A Model-driven Methodology for Generating and Verifying CSP-based Java Code

Julio Mariño [1]   Raúl N.N. Alborodo [2]

[1] Universidad Politécnica de Madrid
Babel research group
julio.marino@upm.es

[2] IMDEA Software Institute
raul.alborodo@imdea.org

Communicating Process Architectures CPA2015
Canterbury, August 24 2015

summary

the paper in a nutshell

this paper is about:

- model-driven development of concurrent software
- specifying process interaction with formal models
- generating code from these models (semi-automatically), and
- verifying the resulting code

our contributions:

- a textual syntax for specifying process interaction models (that we call *shared resources*) as JML-annotated Java interfaces
- a couple of generic templates for translating these models into Java classes using the JCSP (CSP for Java) library
- an strategy for verification of the code generated according to these templates, and
- some experimental results on the mechanical verification using the KeY tool

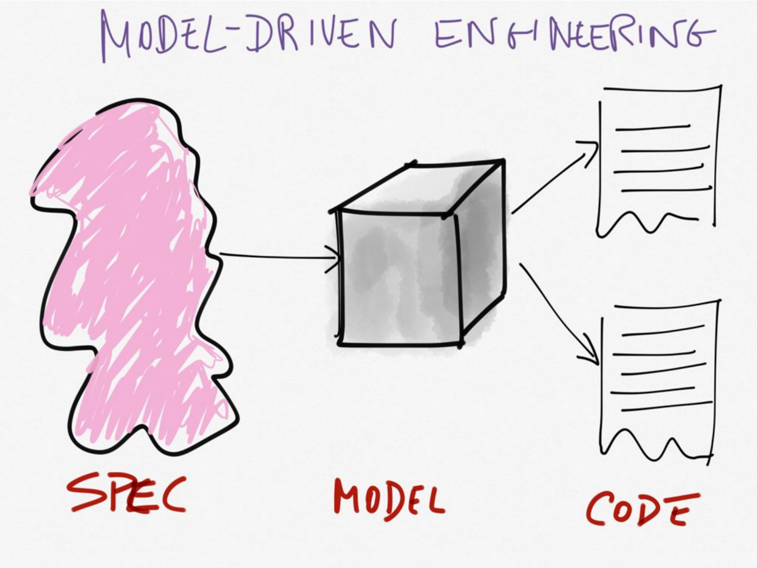
(initial) motivation:

- teaching ~~trying to teach~~ concurrency to undergrad students for more than 15 years

POLITÉCNICA imdea

model-driven software development

workflows



benefits of model-driven software development

why adding may be necessary for simplifying things

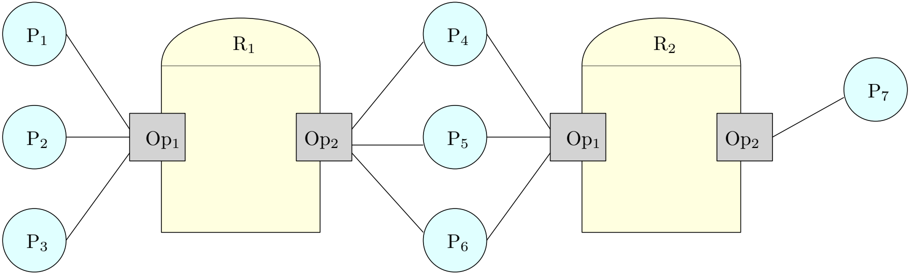
1. Formalizing (part of) the requirements reduces ambiguity in the problem statement.
2. Formal models can be the subject of *experiments* aimed at *early requirement validation*. That is, a mathematical model can be formally verified for detecting inconsistencies or other flaws.
3. Code is not written from scratch but *generated* or *distilled* (semiautomatically) from the model. This brings several benefits. One of them is *portability*. This is specially relevant for concurrent software production, given the volatility of certain languages. A second benefit is robustness against changes in the requirements – modifying concurrent code by hand may introduce more errors than re-generating it. Finally, the generative approach may reduce production costs at this stage.
4. Models can help in the validation, verification and test case generation of the code obtained from the previous phases.

