Formal Models for Programming and Composing Correct Distributed Systems

Ludovic Henrio

LIAMA open day

Shanghai – April 2013

OASIS Team
INRIA – UNS – I3S – CNRS
Sophia Antipolis

Objective

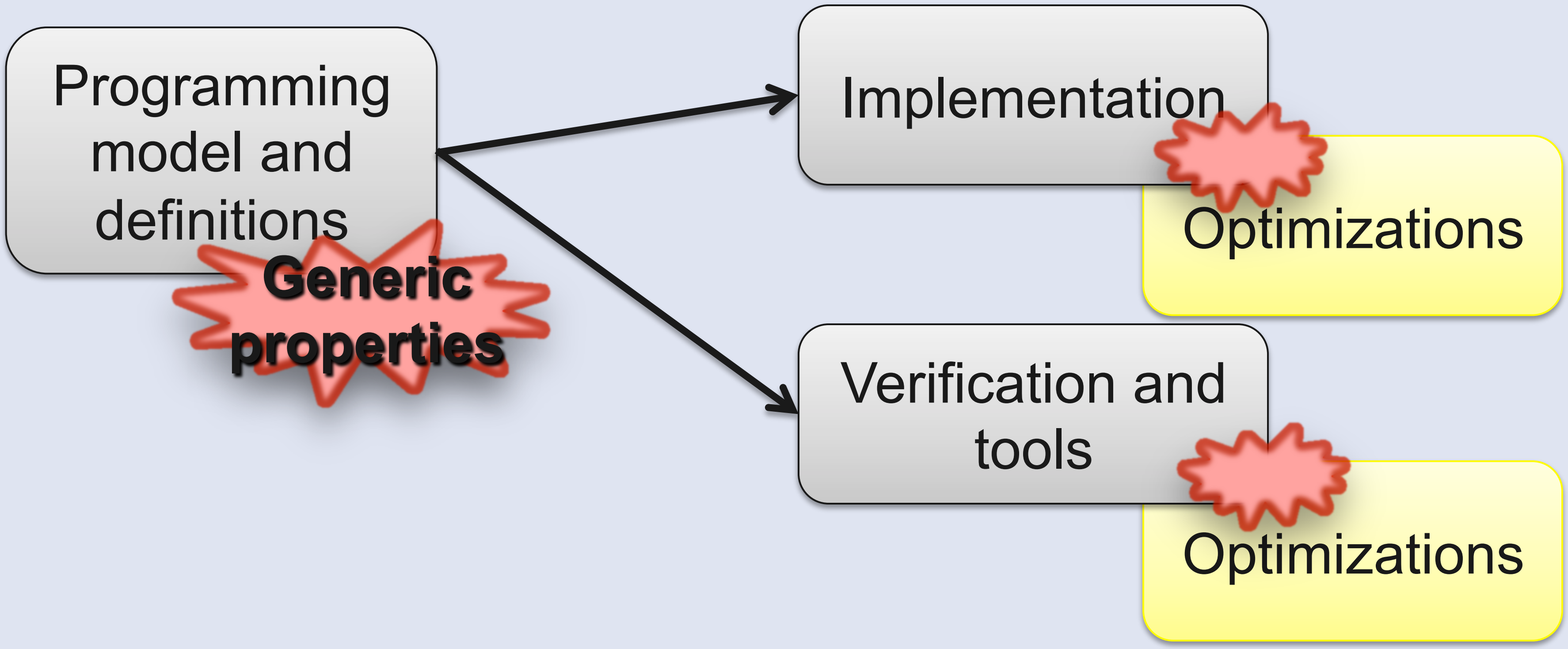
Help the programmer write
correct distributed applications, and run them safely.

- By designing languages and middlewares
- By proving their properties
- By providing tools to support the development and proof of correct programs

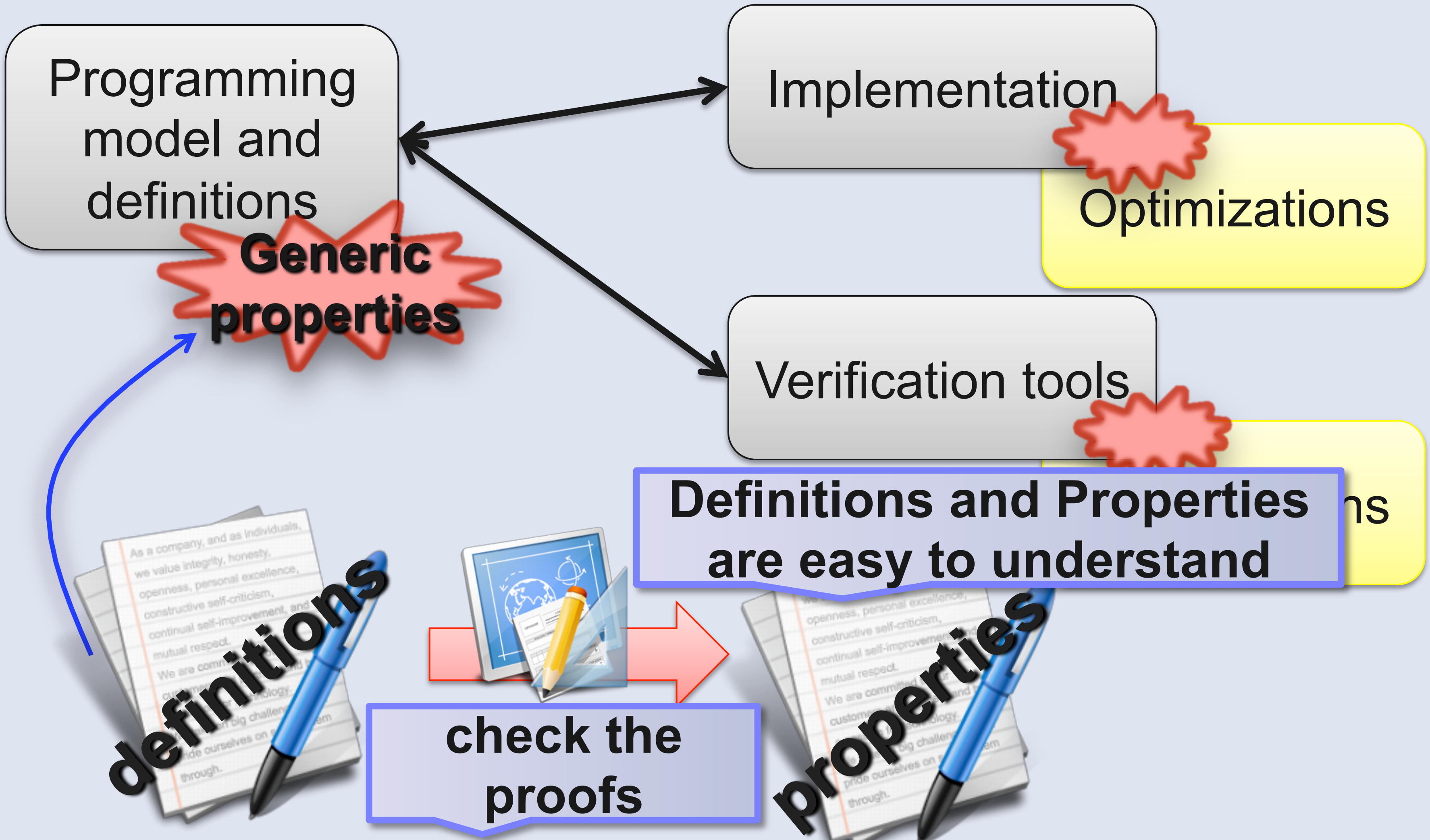
*Programming **easily correct** concurrent programs is still a challenge*

***Distributed** programming is even more difficult, and more and more useful (cloud computing, service oriented computing …)*

