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
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

![Diagram showing concurrency, interaction, and synchronization relationships]
shared resources by example
readers & writers

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

\[
\begin{align*}
&w = 1 \quad \text{AW} \quad w = 0 \quad \text{BR} \quad w = 1 \quad \text{BR} \quad w = 0 \\
&r = 0 \quad \sim \quad r = 0 \quad \sim \quad r = 1 \quad \sim \quad r = 2
\end{align*}
\]

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** Readers Writers

**OPERATIONS**

**ACTION** BeforeRead; AfterRead; BeforeWrite; AfterWrite:

**SEMANTICS**

**DOMAIN:**

**STATE:** \((\text{readers} : \mathbb{N} \times \text{writers} : \mathbb{N})\)

**INVVT:** \((\text{readers} > 0 \Rightarrow \text{writers} = 0) \land \)

\((\text{writers} > 0 \Rightarrow \text{readers} = 0 \land \text{writers} = 1)\)

**INITIAL:** \(\text{writers} = 0 \land \text{readers} = 0\)

**CPRE:** \(\text{writers} = 0 \land \text{readers} = 0\)

BeforeWrite

**POST:** \(\text{writers} = 1\)

**PRE:** \(\text{writers} = 1\)

**CPRE:** \(\text{true}\)

AfterWrite

**POST:** \(\text{writers} = 0\)

**CPRE:** \(\text{writers} = 0\)

BeforeRead

**POST:** \(\text{readers} = 1 + \text{readers}^{\text{in}}\)

**PRE:** \(\text{readers} > 0\)

**CPRE:** \(\text{true}\)

AfterRead

**POST:** \(\text{readers} = \text{readers}^{\text{in}} - 1\)

- **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
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

```java
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();
}
```
shared resource specifications as JML-annotated Java interfaces (cont’d.)
a textual, convenient and ready-to-compile representation

```java
@ 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();
```
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 \textsc{Cpres}

- When shared resource operations take no arguments or the operation’s \textsc{Cpre} does not depend on them, \textit{one channel per operation} and channel enabled when \textsc{Cpre} holds (see, for instance, readers & writers).

- When \textsc{Cpres} may vary depending on the actual parameters operations can take there are two basic approaches:
  - \textbf{channel replication:} Instantiate \textsc{Cpres} with all their possible values, take classes modulo logical equivalence, then assign a channel to each class. Enable channels according to each \textsc{Cpre}.
  - \textbf{deferred requests:} one (always open) channel per operation, requests are stored in the server until \textsc{Cpre} holds.
CPREs depending on their parameters

multibuffer

CADT Multibuffer

OPERATIONS

ACTION Put: $\text{Sequence}(\text{ANY})[i]

ACTION Get: $\mathbb{N}[i] \times \text{Sequence}(\text{ANY})[o]

SEMANTICS

DOMAIN:

STATE: self = $\text{Sequence}(\text{ANY})$

INV: Length(self) $\leq$ MAX

INITIAL: Length(self) = 0

PRE: $1 \leq \text{Length}(r) \leq \lfloor \text{MAX/2} \rfloor$

CPRE: $1 \leq \text{Length}(r) \leq \text{MAX} - \text{Length}(\text{self})$

Put(r)

POST: self = self$^{\text{in}} + r$

PRE: $1 \leq n \leq \lfloor \text{MAX/2} \rfloor$

CPRE: $1 \leq n \leq \text{Length}(\text{self})$

Get(n, s)

POST: self$^{\text{in}} = \text{self} + s$
channel replication
multibuffer

Considering Multibuffer example with MAX = 4

\[ p_{e_i} : E_i \rightarrow \mathbb{N} \]
\[ p_{e_{PUT}}([a_1, \ldots, a_n]) = n \]
\[ p_{e_{GET}}(n) = \text{MAX}/2 + n \]

\[ p_{x_i} : D_x \rightarrow E_i \]
\[ p_{x_{PUT}}([a_1, \ldots, a_k]) = [0_1, \ldots, 0_k] \]
\[ p_{x_{GET}}(n) = n \]
channel replication
multibuffer