shared resources

what is so relevant that deserves to be modeled

key abstractions

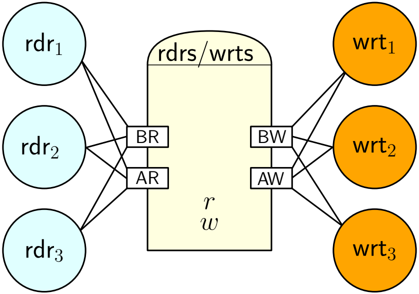
concurrency	=	simultaneous execution	+
		nondeterminism	+
		interaction	
interaction	=	communication	+
		synchronization	
synchronization	=	~~mutual exclusion~~ serializability	+
		condition synchronization	



POLITÉCNICA imdea

shared resources by example

readers & writers



communication: takes place via change of the resource's internal state, after applying a sequence of (serial) operations:

$$\boxed{\begin{matrix} w = 1 \\ r = 0 \end{matrix}} \overset{AW}{\leadsto} \boxed{\begin{matrix} w = 0 \\ r = 0 \end{matrix}} \overset{BR}{\leadsto} \boxed{\begin{matrix} w = 1 \\ r = 1 \end{matrix}} \overset{BR}{\leadsto} \boxed{\begin{matrix} w = 0 \\ r = 2 \end{matrix}}$$

synchronization: consists in restricting the set of valid sequences of operations (internal language of the shared resource):

- valid traces: BR; AR; BW; AW; BR; BR; AR; AR; BW; AW; ...
- invalid traces: BR; BW; AW; BR; BR; AR; AR; BW; AW; AR; ...

formal specification of a shared resource

readers & writers

CADT ReadersWriters
OPERATIONS
ACTION BeforeRead;AfterRead;BeforeWrite;AfterWrite:
SEMANTICS
DOMAIN:
STATE: $(readers : \mathbb{N} \times writers : \mathbb{N})$
INVT: $(readers > 0 \Rightarrow writers = 0) \wedge$
$(writers > 0 \Rightarrow readers = 0 \wedge writers = 1)$
INITIAL: $writers = 0 \wedge readers = 0$

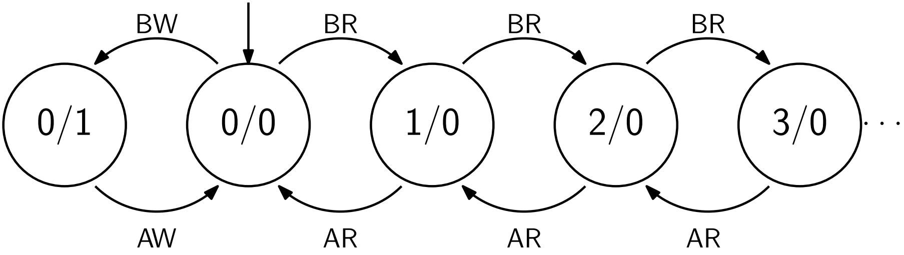
CPRE: $writers = 0 \wedge readers = 0$
BeforeWrite
POST: $writers = 1$
PRE: $writers = 1$
CPRE: *true*
AfterWrite
POST: $writers = 0$
CPRE: $writers = 0$
BeforeRead
POST: $readers = 1 + readers^{\mathrm{in}}$
PRE: $readers > 0$
CPRE: *true*
AfterRead
POST: $readers = readers^{\mathrm{in}} - 1$

- *preconditions* (PRES) are often independent from the resource's state
- The *invariant* (INVARIANT) maps to the loop invariant within the server code.
- The *concurrent or synchronization pre-condition* (CPRE) must hold right before entering the code for each operation (might block execution)
- The *post-condition* (POST) must hold on exit of the code of each operation

POLITÉCNICA imdea

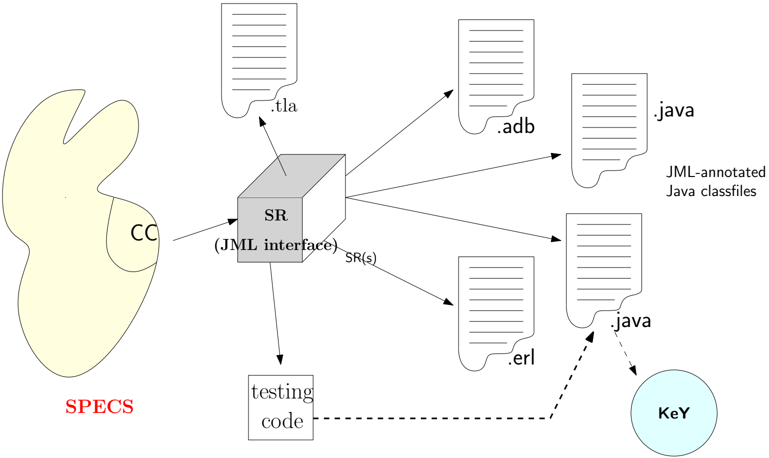
shared resources as abstract state machines

