Formalism and Platform for Autonomous Distributed Components

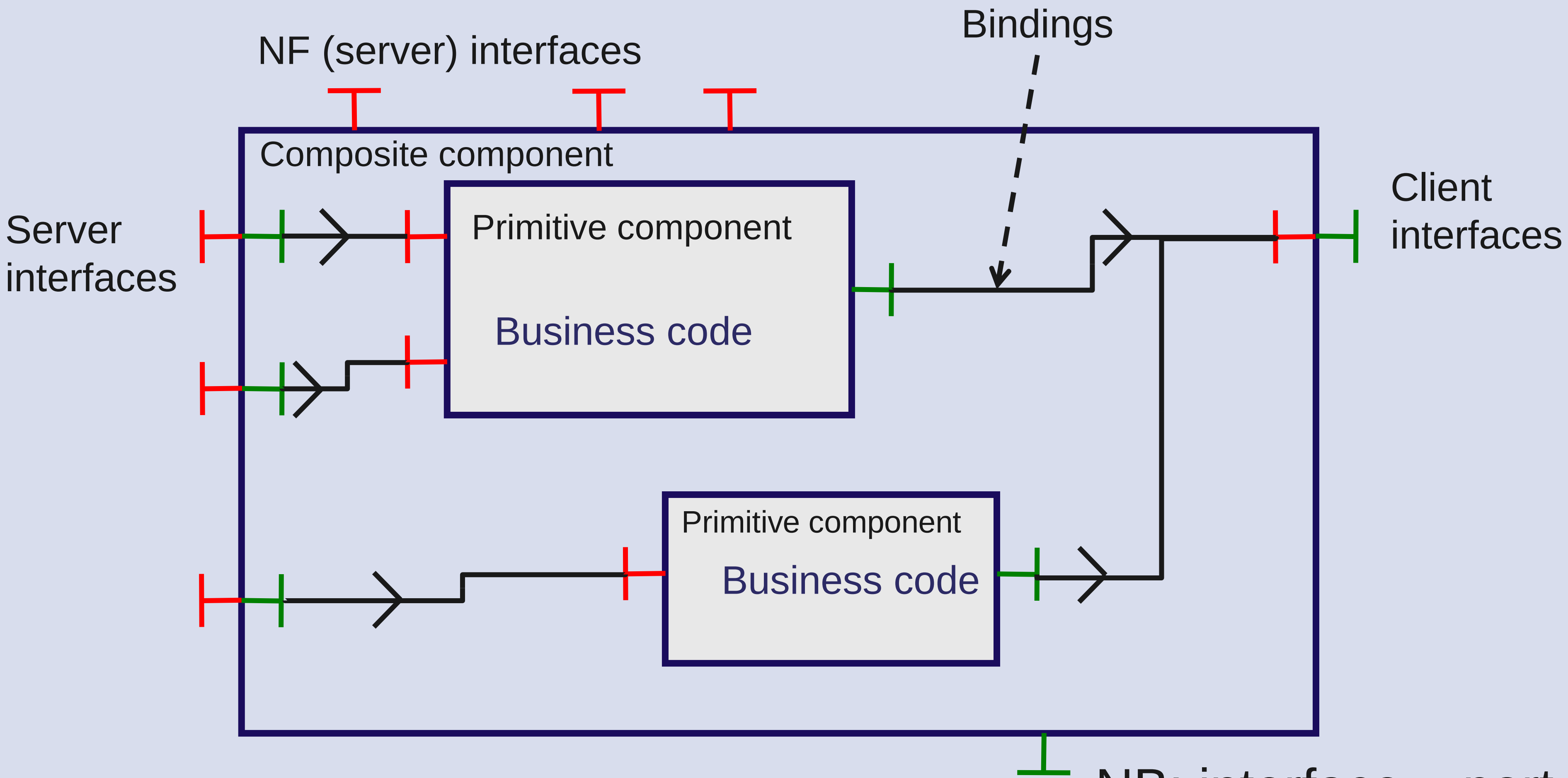
OASIS team - Ludovic Henrio

- A Distributed Component Model
- Formalisation in Isabelle
- Autonomous components:
  Componentise component management and distributed reconfiguration

Reseco 2009

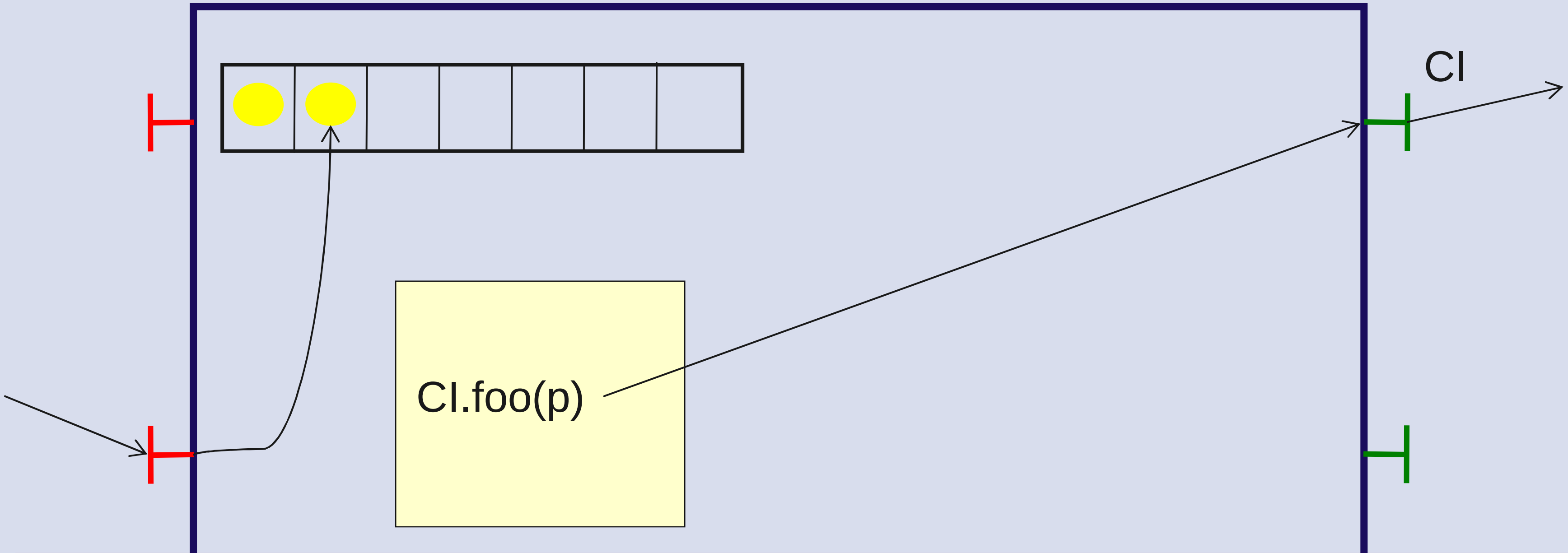
A DISTRIBUTED COMPONENT MODEL

What are (GCM) Components?