Considering *Multibuffer* example with $\text{MAX} = 4$

\[
\begin{align*}
pe_i : E_i &\rightarrow \mathbb{N} \\
p_{e_{\text{put}}}([a_1, \ldots, a_n]) &= n \\
p_{e_{\text{get}}}(n) &= \text{MAX}/2 + n \\
px_i : D_x &\rightarrow E_i \\
p_{x_{\text{put}}}([a_1, \ldots, a_k]) &= [0_1, \ldots, 0_k] \\
p_{x_{\text{get}}}(n) &= n
\end{align*}
\]
deferral requests

**CPRE** depends on some operation parameters \( x (D_x \text{ potentially infinite}) \)

- 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.
deferred requests
multibuffer

Considering Multibuffer example with $\text{MAX} = 4$
deferred requests: multibuffer example

**wrapper**

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

```java
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).

```java
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 typical errors programmers make in applying the template
- actual proof obligations derived from both template and shared resource specification

proof obligations for the server component

- \textit{prop\_cs\_preservation}: immediately after the conditional statement that decides upon the index that tells the server which call must serve, the \texttt{CPRE} of that call must hold.
- \textit{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.
- \textit{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

```java
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){
        ... (chosenService);
        case METHOD\(_i\):
            //@ assert CPRE\(_i\)(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**

```java
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] == C02E);*/
    chosenService = fairSelect(syncCond, services);

    switch(chosenService){
    ... case METHOD1:
        //@ assert C02E(chosenService);
        ... break;
    ...
    }
}
```

**Instrumented Code**

```java
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 METHOD1:
            cprePreservation &= C02E(chosenService);
            ... break;
        ...
        }
    }
}```
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 \( \text{synCond} \)
- their length must be the equal.
verification: channel replication

*prop_safe_selection*

Generated Code

```java
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 on chosenService
    }
}
```
verification: channel replication

prop_safe_selection

Generated Code

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

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

        /*@ assert (\forall int j;
            @ 0<=j && j<syncCond.length;
            @ syncCond[j] == CPRE_i);
        @*/
        wellFormedSyncCond &= syncCond.length == guards.length;

        ... process a request on chosenService
    }
}
```

Instrumented Code

```java
//@ 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 on chosenService
    }
}
```
verification: channel replication

Generated Code

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

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

Errors that can be found: poorly updates of syncCond

Instrumented Code

```java
//@ 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);
        ...
    }
}
```
verification: channel replication

prop_only_one_request

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

Generated Code

```java
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:\
                //@ assert CPRE(chosenService);
                ...
                break;
                ...
        }
    }
}
```
verification: channel replication

prop_only_one_request

Only one request is processed per server iteration. If using nested \textit{if}, is already guaranteed if using nested \textit{if} statements, but when using \textit{switch}, the execution of more than one branch is possible.

Generated Code

```java
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 METHOD1:
                //@ assert CPRE1(chosenService);
                ...
                break;
            ...
        }
    }
}
```

Instrumented Code

```java
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 METHOD1:
                ...
                processedMessages ++;
                break;
            ...
        }
        oneMessageProcessed &= processedMessages == 1;
    }
}
```
verification: channel replication

prop_only_one_request

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

Instrumented Code

Generated Code

```java
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):
        //@ assert CPRE(chosenService);
        ... break;
    ...
  }
}
```

Errors that can be found: missing break statements in each switch pattern.
verification: deferred requests

proof obligations

proof obligations for the server component

1. **prop_cs_preservation**: immediately after the server code that retrieves a request to be processed, the $\texttt{CPre}$ 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.

2. **prop_completeness**: If the server exits the code for processing deferred requests – and is about to loop back to the $\texttt{fairSelect}$ – 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.
Generated Code

```java
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

Generated Code

```java
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

```java
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
}
verification: deferred requests

prop_completeness

We need to ensure that no pending request can be processed. A request is either processed (if its \( C_{\text{PRE}} \) holds) or enqueued again. If it is true, property (\( prop_{-cs\_preservation} \)) guarantees that is going to be processed. Otherwise, (\( C_{\text{PRE}} \) does not hold) two cases can be distinguished.
verification: deferred requests

\textit{prop\_completeness}

We need to ensure that no pending request can be processed. A request is either processed (if its \texttt{CPRE} holds) or enqueued again. If it is true, property (\textit{prop\_cs\_preservation}) guarantees that is going to be processed. Otherwise, (\texttt{CPRE} does not hold) two cases can be distinguished.

