Aalto University
School of Science

Object-Oriented Programs as Parametrised Systems

Antti Siirtola

Department of Information and Computer Science
Aalto University, School of Science
antti.siirtola@aalto.fi

Rich Model Toolkit COST Action Meeting, June 17th, 2013

Outline

Introduction

Model of Computation

Parametrised LTSs

Modelling Object-Oriented Programs

Verification Techniques for Parametrised Parts

Conclusions

Introduction

Formal Verification

Main Question

Does a given system implementation meet its specification?

Formal Verification

Main Question

Does a given system implementation meet its specification?

Formally

$\mathit{Sys} \preceq \mathit{Spec}$?

Formal Verification

Main Question

Does a given system implementation meet its specification?

Formally

$Sys \preceq Spec$?

Not natural to object-oriented programs

Formal Verification

Main Question

Does a given system implementation meet its specification?

Formally

$\mathit{Sys} \preceq \mathit{Spec}$?

Not natural to object-oriented programs

Why?

Object-Oriented Programs

OO Programs have many natural parameters:

- the number of concurrent threads,
- the number of replicated objects,
- the size of stack = the number of recursions.

Object-Oriented Programs

OO Programs have many natural parameters:

- the number of concurrent threads,
- the number of replicated objects,
- the size of stack = the number of recursions.

OO Programs are usually not representable as a single (finite-state) system!

Parametrised Verification

Main Question

Does a given parametrised system implementation meet its parametrised specification for every parameter value?

Parametrised Verification

Main Question

Does a given parametrised system implementation meet its parametrised specification for every parameter value?

Formally

$\mathit{Sys}(i) \preceq \mathit{Spec}(i)$ for all $i \in I$?

Parametrised Verification

Main Question

Does a given parametrised system implementation meet its parametrised specification for every parameter value?

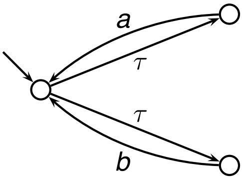
Formally

$Sys(i) \preceq Spec(i)$ for all $i \in I$?

Natural to object-oriented programs!

Model of Computation

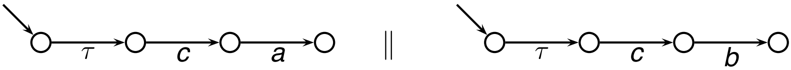
Labelled Transition System



$$\Sigma = \{a, b\}$$

Parallel Composition

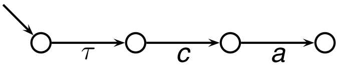
Hoare-style alphabet-based synchronisation



$\Sigma = \{a, c\}$ $\qquad\qquad$ $\Sigma = \{b, c\}$

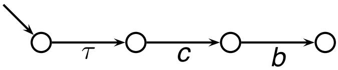
Parallel Composition

Hoare-style alphabet-based synchronisation



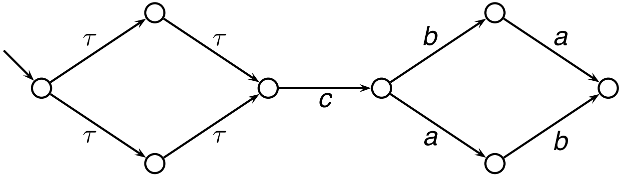
$\Sigma = \{a, c\}$

$\|$



$\Sigma = \{b, c\}$

$=$



$\Sigma = \{a, b, c\}$

Parametrised LTSs

Parametrised LTS

Parametrised action: $c(x_1, \ldots, x_k)$

- c is a channel
- $x_1, \ldots, x_k$ are variables over parametrised types $T_1, \ldots, T_k$
- $T_1, \ldots, T_n$ represent arbitrary large sets

Parametrised LTS (PLTS)

- An LTS with parametrised actions is a PLTS
- There are also other PLTSs

Parametrised Parallel Composition

Syntax: $\| \, x : P$

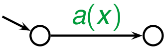
- x is a variable of a type T
- $T = \{t0, t1, \ldots, tn\}$ where $n \in \mathbb{Z}_+$
- P is a parametrised LTS

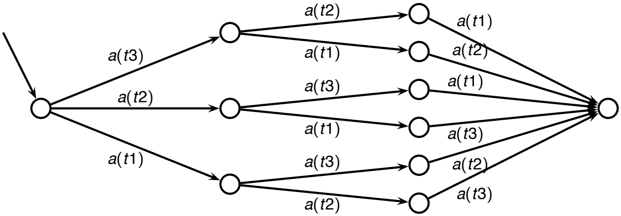
Parametrised Parallel Composition

Syntax: $\parallel x : P$

- x is a variable of a type T
- $T = \{t0, t1, \ldots, tn\}$ where $n \in \mathbb{Z}_+$
- P is a parametrised LTS