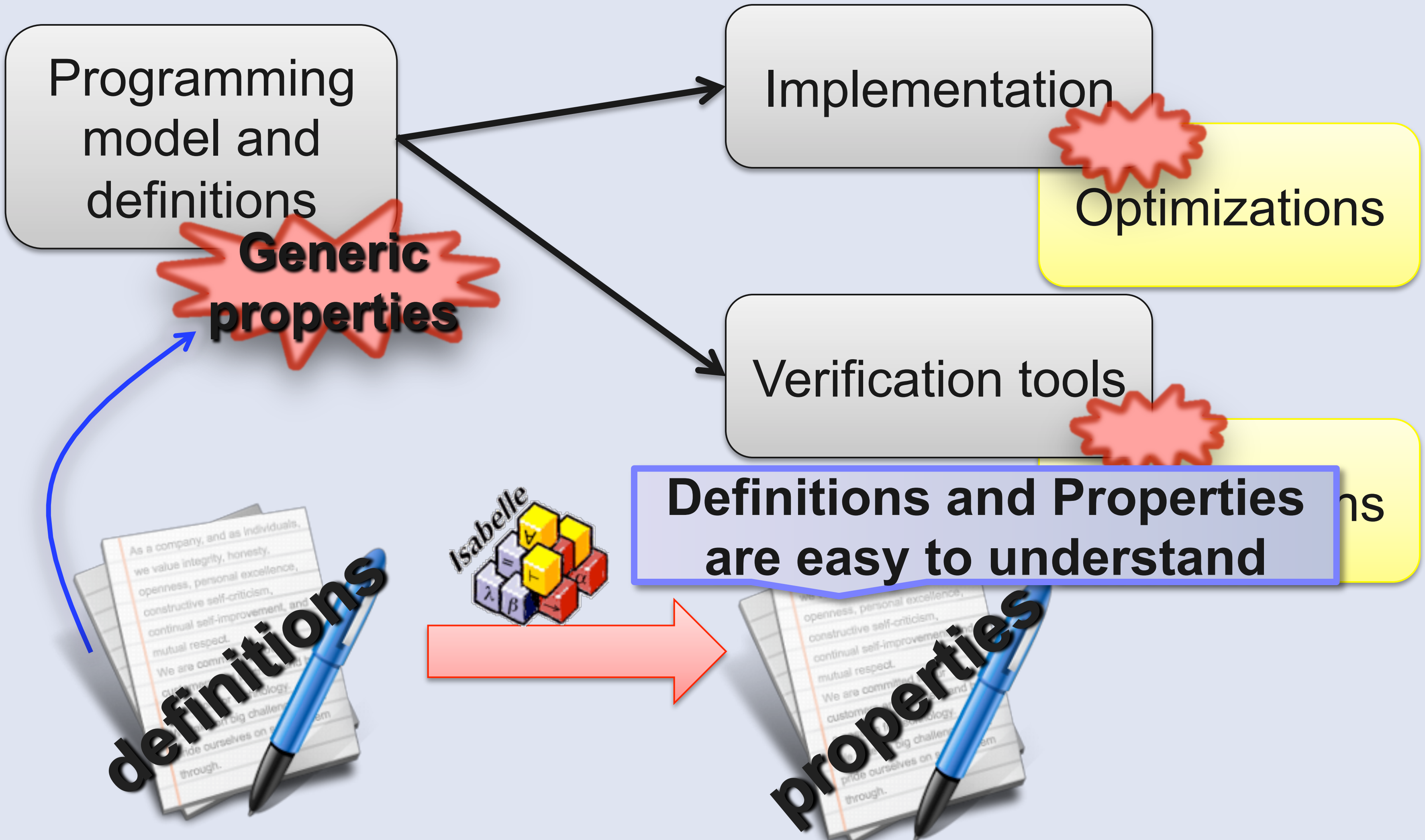
General approach



General approach



General approach



Agenda

I. Introduction: Formal Methods

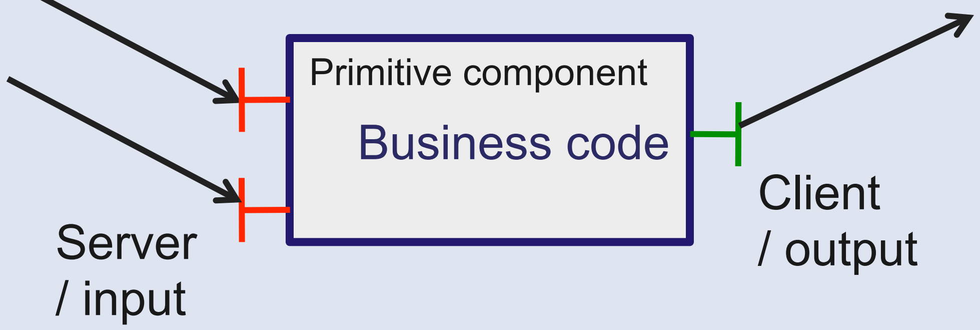
II. A Distributed Component Model

III. Behavioural Specification

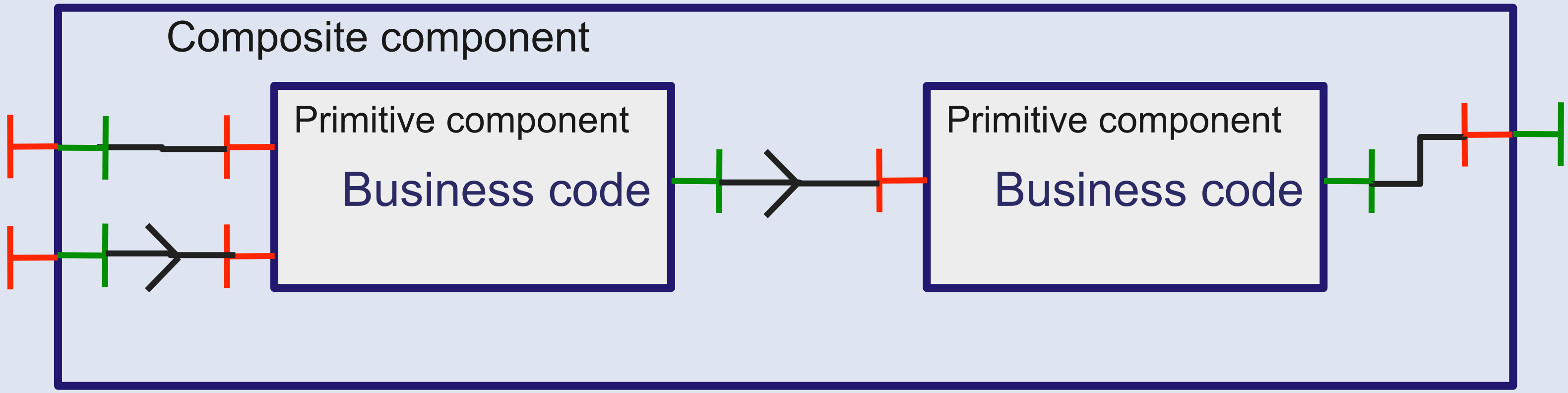
IV. Dynamicity

V. A Few Hot Topics

What are (our) Components?



What are (our) Components?



➢ **Grid Component Model (GCM)**

An extension of Fractal for Distributed computing

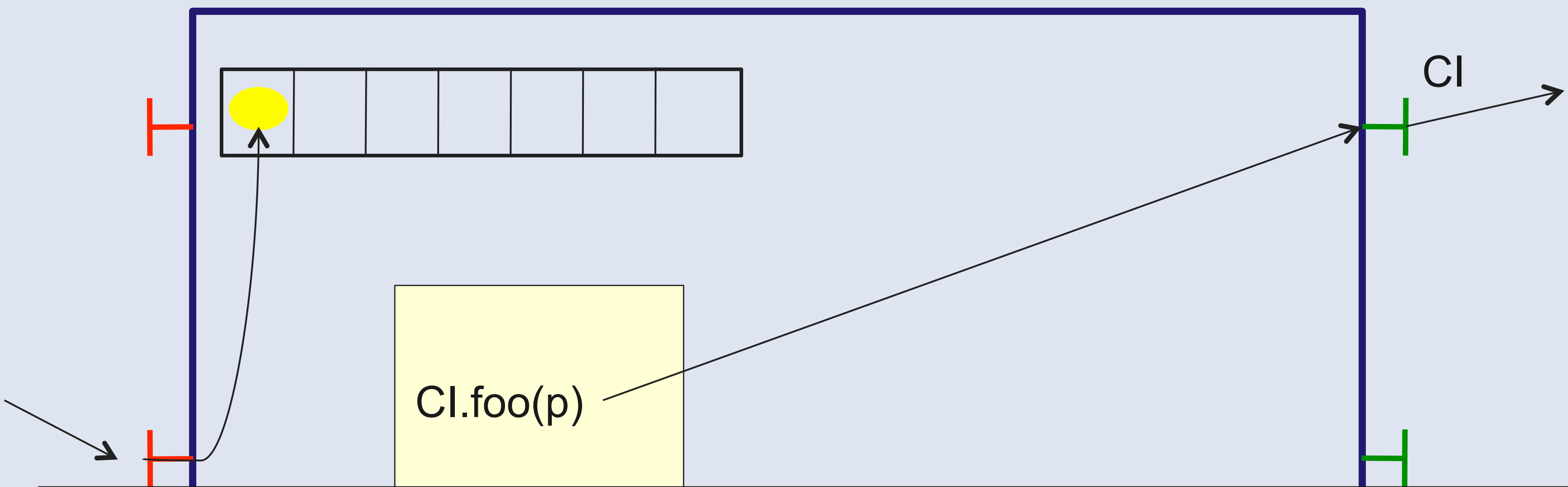
But what is a Good size for a (primitive) Component?

