Protocol Conformance Checking – Steps Towards Practical Applicability

Andreas Both (speaker), Wolf Zimmermann

Institute for Computer Science
University of Halle, Germany
http://swt.informatik.uni-halle.de/

December 10, 2008
Summary of Yesterday

Problems with Non-Conservative Approximation

- if behaviour is not representable, False Positives are possible
- parallel and sequential behaviour

⇒ Usable, if difficult parts are not captured?

Our Solution

- use PRS as more powerful representation of behaviour
- contain fundamental concepts like recursion and parallelism in one representation

⇒ ensure conservative abstraction of source code

- not Turing powerful

⇒ model checking possible
Summary of Yesterday

Problems with Non-Conservative Approximation

- if behaviour is not representable, False Positives are possible
- parallel and sequential behaviour
  ⇒ Usable, if difficult parts are not captured?

Our Solution

- use PRS as more powerful representation of behaviour
- contain fundamental concepts like recursion and parallelism in one representation
  ⇒ ensure conservative abstraction of source code
- not Turing powerful
  ⇒ model checking possible
Summary of Yesterday: Full Process

- single component
- create abstraction (as PAN)
- single component abstractions
- create system abstraction
- system abstraction
- combine abstraction and protocol
- Combined Abstraction
- model checking
- counter-example(s)

Properties of the process

- implementable for OO-programming languages
- automatic process
- protocol can be defined easily

⇒ developer has to know his component only

Usage

- if his component fits into system
- which component works with his component
Summary of Yesterday: Full Process

- single component
  - create abstraction (as PAN)
- single component abstractions
  - create system abstraction
- system abstraction
  - combine abstraction and protocol
- Combined Abstraction
  - model checking
  - counter-example(s)

Properties of the process

- implementable for OO-programming languages
- automatic process
- protocol can be defined easily
  - developer has to know his component only

Usage

- if his component fits into system
- which component works with his component
Properties of the process

- implementable for OO-programming languages
- automatic process
- protocol can be defined easily

⇒ developer has to know his component only

Usage

- if his component fits into system
- which component works with his component
Model Checking is Hard to Solve

Problems
- model checking is hard to solve
- abstractions could grow huge
- many False Negatives blur the real problems in the source code

Yet another approach that is not applicable.
Is this approach at all applicable?

Our Answer
(probably) Yes.
Model Checking is Hard to Solve

Problems
- model checking is hard to solve
- abstractions could grow huge
- many False Negatives blur the real problems in the source code

Yet another approach that is not applicable.
Is this approach at all applicable?

Our Answer
(probably) Yes.
Model Checking is Hard to Solve

Problems

- model checking is hard to solve
- abstractions could grow huge
- many False Negatives blur the real problems in the source code

Yet another approach that is not applicable.

Is this approach at all applicable?

Our Answer

(probably) Yes.
Applicability of PA

Properties
- no synchronization
- model checking much faster

Discussion
- often parallel join operator is dispensable (except: BPEL)
- thus only relevant if interaction in min. 2 parallel threads
  ⇒ happens not often
  ⇒ use PA, if no join-operator is present

Applicability of PA

Properties
- no synchronization
- model checking much faster

Discussion
- often parallel join operator is dispensable (except: BPEL)
- thus only relevant if interaction in min. 2 parallel threads
  ⇒ happens not often
  ⇒ use PA, if no join-operator is present

Why using PAN?

Pro of PA
- can find every error in Combined Abstraction
- faster model checking

Contra of PA
- PA creates much more spurious False Negatives

Andreas Both, Wolf Zimmermann
Why using PAN?

Pro of PA
- can find every error in Combined Abstraction
- faster model checking

Contra of PA
- PA creates much more spurious False Negatives

Andreas Both, Wolf Zimmermann
func main() {
    fork(V,W);
    ...
    join(V,W);
    ...
    c(...);
}

func c(...) {
    if (...) {
        c();
        t();
    }
}

Process Algebra
- 2 threads: $U \xrightarrow{\lambda} V || W$
- 1. Thread call $x()$: $V \xrightarrow{x} \varepsilon$
- 2. Thread call $b()$: $W \xrightarrow{b} S$
- no join: $S \xrightarrow{\lambda} Q$
- can terminate: $Q \xrightarrow{\lambda} \varepsilon$
- recursion: $Q \xrightarrow{c} Q.R$

CE of PA
- $x\ b\ c^n\ t^n$
- $b\ x\ c^n\ t^n$
- $b\ c^i\ x\ c^{n-i}\ t^n$
- ...