Instance of ($\parallel x : P$) when $n = 3$ and $P =$





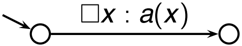
Parametrised Choice

Applied to transitions within parametrised LTSs

Parametrised Choice

Applied to transitions within parametrised LTSs

Example

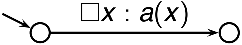
Parametrised LTS: 

Parametrised Choice

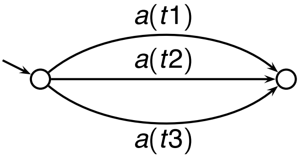
Applied to transitions within parametrised LTSs

Example

Parametrised LTS:



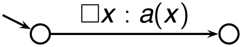
Instance when $n = 3$:

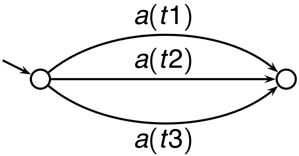


Parametrised Choice

Applied to transitions within parametrised LTSs

Example

Parametrised LTS: 

Instance when $n = 3$: 

PLTS Verification

Theorem (Siirtola & Heljanko 2013)

Trace inclusion problem of two PLTSs, where the implementation PLTS may involve hiding, is decidable when

- *no two variables of the same type occur both in a parametrised parallel composition and a parametrised choice and*
- *the specification PLTS is deterministic.*

Otherwise, the problem is undecidable.

PLTS Verification

Theorem (Siirtola & Heljanko 2013)

Trace inclusion problem of two PLTSs, where the implementation PLTS may involve hiding, is decidable when

- *no two variables of the same type occur both in a parametrised parallel composition and a parametrised choice and*
- *the specification PLTS is deterministic.*

Otherwise, the problem is undecidable.

The technique is implemented in our Bounds tool.

Modelling Object-Oriented Programs

Specifications

Concurrency related properties

- Mutual exclusion, deadlocks, . . .
- Control flow and references to objects must be preserved
- Data can be abstracted away

Case: Shared Resource System (SRS)

- An arbitrary number of Resource objects is created
- An arbitrary number of User threads is started
- The users should access a resource in a mutually exclusive way

Case: Shared Resource System (SRS)

- An arbitrary number of Resource objects is created
- An arbitrary number of User threads is started
- The users should access a resource in a mutually exclusive way

```
class User runnable {
    run: () => () {
        # repeat arbitrarily long
        while(true) {
            # pick a resource
            Resource r;
            while(r = null){
                r := SRS.getFirstRes();
                while(true){
                    r := r.getNext();
                }until(true);
            }
            # lock the resource
            synchronised(r){
                # use the resource
                while(true){
                    r.use();
                }until(true);
            }
```

```
class Resource {
    # pointer to the next resource
    Resource nxt;
    # abstract use method
    # does not change the structure
    use: () => () const(nxt);
    # returns the next resource
    getNext: (Resource) => (){
        return(nxt);
    }
    # sets the next resource
    setNext: () => (Resource r){
        nxt := r;
        return();
    }
}
```

Parameters

- A finite set of thread ids (for each runnable class)
  - *n* User-threads *User1*, ..., *Usern* plus the main thread *SRS*

Parameters

- A finite set of thread ids (for each runnable class)
  - *n* User-threads *User1*, $\ldots$, *Usern* plus the main thread *SRS*
- A finite set of objects ids (for each class)
  - *m* Resource objects *Res1*, $\ldots$, *Resm*

Parameters

- A finite set of thread ids (for each runnable class)
  - *n* User-threads *User1*, $\ldots$, *Usern* plus the main thread *SRS*
- A finite set of objects ids (for each class)
  - *m* Resource objects *Res1*, $\ldots$, *Resm*
- A finite ordered set of stack positions
  - Only two levels of nested method calls in the example
  - Two stack positions: $s1 < s2$

Actions

Each method call/operation is represented as two actions

Example

Method call: $o.setValue(v)$

Actions: $begSetValue(t, s, o, v)$

$endSetValue(t, s, o, v)$

- t is the id of the thread that makes the call
- s denotes the next stack position

LTSs I

A control flow LTS *Ctrl*(u, s) for each runnable object u and stack position s

```
lts
    I = begUserRun(u,s1) -> User72
    User72 = [true] tau -> User76(rNull)
       [] [true] tau -> User99
    User76(r) = [r = rNull] tau -> User78(r)
       [] [!r = rNull] tau -> User88(r)
    User78(r) = begSrsGetFirstRes(u,s2,m) -> User78e(r)
    User78e(r) = []r: endSrsGetFirstRes(u,s2,m,r) -> User79(r)
    User79(r) = [true] tau -> User81(r)
       [] [true] tau -> User76(r)
    User81(r) = begResGetNext(u,s2,r) -> User81e(r)
    User81e(r) = []r2: endResGetNext(u,s2,r,r2) -> User79(r2)
    User88(r) = resSync(u,s1,r) -> User91(r)
    User91(r) = [true] tau -> User93(r)
       [] [true] tau -> User96(r)
    User93(r) = begResUse(u,s2,r) -> User93b(r)
    User93b(r) = endResUse(u,s2,r) -> User91(r)
    User96(r) = resUnsync(u,s1,r) -> User72(r)
    User99 = endUserRun(u,s1) -> STP
from I
```