Not a strict requirement, but somehow imposed by the model design

- According to CCA or SCA, a *service* (a component contains a provided business function)
- According to Fractal, a *few objects*
- According to GCM, a *process*

> ➢ *In GCM/ProActive,*
>
> *1 Component (data/code unit)*
> *= 1 Active object (1 thread = unit of concurrency)*
> *= 1 Location (unit of distribution)*

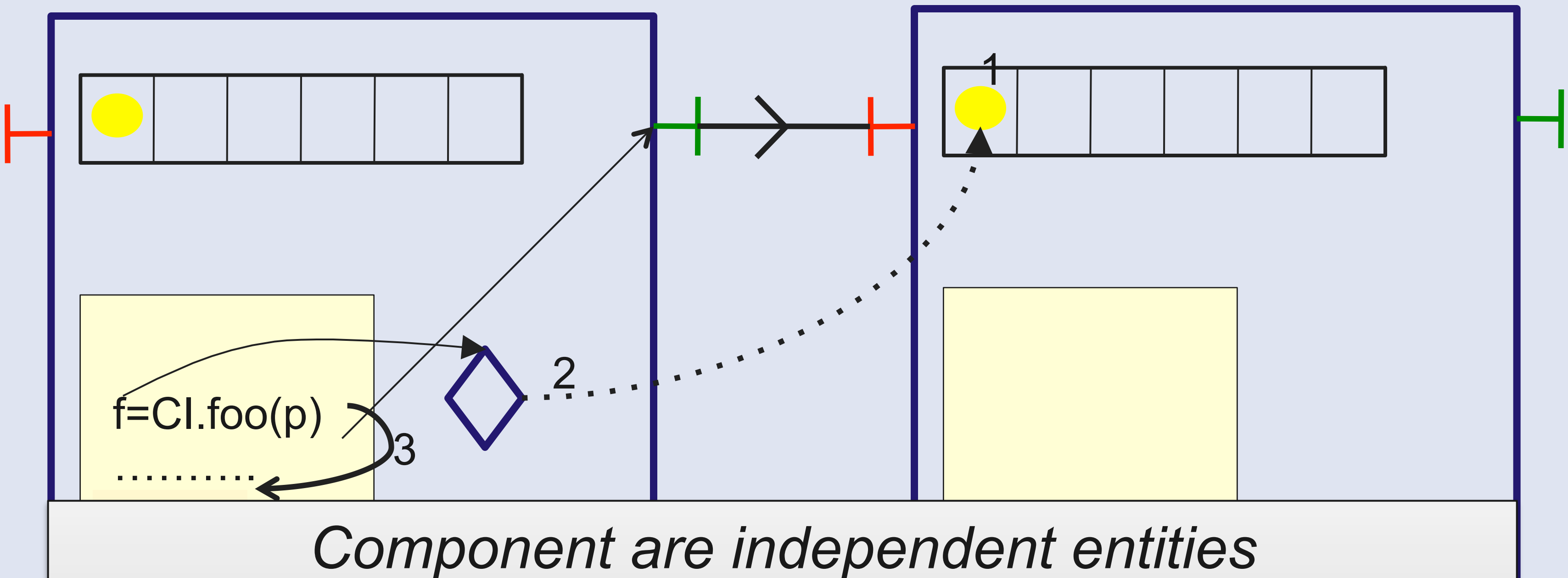
A Primitive GCM Component



In ProActive/GCM a primitive component is an active object

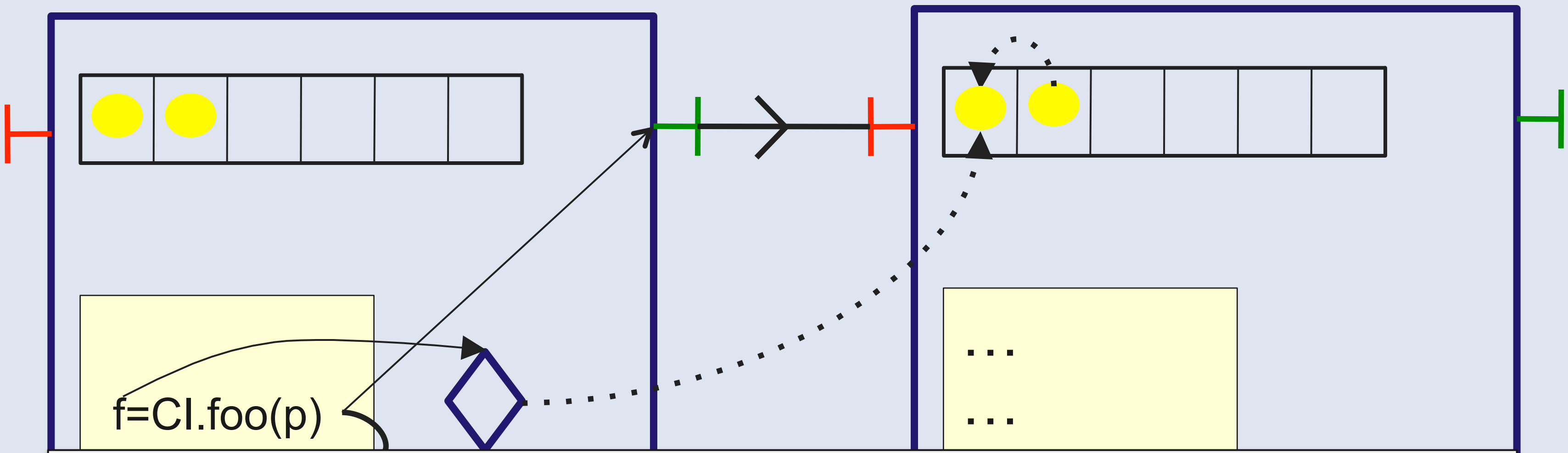
- *Primitive components communicating by asynchronous requests on interfaces*
- *Components abstract away distribution and concurrency*

Futures for Components



Component are independent entities
(threads are isolated in a component)
+
Asynchronous requests with results
⬇
Futures are necessary