CE of PAN
- $x\ b\ c^n\ t^n$
- $b\ x\ c^n\ t^n$
Example: Problems with PA-Abstractions (Combined Abstraction)

```
func main() {
    for (V,W) {
        ...
        join (V,W);
        ...
        c (...);
    }
}

func c(...) {
    if (...) {
        c();
        t();
    }
}
```

Process Algebra

- 2 threads: $U \xrightarrow{\lambda} V || W$
- 1. Thread call $x()$: $V \xrightarrow{x} \varepsilon$
- 2. Thread call $b()$: $W \xrightarrow{b} S$
- no join: $S \xrightarrow{\lambda} Q$
- can terminate: $Q \xrightarrow{\lambda} \varepsilon$
- recursion: $Q \xrightarrow{c} Q.R$

CE of PA

$x b c^n t^n$
$b x c^n t^n$
$c^i x c^{n-i} t^n$
...

CE of PAN

$x b c^n t^n$
$b x c^n t^n$
Example: Problems with PA-Abstractions (Combined Abstraction)

```go
func main() {
    for (V,W) {
        ...
    }
    join (V,W);
    ...
    c (...);
}

func c(...) {
    if (...) {
        c();
        t();
    }
}
```

Process Algebra

- 2 threads: \( U \xrightarrow{\lambda} V || W \)
- 1. Thread call \( x() : V \xrightarrow{x} \varepsilon \)
- 2. Thread call \( b() : W \xrightarrow{b} S \)
- no join: \( S \xrightarrow{\lambda} Q \)
- can terminate: \( Q \xrightarrow{\lambda} \varepsilon \)
- recursion: \( Q \xrightarrow{c} Q.R \)

CE of PA

- 1. \( x b c^n t^n \)
- 2. \( b x c^n t^n \)
- \( b c^i x c^{n-i} t^n \)

CE of PAN

- \( x b c^n t^n \)
- \( b x c^n t^n \)

...
Example: Problems with PA-Abstractions (Combined Abstraction)

```go
func main() {
    fork(V, W);
    ...
    join(V, W);
    ...
    c(...);
}

func c(...) {
    if (...) {
        c();
        t();
    }
}
```

Process Algebra Net

- 2 threads: $U \xrightarrow{\lambda} V || W$
- 1. Thread call $x()$: $V \xrightarrow{x} P$
- 2. Thread call $b()$: $W \xrightarrow{b} S$
- with join: $S || P \xrightarrow{\lambda} Q$
- can terminate: $Q \xrightarrow{\lambda} \epsilon$
- recursion: $Q \xrightarrow{c} Q.R$

CE of PA

```text
x b c^n t^n
b x c^n t^n
b c^i x c^{n-i} t^n
...```

CE of PAN

```text
x b c^n t^n
b x c^n t^n
...```
Example: Problems with PA-Abstractions (Combined Abstraction)

```go
func main() {
    fork(V, W);
    ...
    join(V, W);
    ...
    c(...);
}

func c(...) {
    if (...) {
        c();
        t();
    }
}
```

**Process Algebra Net**

- 2 threads: \( U \xrightarrow{\lambda} V || W \)
- 1. Thread call \( x(): V \xrightarrow{x} P \)
- 2. Thread call \( b(): W \xrightarrow{b} S \)
- with join: \( S || P \xrightarrow{\lambda} Q \)
- can terminate: \( Q \xrightarrow{\lambda} \epsilon \)
- recursion: \( Q \xrightarrow{c} Q.R \)

**CE of PA**

\[
x b c^n t^n \\
b x c^n t^n \\
b c^i x c^{n-i} t^n \\
... \\
\]

**CE of PAN**

\[
x b c^n t^n \\
b x c^n t^n \\
... \\
\]
Applicability of PAN

- using a PAN results less spurious False Negatives (counter examples)
- easier to filter relevant counter examples
- easier to see the behaviour

Disadvantage

- contains reachability problem of Petri nets (EXP-Space hard)
  \[ \Rightarrow \text{reduction of problem size very important} \]

using Process Algebras as preparation and Process Algebra Nets for exact counterexamples

Applicability of PAN

- using a PAN results less spurious False Negatives (counter examples)
- easier to filter relevant counter examples
- easier to see the behaviour

Disadvantage

- contains reachability problem of Petri nets (EXP-Space hard)
  \[ \Rightarrow \text{reduction of problem size very important} \]