NB: interface = port

A Primitive GCM Component

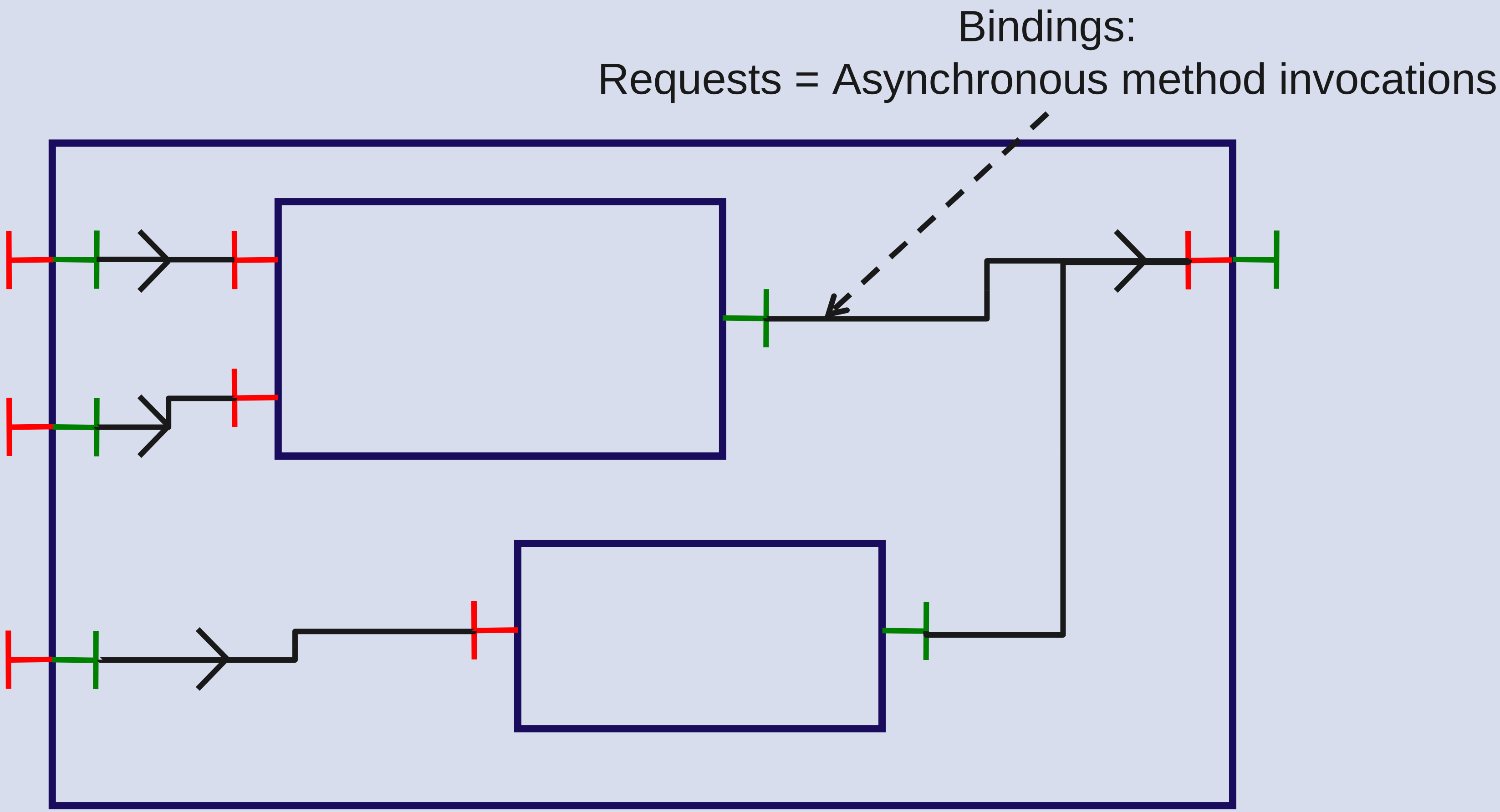


Primitive components communicating by *asynchronous* remote method invocations on interfaces (*requests*)

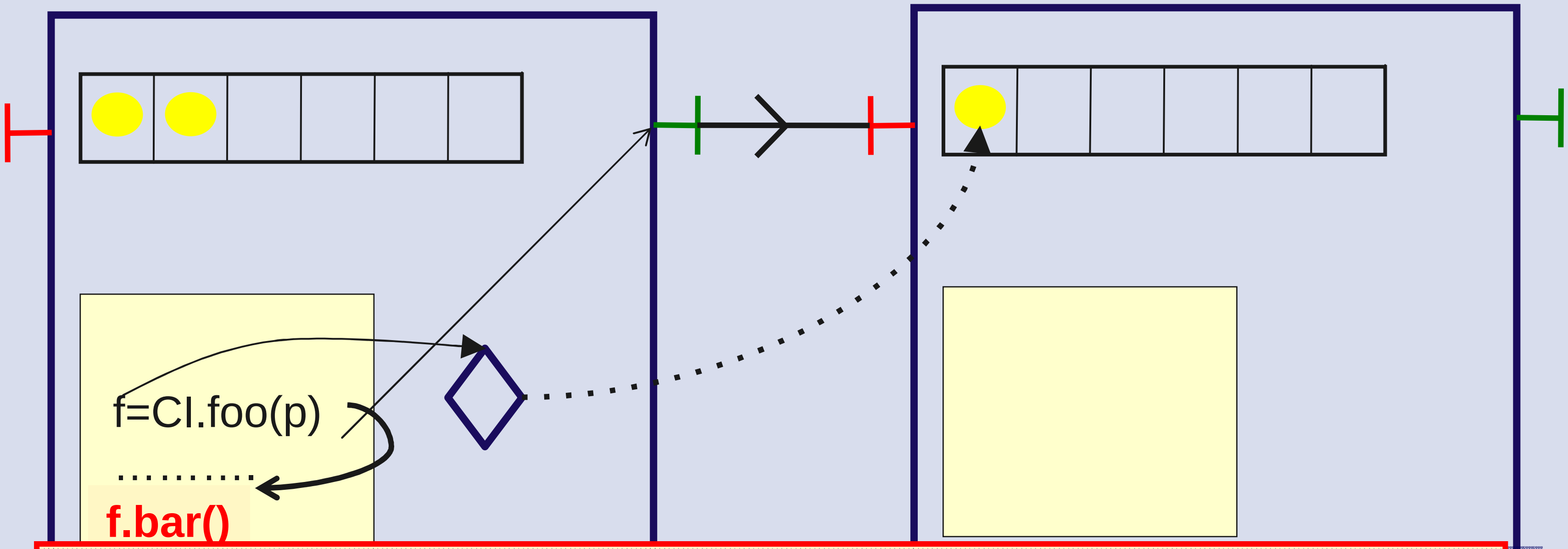
➔ Components abstract away distribution and *concurrency*

in ProActive (reference implementation) components are mono-threaded
➔ **simplifies concurrency** but can create **deadlocks**