readers & writers



model-driven engineering revisited

applying all of this to developing concurrent Java SW



shared resource specifications as JML-annotated Java interfaces

a textual, convenient and ready-to-compile representation

```
package es.upm.babel.ccjml.samples.readerswriters.java;

public interface /*@ shared_resource @*/ ReadersWriters {
  //@ public model instance int readers;
  //@ public model instance int writers;

  /*@ public instance invariant
    @     readers >= 0 && writers >= 0 &&
    @     (readers > 0 ==> writers == 0) &&
    @     (writers > 0 ==> readers == 0 && writers == 1);
    @*/

  //@ public initially readers == 0 && writers == 0;

  /*@ public normal_behaviour
    @    cond_sync writers == 0 && readers == 0;
    @    assignable writers;
    @    ensures writers == 1;
    @*/
  public void beforeWrite();
```

POLITÉCNICA imdea

shared resource specifications as JML-annotated Java interfaces (cont'd.)

a textual, convenient and ready-to-compile representation

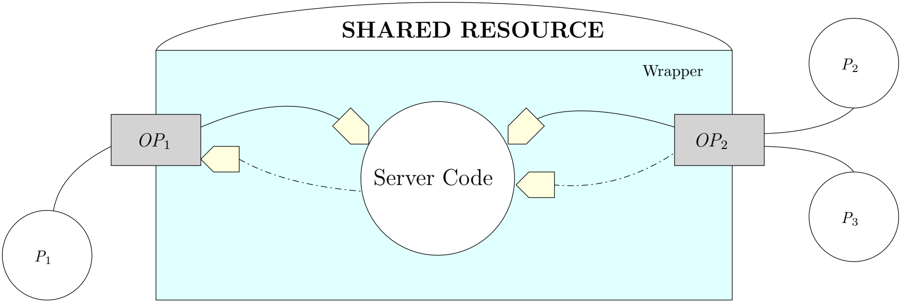
```
    @   requires writers == 1;
    @   assignable writers;
    @   ensures writers == 0;
    @*/
  public void afterWrite();

  /*@ public normal_behaviour
    @   cond_sync writers == 0;
    @   assignable readers;
    @   ensures readers == \old(readers) + 1;
    @*/
  public void beforeRead();

  /*@ public normal_behaviour
    @   requires readers > 0;
    @   assignable readers;
    @   ensures readers == \old(readers) - 1;
    @*/
  public void afterRead();
}
```

POLITÉCNICA imdea

implementing shared resources using JCSP

client-server + RPC + ...

a view from the clients' side:

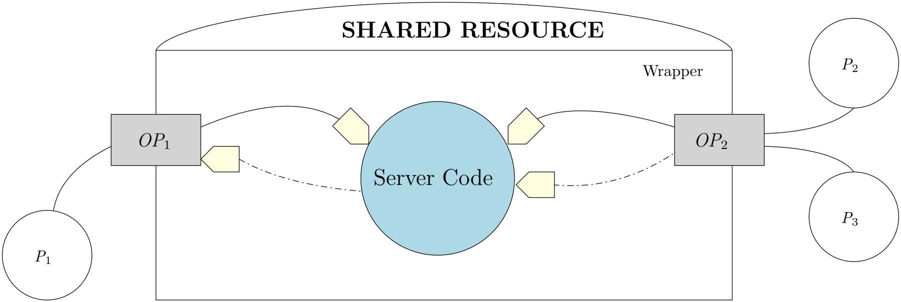


- **Wrapper** Receiving method invocations and propagating them as messages to the server through CSP channels;

implementing shared resources using JCSP

client-server + RPC + ...

server side:



- **Server** Processing the requests received from the wrapper methods and modifying the shared resource inner state

implementing the server

the devil is in the CPREs

- When shared resource operations take no arguments or the operation's CPRE does not depend on them, one channel per operation and channel enabled when CPRE holds (see, for instance, readers & writers).
- When CPREs may vary depending on the actual parameters operations can take there are two basic approaches:
  - channel replication: Instantiate CPREs with all their possible values, take classes modulo logical equivalence, then assign a channel to each class. Enable channels according to each CPRE.
  - deferred requests: one (always open) channel per operation, requests are stored in the server until CPRE holds.