\textbf{CPRE does NOT depend on the input parameters}

\begin{verbatim}
//prop_completeness
//@ ensures \( \bigwedge_{i=1}^{n} \) (method\textsubscript{i}.Requests > 0 \( \Rightarrow \) !CPRE\textsubscript{i});
\end{verbatim}
verification: deferred requests

prop_completeness

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

**CPRE** does NOT depend on the input parameters

\[
// \text{prop\_completeness} \\
// @ ensures \bigwedge_{i=1}^{n} (\text{method}_i\text{Requests} > 0 \Rightarrow !\text{CPRE}_i);
\]

**CPRE** DEPENDS on the input parameters

- Follow a similar approach as for \( \text{prop\_cs\_preservation} \)
- A new variable completeness is defined
- It accumulates the value of the associated \( \text{CPRE} \) of requests.

\[
// \text{prop\_completeness} \\
// @ ensures \sum_{i=1}^{n} \text{method}_i\text{Request.size()} > 0 \Rightarrow \text{completeness};
\]
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

```java
//prop_completeness
//@ ensures \( \bigwedge_{i=1}^{n} (\text{method}_i.\text{Requests} > 0 \implies \neg \text{CPRE}_i) \);  
```

CPRE DEPENDS on the input parameters

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

```java
//prop_completeness
//@ ensures \( \sum_{i=1}^{n} \text{method}_i.\text{Request}.\text{size}() > 0 \implies \text{completeness}; \)
```

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: \( \texttt{cpre} \) depends on \( x \) (with \( D_x \) finite)
  - Deferred requests: \( \texttt{cpre} \) depends on \( x \) (with \( D_x \) potentially infinite)
- Automatic verification of JML-annotated 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 J- COMPILER (E.G. USING/PEN J-)

INTEGRATING THE PRESENTED FRAMEWORK IN E9

EXPERIENCE GAINED WITH INSTRUMENTATION MAY SERVE TO MAKE E9 CONCURRENTLY-AWARE & FIRST STEPS TOWARD CODE COMPILATION FOR SHARED RESOURCES FOR A SUBSET OF THE SHARED RESOURCES SYNTAX (CODENAME \textsc{Razor})

A COLLECTION OF CORRECT CONCURRENT JAVA COLLECTIONS ON THE WAY

J. Mariño & R. Alborodo (UPM & IMDEA)

Model-based Code Generation Using JCSP

CPA2015 31 / 31
conclusions and future work

- JML extension for shared resources presented
- Generation of correct Java code from specifications using model-driven techniques
  - Channel replication: $\text{CPRE}$ depends on $x$ (with $D_x$ finite)
  - Deferred requests: $\text{CPRE}$ 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/](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
Channel Replication

\[ \text{CPRE depends on some operation parameters} \]

\[
\text{CPRE}(\text{op}_i(\vec{x}, \vec{y})) \equiv C_i \]

\[
\begin{cases} 
\text{tautology} & C_i \iff \text{true open channel} \\
\text{depends only on resource state} & C_i = \phi(S) \quad \text{one channel enabled by } \phi \\
\text{may depend on } \vec{x} : C_i = \phi(S, \vec{x}) & \text{channel replication} \\
\text{deferred requests} & 
\end{cases}
\]
Channel Replication: Formalization

Considering one operation $\text{op}_i(x, y)$
- $x \in D_x$ and $y \in D_y$
- $\text{CPRE}_{\text{op}_i} C_i$ only depends on $x$
Channel Replication: Formalization

Considering one operation $op_i(x, y)$
- $x \in D_x$ and $y \in D_y$
- $\text{CPRE}_{op_i} C_i$ only depends on $x$

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

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