Composition in GCM

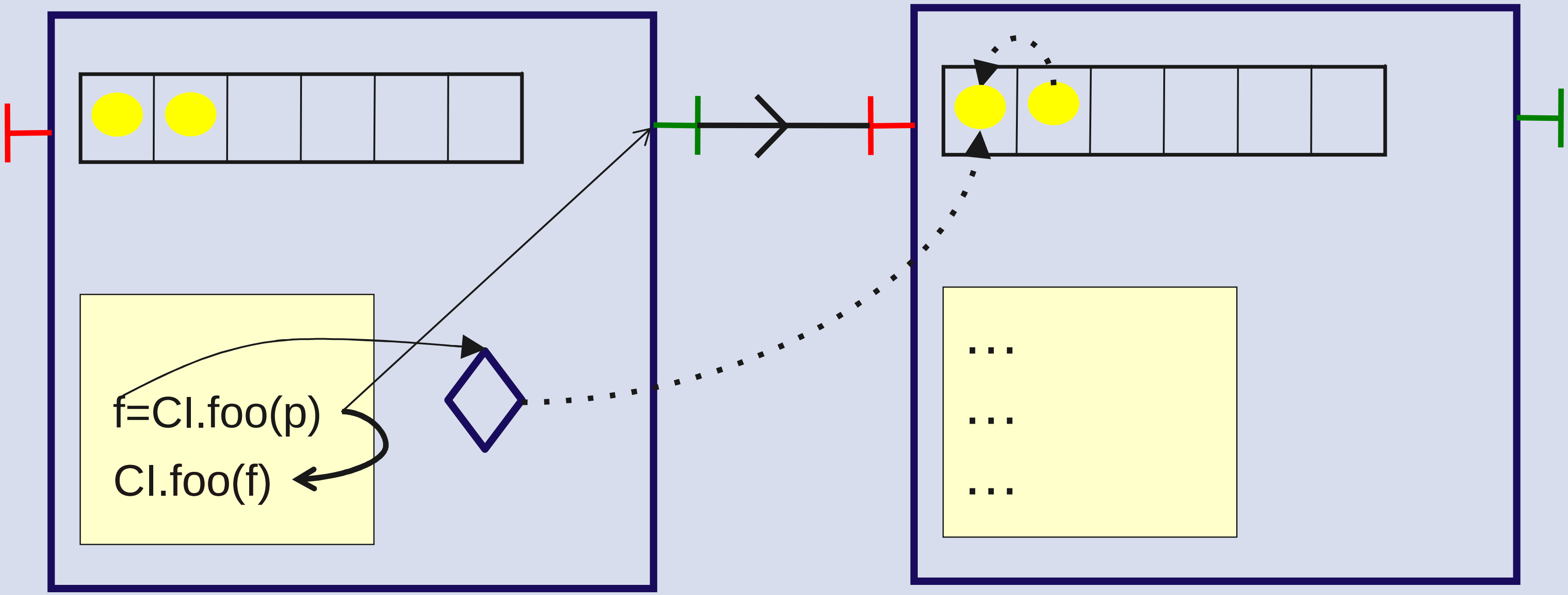


Futures for Components



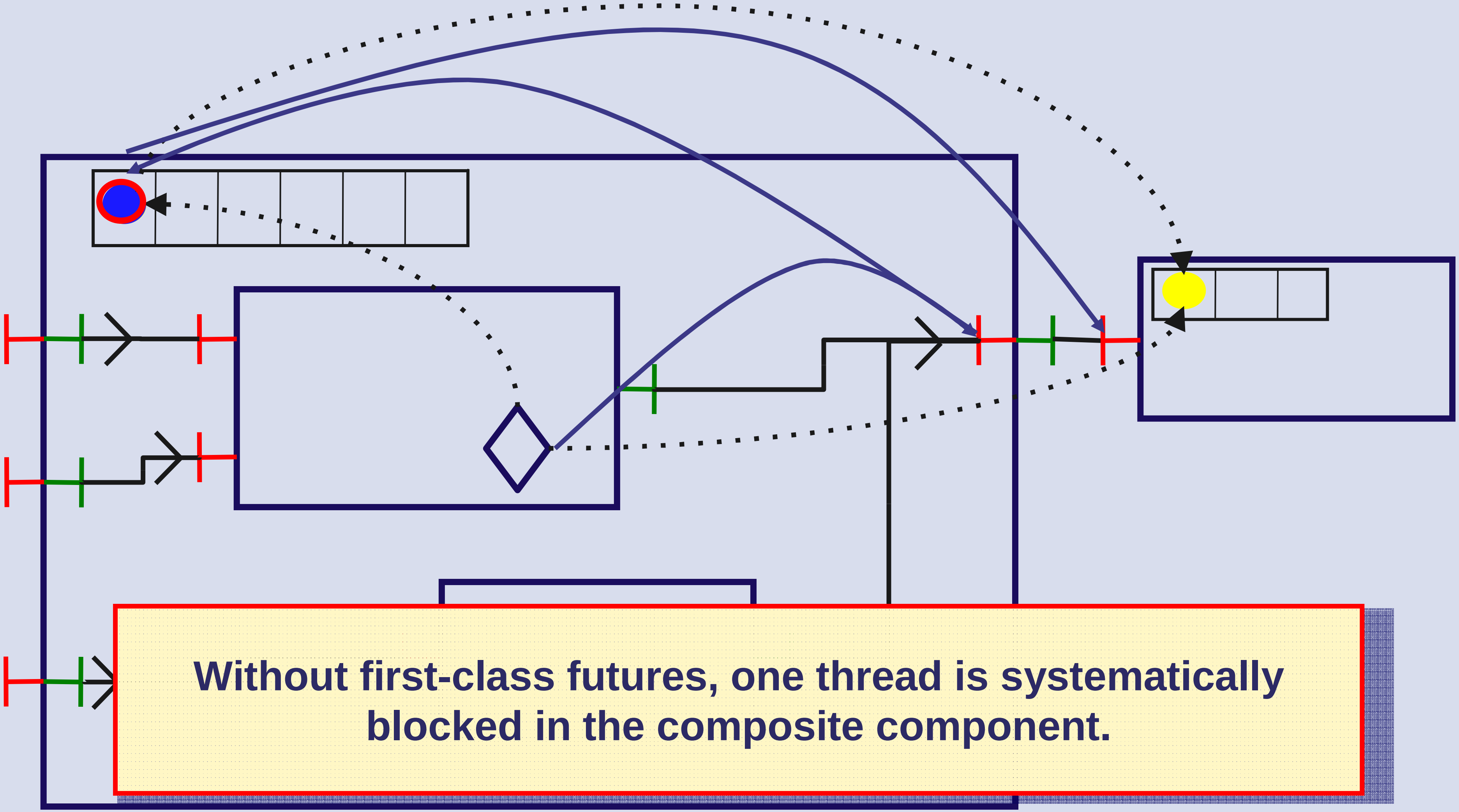
Component are independent entities
(threads are isolated in a component)
+
Asynchronous method invocations with results
⇓
Futures are necessary

First-class Futures



- **Only strict operations are blocking (access to a future)**
- **Communicating a future is not a strict operation**

First-class Futures and Hierarchy



Without first-class futures, one thread is systematically blocked in the composite component.

Approach: a refined GCM model

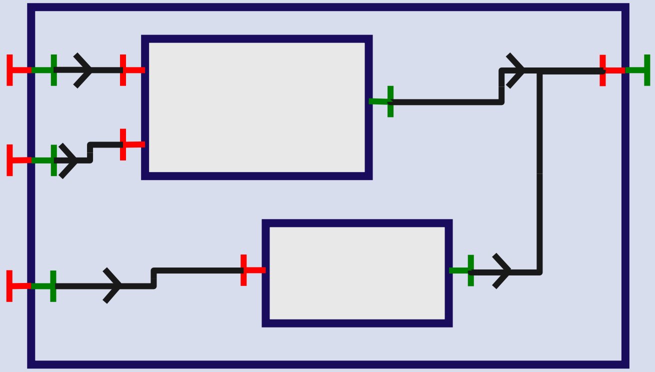
- A model:
  - more precise than GCM, give a semantics to the model:
    - future / requests
    - request queues
    - no shared memory between components
    - notion of request service
  - less precise than ProActive/GCM
    - can be multithreaded
    - no “active object” model

FORMALISATION

Objectives

- A model general enough to study GCM, but also other component models interacting by requests
- In a theorem prover (Isabelle)
- To study
  - the GCM component model,
  - its implementation(s),
  - interaction between futures and components,
  - component reconfiguration and management

Principles



- component architecture:
  - bindings
  - interfaces (only functional)
  - component composition
- communications by requests and futures
  - request queues
  - future references
- abstraction of the business code by a behaviour (~LTS)
- values abstracted away: we just keep track of future references

