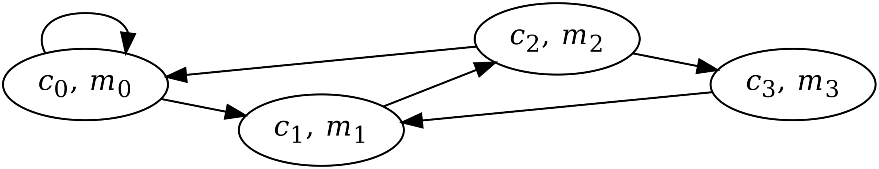
Soundness of Cyclic Proofs in KeY

KeY Symposium 2023

Daniel Drodt
TU Darmstadt



Introduction

Introduction

- Calculus of KeY must be sound
  - Otherwise proof does not guarantee anything

Introduction

- Calculus of KeY must be sound
  - Otherwise proof does not guarantee anything

One Rule of KeY Is Not Properly Proven Sound

Introduction

- Calculus of KeY must be sound
  - Otherwise proof does not guarantee anything

One Rule of KeY Is Not Properly Proven Sound

- Incorrect contracts can be verified
  - A soundness hole

Introduction

- Calculus of KeY must be sound
  - Otherwise proof does not guarantee anything

One Rule of KeY Is Not Properly Proven Sound

- Incorrect contracts can be verified
  - A soundness hole

We explore the underlying problem and discuss possible solutions.

The Problem

Rule useMethodContract

Rule useMethodContract allows usage of contract $(\psi_{\mathit{pre}}, \psi_{\mathit{post}}, \ldots)$:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{\mathit{pre}} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{\mathit{post}} \land \mathit{wellFormed}(h) \land \ldots \land \mathtt{exc} \doteq \mathtt{null} \rightarrow \langle \pi \mathtt{x} = \mathtt{res}; \omega \rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x} = \mathtt{se.m}(\mathtt{a}_1, \ldots, \mathtt{a}_n); \omega \rangle\, \varphi, \Delta}$$

The Problem

Rule useMethodContract

Rule useMethodContract allows usage of contract $(\psi_{pre}, \psi_{post}, \dots)$:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{pre} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{post} \land \mathit{wellFormed}(h) \land \dots \land \mathtt{exc} \doteq \mathtt{null} \rightarrow \langle \pi \mathtt{x = res;} \omega \rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x = se.m(a_1, \dots, a_n);} \omega \rangle\, \varphi, \Delta}$$

The Problem

Rule useMethodContract

Rule useMethodContract allows usage of contract $(\psi_{\mathit{pre}}, \psi_{\mathit{post}}, \ldots)$:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{\mathit{pre}} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{\mathit{post}} \land \mathit{wellFormed}(h) \land \ldots \land \mathtt{exc} \doteq \mathtt{null} \to \langle \pi \mathtt{x} = \mathtt{res};\omega\rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x} = \mathtt{se.m}(\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n});\omega\rangle\, \varphi, \Delta}$$

- The proof now depends on this contract

The Problem

Rule useMethodContract

Rule useMethodContract allows usage of contract $(\psi_{pre}, \psi_{post}, \ldots)$:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{pre} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{post} \land \mathit{wellFormed}(h) \land \ldots \land \mathtt{exc} \doteq \mathtt{null} \rightarrow \langle \pi \mathtt{x} = \mathtt{res};\omega \rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x} = \mathtt{se.m}(\mathtt{a}_1, \ldots, \mathtt{a}_n);\omega \rangle\, \varphi, \Delta}$$

- The proof now depends on this contract
- useMethodContract was proven sound...

The Problem

Rule useMethodContract

Rule useMethodContract allows usage of contract $(\psi_{pre}, \psi_{post}, \dots)$:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{pre} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{post} \land \mathit{wellFormed}(h) \land \dots \land \mathtt{exc} \doteq \mathtt{null} \rightarrow \langle \pi \mathtt{x} = \mathtt{res};\omega\rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x} = \mathtt{se.m}(\mathtt{a}_1, \dots, \mathtt{a}_n);\omega\rangle\, \varphi, \Delta}$$

- The proof now depends on this contract
- useMethodContract was proven sound...
  - ...assuming that the contract is already verified