Applicability of PAN

- using a PAN results less spurious False Negatives (counter examples)
- easier to filter relevant counter examples
- easier to see the behaviour

Disadvantage

- contains reachability problem of Petri nets (EXP-Space hard)
  \[ \Rightarrow \text{reduction of problem size very important} \]

using Process Algebras as preparation and Process Algebra Nets for exact counterexamples

Discussion

- PRS allow us to choose the right representation dynamically
  ⇒ best algorithm selectable

Note: Reachability for PAN
- no algorithm implemented
- Petri net solver as sub process
  ⇒ currently implementing multi-threaded approach
Discussion

- PRS allow us to choose the right representation dynamically
  ⇒ best algorithm selectable

Note: Reachability for PAN
- no algorithm implemented
- Petri net solver as sub process
  ⇒ currently implementing multi-threaded approach
Discussion

- PRS allow us to choose the right representation dynamically
  ⇒ best algorithm selectable

Note: Reachability for PAN
- no algorithm implemented
- Petri net solver as sub process
  ⇒ currently implementing multi-threaded approach
Problem

- abstractions could grow huge

Ideas for Dealing with Huge Abstractions

- use compiler construction technology to create a more concrete abstraction
- reduce the size of the PRS
- implement better model checking algorithm

Andreas Both, Wolf Zimmermann

PCC – Steps Towards Practical Applicability
**Problem**

- abstractions could grow huge

**Ideas for Dealing with Huge Abstractions**

- use compiler construction technology to create a more concrete abstraction
- reduce the size of the PRS
- implement better model checking algorithm
Problem

- abstractions could grow huge

Ideas for Dealing with Huge Abstractions

- use compiler construction technology to create a more concrete abstraction
- reduce the size of the PRS
- implement better model checking algorithm
Reducing the Size of PRS

During Abstraction Process

- we ignore the data flow
- consider only control flow
  ⇒ use dataflow analysis to predict state explosion
  ⇒ reduce many rules in abstraction
Reducing the Size of PRS

During Abstraction Process

- we ignore the data flow
- consider only control flow
  ⇒ use dataflow analysis to predict state explosion
  ⇒ reduce many rules in abstraction
Reducing the Size of PRS: Example, Program Analysis

\[ y = 7; \]
\[ \ldots \]
\[ x = y; \]
\[ \ldots \]
\[ \text{switch } x: \]
\[ \text{case 3: } a(); \text{ end; } \]
\[ \text{case 4: } b(); \text{ end; } \]
\[ \text{case 5: } \ldots ; \text{ end; } \]
\[ \text{case 7: } c(); \text{ end; } \]
\[ \text{other: } \ldots ; \text{ end; } \]
\[ \ldots \]

Note

Optimizations only for one component.
Reducing the Size of PRS: Example, Program Analysis

\[
y = 7; \\
\ldots \\
x = y; \\
\ldots \\
\text{switch } x:\ \\
\quad \text{case 3: } a(); \text{ end; } \\
\quad \text{case 4: } b(); \text{ end; } \\
\quad \text{case 5: } \ldots; \text{ end; } \\
\quad \text{case 7: } c(); \text{ end; } \\
\quad \text{other: } \ldots; \text{ end; } \\
\ldots 
\]

Note
Optimizations only for one component.
Reducing the Size of PRS: Example, Program Analysis

\[ y = 7; \]
\[ \ldots \]
\[ x = y; \]
\[ \ldots \]
\[ \text{switch } x:\]
  \[ \text{case } 3: \ a(); \ \text{end}; \]
  \[ \text{case } 4: \ b(); \ \text{end}; \]
  \[ \text{case } 5: \ \ldots; \ \text{end}; \]
  \[ \text{case } 7: \ c(); \ \text{end}; \]
  \[ \text{other : } \ldots; \ \text{end}; \]
\[ \ldots \]

Note

Optimizations only for one component.
Reducing the Size of PRS: Example, Program Analysis

```plaintext
y = 7;
...
x = y;
...
switch x:
case 3: a(); end;
case 4: b(); end;
case 5: ...; end;
case 7: c(); end;
other: ...; end;
...
```

Note

Optimizations only for one component.
Reducing the Size of PRS: Example, Lower Complexity

class U(Thread):
    def start:
        x();

class V(Thread):
    def start:
        y();

class W(Thread):
    def start:
        calculations;

Andreas Both, Wolf Zimmermann
Reducing the Size of PRS: Example, Lower Complexity

class U(Thread):
    def start:
        x();

class V(Thread):
    def start:
        y();

class W(Thread):
    def start:
        calculations;

