Algorithmic Verification Techniques for Mobile Code

Irem Aktug

CSC KTH
Stockholm, Sweden

KTH Computer Science and Communication

Mobile Code

Applications obtained from an untrusted source to be executed locally

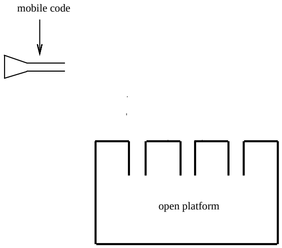


Figure: Mobile Code Joining Platform

- Java Web Applets
- Code embedded in Microsoft Office documents
- Java Card Applets (smart card applications)
- Java Midlets (mobile phone applications)

Mobile Code

Applications obtained from an untrusted source to be executed locally

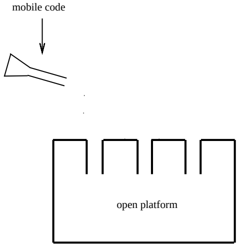


Figure: Mobile Code Joining Platform

- Java Web Applets
- Code embedded in Microsoft Office documents
- Java Card Applets (smart card applications)
- Java Midlets (mobile phone applications)

Mobile Code

Applications obtained from an untrusted source to be executed locally

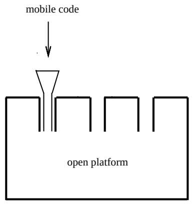


Figure: Mobile Code Joining Platform

- Java Web Applets
- Code embedded in Microsoft Office documents
- Java Card Applets (smart card applications)
- Java Midlets (mobile phone applications)

Two Correctness Problems

Main Problem: How do we check that the system composed with the mobile component works correctly?

Problem 1

- Choose a reasonable property P' for the mobile component
- Build a generic model for P'
- Check if the system composed with this generic model has property P

Problem 2
Check if the mobile component has property P' at time of join.

Part I

(Problem 1)

Part I: State Space Representation

We develop a representation for the state space of open systems to facilitate visualization and verification.

Part I: State Space Representation

We develop a representation for the state space of open systems to facilitate visualization and verification.

The Setting

- Implementations are given as processes in a process algebra.
- Assumptions on components and open system properties are given in modal μ-calculus.

Contribution: *OTA*

We specify open systems with *open process terms with assumptions*.

Contribution: *OTA*

We specify open systems with *open process terms with assumptions*.

Example

- The platform can only do a single *a* action.
- Only one mobile component can execute in the system at a given time.
- The mobile component runs in parallel with the platform.
- The mobile component can only perform a finite number of *a* actions.

Contribution: *OTA*

We specify open systems with *open process terms with assumptions*.

Example

- The platform can only do a single a action.
- Only one mobile component can execute in the system at a given time.
- The mobile component runs in parallel with the platform.
- The mobile component can only perform a finite number of a actions.

$$a.0$$

Contribution: *OTA*

We specify open systems with *open process terms with assumptions*.

Example

- The platform can only do a single a action.
- Only one mobile component can execute in the system at a given time.
- The mobile component runs in parallel with the platform.
- The mobile component can only perform a finite number of a actions.

$$a.0 \quad C$$

Contribution: *OTA*

We specify open systems with *open process terms with assumptions*.

Example

- The platform can only do a single a action.
- Only one mobile component can execute in the system at a given time.
- The mobile component runs in parallel with the platform.
- The mobile component can only perform a finite number of a actions.

$$a.0 \parallel C$$

Contribution: *OTA*

We specify open systems with *open process terms with assumptions*.

Example

- The platform can only do a single *a* action.
- Only one mobile component can execute in the system at a given time.
- The mobile component runs in parallel with the platform.
- The mobile component can only perform a finite number of *a* actions.

$$C : \mu Z.\, [a]\, Z \triangleright a.0 \parallel C$$

Contribution: *EMTS*

We represent the state space of open systems with *extended model transition systems*.

Special features of EMTSs:

- Two types of transitions for the two modalities of the logic
- Coloring of states for fairness constraints

Contribution: *EMTS*

We represent the state space of open systems with *extended model transition systems*.

Special features of EMTSs:

- Two types of transitions for the two modalities of the logic
- Coloring of states for fairness constraints

Example

EMTS for the open system $C : \mu Z.\,[a]\,Z \triangleright a.0 \parallel C$

Contribution: *EMTS*

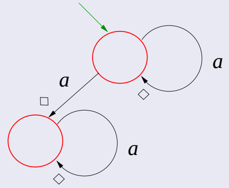
We represent the state space of open systems with *extended model transition systems*.