The Problem

Rule useMethodContract

Rule useMethodContract allows usage of contract $(\psi_{pre}, \psi_{post}, \ldots)$:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{pre} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{post} \land \mathit{wellFormed}(\mathtt{h}) \land \ldots \land \mathtt{exc} \doteq \mathtt{null} \to \langle \pi \mathtt{x} = \mathtt{res};\omega\rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x} = \mathtt{se.m}(\mathtt{a}_1, \ldots, \mathtt{a}_n);\omega\rangle\, \varphi, \Delta}$$

- The proof now depends on this contract
- useMethodContract was proven sound...
  - ...assuming that the contract is already verified

This Does Not Cover Recursion

- Circular reasoning
- Termination is not ensured

The Problem
Rule useMethodContractTotal

Rule useMethodContractTotal covers recursion:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{\mathit{pre}} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange} \land \mathit{term} \prec \mathtt{mby}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{\mathit{post}} \land \mathit{wellFormed}(\mathtt{h}) \land \ldots \land \mathtt{exc} \doteq \mathtt{null} \rightarrow \langle \pi \mathtt{x} = \mathtt{res}; \omega \rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x} = \mathtt{se.m}(\mathtt{a}_1, \ldots, \mathtt{a}_n); \omega \rangle\, \varphi, \Delta}$$

The Problem
Rule useMethodContractTotal

Rule useMethodContractTotal covers recursion:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{\mathsf{pre}} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange} \land \mathit{term} \prec \mathtt{mby}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{\mathsf{post}} \land \mathit{wellFormed}(\mathsf{h}) \land \ldots \land \mathtt{exc} \doteq \mathtt{null} \rightarrow \langle \pi \mathtt{x} = \mathtt{res};\omega\rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\, \langle \pi \mathtt{x} = \mathtt{se.m}(\mathsf{a}_1, \ldots, \mathsf{a}_n);\omega\rangle\, \varphi, \Delta}$$

- mby is the termination witness

The Problem
Rule useMethodContractTotal

TECHNISCHE UNIVERSITÄT DARMSTADT

Rule useMethodContractTotal covers recursion:

$$\frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(\psi_{\mathit{pre}} \land \mathit{wellFormed}(\mathtt{heap}) \land \mathit{paramsInRange} \land \mathit{term} \prec \mathtt{mby}), \Delta \\ \Gamma \vdash \mathcal{U}\mathcal{V}(\psi_{\mathit{post}} \land \mathit{wellFormed}(\mathtt{h}) \land \ldots \land \mathtt{exc} \doteq \mathtt{null} \rightarrow \langle \pi \mathtt{x} = \mathtt{res};\omega \rangle\, \varphi), \Delta\end{array}}{\Gamma \vdash \mathcal{U}\,\langle \pi \mathtt{x} = \mathtt{se.m}(\mathtt{a}_1, \ldots, \mathtt{a}_n);\omega \rangle\, \varphi, \Delta}$$

- mby is the termination witness
- Soundness has not been shown
  - Are there theoretical issues?
  - Are there practical limitations or edge cases?

Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m(int num) {
  if (num == 0)
    return 0;
  return m(num - 1);
}
```

Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m(int num) {
  if (num == 0)
    return 0;
  return m(num - 1);
}
```

- We can verify recursive methods
  - Need termination witness num

Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m(int num) {
  if (num == 0)
    return 0;
  return m(num - 1);
}
```

- We can verify recursive methods
  - Need termination witness num
- The proof is trivial

Recursion

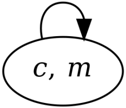
```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m(int num) {
  if (num == 0)
    return 0;
  return m(num - 1);
}
```

- We can verify recursive methods
  - Need termination witness num
- The proof is trivial
- No additional data needed
- Method `m` depends only on itself

Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m(int num) {
  if (num == 0)
    return 0;
  return m(num - 1);
}
```

- We can verify recursive methods
  - Need termination witness num
- The proof is trivial
- No additional data needed
- Method `m` depends only on itself
- We model the dependency in a graph:



Mutual Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m1(int num) {
  return num == 0 ? 0 : m2(num-1);
}
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m2(int num) {
  return num == 0 ? 0 : m1(num-1);
}
```

Mutual Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m1(int num) {
  return num == 0 ? 0 : m2(num-1);
}
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m2(int num) {
  return num == 0 ? 0 : m1(num-1);
}
```