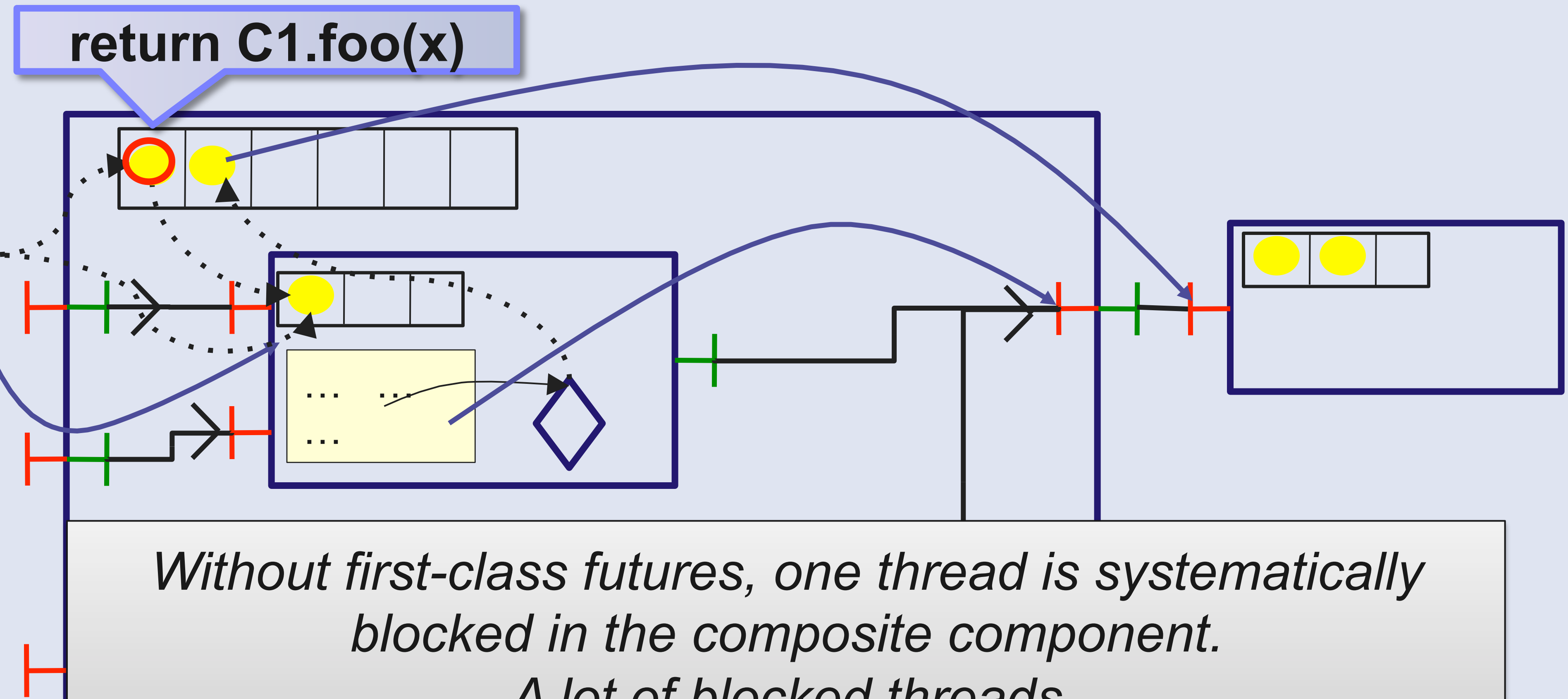
First-class Futures



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

In ProActive and ASP, futures are created and accessed implicitly (no explicit type “future”)
IN contrast with Creol, Jcobox, ...

First-class Futures and Hierarchy



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

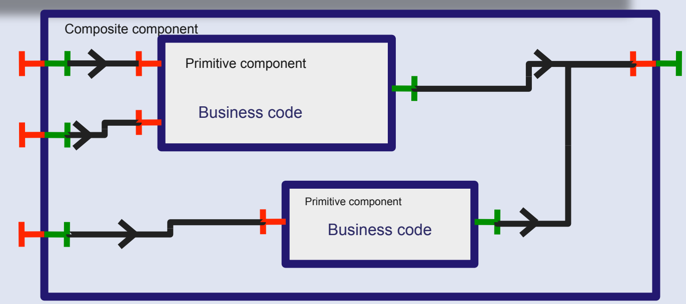
A lot of blocked threads

In GCM/ProActive ➔ systematic deadlock

Back to Formal Methods: a Framework for Reasoning on Components

- Formalise GCM in a theorem prover (Isabelle/HOL) Component hierarchical Structure

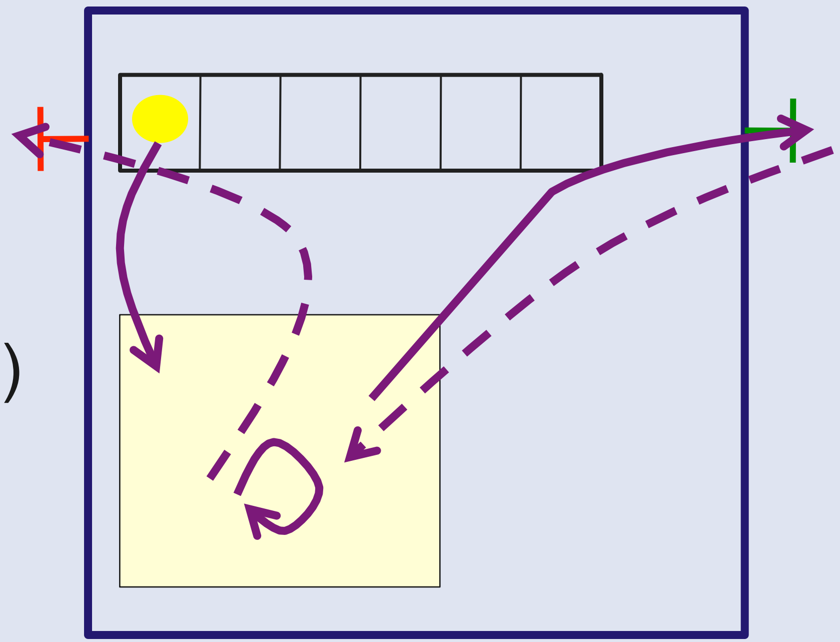
```
datatype Component = Primitive Name Interfaces PrimState
    | Composite Name Interfaces (Component list) (Binding set) CompState
```



- Bindings, etc…
- Design Choices
  - Suitable abstraction level
  - Suitable representation (List / Finite Set, etc …)
- Basic lemmas on component structure

