morphism Problem Reduction

Parse Model

Satisfiable?

Core

Input

Output

Retract

SAT/SMT Encoding

Input Graph

SAT/SMT Solver

Retract

Image

Satisfiable?

SAT/SMT Encoding
Retract Morphism Properties

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

1) Graph morphism property: The morphism $\varphi$ needs to be structure preserving, i.e.
   
   $\varphi(V)(src(e)) = \varphi(E)(e)$
   $\varphi(V)(tgt(e)) = \varphi(E)(e)$
   $\varphi(E)(e) = lab(e)$

2) Subgraph property: The morphism $\varphi$ needs to be a non-surjective endomorphism, i.e.
   
   $\text{dom}(\varphi) = \text{cod}(\varphi)$
   $\exists v \in V : v \not\in \text{img}(\varphi)$

3) Retract property: The morphism $\varphi$ restricted on its image is an identity morphism, i.e.
   
   $\varphi|_{\text{img}(\varphi)} = \text{id}_{\text{img}(\varphi)}$
Retract Morphism Properties

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

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

\[
\begin{align*}
src(\varphi_E(e)) &= \varphi_V(src(e)) \\
tgt(\varphi_E(e)) &= \varphi_V(tgt(e)) \\
lab(\varphi_E(e)) &= lab(e)
\end{align*}
\]
Retract Morphism Properties

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

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

\[
\begin{align*}
src(\varphi(E)) &= \varphi(V)(src(e)) \\
tgt(\varphi(E)) &= \varphi(V)(tgt(e)) \\
lab(\varphi(E)) &= lab(e)
\end{align*}
\]

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

\[
\begin{align*}
dom(\varphi) &= cod(\varphi) \\
\exists v \in V : v \notin \text{img}(\varphi)
\end{align*}
\]
Retract Morphism Properties

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

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

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

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

$$\text{dom}(\varphi) = \text{cod}(\varphi) \quad \exists v \in V: v \notin \text{img}(\varphi)$$

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

$$\varphi|_{\text{img}(\varphi)} = \text{id}_{\text{img}(\varphi)}$$
SMT-LIB2 Encoding of Retract Morphism Properties

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

(declare-datatypes () ((V v1 ... vN))) | ($V = \{v_1, ..., v_n\}$)
(declare-datatypes () ((E e1 ... eM))) | ($E = \{e_1, ..., e_m\}$)
(declare-datatypes () ((L A ...))) | ($\Lambda = \{A, \ldots\}$)
(declare-fun src (E) V) | $src : E \rightarrow V$
(declare-fun tgt (E) V) | $tgt : E \rightarrow V$
(declare-fun lab (E) L) | $lab : E \rightarrow \Lambda$
SMT-LIB2 Encoding of Retract Morphism Properties

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

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

For instance the graph $\xymatrix{1 \ar[r]^A & 2}$ can be encoded in the following way:

$(\text{assert} (= (\text{src} \ e1) v1)) | src(e_1) = v_1$
$(\text{assert} (= (\text{tgt} \ e1) v2)) | tgt(e_1) = v_2$
$(\text{assert} (= (\text{lab} \ e1) A)) | lab(e_1) = A$
SMT-LIB2 Encoding of Retract Morphism Properties

Next, we specify the constraints for the morphism \( \varphi: G \rightarrow G \):

1) Graph morphism property

\[
(\text{declare-fun } v\varphi (V) V) \quad \mid \varphi_V: V \rightarrow V \\
(\text{declare-fun } e\varphi (E) E) \quad \mid \varphi_E: E \rightarrow E \\
(\text{assert (forall ((e E)) (= (src (e\varphi e)) (v\varphi (src e)))))} \mid \text{src}(\varphi_E(e)) = \varphi_V(\text{src}(e)) \\
(\text{assert (forall ((e E)) (= (tgt (e\varphi e)) (v\varphi (tgt e))))}) \mid \text{tgt}(\varphi_E(e)) = \varphi_V(\text{tgt}(e)) \\
(\text{assert (forall ((e E)) (= (lab (e\varphi e)) (lab e))))} \mid \text{lab}(\varphi_E(e)) = \text{lab}(e)
\]
Next, we specify the constraints for the morphism $\varphi : G \rightarrow G$:

1) **Graph morphism property**

   \[
   \begin{align*}
   &\text{(declare-fun vphi (V) V)} \quad | \quad \varphi_V : V \rightarrow V \\
   &\text{(declare-fun ephi (E) E)} \quad | \quad \varphi_E : E \rightarrow E \\
   &\text{(assert (forall ((e E)) (\(= (\text{src (ephi e)})(\text{vphi (src e)})))))} \quad | \quad \text{src} (\varphi_E (e)) = \varphi_V (\text{src} (e)) \\
   &\text{(assert (forall ((e E)) (\(= (\text{tgt (ephi e)})(\text{vphi (tgt e)})))))} \quad | \quad \text{tgt} (\varphi_E (e)) = \varphi_V (\text{tgt} (e)) \\
   &\text{(assert (forall ((e E)) (\(= (\text{lab (ephi e)})(\text{lab e})))))} \quad | \quad \text{lab} (\varphi_E (e)) = \text{lab} (e)
   \end{align*}
   \]