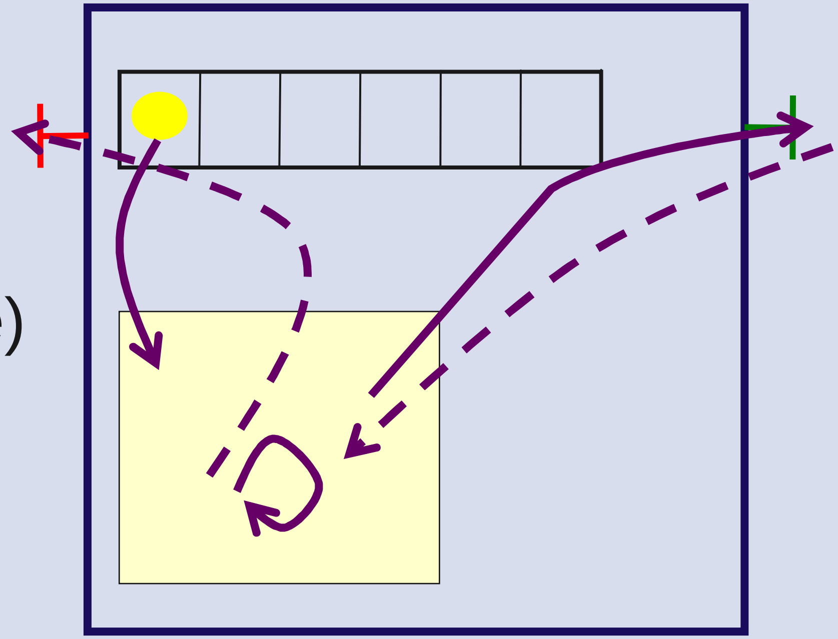
```
types Value = "natx(Fid list)"
```

Primitive Components

- 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,

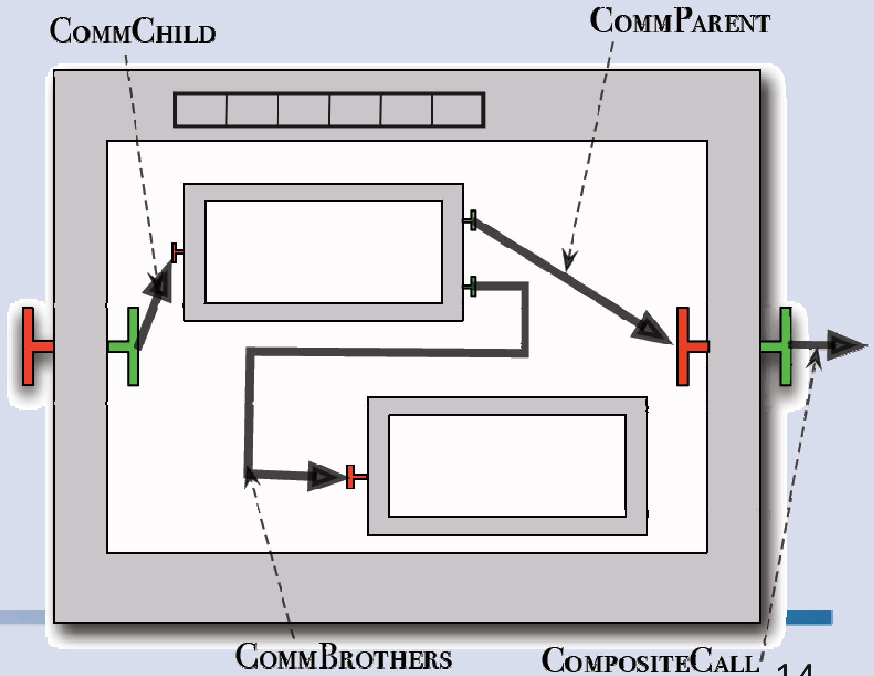
  e.g. one can only send result for a served request



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

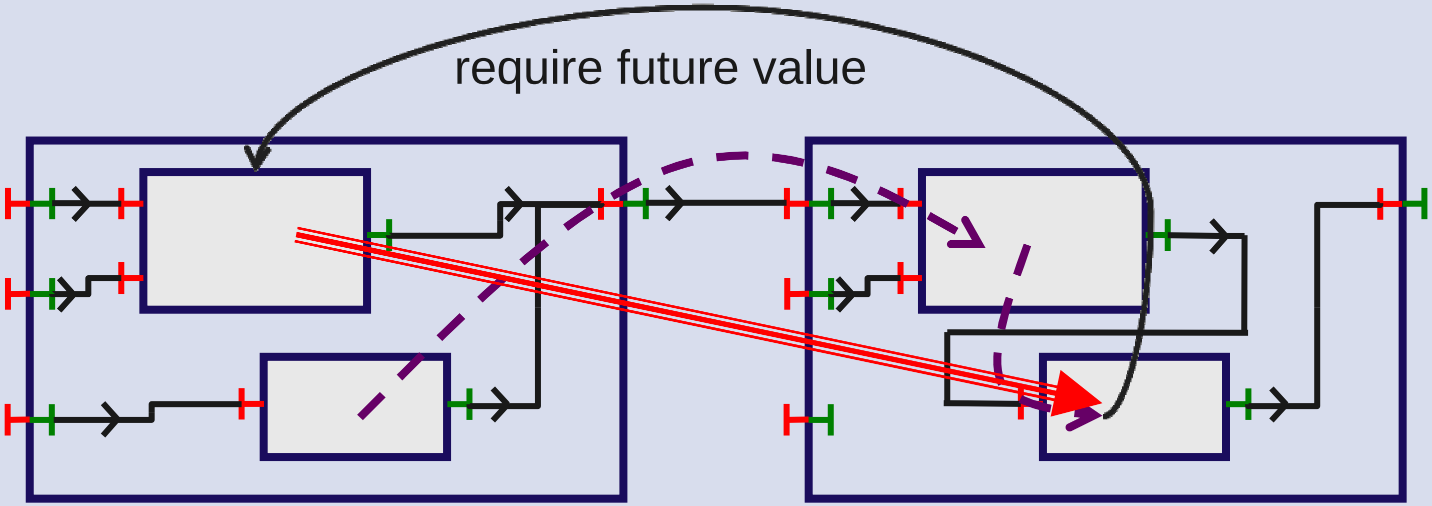
Composite Components

- Composite components are defined by their interfaces + content + bindings
- Semantics
  - Composites also have request queues (futures)
  - Only delegate calls to inner or outer components
  - Use the bindings to know where to transmit requests
  - Plus receive futures (like primitives)



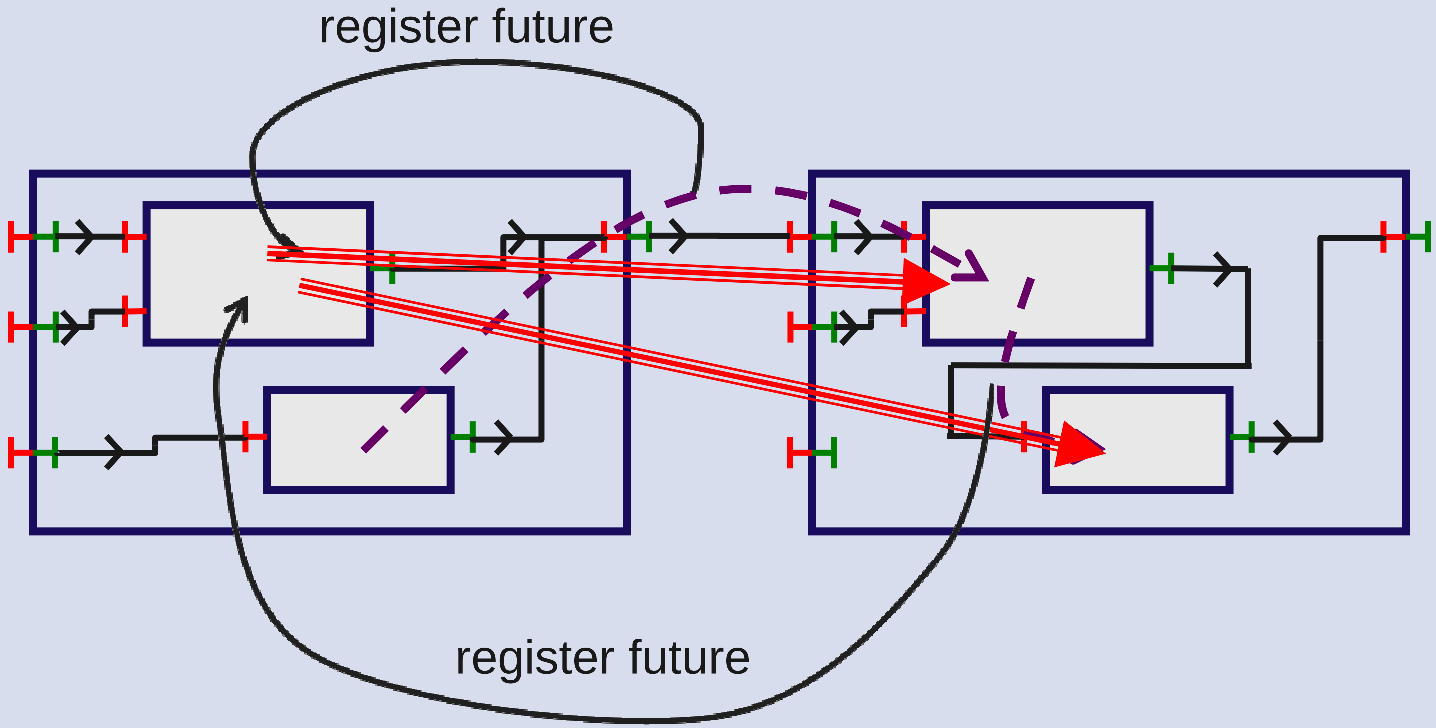
Future Update Strategies

- How to bring future values to components that need them
- Different strategies can be envisioned
- A “naive” approach: Any component can receive a value for a future reference it holds. Not much operational.
- More operational is the lazy approach:



Eager home future update

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



First Proofs (ongoing)

- Future update remove all references to a given future

```
lemma UpdFutRed_futdisappear:
"S -[f, v, N]→F S2  ,RL ⟹ CorrectComponentWeak S ⟶
(S2^^N = Some C⟶f∉set (snd v)⟶f∉ LocalReferencedRqs C)"
```

- All Future references are registered during reduction

```
theorem registeredFutures: "⊢ C1 ↝ C2 ⟹
(GlobalRegisteredFuturesComp C1 ⟶GlobalRegisteredFuturesComp C2)"
```