Special features of EMTSs:

- Two types of transitions for the two modalities of the logic
- Coloring of states for fairness constraints

Example

EMTS for the open system $C : \mu Z.\, [a]\, Z \triangleright a.0 \parallel C$



Contribution: Characteristic Model Construction for Modal μ-calculus

We define a construction that maps the formulae of the logic to EMTSs.
The construction is defined recursively on the structure of the formula.

Contribution: Characteristic Model Construction for Modal μ-calculus

We define a construction that maps the formulae of the logic to EMTSs. The construction is defined recursively on the structure of the formula.

Example

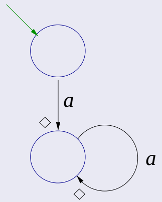
The mobile component can only do a finite number of a's, i.e. it has the property $\mu Z . [a] Z$

Contribution: Characteristic Model Construction for Modal μ-calculus

We define a construction that maps the formulae of the logic to EMTSs. The construction is defined recursively on the structure of the formula.

Example

The mobile component can only do a finite number of *a*'s, i.e. it has the property $\mu Z .\, [a]\, Z$



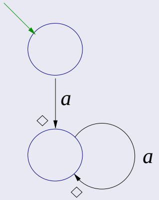
$\varepsilon(Z)$

Contribution: Characteristic Model Construction for Modal μ-calculus

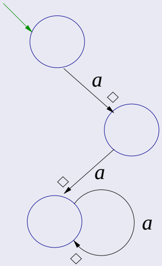
We define a construction that maps the formulae of the logic to EMTSs. The construction is defined recursively on the structure of the formula.

Example

The mobile component can only do a finite number of *a*'s, i.e. it has the property $\mu Z.\,[a]\,Z$



$\varepsilon(Z)$



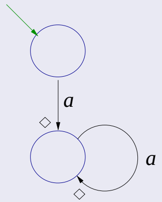
$\varepsilon([a]\,Z)$

Contribution: Characteristic Model Construction for Modal μ-calculus

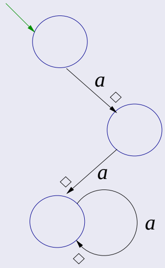
We define a construction that maps the formulae of the logic to EMTSs. The construction is defined recursively on the structure of the formula.

Example

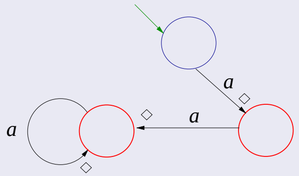
The mobile component can only do a finite number of *a*'s, i.e. it has the property $\mu Z.\,[a]\,Z$



$\varepsilon(Z)$



$\varepsilon([a]\,Z)$



$\varepsilon(\mu Z.\,[a]\,Z)$

Correctness of Construction I

Characteristic Model Property

Given a property $\mathcal{P}$, the EMTS constructed simulates exactly those processes that have property $\mathcal{P}$.

Contribution: Model Construction for OTA

The construction is defined recursively on the structure of the process term with assumptions.
Uses characteristic model of the given property for mobile components.

Contribution: Model Construction for OTA

The construction is defined recursively on the structure of the process term with assumptions.
Uses characteristic model of the given property for mobile components.

Example

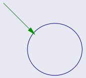
$$C : \nu Z.\, [a]\, Z \triangleright a.0 \parallel C$$

Contribution: Model Construction for OTA

The construction is defined recursively on the structure of the process term with assumptions.
Uses characteristic model of the given property for mobile components.

Example

$$C : \nu Z.\, [a]\, Z \triangleright a.0 \parallel C$$



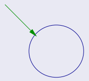
$\varepsilon(0)$

Contribution: Model Construction for OTA

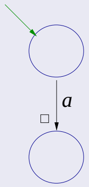
The construction is defined recursively on the structure of the process term with assumptions.
Uses characteristic model of the given property for mobile components.

Example

$$C : \nu Z . [a] Z \triangleright a.0 \parallel C$$



$\varepsilon(0)$



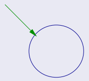
$\varepsilon(a.0)$

Contribution: Model Construction for OTA

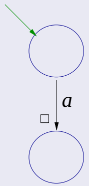
The construction is defined recursively on the structure of the process term with assumptions.
Uses characteristic model of the given property for mobile components.