2) **Subgraph property**

   \[
   \begin{align*}
   &\text{(assert (exists ((v1 V)) not(exists ((v2 V)) (\(= v1 (\text{vphi v2}))))))} \quad | \quad \exists v_1 \in V \neg \exists v_2 \in V : \\
   &\quad v_1 = \varphi_V (v_2)
   \end{align*}
   \]
We need to specify that the retract property $\varphi|_{\text{img}(\varphi)} = \text{id}_{\text{img}(\varphi)}$ holds. We rephrase this requirement in the following way:

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

Every element in the image of $\varphi$ is part of the retract and therefore always has to be mapped to itself.
We need to specify that the retract property $\varphi|_{\text{img}(\varphi)} = \text{id}_{\text{img}(\varphi)}$ holds. We rephrase this requirement in the following way:

\[ \forall x \in G \left( \exists y \in G \ (\varphi(y) = x) \right) \implies \varphi(x) = x \]

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

3) Retract property

\begin{align*}
\text{(assert (forall ((v1 V)) (=> (exists ((v2 V)) (= v1 (vphi v2))) (= v1 (vphi v1)))))} \\
\text{(assert (forall ((e1 E)) (=> (exists ((e2 E)) (= e1 (ephi e2))) (= e1 (ephi e1)))))}
\end{align*}
Example Graph

\[
\begin{align*}
&v_1 & & v_3 \\
& & A & & A \\
& A & & e_0 & e_1 \\
& e_2 & & v_2 & v_3 \\
& A & & & A \\
& & e_3 & & \\
& A & & v_1 & v_4
\end{align*}
\]
The SAT encoding is more tedious to achieve.

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

For a pair of nodes $(x, y) \in V \times V$ we use $A_{x-y}$ with $A \ni A_{x-y} \equiv \text{true}$ iff $\phi_V(x) = y$ holds.

The node mapping must be a function.

Additional requirement $\bigwedge x \in V \bigvee y \in V (A_{x-y} \land \bigwedge z \in V \setminus \{y\} \neg A_{x-z})$.

$\forall x \exists ! y \phi_V(x) = y$.
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

⇝ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

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

For a pair of nodes $(x, y) \in V \times V$ we use $A_{x-y}$ with $A \ni A_{x-y} \equiv \text{true}$ iff $\phi_V(x) = y$ holds.

The node mapping must be a function.

Additional requirement $\bigwedge x \in V \bigvee y \in V (A_{x-y} \land (\bigwedge z \in V \{y\} \neg A_{x-z}))$ for all $x \exists ! y \phi_V(x) = y$.
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 ⇔ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

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

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

$$A \ni Ax-y \equiv \text{true} \iff \varphi_V(x) = y \text{ holds.}$$
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 ⇔ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

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

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

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

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 $\implies$ Find a node mapping describing the retract since the corresponding edge mappings can be derived from it.

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

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

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

The node mapping must be a function.

Additional requirement

$$\forall x \in V \forall y \in V \left( Ax-y \land (\forall z \in V \setminus \{y\} \neg Ax-z) \right) \mid \forall x \exists! y \varphi_V(x) = y$$
SAT Encoding of Retract Morphism Properties

1) Graph morphism property

\[ \bigwedge_{e \in E} \bigvee_{e' \in E_{lab(e)}} \left( (Asrc(e) - src(e')) \land (Atgt(e) - tgt(e')) \right) \]

2) Subgraph property

\[ \bigvee_{x \in V} \left( \bigwedge_{y \in V} \neg Ay-x \right) \quad \mid \exists x \forall y \ \varphi(y) \neq x \]

3) Retract property

\[ \bigwedge_{x \in V} \left( \bigvee_{y \in V} Ay-x \Rightarrow Ax-x \right) \quad \mid \varphi|_H = id_H \]
SAT Encoding of Retract Morphism Properties

1) Graph morphism property

\[ \bigwedge_{e \in E} \bigvee_{e' \in E_{lab(e)}} \left( (Asrc(e) - src(e')) \land (Atgt(e) - tgt(e')) \right) \]

2) Subgraph property

\[ \bigvee_{x \in V} \left( \bigwedge_{y \in V} \neg A_y - x \right) \quad \mid \exists x \forall y \quad \varphi(y) \neq x \]

3) Retract property

\[ \bigwedge_{x \in V} \left( (\bigvee_{y \in V} A_y - x) \Rightarrow A_x - x \right) \quad \mid \varphi|_H = id_H \]

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 $\rho$ for an edge to exist.

