Asynchronous Components with Futures: Semantics and Proofs in Isabelle/HOL

Ludovic Henrio, Muhammad Uzair Khan

- A Distributed Component Model with Futures Mechanised in Isabelle/HOL
- A study of future update strategies
  - Specification of future update protocol
  - Properties of one strategy
  ➔ Correctness of the component model implementation
Context and Objectives
What are (GCM) Components?

Bindings

Server interfaces

Composite component

Primitive component

Business code

Client interfaces

Primitive component

Business code

interface = port
Our component model

Characteristics of the software component model:
- structured, hierarchical, and distributed
- support for runtime representation and manipulation → handle reconfiguration, component structure
- (separation between functional and non-functional concerns)

An implementation of GCM: ProActive/GCM
- based on the ProActive Java library
- developed in our team
- Components are monothreaded
Approach: a refined GCM model

• More precise than GCM, give a semantics to the model:
  – asynchronous communications: future / requests
  – request queues
  – no shared memory between components
  – notion of request service
• Less precise than ProActive/GCM
  – can be multithreaded
  – no active object, not particularly object-oriented

Similarities with: SCA and Fractal (structure), Creol (futures)
Objectives

- A model general enough to study GCM, but also other component models interacting by requests
- In a theorem prover (Isabelle) → Mechanized proofs
- To study
  - the GCM component model and its implementation(s)
  - interaction between futures and components,
  - correctness of adaptation procedures and optimisations done in the library → proofs

Here: Prove correctness of future update strategies → proofs about the middleware
Communication Model
A Primitive GCM Component

Primitive components communicating by *asynchronous* requests on interfaces

→ Components abstract away distribution and *concurrency*
Primitive Components in Isabelle

- Primitive components are defined by interfaces plus an internal behaviour, they can:
  - emit requests
  - serve requests
  - send results
  - receive results (at any time)
  - do internal actions

Some rules define a correct behaviour, e.g. one can only send results for a served request.

- We define the behaviour of the whole components as small-step operational semantics.

\[
RcvResultPrim
\]

\[
(s, ReceiveResult f v, s') \in \text{behaviour}(s)
\]

\[
s'' = s'(\mathbf{results} = [f_i \mapsto v_i | f_i \in \text{dom}(\text{results}(s)) \land v_1 = \text{updateFV}(\text{results}(s)(f_i), f, v)],
\]

\[
\text{queue} = \left[ [f_i, v_i, \text{itf}_i] | [f_i, v'_i, \text{itf}_i] \in s.\text{queue} \land v_i = \text{updateFV}(v'_i, f, v) \right]
\]

\[
S \vdash \text{Prim}[N, \text{itfs}, s] \rightarrow f, v, N \Rightarrow F \text{Prim}[N, \text{itfs}, s''], [(f'', N) f'' \in \text{RefFutSet}(v)]
\]

\[
RcvResult[simp, intro!]:
\]

"\langle\langle\text{PintState s}, \text{ReceiveResult f v}, s2\rangle \in \langle\text{Behaviour s}\rangle, \rangle \rightarrow
\langle\text{Primitive N itf s} \rangle - [f, v, N'] = f (\text{ if N = N' then

\langle\text{Primitive N itf s}\rangle\langle\text{PintState:= s2},
Pqueue:=\text{map (A R) parameter:=UpdateFuture f v (parameter R)}\rangle\langle Pqueue\ s\rangle,
PcomputedResults:=\text{map (A R) fValue:=UpdateFuture f v (fValue R)}\rangle\langle PcomputedResults\ s\rangle\rangle
\text{ else \langle Primitive N itf s \rangle },
\text{ if N = N' then \langle map (A id, (id,N)) (snd(v)) \rangle \text{ else } [] \rangle \text{ "}
"
Communication inside Composites

- Composites only delegate calls between components
- Use the bindings to know where to transmit requests
- Plus receive futures (like primitives)
- Composite behaviour expressed as small step semantics and encoded in Isabelle
Futures
Futures for Components

Component are independent entities (threads are isolated in a component) +

Asynchronous requests with results  

Futures are necessary
First-class Futures

Only strict operations are blocking (access to a future)
Communicating a future is not a strict operation
Future Update Strategies

• How to bring future values to components that need them?
• A “naive” approach: Any component can receive a value for a future reference it holds.
• More operational is the lazy approach:

Lazy future update -- « On demand »

No-unnecessary transfer of values

Two phases update: « registration delay + time for transfer »

Results stored for long term ➔ Not much operational.
Eager forward-based strategy

• A strategy avoiding to store future values indefinitely
• Future updates follow the same path as future flow
• Each component remembers only the components to which it forwarded the future

Results sent as soon as available
No additional message
Future updates form a chain ➔ intermediate components
Easy to garbage collect computed results
Eager home-based future update

- A strategy avoiding to store future values indefinitely
- Relies on future registration and sends the value as soon as it is calculated

Results sent as soon as available

Every component with future reference is registered

Un-necessary transfers

Garbage collection of computed results possible

Formalised in Isabelle
Formalisation and properties
GCM-like Component with Eager-home