A semantics of 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

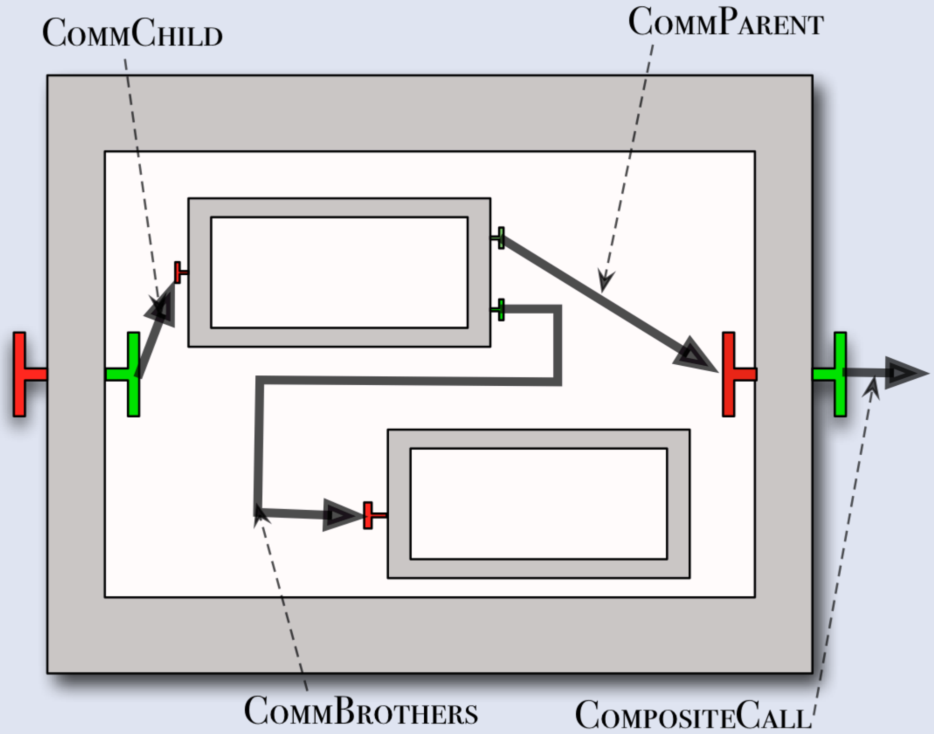


Components can have any behaviour

BUT some rules define a correct behaviour,

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

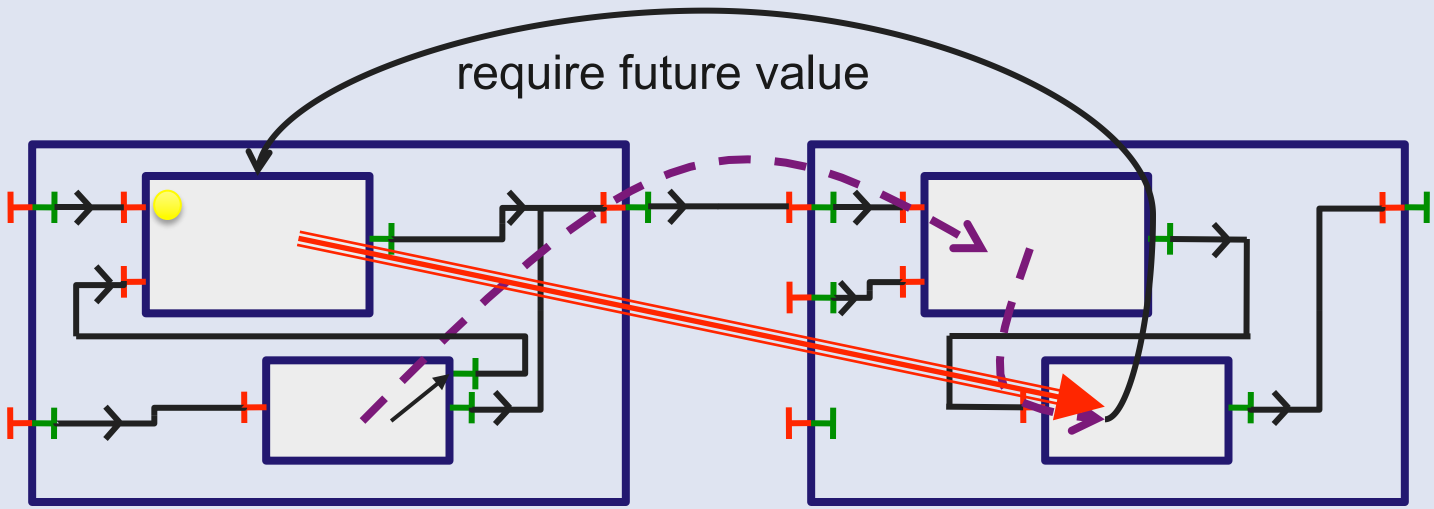
Communication inside Composites



- Composites only delegate calls between components
- Use the bindings to know where to transmit requests
- Component system behaviour is expressed as a small step semantics, and specified on paper and in Isabelle/HOL

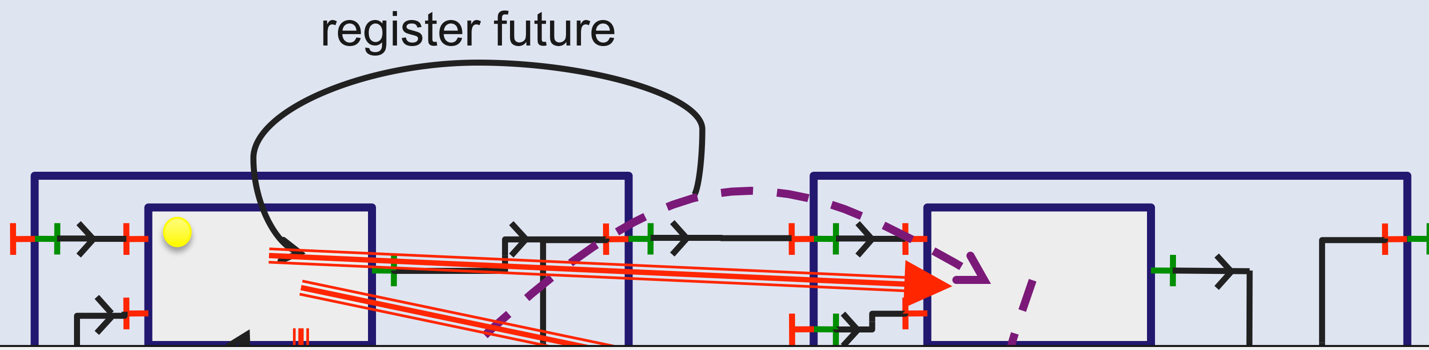
Future Update Strategies (Muhammad Khan)

- 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:



Eager home-based future update

- avoids to store future values indefinitely
- Relies on future registration



Results sent as soon as available

Every component with future reference is registered

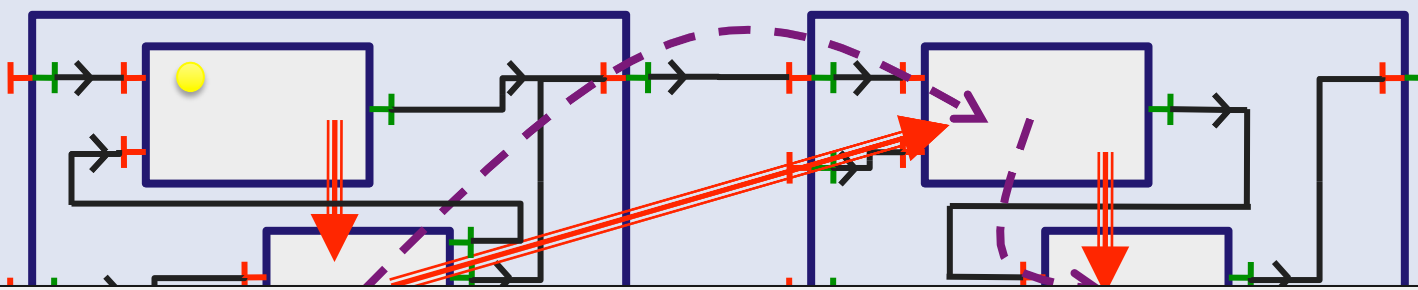
Un-necessary transfers

Garbage collection of computed results possible

Formalised in Isabelle

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

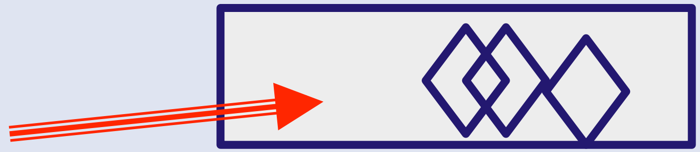
Future updates form a chain (intermediate components

Easy to garbage collect computed results

Properties on Future updates

- Future updates remove all references to a given future

```
lemma UpdatedFutureDisappear:
"⟦ S ⊣f,v,N⟼F S2, RL; CorrectComponent S; (S2^^N) = Some C ; f ∉ set (snd v)⟧
⟹ f ∉ LocalRefFutSet C)"
```

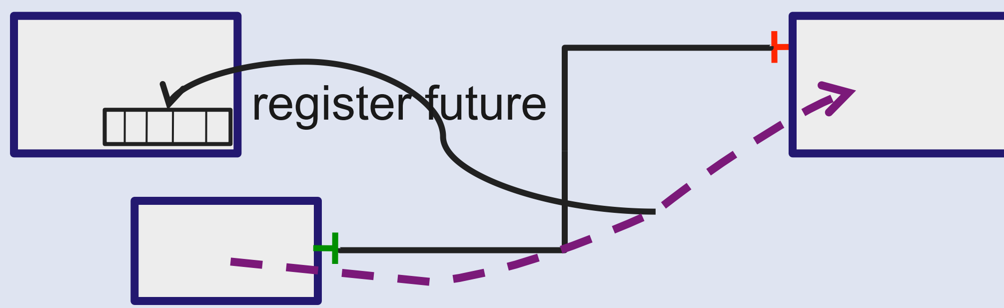


Garbage collectable

- All Future references are registered during reduction

```
theorem FuturesRegistered:
"⟦ ⊢ C1 ⇝ C2; CorrectComponent C1; GlobalRegisteredFuturesComp C1⟧
⟹ GlobalRegisteredFuturesComp C2"
```

Complete registration



A “refined” GCM model in Isabelle/HOL

- 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
- More abstract than ProActive/GCM
  - can be multithreaded
  - no active object, not particularly object-oriented