Example

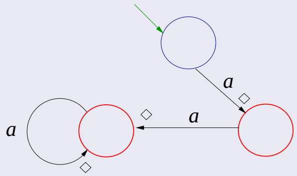
$$C : \nu Z.\,[a]\,Z \triangleright a.0 \parallel C$$



$\varepsilon(0)$



$\varepsilon(a.0)$



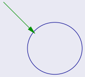
$\varepsilon(\mu Z.\,[a]\,Z)$

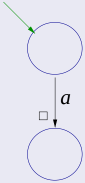
Contribution: Model Construction for OTA

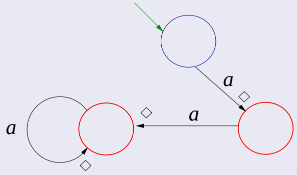
The construction is defined recursively on the structure of the process term with assumptions.
Uses characteristic model of the given property for mobile components.

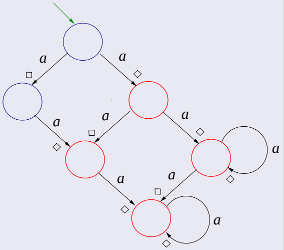
Example

$$C : \nu Z.\,[a]\,Z \triangleright a.0 \parallel C$$









$\varepsilon(0)$ $\varepsilon(a.0)$ $\varepsilon(\mu Z.\,[a]\,Z)$

Correctness of Construction II

Model Construction Correctness

1. Given an open process term with assumptions with a single mobile component, the constructed EMTS simulates exactly those closed systems specified by the OTA, as long as the OTA does not have dynamic process creation.
2. Given an open process term with assumptions, the constructed EMTS simulates all closed systems specified by the OTA (and possibly more), as long as the OTA does not have dynamic process creation.

Proving Properties of Open Systems

We prove modal μ-calculus properties of open systems using a proof system.

Proving Properties of Open Systems

We prove modal μ-calculus properties of open systems using a proof system.

Proof System Properties

The proof system is sound in general and complete for prime formulae of the logic.

Publications

1. I. Aktug and D. Gurov, "Towards State Space Exploration Based Verification of Open Systems" to appear in Proceedings of the 4^{th} International Workshop on Automated Verification of Infinite-State Systems (AVIS'05), April 2005, Edinburgh, Scotland
2. I. Aktug and D. Gurov, "State Space Representation for Verification of Open Systems", in Proceedings of the 11^{th} International Conference on Algebraic Methodology and Software Technology (AMAST '06), volume 4019 of Lecture Notes in Computer Science, pages 5-20, July 2006, Kuressaare, Estonia

Part II

(Problem 1)

Part II: Program Models

We extend a compositional verification framework for handling exceptional and multi-threaded behavior.

Part II: Program Models

We extend a compositional verification framework for handling exceptional and multi-threaded behavior.

The Setting

- Implementations are given as Java bytecode programs.
- Assumptions on components and open system properties are given in a fragment of modal μ-calculus.

Compositional Verification Framework

Developed by Gurov, Huisman and Sprenger

- Structure: control-flow graphs with procedures, without data
- Behavior: a set of (possibly infinite-length) executions induced from the program structure
- Structure extraction from bytecode: Defined, Implemented using SOOT.
- Structural and behavioral properties of programs
- Structural properties can be checked using model checking
- Behavioral properties can be checked using PDA-based model checking

Contributions

Exceptional Control Flow

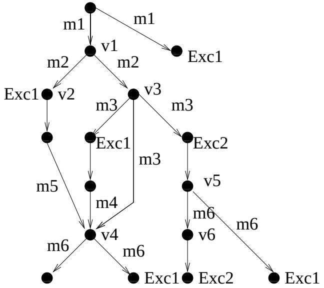
- Structure: Extended with a set of exceptions, such that control points of the flow graph may be labeled with an exception.
- Behavior: Throws and catches are reflected.
- Structure extraction from bytecode: Defined, Implemented by extending the extraction tool for basic model.

Contributions

Exceptional Control Flow

- Structure: Extended with a set of exceptions, such that control points of the flow graph may be labeled with an exception.
- Behavior: Throws and catches are reflected.
- Structure extraction from bytecode: Defined, Implemented by extending the extraction tool for basic model.

```
m1();
try { m2();
      try { m3(); }
      catch Exc1 { m4(); }
    }
catch Exc1 { m5(); }
finally { m6(); }
```



Contributions

Exceptional Control Flow

- Structure: Extended with a set of exceptions, such that control points of the flow graph may be labeled with an exception.
- Behavior: Throws and catches are reflected.
- Structure extraction from bytecode: Defined, Implemented by extending the extraction tool for basic model.

Model With Multi-threading

- Structure: We extend the basic program model with a set of thread id's and lock names.
  We add new labels (e.g. spawn, lock, unlock, wait, notify).
- Behavior: We maintain a configuration per active thread, a lock map and a wait map.
- Structure extraction from bytecode: Defined.