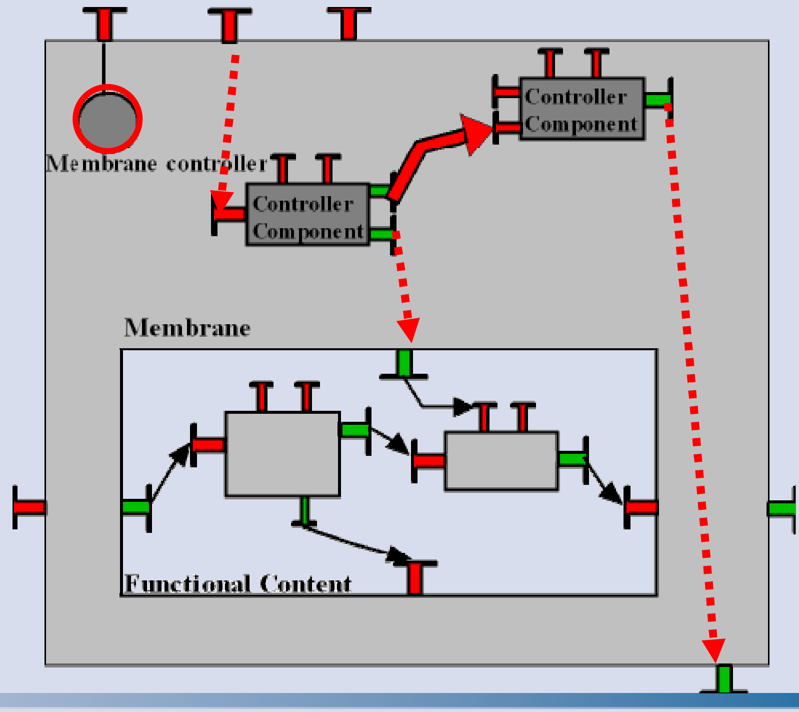
A formalisation in Isabelle of Component structure + request / futures

Middle-term goal: correctness of various strategies

A PLATFORM FOR AUTONOMOUS COMPONENTS

Non-functional Component Structure

- Non-functional aspects as a composition of components (inside a membrane)
  - A component structure for the membrane
  - New kind of interfaces and bindings
  - An API for reconfiguring the membrane
  - Non-functional code is
    - components or objects
    - distributed or not



Adaptation in the GCM

- **Functional adaptation**: adapt the architecture + behaviour of the application to new requirements/objectives
  - add a new functionality
- **Non-functional adaptation**: adapt the architecture of the container+middleware to changing environment/NF requirements (QoS …)
  - Change communication protocol
  - Update security policy

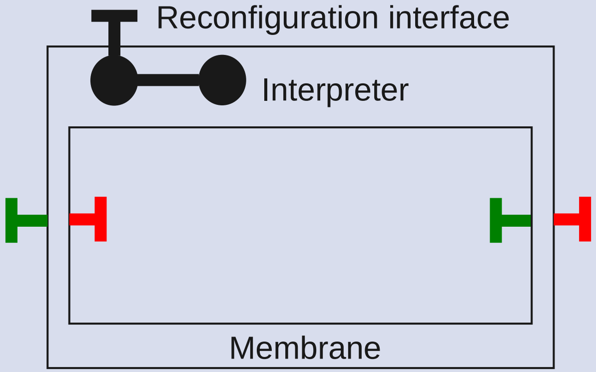
Both functional and non-functional adaptation are expressed as reconfigurations

How to express reconfigurations?

- Fractal / GCM defines an API for reconfiguring components
- We start from FScript:
  - A Scripting reconfiguration language
  - Dedicated to Fractal components
  - FPath expressions: navigate and select elements in the components architecture
  - Centralized execution

A Controller for Reconfigurations

- Manages and allows the invocation of the script interpreter
- Is collocated with the component
- Exposes methods for reconfiguration
  - `setInterpreter(interpreterClassName)`
  - `loadScript(scriptFileName)`
  - `executeAction(actionName, arguments...)`

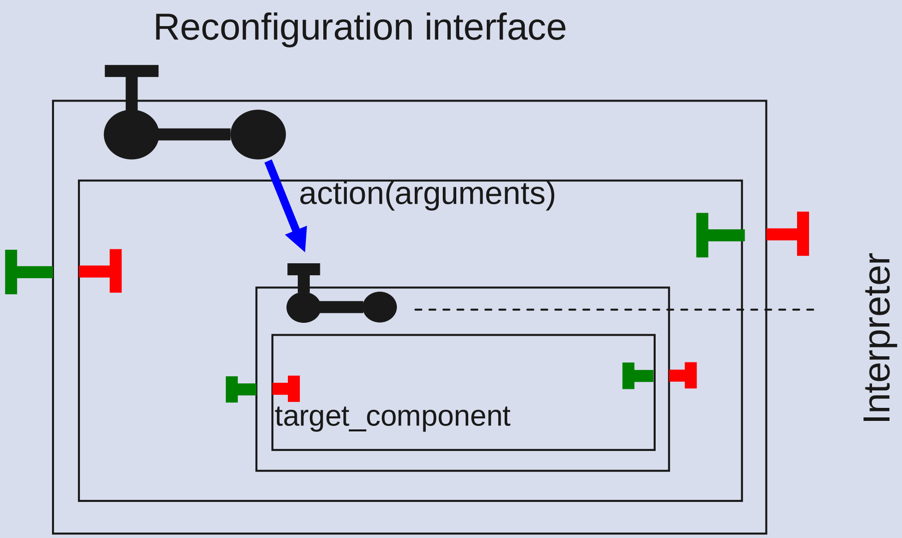


Triggering Distributed Reconfigurations in the Scripting Language

A primitive for the distributed script interpretation

```
remote_call(target_component, action_name, parameters, ...);
```

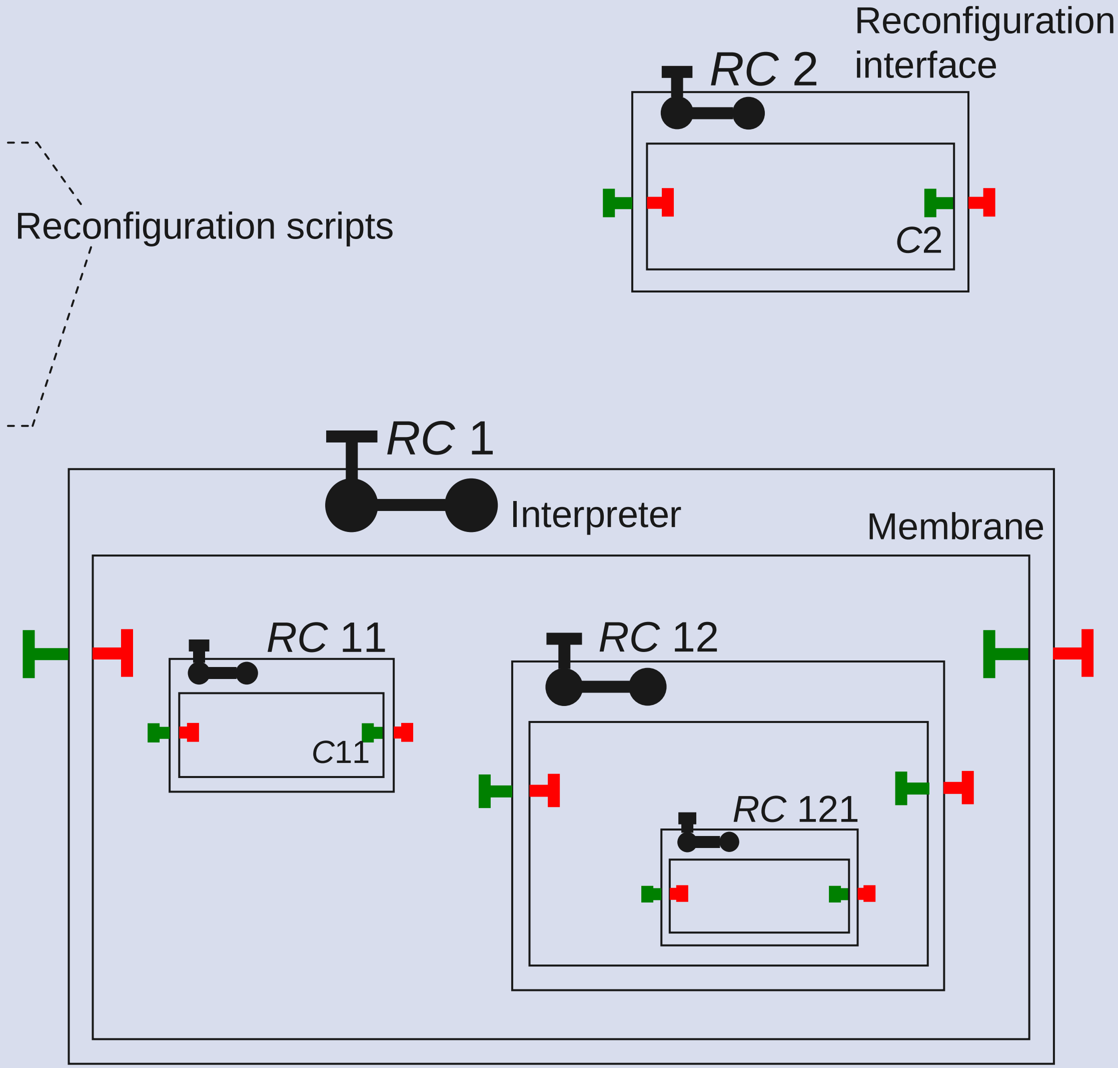
- Triggers the action `action_name` by the interpreter located in `target_component`
- Receives action arguments as parameter