A guide for implementing and proving properties of component middlewares

Agenda

I. Introduction: Formal Methods

II. A Distributed Component Model

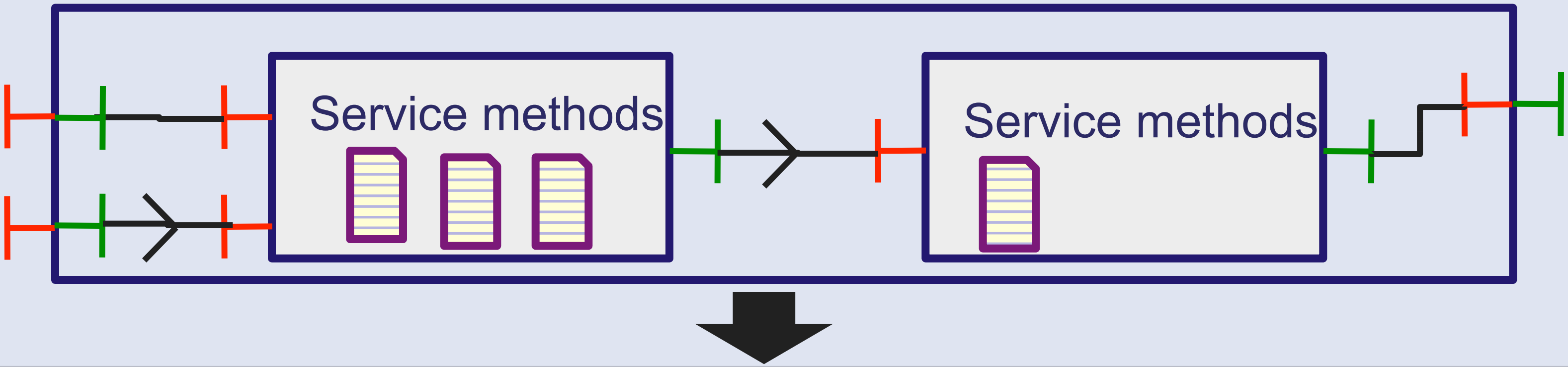
III. Behavioural Specification

IV. Dynamicity

V. A Few Hot Topics

How to ensure the correct behaviour of a given program?

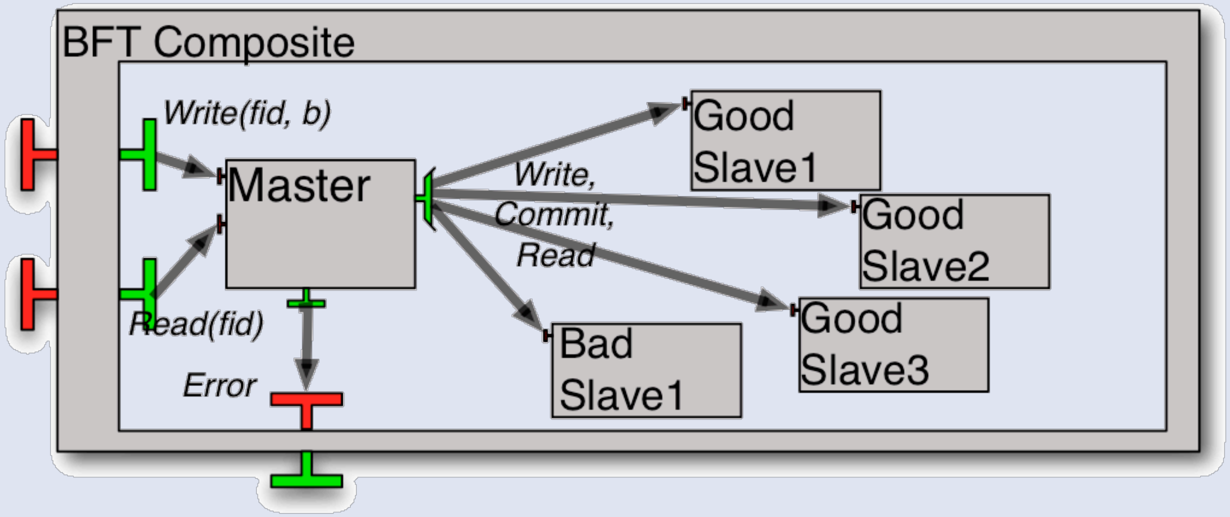
- Theorem proving too complicated for the ProActive programmer
- Our approach: behavioural specification



- ➢ *Trust the implementation step*
- ➢ *Or static analysis*
- ➢ ***Generate correct (skeletons of) components***
  (+static and/or runtime checks)

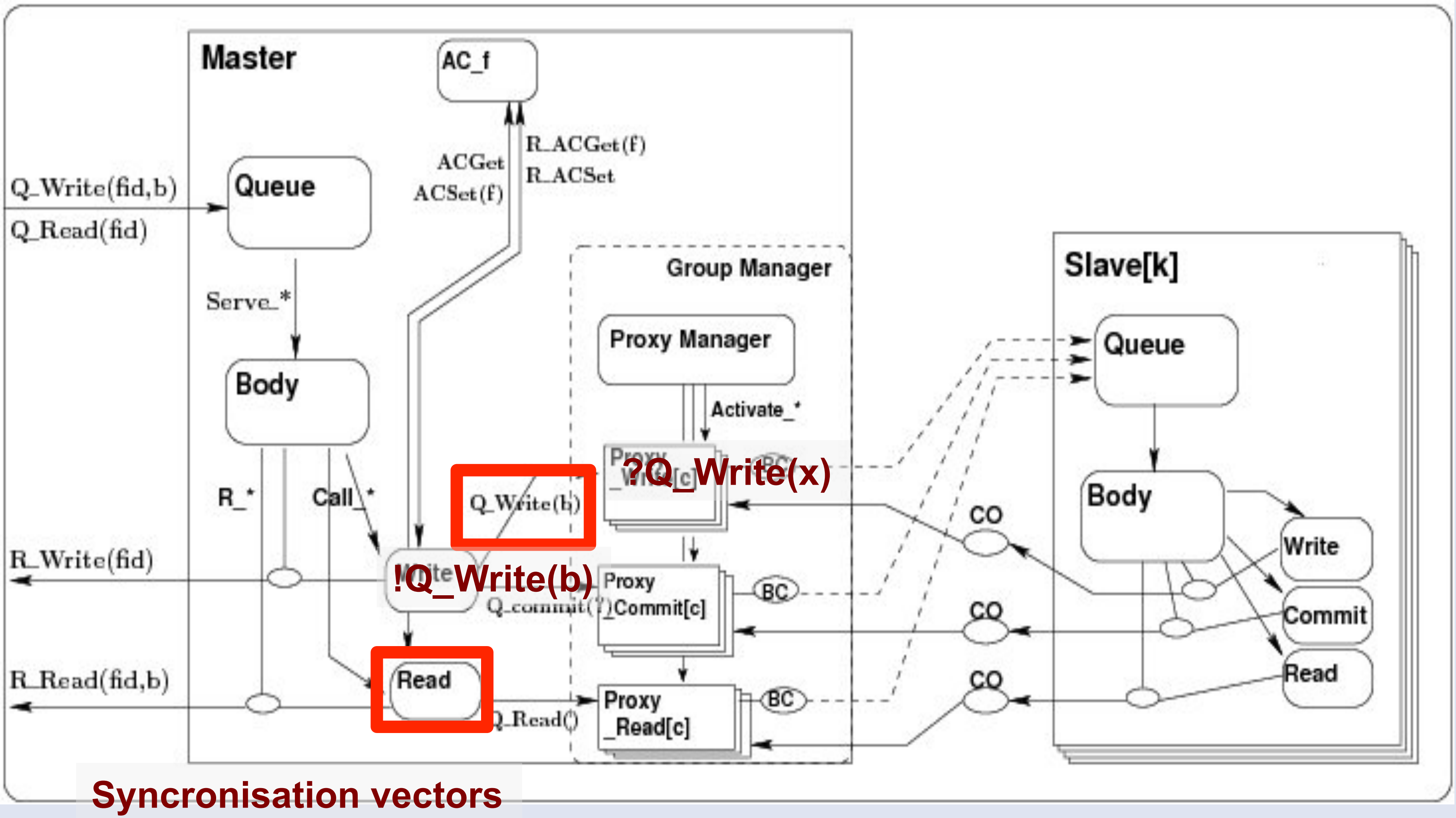
Use-case: Fault-tolerant storage

- 1 composite component with 2 external services Read/Write.
- The service requests are delegated to the Master.