- We can verify m1 and m2 separately

Mutual Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m1(int num) {
  return num == 0 ? 0 : m2(num-1);
}
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m2(int num) {
  return num == 0 ? 0 : m1(num-1);
}
```

- We can verify `m1` and `m2` separately
- Depend on each other

Mutual Recursion

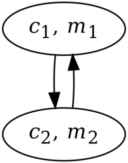
```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m1(int num) {
  return num == 0 ? 0 : m2(num-1);
}
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m2(int num) {
  return num == 0 ? 0 : m1(num-1);
}
```

- We can verify `m1` and `m2` separately
- Depend on each other
- Recursion is still bounded by `num`

Mutual Recursion

```
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m1(int num) {
  return num == 0 ? 0 : m2(num-1);
}
/*@ normal_behavior
  @ requires num >= 0;
  @ measured_by num;
  @ ensures \result == 0;
  @*/
int m2(int num) {
  return num == 0 ? 0 : m1(num-1);
}
```

- We can verify m1 and m2 separately
- Depend on each other
- Recursion is still bounded by num
- We have mutual recursion
- More complex cycle:



Mutual Recursion

Unsound Proofs

```
/*@ normal_behavior
  @ ensures false;
  @*/
void m1() {
  m2();
}

/*@ normal_behavior
  @ ensures false;
  @*/
void m2() {
  m1();
}
```

Mutual Recursion

Unsound Proofs

```
/*@ normal_behavior
  @ ensures false;
  @*/
void m1() {
  m2();
}

/*@ normal_behavior
  @ ensures false;
  @*/
void m2() {
  m1();
}
```

- KeY allows verification of m1

Mutual Recursion

Unsound Proofs

```
/*@ normal_behavior
  @ ensures false;
  @*/
void m1() {
  m2();
}

/*@ normal_behavior
  @ ensures false;
  @*/
void m2() {
  m1();
}
```

- KeY allows verification of `m1`
  - Assumes `m2` is correct

Mutual Recursion
Unsound Proofs

```
/*@ normal_behavior
  @ ensures false;
  @*/
void m1() {
  m2();
}

/*@ normal_behavior
  @ ensures false;
  @*/
void m2() {
  m1();
}
```

- KeY allows verification of `m1`
  - Assumes `m2` is correct
- Will then disallow `m2` depending on `m1`

Mutual Recursion

Unsound Proofs

```
/*@ normal_behavior
  @ ensures false;
  @*/
void m1() {
  m2();
}

/*@ normal_behavior
  @ ensures false;
  @*/
void m2() {
  m1();
}
```

- KeY allows verification of `m1`
  - Assumes `m2` is correct
- Will then disallow `m2` depending on `m1`
- We can close KeY and then verify `m2`
  - KeY loses information about dependencies

Mutual Recursion

Unsound Proofs

```
/*@ normal_behavior
  @ ensures false;
  @*/
void m1() {
  m2();
}

/*@ normal_behavior
  @ ensures false;
  @*/
void m2() {
  m1();
}
```

- KeY allows verification of `m1`
  - Assumes `m2` is correct
- Will then disallow `m2` depending on `m1`
- We can close KeY and then verify `m2`
  - KeY loses information about dependencies
- Might happen accidentally

Mutual Recursion

Unsound Proofs

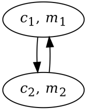
```
/*@ normal_behavior
  @ ensures false;
  @*/
void m1() {
  m2();
}