POLITÉCNICA imdea

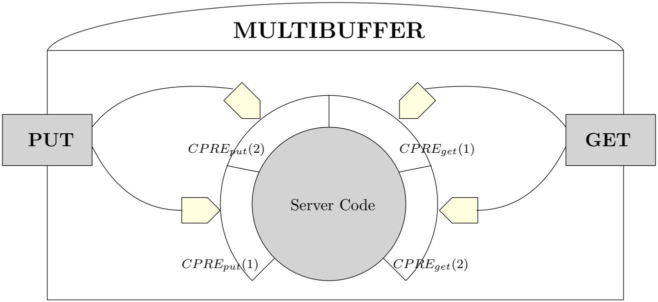
CADT Multibuffer
OPERATIONS
ACTION Put: $Sequence(ANY)[i]$
ACTION Get: $\mathbb{N}[i] \times Sequence(ANY)[o]$
SEMANTICS
DOMAIN:
STATE: $\mathrm{self} = Sequence(ANY)$
INVT: $\text{Length}(\mathrm{self}) \leq MAX$
INITIAL: $\text{Length}(\mathrm{self}) = 0$

PRE: $1 \leq \text{Length}(r) \leq \lfloor MAX/2 \rfloor$
CPRE: $1 \leq \text{Length}(r) \leq MAX - \text{Length}(\mathrm{self})$
Put(r)
POST: $\mathrm{self} = \mathrm{self}^{\mathrm{in}} + r$
PRE: $1 \leq n \leq \lfloor MAX/2 \rfloor$
CPRE: $1 \leq n \leq \text{Length}(\mathrm{self})$
Get(n, s)
POST: $\mathrm{self}^{\mathrm{in}} = \mathrm{self} + s$

channel replication

multibuffer

Considering *Multibuffer* example with MAX = 4



$$pe_i : E_i \nrightarrow \mathbb{N}$$

$$pe_{put}([a_1, \ldots, a_n]) = n$$
$$pe_{get}(n) = MAX/2 + n$$

$$px_i : D_x \rightarrow E_i$$

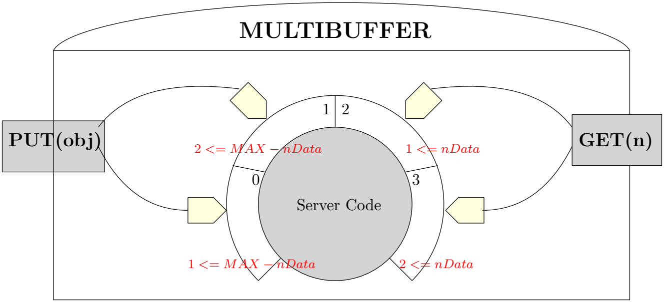
$$px_{put}([a_1, \ldots, a_k]) = [0_1, \ldots, 0_k]$$
$$px_{get}(n) = n$$

POLITÉCNICA imdea

channel replication

multibuffer

Considering *Multibuffer* example with MAX = 4



$pe_i : E_i \nrightarrow \mathbb{N}$

$px_i : D_x \rightarrow E_i$

$pe_{put}([a_1, \ldots, a_n]) = n$
$pe_{get}(n) = MAX/2 + n$

$px_{put}([a_1, \ldots, a_k]) = [0_1, \ldots, 0_k]$
$px_{get}(n) = n$

deferred requests

CPRE depends on some operation parameters x (D_x potentially infinite)

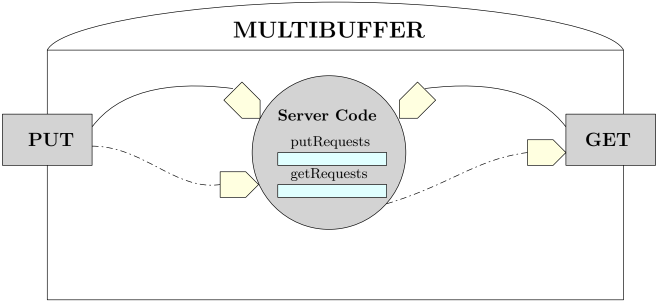
- Every request is stored in some data structure as soon as it is received by the server. Typically, there will be one collection per method;
- It must be ensured that no pending request whose synchronization condition holds is left unattended before entering into a new iteration of the service loop;
- Finally, mutual exclusion of the servicing of the requests must be guaranteed by the server implementation.