Benefits

- More precise models: Exits due to uncaught exceptions are added
- Properties related to exceptional/multi-threaded behavior can be shown:
  - Exception e is never thrown
  - Exception e is always caught within the method where it is thrown
  - Method m can only be called by thread t, if t has lock l

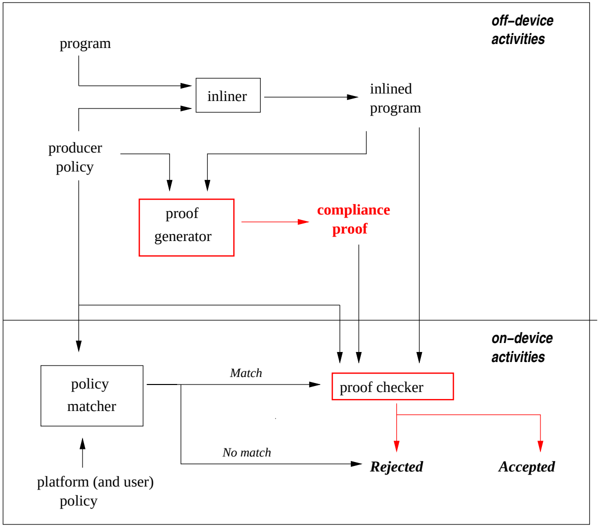
Publications

1. M. Huisman, I. Aktug and D. Gurov, “Program Models for Compositional Verification”, in Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM’08), volume 5256 of Lecture Notes in Computer Science, pages 147-166, October 2008, Kitakyushu-City, Japan

Part III

(Problem 2)

The Framework



Provably Correct Inlining

An inlined program is *correctly inlined* for policy $\mathcal{P}$, if the program adheres to the policy.
For a given inlined program, we want to create a proof of correct inlining such that:

- can be automatically generated,
- efficiently checkable.

Provably Correct Inlining

An inlined program is *correctly inlined* for policy $\mathcal{P}$, if the program adheres to the policy.
For a given inlined program, we want to create a proof of correct inlining such that:

- can be automatically generated,
- efficiently checkable.

The Setting

- Mobile component implementations are sequential Java bytecode programs.
- Our security relevant actions are calls to and returns from a fixed API.
- The policies (or properties) consist of a set of security relevant action sequences.

Contribution: The Policy Language ConSpec

- Adapted from PSLang of Erlingsson and Schneider
- Semantics given through security automata
- A restricted language to allow the formal treatment of several activities of the framework
  - Bounded domains for state variables to enable matching
  - Restricted language for state updates (e.g. no loops) for automatic proof generation

Policies

Example: ConSpec Language

"*After applications access an existing file, they are required to obtain approval from the user each time a connection is to be opened.*"

Policies

Example: ConSpec Language

"After applications access an existing file, they are required to obtain approval from the user each time a connection is to be opened."

```
SECURITY   STATE
           bool accessed = false;
           bool permission = false;

BEFORE     File.Open(string path, string mode, string access)
PERFORM    mode.equals(CreateNew)      → { skip; }
           !mode.equals(CreateNew) → { accessed = true; }

AFTER      bool answer = GUI.AskConnect()
PERFORM    answer → { permission = true; }
           !answer → { permission = false; }

BEFORE     Connection.Open(string type, string address)
PERFORM    !accessed || permission -> { permission = false; }
```

Contribution: Annotation Scheme for Specifying Correct Monitor Inlining

The target program is annotated in two steps:

Contribution: Annotation Scheme for Specifying Correct Monitor Inlining

The target program is annotated in two steps:

1. Level I: we insert a correct monitor into the program using specification variables.

Contribution: Annotation Scheme for Specifying Correct Monitor Inlining

The target program is annotated in two steps:

1. Level I: we insert a correct monitor into the program using specification variables.

2. Level II: we specify that an embedded state exists such that:
    - the embedded monitor is in "synch" with the specified monitor immediately prior to execution of a security relevant action, and
    - the updates to the embedded state are made locally, that is by the method that executes the security relevant method call.