/*@ normal_behavior
  @ ensures false;
  @*/
void m2() {
  m1();
}
```

- KeY allows verification of m1
    - Assumes m2 is correct
- Will then disallow m2 depending on m1
- We can close KeY and then verify m2
    - KeY loses information about dependencies
- Might happen accidentally

When is the rule application sound?

Cyclic Dependencies

...and their soundness

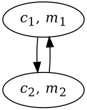


Problem

- Cyclic dependencies; units depending on themselves

Cyclic Dependencies

...and their soundness

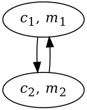


Problem

- Cyclic dependencies; units depending on themselves
- Common (theorem provers, package managers, ...)

Cyclic Dependencies

...and their soundness



Problem

- Cyclic dependencies; units depending on themselves
- Common (theorem provers, package managers, ...)

Intuitive Solution

When the cycle (recursion) is bounded, we can allow it

Modeling Proof Dependencies

Contract Dependency Graph

Modeling Proof Dependencies

Contract Dependency Graph

- Vertices are pairs of contracts *c* and methods *m*

Modeling Proof Dependencies

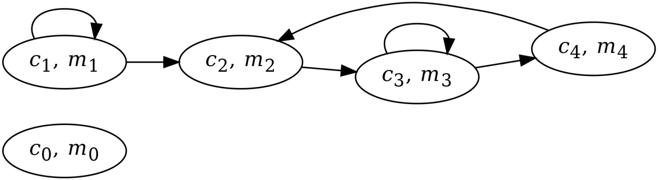
Contract Dependency Graph

- Vertices are pairs of contracts *c* and methods *m*
- Arc from (c_1, m_1) to (c_2, m_2) iff proof for (c_1, m_1) depends on (c_2, m_2)

Modeling Proof Dependencies

Contract Dependency Graph

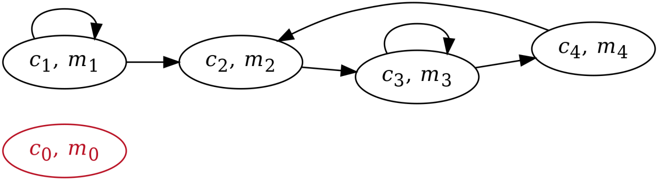
- Vertices are pairs of contracts c and methods m
- Arc from (c_1, m_1) to (c_2, m_2) iff proof for (c_1, m_1) depends on (c_2, m_2)



Modeling Proof Dependencies

Contract Dependency Graph

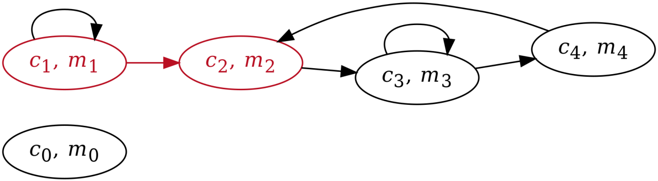
- Vertices are pairs of contracts c and methods m
- Arc from (c_1, m_1) to (c_2, m_2) iff proof for (c_1, m_1) depends on (c_2, m_2)



Modeling Proof Dependencies

Contract Dependency Graph

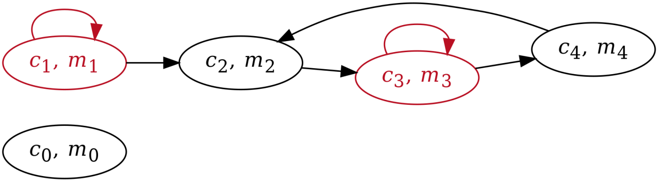
- Vertices are pairs of contracts c and methods m
- Arc from (c_1, m_1) to (c_2, m_2) iff proof for (c_1, m_1) depends on (c_2, m_2)



Modeling Proof Dependencies

Contract Dependency Graph

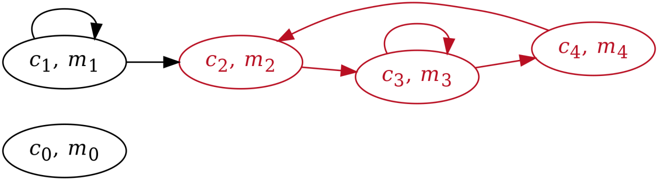
- Vertices are pairs of contracts c and methods m
- Arc from (c_1, m_1) to (c_2, m_2) iff proof for (c_1, m_1) depends on (c_2, m_2)



Modeling Proof Dependencies

Contract Dependency Graph

- Vertices are pairs of contracts c and methods m
- Arc from (c_1, m_1) to (c_2, m_2) iff proof for (c_1, m_1) depends on (c_2, m_2)



Terminating Cycle Theory

Terminating Graphs

Terminating Cycle Theory

Terminating Graphs

- Strongly connected component is terminating iff
  - It contains no arc or
  - Every contract has termination witness

Terminating Cycle Theory

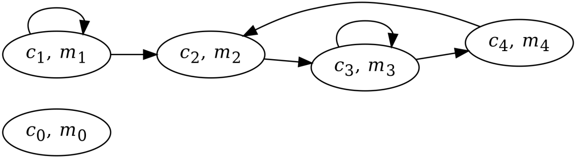
Terminating Graphs

- Strongly connected component is terminating iff
  - It contains no arc or
  - Every contract has termination witness
- Graph is called terminating iff every strongly connected component is terminating

Terminating Cycle Theory

Terminating Graphs

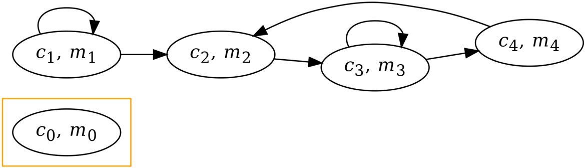
- Strongly connected component is terminating iff
  - It contains no arc or
  - Every contract has termination witness
- Graph is called terminating iff every strongly connected component is terminating



Terminating Cycle Theory

Terminating Graphs

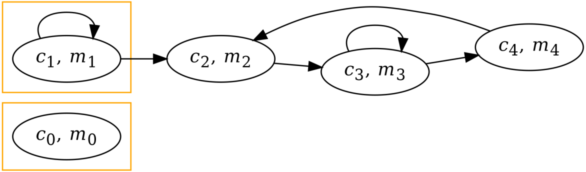
- Strongly connected component is terminating iff
  - It contains no arc or
  - Every contract has termination witness