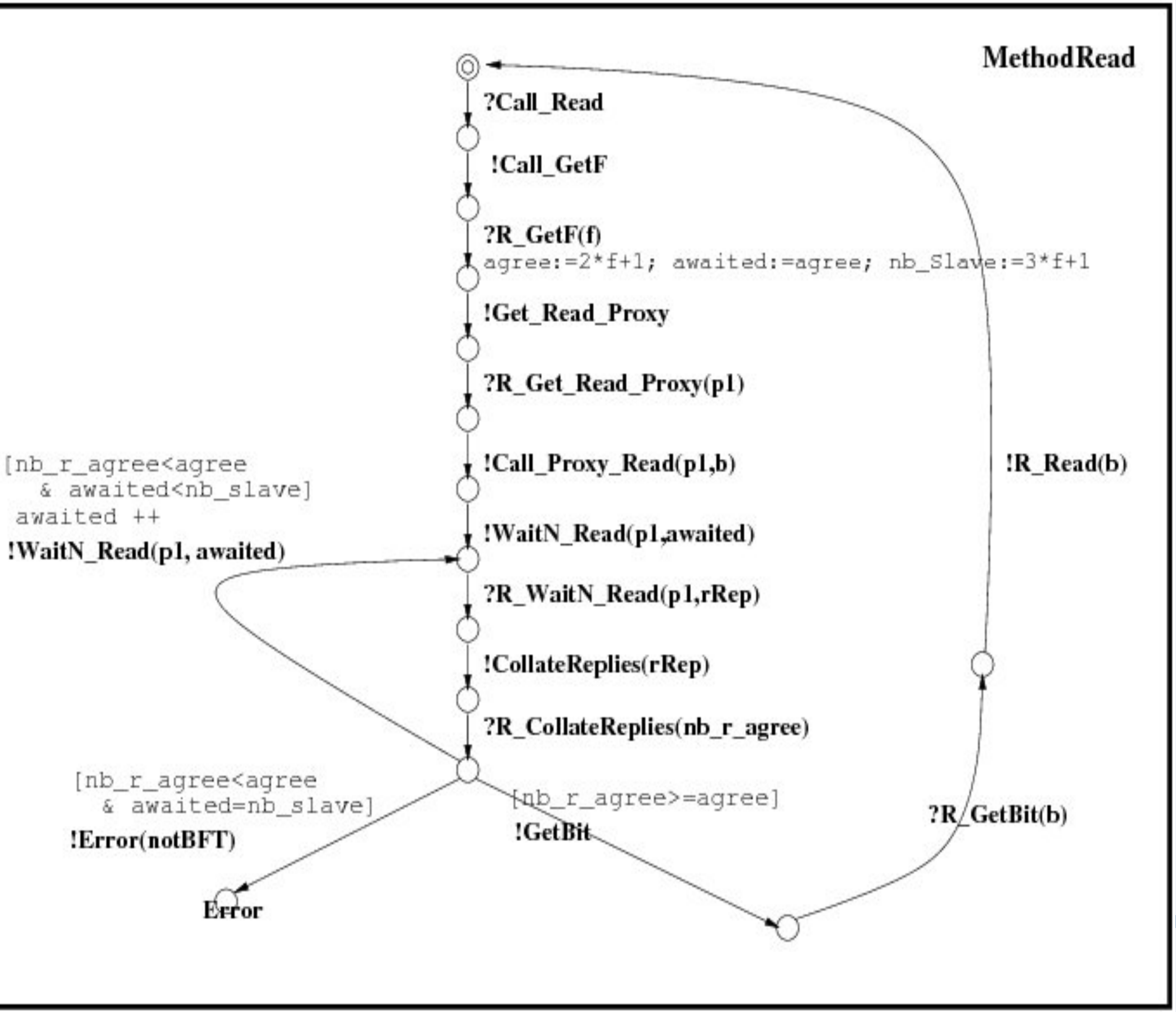
- 1 multicast interface sending write/read/commit requests to all slaves.
- the slaves reply asynchronously, the master only needs enough coherent answers to terminate

Full picture: a pNet



$$\langle -, -, -, -, -, (k \mapsto !Q\_m_n(f, arg), k' \mapsto ?Q\_m'_n(f, arg)) \rangle \rightarrow Q\_m_n(f, arg)$$

Basic pNets: parameterized LTS



Labelled transition systems, with:

- Value passing
- Local variables
- Guards….

Can be written as a UML diagram

Properties proved

- Reachability(*):

1- **The Read service can terminate**

∀ fid:nat among {0...2}. ∃ b:bool. <true* . {!R_Read !fid !b}> true

2- **Is the BFT hypothesis respected by the model ?**

< true* . 'Error (NotBFT)'> true

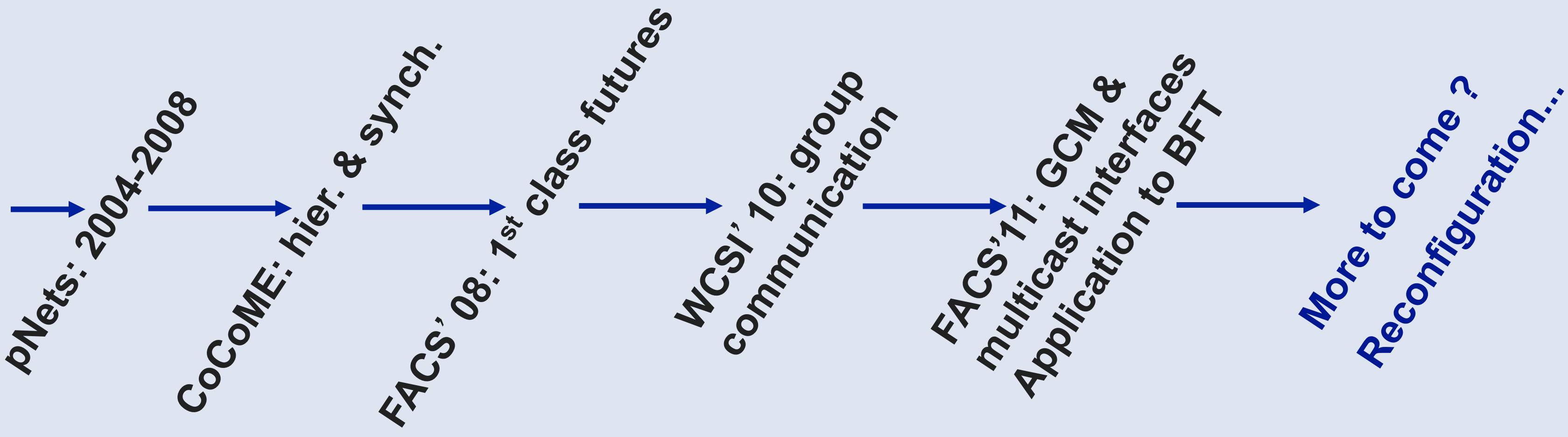
- **Termination**:

After receiving a Q_Write(f,x) request, it is (fairly) inevitable that the Write services terminates with a R_Write(f) answer, or an Error is raised.

Prove

- *generic properties like absence of deadlock*
- *or properties specific to the application logic*

Recent advances in behavioural specification for GCM



- Scaling-up : gained orders of magnitude by a combination of:
  - data abstraction,
  - compositional and contextual minimization,
  - distributed state-space generation.
- Specified formally the whole generation process (submitted)

Agenda

I. Introduction: Formal Methods

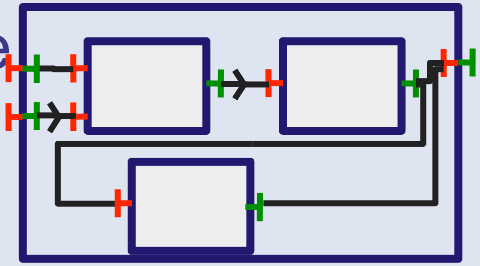
II. A Distributed Component Model

III. Behavioural Specification

IV. Dynamicity

V. A Few Hot Topics

Adaptation in the GCM



- Functional adaptation: adapt the architecture + behaviour of the application to new requirements/objectives
- Non-functional adaptation:
  adapt the architecture of the container+middleware to

Both functional and non-functional adaptation are expressed as reconfigurations

Language support for distributed reconfiguration: GCM-script

A platform for designing and running autonomic components

Formalising Reconfiguration (preliminary)