Level I Annotations
Policy

"After applications access an existing file, they are required to obtain approval from the user each time a connection is to be opened."

```
SECURITY    STATE
            bool accessed = false;
            bool permission = false;

BEFORE      File.Open(string path, string mode, string access)
PERFORM     mode.equals(CreateNew)      → { skip; }
            !mode.equals(CreateNew) → { accessed = true; }

AFTER       bool answer = GUI.AskConnect()
PERFORM     answer → { permission = true; }
            !answer → { permission = false; }

BEFORE      Connection.Open(string type, string address)
PERFORM     !accessed || permission -> { permission = false; }
```

Level I Annotations

Target Program

L	$M[L]$
L1	`aload r0`
L2	`getfield gui`
L3	`dup`
L4	`astore r1`
L5	`invokevirtual GUI/AskConnect()Z`
L6	`istore r2`
L7	`aload r1`
L8	`instanceof GUI`
L9	`ifeq L12`
L10	`iload r2`
L11	`putstatic SecState/permission`
L12	`iload r2`
L13	`ireturn`

Figure: A target application method

Level I Annotations

Target Program

L	$M[L]$
L1	`aload r0`
L2	`getfield gui`
L3	`dup`
L4	`astore r1`
L5	`invokevirtual GUI/AskConnect()Z`
L6	`istore r2`
L7	`aload r1`
L8	`instanceof GUI`
L9	`ifeq L12`
L10	`iload r2`
L11	`putstatic SecState/permission`
L12	`iload r2`
L13	`ireturn`

Figure: A target application method

Level I Annotations

Example: Level I Annotations

$A^{\mathrm{I}}[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\{\mathit{Defined}(gs)\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{gs := \delta_{\perp}(gs, a)\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
	L13	`ireturn`

Figure: An application method with level I annotations for the example policy

Level I Annotations

Theorem: Correctness of Level I Annotations

Program T annotated with level I annotations for policy $\mathcal{P}$ is valid, if and only if T adheres to $\mathcal{P}$.

Level II Annotations

Example: Level I Annotations

$A^{\mathrm{I}}[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\{\mathit{Defined}(gs)\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{gs := \delta_{\perp}(gs, a)\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
	L13	`ireturn`

Figure: An application method with level I annotations for the example policy

Level II Annotations

Example: Level II Annotations

$A^{\mathrm{II}}[L]$	L	$M[L]$
$\{gs = \texttt{SecState}\}$	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\{Defined(gs) \wedge gs = \texttt{SecState}\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{gs := \delta_{\perp}(gs, a)\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
$\{gs = \texttt{SecState}\}$	L13	`ireturn`

Figure: An application method with level II annotations for the example policy

Level II Annotations

Theorem: Level II Characterization

The level II annotations of T for policy $\mathcal{P}$ with the embedded state $\overrightarrow{ms}$ is valid if, and only if, $\overrightarrow{ms}$ identifies a method-local monitor for $\mathcal{P}$.

Contribution: Proofs of Correct Inlining

If T is a "nicely" inlined program then level II annotations can be completed to full annotations using weakest precondition calculation on inlined blocks.
The full annotations can be used as the proof of correct inlining.
Full annotations generated by:

1. Adding the synchronization annotation as precondition to uninlined instructions
2. Propagating the synchronization annotation from the bottom to the top of the inlined blocks using a weakest precondition calculator

Full annotations are checked by checking the proof local validity:

- constructing verification conditions using the axiomatic semantics of single instructions
- discharging the resulting verification conditions

Correct Monitor Inlining

Level III ("Full") Annotations for the Inliner

Proof of correct inlining can be constructed for nicely inlined programs.

Correct Monitor Inlining

Level III ("Full") Annotations for the Inliner

Proof of correct inlining can be constructed for nicely inlined programs.

A program is "nicely" inlined if

- the problem of computing the weakest precondition of inlined blocks is decidable,
- the problem of discharging the verification conditions arising from the local validity of the full annotations is decidable

Benefits

- The annotation scheme facilitates generation of correct inlining proof. Proof generation and check is efficient.
  Such a proof can be used in a proof-carrying code setting for certifying policy compliance to the platform.
- The annotation scheme can be used to show correctness of an inliner.

Publications

1. I. Aktug and K. Naliuka, “ConSpec: A Formal Language for Policy Specification”, in Proceedings of The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM’07), volume 197-1 of Electronic Notes in Theoretical Computer Science, pages 45-58, September 2007, Dresden, Germany
   Full version accepted for publication in *Science of Computer Programming*
2. I. Aktug, M. Dam and D. Gurov, “Provably Correct Runtime Monitoring”, in the Proceedings of the 15^{th} International Symposium on Formal Methods (FM ’08), volume 5014 of Lecture Notes in Computer Science, pages 262-277, May 2008, Turku, Finland,
   Full version accepted for publication in *Journal of Logic and Algebraic Programming*

Thank God It's Over Slide

Thank You!