- Graph is called terminating iff every strongly connected component is terminating



Terminating Cycle Theory

Terminating Graphs

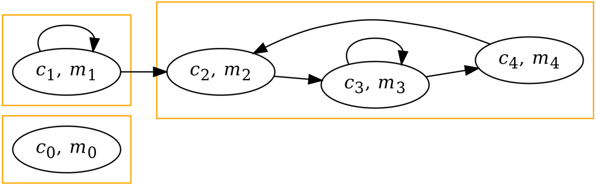
- Strongly connected component is terminating iff
  - It contains no arc or
  - Every contract has termination witness
- Graph is called terminating iff every strongly connected component is terminating



Terminating Cycle Theory

Terminating Graphs

- Strongly connected component is terminating iff
  - It contains no arc or
  - Every contract has termination witness
- Graph is called terminating iff every strongly connected component is terminating



Terminating Cycle Theory

Application

Restriction to Rule Applications

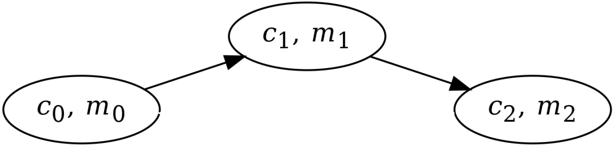
Only permit rule applications that result in a terminating Contract Dependency Graph

Terminating Cycle Theory

Application

Restriction to Rule Applications

Only permit rule applications that result in a terminating Contract Dependency Graph

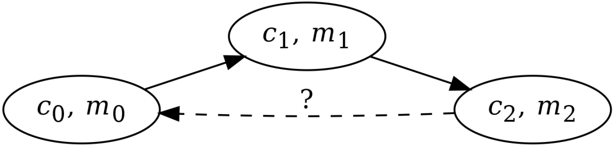


Terminating Cycle Theory

Application

Restriction to Rule Applications

Only permit rule applications that result in a terminating Contract Dependency Graph

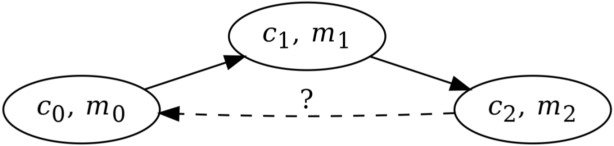


Terminating Cycle Theory

Application

Restriction to Rule Applications

Only permit rule applications that result in a terminating Contract Dependency Graph



- Restriction has been proven to ensure soundness
- Not too restrictive

Implementation

Existing Checks

Implementation

Existing Checks

- Similar to the restriction above
- Constructs (a subgraph of) the Contract Dependency Graph

Implementation

Existing Checks

- Similar to the restriction above
- Constructs (a subgraph of) the Contract Dependency Graph
- Only considers loaded proofs
- Prone to (accidental) exploits

Implementation

Existing Checks

- Similar to the restriction above
- Constructs (a subgraph of) the Contract Dependency Graph
- Only considers loaded proofs
- Prone to (accidental) exploits
- Additional tools exists, e.g., by Wolfram Pfeifer

Implementation

Existing Checks

- Similar to the restriction above
- Constructs (a subgraph of) the Contract Dependency Graph
- Only considers loaded proofs
- Prone to (accidental) exploits
- Additional tools exists, e.g., by Wolfram Pfeifer

We need to have persistent information of the global proof state!

Implementation

Original Approach

KeY has no notion of “project”

Per-Folder Dependencies

- Persistent, but no “project”

Implementation

Original Approach

KeY has no notion of “project”

Per-Folder Dependencies

- Persistent, but no “project”
- Dependency information independent of environments and proofs

Implementation

Original Approach

KeY has no notion of “project”

Per-Folder Dependencies

- Persistent, but no “project”
- Dependency information independent of environments and proofs
- When loading folder, parsing Java, creating environment, ...
  - Create dependency repository, load dependency files

Implementation
Original Approach

KeY has no notion of “project”

Per-Folder Dependencies

- Persistent, but no “project”
- Dependency information independent of environments and proofs
- When loading folder, parsing Java, creating environment, ...
  - Create dependency repository, load dependency files
- Dependency files contain
  - Dependencies of all proofs of a folder
  - Hashes of contract and method

Dependency Files