Results received from other components

Incoming Requests

Request queue

Request service

End of service

Request sent

Results

Future Recipient list

Results returned to components in Future Recipient list
Composite Component Semantics – CommChild

\[
\text{CommChild} \\
\text{queue}(s) = [f, v, if\ f] \# Q \quad [This.\ itf, N'.\ itf'] \in \text{bindings} \quad f' \notin \text{RefFutSet}(S) \\
\text{host}(f') = N' \quad \text{subCp'} = \text{subCp} \leftarrow (\text{Enqueue}((\text{subCp} \uparrow N'), [f', v, if\ f'])) \\
s' := s([\text{queue} := Q, \text{results} := \text{results}(s)[f \mapsto (V_f, f')]]) \\
S \vdash \text{Comp}[N_0, Itfs, subcp, \text{bindings}, s] \rightarrow_R \text{Comp}[N_0, itfs, subcp', \text{bindings}, s'], \\
[f', N_0] \# [(f'', N')] \quad f'' \in \text{RefFutSet}(v)
\]

CommChild[simp, intro!":"
[[ C\ queue\ s = R\#Q; \{\text{src=This (invokedIf}\ R)},\text{dest=N'},\text{f id i2}\} \in \text{bindings;} \\
\ f\notin(\text{set } (\text{RqIdList } S)) ; \text{subCp'}\uparrow N' = \text{Some } C'] \rightarrow \\
S \vdash (\text{Composite } N \ itf \ subCp \text{ bindings } s) \rightarrow_R \text{Composite } N \ itf \\
\langle\text{subCp}<\langle C'\leftarrow\langle i\text{id=f, parameter=(parameter } R\rangle,\text{invokedIf}=i2\rangle\rangle \text{ bindings} \\
\langle\text{s}\langle C\ queue\ :=\ Q, C\ computed\ Results\ :=\ C\ computed\ Results\ s@ [(f\text{id= id R, f\Value=O,}[f])]]\rangle, \\
\langle f, N\rangle\#(\text{map } \lambda\text{id, (id, N')} \langle\text{snd}(\text{parameter } R)\rangle)"
\]
Properties on Future updates

- Future updates remove all references to a given future

```
lemma UpdatedFutureDisappear:
"\[[ S \vdash f, v, N \mapsto F S2, RL; CorrectComponent S; (S2 \sim N) = Some C ; f \notin set (snd v)]
\rightarrow f \notin LocalRefFutSet C)"
```

- All Future references are registered during reduction

```
theorem FuturesRegistered:
"\[ \vdash C1 \sim C2; CorrectComponent C1; GlobalRegisteredFuturesComp C1\]
\rightarrow GlobalRegisteredFuturesComp C2"
```
CONCLUSION
Formalisation in Isabelle

- > 4000 lines,
- Almost 300 lemmas and theorems,
- ~ 500 lines definitions: component model and semantics
- 1800 lines for future update specific properties
- A few tricky design choices (lists vs sets, how to store names, …)
- A couple of axioms remaining (reduction correctness)

**short-term goal: correctness of various strategies**

**Longer term: reconfiguration and execution**
Conclusion

- GCM-like Component Model
  - Hierarchical components
  - Asynchronous communication
  - First class futures

- Formalization in Isabelle/HOL
  - Formal specification + one future update strategy
  - Tools (lemmas + constructs) for manipulating components + futures

- Proofs of correctness for future update and registration

- A framework for mechanical proofs:
  - Formal constructs to study components, futures, reconfiguration, …
THANK YOU !
Without first-class futures, one thread is systematically blocked in the composite component. A lot of blocked threads
Auxiliary lemmas

- Registered futures are preserved during reduction

\[
\text{lemma } \text{R\_maintainsRegFutures: }\\
\left[ S \vdash C \rightarrow_R C', \text{RL};\text{CorrectComponent } C;\text{RegisteredFuture } f \text{ N } C; C \in cpSet S \right] \implies \text{RegisteredFuture } f \text{ N } C''
\]

- Futures references from a subcomponent in C2 are either registered in S or are in RL

\[
\text{lemma } \text{registeredFutures}_R:\\\n\left[ S \vdash C \rightarrow_R C', \text{RL}; C \in cpSet S; \forall C \in cpSet C1. \text{LocalRegisteredFuturesComp } C S; C' \in cpSet C2; f \in \text{LocalRefFutSet } C' \right] \implies \text{RegisteredFuture } f \text{ (getName } C') S \lor (f, \text{getName } C') \in \text{set RL}
\]

- Basic lemmas

\[
\text{lemma } \text{cpSetFirst: } C \in cpSet C
\]

\[
\text{lemma } \text{getCp\_inlist: } CL \land N = \text{Some } C \implies C \in \text{set } CL \land \text{getName } C = N
\]

\[
\text{lemma } \text{red\_names\_eq: } \left[ S \vdash c1 \rightarrow_R c2, \text{RL}; \text{CorrectComponentWeak } c1 \right] \implies \text{getName } ' (cpSet c2) = \text{getName } ' (cpSet c1)
\]
Remaining axioms

axioms Fut_maintains_WF:
"S :-[f, v, N]→F S2 ,RL ⟹ CorrectComponentWeak S ⟹ CorrectComponentWeak S2"

axioms comm_maintains_WF[rule_format]:
"S ∪ C :-[i1, i, v]→O C' ⟹ CorrectComponentWeak C ⟹ CorrectComponentWeak C'"

axioms red_maintains_WF[rule_format]:
"S ∪ C →R C',RL ⟹ CorrectComponentWeak C ⟹ CorrectComponentWeak C'"

axioms red_maintains_WF_Strong[rule_format]:
"S ∪ C →R C',RL ⟹ CorrectComponent C ⟹ CorrectComponent C'"

axioms fut_maintains_WF_Strong[rule_format]:
"S :-[f,v,N]→fS2,RL ⟹ CorrectComponent C ⟹ CorrectComponent C'"