Example

```
definition action1
        ...
remote_call(C2,'action2')
remote_call(C11,'action11')
remote_call(C12,'action12')
        ...
```

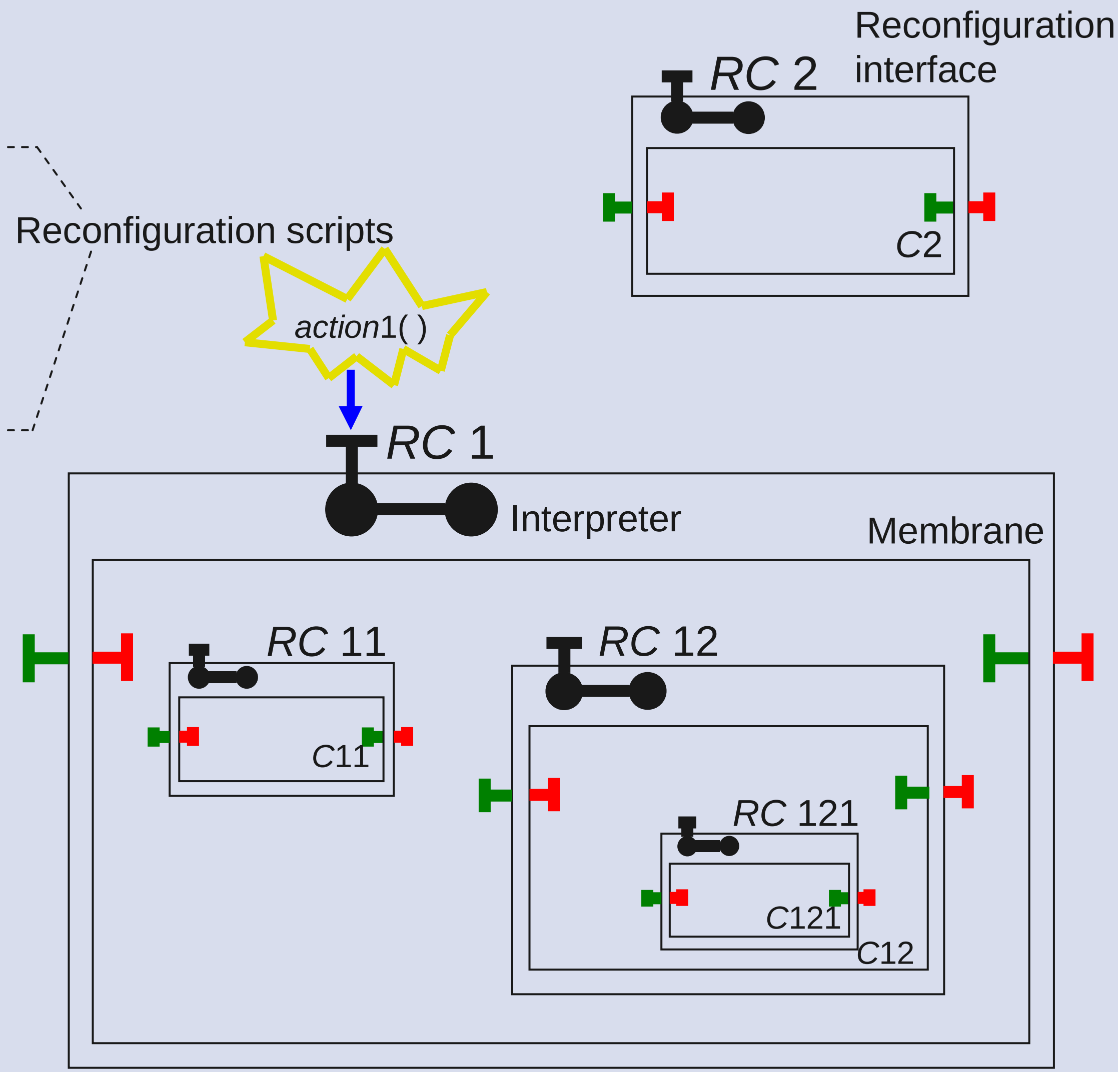
```
definition action12
        ...
remote_call(C121,'action121')
        ...
```



Example

```
definition action1
        ...
remote_call(C2,'action2')
remote_call(C11,'action11')
remote_call(C12,'action12')
        ...
```

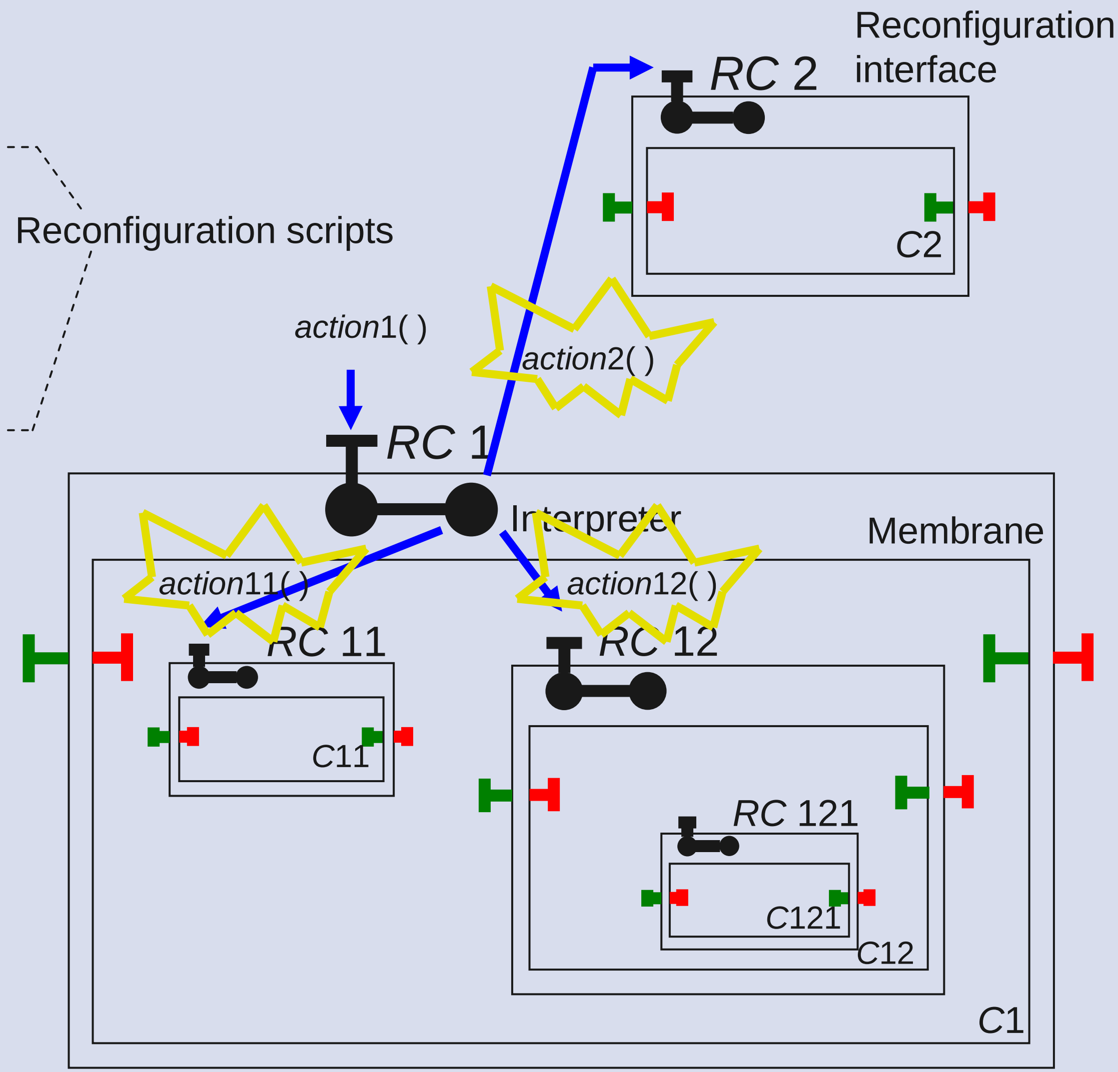
```
definition action12
        ...
remote_call(C121,'action121')
        ...
```