POLITÉCNICA imdea

deferred requests

multibuffer

Considering Multibuffer example with `MAX = 4`



deferred requests: multibuffer example

wrapper

- *single send*: when the footprint contains all the actual parameter (e.g. the `get` operation)

```
    One2OneChannel innerChannel = Channel.one2one();
    chGet.out().write(new GetRequestCSP(n,innerChannel));
    Object[] res = (Object[]) innerChannel.in().read(); // blocks
    return res;
  }
```

- *double send*: when the footprint does not contain all the parameter information (e.g. the `put` operation).

```
    One2OneChannel innerChannel = Channel.one2one();
    chPut.out().write(new PutRequestCSP(els.length,innerChannel));
    // send the data to be inserted
    innerChannel.out().write(els); // blocks until server can take it
    innerChannel.in().read();      // wait for server to finish
  }
```

verification : channel replication

proof obligations

key ideas:

- code form follows function (template-based programming), so we can JML-annotate crucial points in the code
- goal: reveal tpical errors programmers make in applying the template
- actual prrof obligations derived from both template and shared resource specification

proof obligations for the server component

- *prop_cs_preservation:* immediately after the conditional statement that decides upon the index that tells the server which call must serve, the CPRE of that call must hold.
- *prop_safe_selection:* the server code must guarantee that a valid service is selected in each iteration, i.e. the selected service *s* must belong to *pe* range, and has a message waiting to be read.
- *prop_only_one_request:* only one request is processed in each iteration. Server code must guarantee this in order to avoid losing requests.

verification: channel replication

prop_cs_preservation

Immediately after the `switch` statement determines which branch will execute, the corresponding synchronization condition must hold.

Generated Code

```
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    /*@ assert (\forall int j;
      @              0<=j && j<syncCond.length;
      @              syncCond[j] == CPRE_i);
      @*/
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_l:
        //@ assert CPRE_l(chosenService);
        ...
        break;
      ...
    }
  }
}
```

verification: channel replication

prop_cs_preservation

Immediately after the `switch` statement determines which branch will execute, the corresponding synchronization condition must hold.

Generated Code

```
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    /*@ assert (\forall int j;
      @               0<=j && j<syncCond.length;
      @               syncCond[j] == CPRE_j);
      @*/
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_l:
        //@ assert CPRE_l(chosenService);
        ...
        break;
      ...
    }
  }
}
```

Instrumented Code

```
public boolean cprePreservation;
public boolean oneMessageProcessed;
...
//@ ensures cprePreservation;
public void run(){
  ...
  cprePreservation = true;
  int chosenService = 42;

  while (chosenService != -1){
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_l:
        cprePreservation &= CPRE_l(chosenService);
        ...
        break;
      ...
    }
  }
}
```

verification: channel replication

prop_safe_selection

Server code must guarantee that a valid service is selected in each iteration, i.e. the selected service s must belong to pe range, and has a message waiting to be read. The aims are:

- `services` must include all input channels and its length must be equal to $\#rg(pe)$
- a channel in a position `i` in `services` must have its synchronization predicate in the position `i` of `synCond`
- their length must be the equal.

POLITÉCNICA imdea

Generated Code

```
public void run(){
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    /*@ assert (\forall int j;
      @              0<=j && j<syncCond.length;
      @              syncCond[j] == CPRE_i);
      @*/
    chosenService = fairSelect(syncCond,services);

    ...  process a request onchosenService
  }
}
```

verification: channel replication

prop_safe_selection

Generated Code

```
public void run(){
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    /*@ assert (\forall int j;
      @              0<=j && j<syncCond.length;
      @              syncCond[j] == CPRE_i);
      @*/
    chosenService = fairSelect(syncCond,services);

    ...  process a request onchosenService
  }
}
```