LTSs II

An LTS *MemVar*(r, v_r) for each member variable v_r of each object r

```
lts
    I(r2) = []u,s: begResNxtGet(u,s,r) -> I(r2)
        [] []u,s: endResNxtGet(u,s,r,r2) -> I(r2)
        [] []u,s,r2: begResNxtSet(u,s,r,r2) -> I(r2)
        [] []u,s: endResNxtSet(u,s,r) -> I(r2)
from I(rNull)
```

LTSs II

An LTS *MemVar*(r, v_r) for each member variable v_r of each object r

```
lts
    I(r2) = []u,s: begResNxtGet(u,s,r) -> I(r2)
        [] []u,s: endResNxtGet(u,s,r,r2) -> I(r2)
        [] []u,s,r2: begResNxtSet(u,s,r,r2) -> I(r2)
        [] []u,s: endResNxtSet(u,s,r) -> I(r2)
from I(rNull)
```

An LTS *New*(r) for each object r (to guarantee unique creation)

```
lts
    I = []u,s: endResNew(u,s,r) -> STP
from I
```

LTSs III

An LTS *Sync*(*c*) for each class *c* (with synchronised methods)

```
lts
    I = []r: resSync(u,s,r) -> S1(r)
       [] []r: resSync(u2,s0,r) -> S2(r)
    S1(r) = resUnsync(u,s,r) -> I
       [] []r2: [!r=r2] resSync(u2,s0,r2) -> S1(r)
       [] []r2: [!r=r2] resUnsync(u2,s0,r2) -> S1(r)
    S2(r) = resUnsync(u2,s0,r) -> I
       [] []r2: [!r=r2] resSync(u,s,r2) -> S2(r)
       [] []r2: [!r=r2] resUnsync(u,s,r2) -> S2(r)
from I
```

Parametrised Implementation LTS

Compose all the parts in parallel

$$(\| \ u, s : \mathit{Ctrl}(u, s)) \parallel (\| \ r, v_r : \mathit{MemVar}(r, v_r)) \\ \parallel (\| \ r : \mathit{New}(r)) \parallel (\| \ c : \mathit{Sync}(c))$$

- u ranges over thread ids
- s ranges over stack positions
- r ranges over objects
- v_r ranges over member variables of the object r
- c ranges over classes with synchronised methods

Verification Techniques for Parametrised Parts

Dealing with Objects

Data independence (Lazić 1999)

- a cut-off/threshold for the number of replicated objects
- in our case, 113 resources is sufficient
- the cut-off of 113 resource can be pushed down to 3

Dealing with Objects

Data independence (Lazić 1999)

- a cut-off/threshold for the number of replicated objects
- in our case, 113 resources is sufficient
- the cut-off of 113 resource can be pushed down to 3

Limitation

Abstraction is needed:

- a member variable may change its value independently
- a call to a new operator may give an existing object

Dealing with Threads

Precongruence reduction (Siirtola & Kortelainen 2009)

- a cut-off/threshold for the number of concurrent threads
- in our case, 2 users is sufficient

Dealing with Threads

Precongruence reduction (Siirtola & Kortelainen 2009)

- a cut-off/threshold for the number of concurrent threads
- in our case, 2 users is sufficient

Limitation

The same as above: abstraction is needed.

Dealing with Stack

Behavioural fixed point (BFP) method (Valmari & Tienari 1991)

- the control flow from the viewpoint of any two threads and any two stack positions (= behavioural fixed point)
- strictly linear topology $\rightarrow$ totally ordered topology
- with the BFP method, a cut-off/threshold for the number of stack positions (Siirtola 2010)
- not needed in our example (no real recursion)

Dealing with Stack

Behavioural fixed point (BFP) method (Valmari & Tienari 1991)

- the control flow from the viewpoint of any two threads and any two stack positions (= behavioural fixed point)
- strictly linear topology $\rightarrow$ totally ordered topology
- with the BFP method, a cut-off/threshold for the number of stack positions (Siirtola 2010)
- not needed in our example (no real recursion)

Limitation

Fixed point may not exist

Verification of SRS

- The instances up to 2 users and 3 resources were found to be correctly synchronised
- Hence, SRS is correctly synchronised for any number of users and resources

Conclusions

Conclusions

- Object-oriented programs are naturally modelled as parametrised systems
- There exists techniques and tools for the verification of such models