Example

```
definition action1
        ...
remote_call(C2,'action2')
remote_call(C11,'action11')
remote_call(C12,'action12')
        ...
```

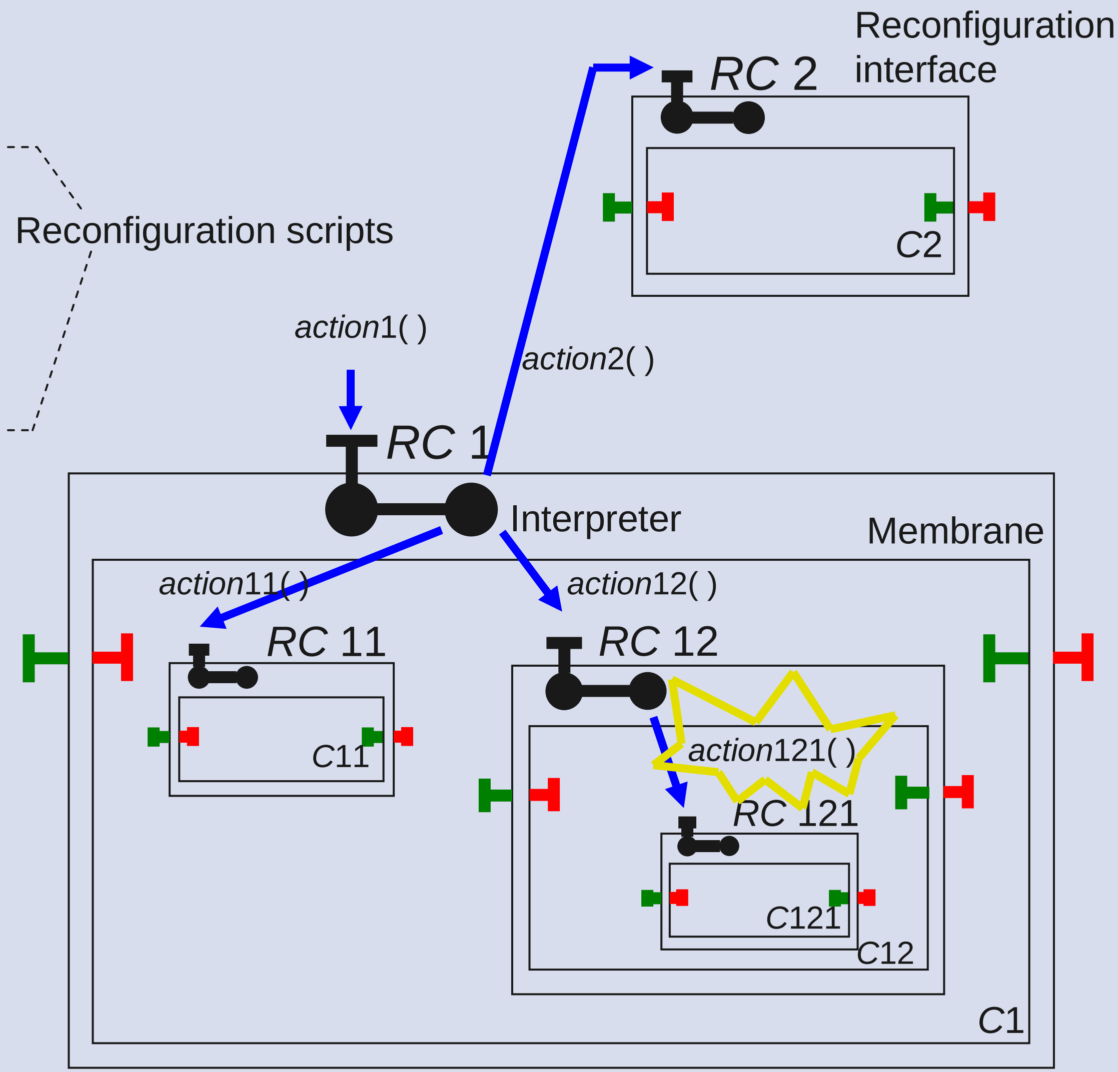
```
definition action12
        ...
remote_call(C121,'action121')
        ...
```



Example

```
definition action1
        …
remote_call(C2,'action2')
remote_call(C11,'action11')
remote_call(C12,'action12')
        …
```

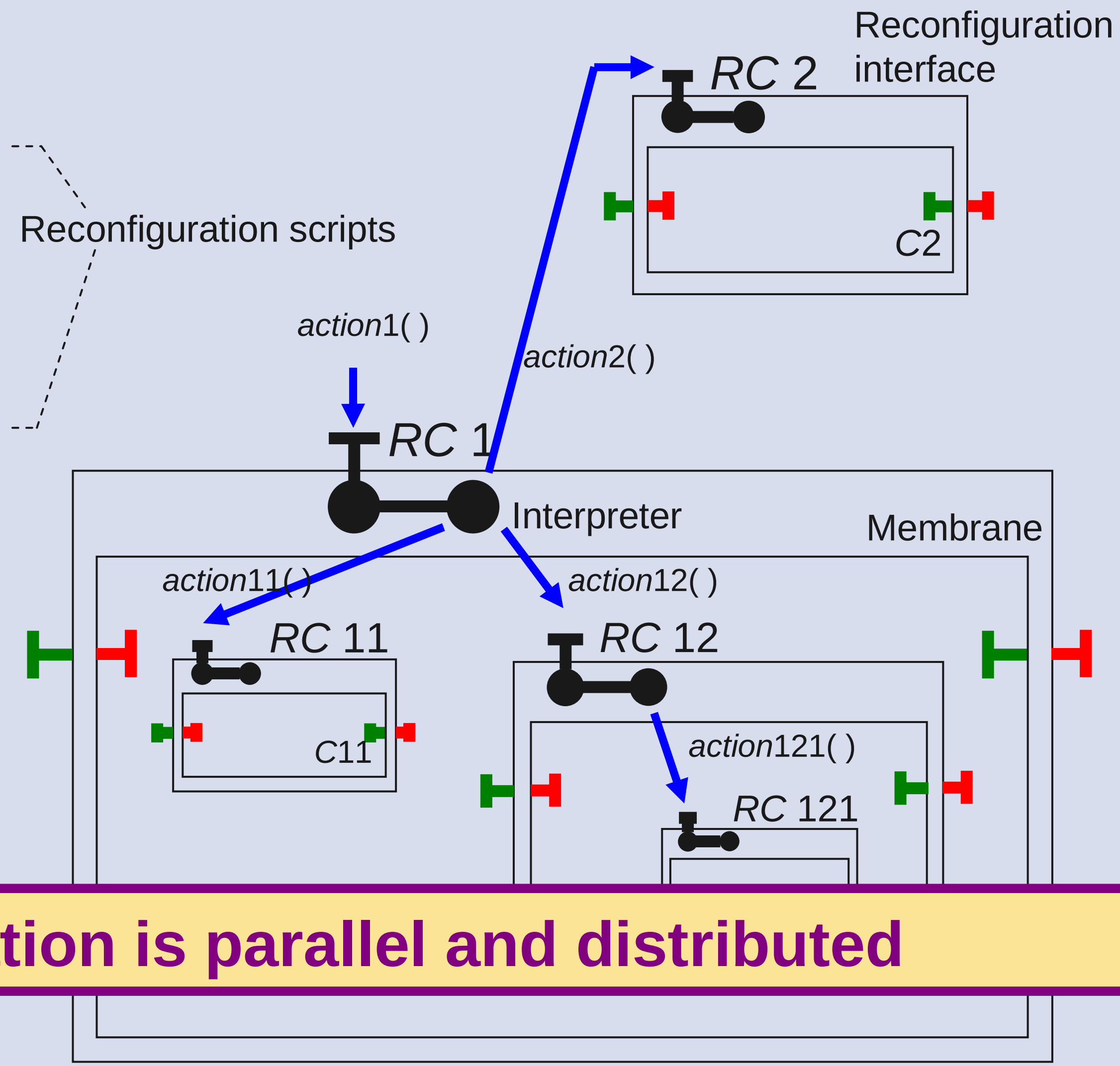
```
definition action12
        …
remote_call(C121,'action121')
        …
```