Instrumented Code

```
//@ ensures wellFormedSyncCond;
public void run(){
  wellFormedSyncCond = true;

  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];
  int chosenService = 42;
  while (chosenService != -1 ) {
    ...  update syncCond array
    for (int i =0 ; i < syncCond.length ; i++ ) {
      wellFormedSyncCond &= (syncCond[i] == CPRE_i);
    }
    wellFormedSyncCond &=
            syncCond.length == guards.length;

    chosenService =
            JCSPKeY.fairSelect(syncCond, guards);
    ...  process a request onchosenService
  }
}
```

verification: channel replication

prop_safe_selection

Generated Code

```
public void run(){
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    /*@ assert (\forall int j;
      @              0<=j && j<syncCond.length;
      @              syncCond[j] == CPRE_i);
      @*/
    chosenService = fairSelect(syncCond,services);

    ...  process a request onchosenService
  }
}
```

Instrumented Code

```
//@ ensures wellFormedSyncCond;
public void run(){
  wellFormedSyncCond = true;

  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];
  int chosenService = 42;
  while (chosenService != -1 ) {
    ...  update syncCond array
    for (int i =0 ; i < syncCond.length ; i++ ) {
      wellFormedSyncCond &= (syncCond[i] == CPRE_i);
    }
    wellFormedSyncCond &=
            syncCond.length == guards.length;

    chosenService =
            JCSPKeY.fairSelect(syncCond, guards);
    ...  process a request onchosenService
  }
}
```

Errors that can be found: poorly updates of syncCond

verification: channel replication

prop_only_one_request

Only one request is processed per server iteration. If using nestsed *if*, is already guaranteed if using nested `if` statements, but when using `switch`, the execution of more than one branch is possible.

Generated Code

```
public void run(){
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_l:
        //@ assert CPRE_l(chosenService);
        ...
        break;
      ...
    }
  }
}
```

POLITÉCNICA imdea

verification: channel replication

prop_only_one_request

Only one request is processed per server iteration. If using nestsed *if*, is already guaranteed if using nested `if` statements, but when using `switch`, the execution of more than one branch is possible.

Generated Code

```
public void run(){
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_I:
        //@ assert CPRE_I(chosenService);
        ...
        break;
      ...
    }
  }
}
```

Instrumented Code

```
public boolean oneMessageProcessed;
...
//@ ensures oneMessageProcessed;
public void run(){
  ...
  oneMessageProcessed = true;
  int chosenService = 42;

  while (chosenService != -1){
    int processedMessages = 0;
    ...  update syncCond array
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_I:
        ...
        processedMessages ++;
        break;
      ...
    }
    oneMessageProcessed &= processedMessages == 1;
  }
}
```

verification: channel replication

prop_only_one_request

Only one request is processed per server iteration. If using nestsed *if*, is already guaranteed if using nested `if` statements, but when using `switch`, the execution of more than one branch is possible.

Generated Code

```
public void run(){
  int chosenService = 42;
  int[] services = {...};
  boolean[] syncCond = new boolean[#rg(pe)];

  while (chosenService != -1 ){
    ...  update syncCond array
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_I:
        //@ assert CPRE_I(chosenService);
        ...
        break;
      ...
    }
  }
}
```

Instrumented Code

```
public boolean oneMessageProcessed;
...
//@ ensures oneMessageProcessed;
public void run(){
  ...
  oneMessageProcessed = true;
  int chosenService = 42;

  while (chosenService != -1){
    int processedMessages = 0;
    ...  update syncCond array
    chosenService = fairSelect(syncCond,services);

    switch(chosenService){
      ...
      case METHOD_I:
        ...
        processedMessages ++;
        break;
      ...
    }
    oneMessageProcessed &= processedMessages == 1;
```

Errors that can be found: missing `break` statements in each `switch` pattern.

proof obligations for the server component

- *prop_cs_preservation:* immediately after the server code that retrieves a request to be processed, the CPRE of the method associated with the request must hold. This restriction ensures *safety* of the processing code because changes to the inner state of the resources are performed only for those requests that represent valid invocations.
- *prop_completeness:* If the server exits the code for processing deferred requests – and is about to loop back to the `fairSelect` – there should be no valid pending requests.