```
"/path/to/folder/MyClass1.java" {
}

"/path/to/folder/MyClass2.java" {
  "MyClass2[m1(int)].JML normal_behavior ..."|-217247427|-979473634 {
    "MyClass1[helper()].JML normal..."|102592814|280909408
  }
  "MyClass2[helper()].JML normal_behavior ..."|40138075|-7495401875 {
    "MyClass2[helper()].JML normal_behavior ..."|102592814|280909408
    "MyClass2[m2()].JML normal_behavior ..."|4910046826|-184653318
  }
}
```

Current Implementation Plan

Shortcomings of Dependency Files

Current Implementation Plan

Shortcomings of Dependency Files

- Sensible compromise to change little of KeY’s structure

Current Implementation Plan

Shortcomings of Dependency Files

- Sensible compromise to change little of KeY’s structure
- Additional files are not ideal
- Similar files are necessary/helpful for better proof management

Current Implementation Plan

Shortcomings of Dependency Files

- Sensible compromise to change little of KeY’s structure
- Additional files are not ideal
- Similar files are necessary/helpful for better proof management
  - Completed proofs
  - Changed files

Current Implementation Plan

Shortcomings of Dependency Files

- Sensible compromise to change little of KeY's structure
- Additional files are not ideal
- Similar files are necessary/helpful for better proof management
  - Completed proofs
  - Changed files

Introducing KeY Projects

Current Implementation Plan

Shortcomings of Dependency Files

- Sensible compromise to change little of KeY’s structure
- Additional files are not ideal
- Similar files are necessary/helpful for better proof management
  - Completed proofs
  - Changed files

Introducing KeY Projects

- What approaches and tools exist?
- How to implement this?

Current Implementation Plan

Shortcomings of Dependency Files

- Sensible compromise to change little of KeY's structure
- Additional files are not ideal
- Similar files are necessary/helpful for better proof management
    - Completed proofs
    - Changed files

Introducing KeY Projects

- What approaches and tools exist?
- How to implement this?

⇒ Bachelor thesis/project in cooperation with KIT

Conclusion

Conclusion

- Theoretical foundation for cyclic dependencies ✓
  - Provided proper proof of intuitive solution ✓

Conclusion

- Theoretical foundation for cyclic dependencies ✓
  - Provided proper proof of intuitive solution ✓
- Added circularity checks for model methods ✓

Conclusion

- Theoretical foundation for cyclic dependencies ✓
  - Provided proper proof of intuitive solution ✓
- Added circularity checks for model methods ✓
- Proposal for solving soundness issues without undue restrictions ✓

Conclusion

- Theoretical foundation for cyclic dependencies ✓
  - Provided proper proof of intuitive solution ✓
- Added circularity checks for model methods ✓
- Proposal for solving soundness issues without undue restrictions ✓
- Begin work on improved proof management

Conclusion

- Theoretical foundation for cyclic dependencies ✓
  - Provided proper proof of intuitive solution ✓
- Added circularity checks for model methods ✓
- Proposal for solving soundness issues without undue restrictions ✓
- Begin work on improved proof management
  - Introduce KeY projects

Conclusion

- Theoretical foundation for cyclic dependencies ✓
  - Provided proper proof of intuitive solution ✓
- Added circularity checks for model methods ✓
- Proposal for solving soundness issues without undue restrictions ✓
- Begin work on improved proof management
  - Introduce KeY projects

Overall: Improve correctness of KeY and increase trust in proofs

Conclusion

- Theoretical foundation for cyclic dependencies ✓
  - Provided proper proof of intuitive solution ✓
- Added circularity checks for model methods ✓
- Proposal for solving soundness issues without undue restrictions ✓
- Begin work on improved proof management
  - Introduce KeY projects

Overall: Improve correctness of KeY and increase trust in proofs

Thank you for your attention!