Example

```
definition action1
        ...
remote_call(C2,'action2')
remote_call(C11,'action11')
remote_call(C12,'action12')
        ...
```

```
definition action12
        ...
remote_call(C121,'action121')
        ...
```



The interpretation is parallel and distributed

Conclusion

- A component model for adaptive autonomous components
  - structured membrane
  - distributed reconfiguration
- A platform supporting adaptation and distribution
  - based on ProActive (active objects, distribution)
  - ADL (membrane and business code composition)
- A formalisation to study the component model and its implementation
  - toward verification of component management procedures

Ongoing and Future Works

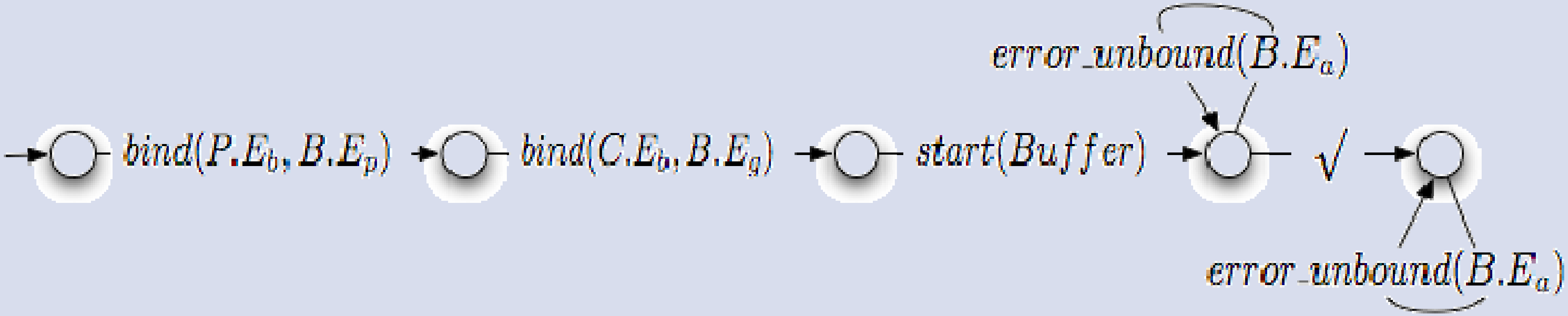
- Formalisation is a long process:

  For the moment, more a formal specification + a few proofs than a complete verification environment

- Verification of (re)configuration procedures by model checking
  - Complementary with Theorem proving approach application specific vs. model properties
  - Already model generation relies on ASP properties

Verification of Properties: Deployment

start Buffer without linking the alarm interface



- The deployment is always successful

  [(not √)* **] <** true * .√ **>** true

- But Error during deployment

  [(not √) * . O_E **]** false

Verification of Properties

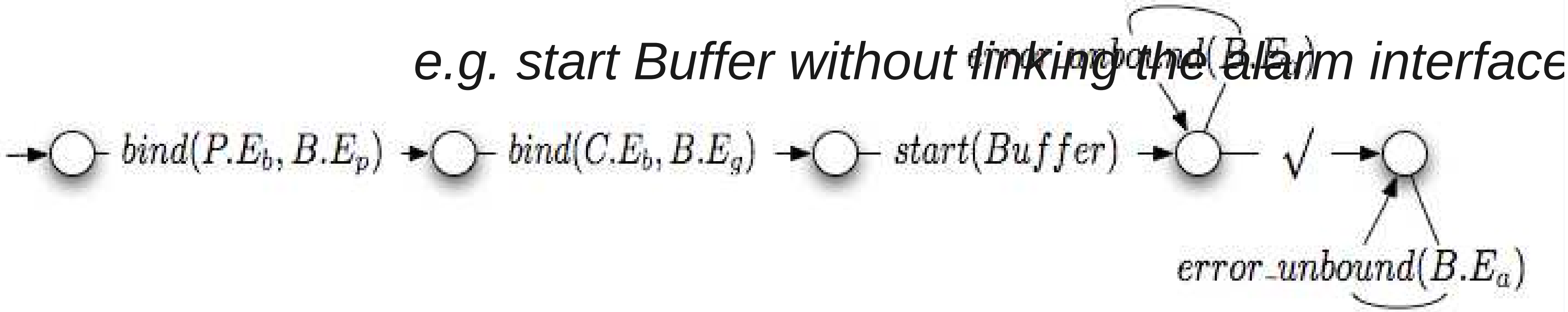
regular μ-calculus (Mateescu'2004)

- Deployment
- (on the Static Automaton with successful synchronisation visible)
  - The deployment is always successful

    [(not √)* **] <** true * .√ **>** true
  - No Error during deployment

    [(not √) * . O_E **]** false

e.g. start Buffer without linking the alarm interface

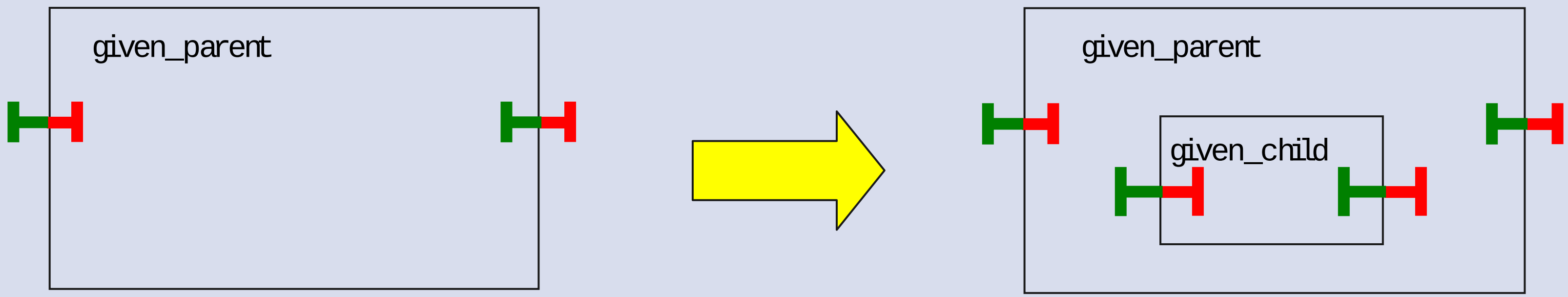


Verification of Properties

- Functional properties under reconfiguration (respecting the topology)
  - Future update (asynchronous result messages) independent of life-cycle or binding reconfigurations
  - Build a model including life-cycle controllers,
    with the reconfiguration actions visible: $?unbind(C.E_b, B.E_g)$ $?stop(C)$
  - Then we can prove:
    [true*.Req_Get()] μX. (< true > true ∧
    [¬Resp_Get()] X)

Example

- Add the component `given_child` to the composite `given_parent`




```
remote_call($given_parent, 'add', '$given_parent', 'given_child');
```

GCM API for Reconfiguration

- Life-cycle controller

```
string getFcState ();
void startFc () throws IllegalLifeCycleException;
void stopFc () throws IllegalLifeCycleException;
```

- Binding controller

```
string[] listFc ();
any lookupFc (string clientItfName)
            throws NoSuchInterfaceException;
void bindFc (string clientItfName, any serverItf) throws ...
void unbindFc (string clientItfName) throws...
```

- Content Controller

```
any[] getFcInternalInterfaces ();
any getFcInternalInterface (string itfName) throws ... Component[] getFcSubComponents
();
void addFcSubComponent (Component c) throws ...
void removeFcSubComponent (Component c) throws ...
```