Andreas Both, Wolf Zimmermann
Reducing the Size of PRS: Example, Lower Complexity

class U(Thread):
def start:
    x();

class V(Thread):
def start:
    y();

class W(Thread):
def start:
calculations;
Reducing the Model Size

- static program analysis
  - Problem: different programming language
  - optimization of created PRS

Note: Optimization of Abstraction and Combined Abstraction
- only Combined Abstraction needed
- industrial partner uses representations
Reducing the Model Size

- static program analysis
  - Problem: different programming language
- optimization of created PRS

Note: Optimization of Abstraction and Combined Abstraction

- only Combined Abstraction needed
- industrial partner uses representations
Discussion

Reducing the Model Size

- static program analysis
  - Problem: different programming language
- optimization of created PRS

Note: Optimization of Abstraction and Combined Abstraction

- only Combined Abstraction needed
- industrial partner uses representations
Reducing the Model Size

- static program analysis
  - Problem: different programming language
- optimization of created PRS

Note: Optimization of Abstraction and Combined Abstraction

- only Combined Abstraction needed
- industrial partner uses representations
### Case Study

**Partner:** OR Soft Jänicke GmbH, Merseburg, Germany

- Implement comfort GUI based on SAP
- Powerful tool for planners of industrial processes

**Size of Considered Source Code**

- > 300 C++ components, > 1000 Python components
- Composed to different products

**Behaviour of Components**

- No restrictions on behaviour
- Communication: native interfaces, boost, XML-RPC (and Web Services)
Case Study

Partner: OR Soft Jänicke GmbH, Merseburg, Germany
- implement comfort GUI based on SAP
- powerful tool for planners of industrial processes

Size of Considered Source Code
- > 300 C++ components, > 1000 Python components
- composed to different products

Behaviour of Components
- no restrictions on behaviour
- communication: native interfaces, boost, XML-RPC (and Web Services)
Scopes of Application

Orchestration

- verify if components are used in the defined manner (e.g. Web Service orchestration)
  1. static binding: can we build a program like this
  2. dynamic binding (components out of a finite set): which components can be used to ensure the behaviour
  3. fully dynamic binding: can not be handled

Many more

...
Case Study: Implemented Architecture

3 Web Services *(still in development)*

A creating abstractions and Combined Abstraction
B verifying Combined Abstraction
C GUI: create scenarios, control verification process
Case Study: Implemented Architecture
Case Study: Implemented Architecture

Andreas Both, Wolf Zimmermann
Early Results of Industrial Case Study

Observations during Creation of Abstractions

- creating abstractions of C++ and Python source code is very difficult
- risk: explosion of state space
- using program analyses lead to much smaller abstractions
  ⇒ worthy to invest in program analyses

Observations on Protocols

Observations on Combined Abstraction
Observations during Creation of Abstractions

Observations on Protocols

- protocol reduce the complexity of Combined Abstraction in contrast to abstraction
- not as large as expected
  ⇒ many $\lambda$-rules, good reduction possible
  ⇒ approach leading to smaller model size

Observations on Combined Abstraction
Early Results of Industrial Case Study

Observations during Creation of Abstractions

Observations on Protocols

Observations on Combined Abstraction

- get a good reduction rate of abstractions and Combined Abstractions
- reduce the problem to a lower class

⇒ **worthy** to invest in optimizations of Combined Abstraction
Conclusions

- protocol for the receivable interaction of *one* component is very useful
- verification for only one component can be improved easily
- even at dynamically chosen components
- found errors
- displace the responsibility to user of component
- "educational" effect

(Early) Conclusion
approach is applicable
Conclusions

- protocol for the receivable interaction of one component is very useful
- verification for only one component can be improved easily
- even at dynamically chosen components
- found errors
- displace the responsibility to user of component
- "educational" effect

(Early) Conclusion

approach is applicable
Thank you for your attention.

Protocol Conformance Checking – Steps Towards Practical Applicability

Andreas Both (speaker), Wolf Zimmermann

Institute for Computer Science
University of Halle, Germany
http://swt.informatik.uni-halle.de/

December 10, 2008
Often academic approaches do not make it to industrial application for several reasons. We are very interested in verifying and improving our approach in an industrial setting. In this talk we will see, how we can use the approach presented in the first talk to solve real problems in component-based systems. To show this, we will give an overview of several properties and optimizations of the defined verification process, we are working on. Last we will show the corner marks of the case study we currently perform with our industrial partner and early results.

maximum time: 30 minutes