verification: deferred requests

prop_cs_preservation

Immediately after the server starts processing a deferred request, the CPRE for the relevant operation must hold

verification: deferred requests

prop_cs_preservation

Generated Code

```
...
public void run(){
  ...
  // process deferred requests for operation k
  for (int i = 0; i < operation_kRequest.size()) {
    ... dequeue request item from operation_k_Request
    ... extract operation_k_footprint from the request item
    if (condition_k (operation_k_footprint) {
      /*@ assert resource_Invariant && condition_k←↩
              (operation_k_footprint)
        @   ==> CPRE_k;
        @*/
      ... extract the channel, innerChannel, from the request item
      ... input remaining operation_k parameters, if any, from
            innerChannel
      ... apply operation_k to the resource, using footprint and
            parameters
      //@ assume resource_Invariant && POST_k;
      ... send operation_k results (or null) down innerChannel
    } else {
      ... enqueue item back on operation_k_Request
    }
  }
  ... process deferred requests for all the other operations similarly
}
```

verification: deferred requests

prop_cs_preservation

Generated Code

```
...
public void run(){
  ...
  // process deferred requests for operation k
  for (int i = 0; i < operation_kRequest.size()) {
    ...  dequeue request item from operation_k_Request
    ...  extract operation_k_footprint from the request item
    if (condition_k (operation_k_footprint) {
      /*@ assert resource_Invariant && condition_k←↩
              (operation_k_footprint)
        @    ==> CPRE_k;
        @*/
      ...  extract the channel, innerChannel, from the request item
      ...  input remaining operation_k parameters, if any, from
             innerChannel
      ...  apply operation_k to the resource, using footprint and
             parameters
      //@ assume resource_Invariant && POST_k;
      ...  send operation_k results (or null) down innerChannel
    } else {
      ...  enqueue item back on operation_k_Request
    }
  }
  ...  process deferred requests for all the other operations similarly
}
```

Instrumented Code

```
boolean cprePreservation;
...
//@ ensures cprePreservation;
public void processDeferredRequests(){
  ...
  // process deferred requests for operation k
  for (int i = 0; i < operation_kRequest.size()) {
    ...  dequeue request item from operation_k_Request
    ...  extract operation_k_footprint from the request item
    if (condition_k (operation_k_footprint) {
      /*@ assert resource_Invariant && condition_k←↩
              (operation_k_footprint)
        @    ==> CPRE_k;
        @*/
      cprePreservation &= CPRE_k;  // let's see if←↩
              it's true
      ...  extract the channel, innerChannel, from the request item
      ...  input remaining operation_k parameters, if any, from
             innerChannel
      ...  apply operation_k to the resource, using footprint and
             parameters
      //@ assume resource_Invariant && POST_k;
      ...  send operation_k results (or null) down innerChannel
    } else {
      ...  enqueue item back on operation_k_Request
    }
  }
  ...  process deferred requests for all the other operations similarly
}
```

POLITÉCNICA imdea

verification: deferred requests

prop_completeness

We need to ensure that no pending request can be processed. A request is either processed (if its CPRE holds) or enqueued again. If it is true, property (*prop_cs_preservation*) guarantees that is going to be processed. Otherwise,(CPRE does not hold) two cases can be distinguished.

verification: deferred requests

prop_completeness

We need to ensure that no pending request can be processed. A request is either processed (if its CPRE holds) or enqueued again. If it is true, property (*prop_cs_preservation*) guarantees that is going to be processed. Otherwise,(CPRE does not hold) two cases can be distinguished.

CPRE does NOT depend on the input parameters

```
//prop_completeness
```

//@ ensures $\bigwedge_{i=1}^{n}$ (method$_i$Requests > 0 ==> !CPRE$_i$);

verification: deferred requests

prop_completeness

We need to ensure that no pending request can be processed. A request is either processed (if its CPRE holds) or enqueued again. If it is true, property (*prop_cs_preservation*) guarantees that is going to be processed. Otherwise,(CPRE does not hold) two cases can be distinguished.

CPRE does NOT depend on the input parameters

$$\texttt{//prop\_completeness}$$

$$\texttt{//@ ensures } \bigwedge_{i=1}^{n} \texttt{(method}_i\texttt{Requests > 0 ==> !CPRE}_i\texttt{);}$$

CPRE DEPENDS on the input parameters

- Follow a similiar approach as for *prop_cs_preservation*
- A new variable `completeness` is defined
- It accumulates the value of the associated CPRE of requests.

$$\texttt{//prop\_completeness}$$

$$\texttt{//@ ensures } \sum_{i=1}^{n} \texttt{method}_i\texttt{Request.size() > 0 ==> completeness;}$$

verification: deferred requests

prop_completeness

We need to ensure that no pending request can be processed. A request is either processed (if its CPRE holds) or enqueued again. If it is true, property (*prop_cs_preservation*) guarantees that is going to be processed. Otherwise,(CPRE does not hold) two cases can be distinguished.