SAT (Limboole) vs SMT (Z3)

| $|V|$ | $|\Lambda|$ | 0.5   | 0.8   | 1.0   | 1.2   | 1.5   |
|------|--------|-------|-------|-------|-------|-------|
|      | SAT    | SMT   | SAT   | SMT   | SAT   | SMT   | SAT   | SMT   |
| 16   |        |       |       |       |       |       |       |       |
| 1    | .075   | .116  | .078  | .344  | .078  | .733  | .071  | 1.17  |
| 2    | .067   | .155  | .096  | .463  | .080  | 1.12  | .079  | 2.11  |
| 3    | .063   | .172  | .100  | .548  | .074  | 1.14  | .071  | 2.02  |
| 32   |        |       |       |       |       |       |       |       |
| 1    | .301   | .620  | .306  | 4.58  | .396  | 12.4  | .424  | 27.4  |
| 2    | .389   | 1.08  | .407  | 7.27  | .415  | 14.9  | .447  | 37.6  |
| 3    | .322   | 1.52  | .383  | 5.27  | .365  | 19.3  | .391  | 40.3  |

0.5, 0.8, 1.0, 1.2, 1.5

0.75, 0.116, 0.078, 0.344, 0.078, 0.733, 0.071

0.067, 0.155, 0.096, 0.463, 0.080, 1.12, 0.079

0.063, 0.172, 0.100, 0.548, 0.074, 1.14, 0.071

0.301, 0.620, 0.306, 4.58, 0.396, 12.4, 0.424

0.389, 1.08, 0.407, 7.27, 0.415, 14.9, 0.447

0.322, 1.52, 0.383, 5.27, 0.365, 19.3, 0.391

0.7, 1.17, 2.11, 2.02, 2.1, 2.02, 2.02

0.7, 1.17, 2.11, 2.02, 2.1, 2.02, 2.02

0.7, 1.17, 2.11, 2.02, 2.1, 2.02, 2.02

0.7, 1.17, 2.11, 2.02, 2.1, 2.02, 2.02

0.7, 1.17, 2.11, 2.02, 2.1, 2.02, 2.02
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 $\text{Post}_{\{r\}}(\mathcal{L}(T)) \subseteq \mathcal{L}(T)$ hold?

$\text{Post}_{\{r\}}(\mathcal{L}(T))$ 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 $L \to T$ there exists a morphism $R \to T$ such that the diagram below commutes. This implies closure under rewriting.

\[
\begin{array}{ccc}
L & \leftarrow & I \\
\downarrow & & \downarrow \\
T & \rightarrow & R
\end{array}
\]
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

| | | \(V\) | \(\Lambda\) | \(\rho \cdot |V| \cdot |\Lambda|\) |
|---|---|---|---|---|
| 24 | 1 | 0.5 | 0.462 | .619 | 2.39 | 6.63 |
| | 2 | 0.6 | 0.595 | .828 | 2.62 | 8.65 |
| | 3 | 0.7 | 0.309 | .901 | 3.27 | 12.0 |
| 32 | 1 | 0.8 | 0.333 | 1.17 | 3.15 | 12.7 |
| | 2 | 0.9 | 0.351 | 1.11 | 4.45 | 19.4 |
| | 3 | 1.0 | 0.359 | .85 | 5.18 | 21.9 |
| 48 | 1 | 1.1 | 0.388 | .973 | 7.18 | 26.2 |
| | 2 | 1.2 | 0.476 | 1.29 | 5.01 | 22.5 |
| | 3 | 1.3 | 0.371 | 1.04 | 5.93 | 22.1 |
| 64 | 1 | 1.4 | 0.589 | 1.29 | 5.01 | 22.5 |
| | 2 | 1.5 | 0.354 | 1.01 | 5.93 | 22.1 |
| | 3 | 1.0 | 0.589 | 1.29 | 5.01 | 22.5 |
| 96 | 1 | 0.5 | 0.462 | .619 | 2.39 | 6.63 |
| | 2 | 0.6 | 0.595 | .828 | 2.62 | 8.65 |
| | 3 | 0.7 | 0.309 | .901 | 3.27 | 12.0 |
| 128 | 1 | 0.8 | 0.333 | 1.17 | 3.15 | 12.7 |
| | 2 | 0.9 | 0.351 | 1.11 | 4.45 | 19.4 |
| | 3 | 1.0 | 0.359 | .85 | 5.18 | 21.9 |
| 256 | 1 | 1.1 | 0.388 | .973 | 7.18 | 26.2 |
| | 2 | 1.2 | 0.476 | 1.29 | 5.01 | 22.5 |
| | 3 | 1.3 | 0.371 | 1.04 | 5.93 | 22.1 |
| 512 | 1 | 1.4 | 0.589 | 1.29 | 5.01 | 22.5 |
| | 2 | 1.5 | 0.354 | 1.01 | 5.93 | 22.1 |
| | 3 | 1.0 | 0.589 | 1.29 | 5.01 | 22.5 |