CPRE does NOT depend on the input parameters

//prop_completeness

//@ ensures $\bigwedge_{i=1}^{n}$ (method$_i$Requests > 0 ==> !CPRE$_i$);

CPRE DEPENDS on the input parameters

- Follow a similiar approach as for *prop_cs_preservation*
- A new variable `completeness` is defined
- It accumulates the value of the associated CPRE of requests.

//prop_completeness

//@ ensures $\sum_{i=1}^{n}$ method$_i$Request.size() > 0 ==> completeness;

Errors that can be found: *ping-pong* effect, bad conditions for processing requests, ...

POLITÉCNICA imdea

experimental results

using KeY to certify shared resource implementations

- Correct implementations (both approaches)
  - implementations following the templates
  - optimized versions of the previous implementations.
- Erroneous/buggy implementations
  - Channel replication:
    - implementations with erroneous or incomplete update of the `syncCond` array.
    - missing `break` statements in `switch` code;
  - Deferred requests:
    - incorrect optimizations on the code processing the pending requests
    - violations of *protocol* definitions.
    - not taking into account *ping-pong* effects

conclusions and future work

- JML extension for shared resources presented
- Generation of correct Java code from specifications using model-driven techniques
  - Channel replication: CPRE depends on x (with D_x finite)
  - Deferred requests: CPRE depends on x (with D_x potentially infinite)
- Automatic verification of JML-anotated implementations using the KeY tool and lots of instrumentation
- Examples, including specifications, implementations and verification annotations, can be found at `http://babel.upm.es/~rnnalborodo/sr_web/`.

conclusions and future work

- JML extension for shared resources presented
- Generation of correct Java code from specifications using model-driven techniques
  - Channel replication: CPRE depends on x (with D_x finite)
  - Deferred requests: CPRE depends on x (with D_x potentially infinite)
- Automatic verification of JML-anotated implementations using the KeY tool and lots of instrumentation
- Examples, including specifications, implementations and verification annotations, can be found at `http://babel.upm.es/~rnnalborodo/sr_web/`.

- Completing the experiments with more implementations of the base test suite, perhaps optimized in non-trivial ways.
- Actually extending the JML compiler (e.g. using OpenJML)
- Integrating the presented framework in KeY
  - Experience gained with instrumentation may serve to make KeY concurrency-aware
- First steps towards code compilation for shared resources
  - for a subset of the shared resource syntax (codename *razor*)
- More examples to show practicality and scalability of the approaches
  - A collection of correct concurrent Java collections on the way

POLITÉCNICA imdea

Channel Replication

CPRE depends on some operation parameters

$$\textsc{cpre}(op_i(\vec{x},\vec{y})) \equiv C_i \begin{cases} \textit{tautology} & C_i \Leftrightarrow \textit{true} \quad \textit{open channel} \\ \textit{depends only on resource state} & C_i = \phi(S) \quad \textit{one channel enabled by } \phi \\ \textit{may depend on } \vec{x} : C_i = \phi(S, \vec{x}) & \begin{cases} \textit{channel replication} \\ \textit{deferred requests} \end{cases} \end{cases}$$

Channel Replication: Formalization

Considering one operation $op_i(x, y)$

- $x \in D_x$ and $y \in D_y$
- $\text{CPRE}_{op_i}\ C_i$ only depends on x

Channel Replication: Formalization

Considering one operation $op_i(x, y)$

- $x \in D_x$ and $y \in D_y$
- $\textsc{cpre}_{op_i}$ C_i only depends on x

C_i is **independent** from y iff $\forall\, a \in D_x . \forall\, b, b' \in D_y . C_i[a/x, b/y] \Leftrightarrow C_i[a/x, b'/y]$

C_i is **dependent** from x iff $\exists\, a, a' \in D_x . C_i[a/x] \not\Leftrightarrow C_i[a'/x]$

Channel Replication: Formalization (Cont.)

$a, a' \in D_x$ are **equivalent** iff $C_i[a/x]$ and $C_i[a'/x]$

Let E_i be the (finite) set of equivalence classes

$op_i(a, b)$ and $op_i(a', b)$ will be routed to the **same channel** if the precondition holds (or fails) for them both