- In our Isabelle/HOL model component structure known at runtime
  i.e. semantic rules reason on the component structure
  - Two reconfiguration primitives formalised (remove and replace)
  - Illustrates the flexibility of the approach
  - Basic proofs
- In our pNets model
  - Old results: start/stop/bind for Fractal components
  - Concerning GCM: formalisation and experiments in progress

Agenda

I. Introduction: Formal Methods

II. A Distributed Component Model

III. Behavioural Specification

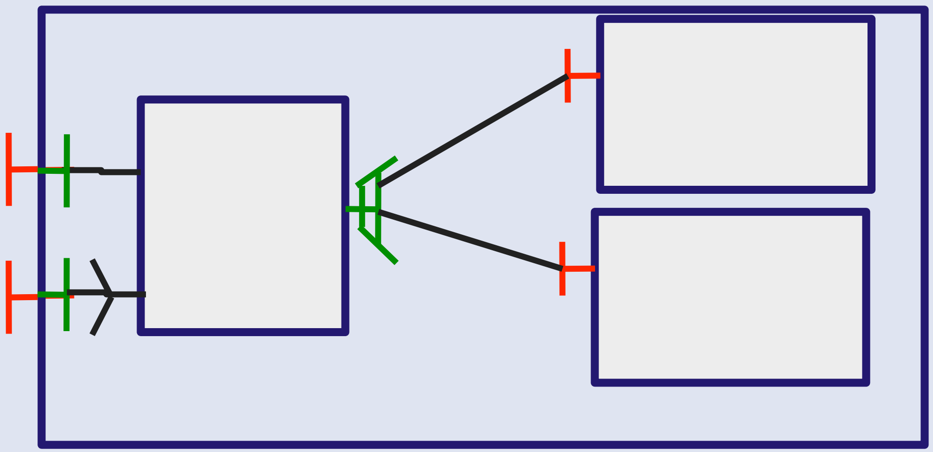
IV. Dynamicity

V. A Few Hot Topics

Correct component reconfiguration Towards safety and autonomicity

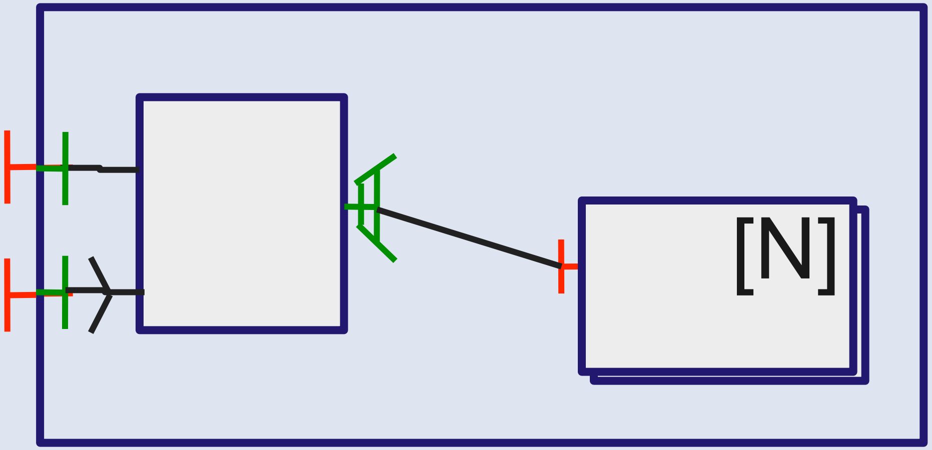
- Verify reconfiguration procedures
- Safe adaptation procedures as a solid ground for autonomic applications
- Some directions:
  - Parameterized topologies: ADL[N] ; behavioural specification (pNets) + reconfiguration primitives
  - Use of theorem proving + prove equivalence between the Isabelle GCM model and the behavioural specification

    ➔prove and use generic properties of reconfiguration procedures



Correct component reconfiguration
Towards safety and autonomicity

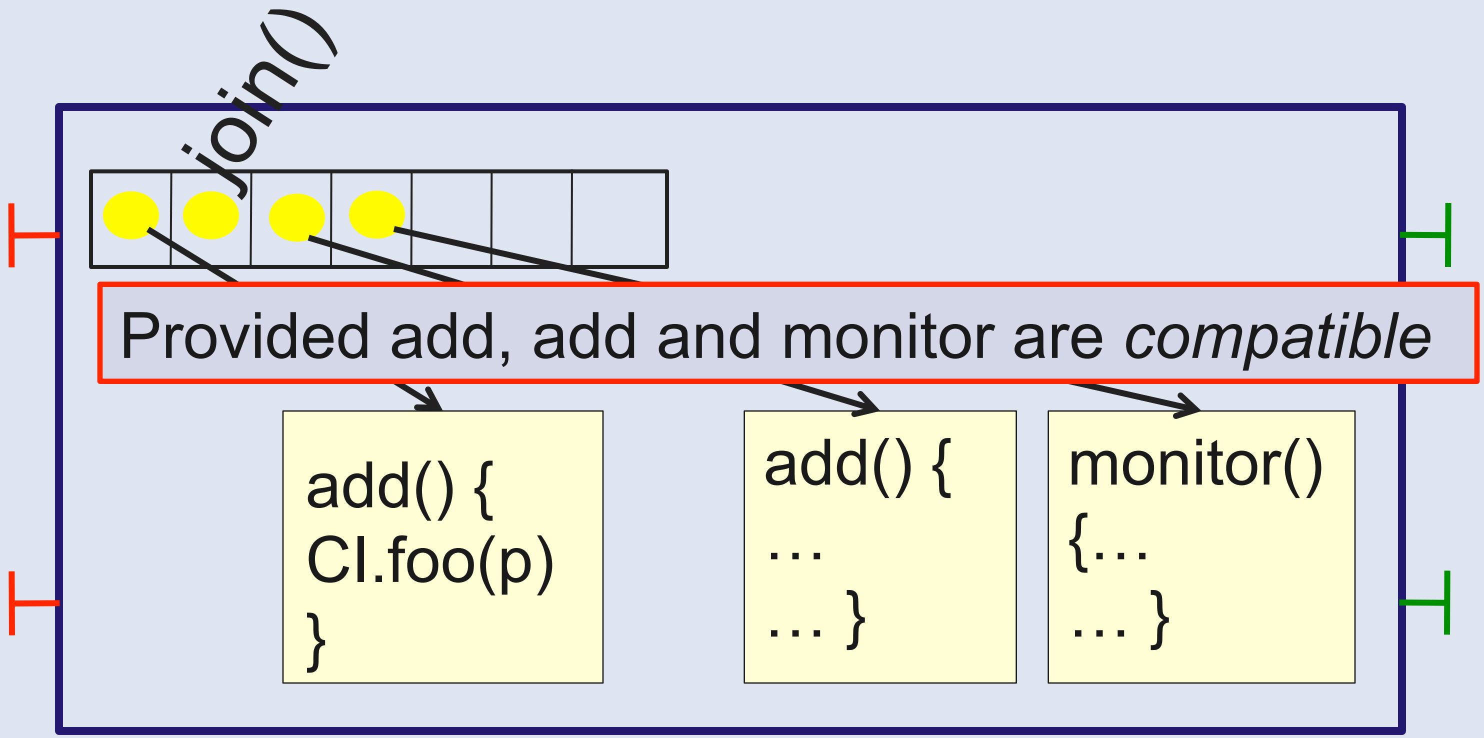
- Verify reconfiguration procedures
- Safe adaptation procedures as a solid ground for autonomic applications



- Some directions:
  - Parameterized topologies: ADL[N] ; behavioural specification (pNets) + reconfiguration primitives
  - Use of theorem proving + prove equivalence between the Isabelle GCM model and the behavioural specification

➔prove and use generic properties of reconfiguration procedures

Primitive Multi-Active GCM Component

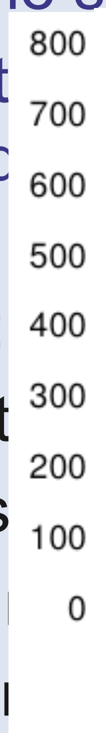


- Our proposal, a programming model that mixes local parallelism and distribution with high-level programming constructs

Henrio, Ludovic, Fabrice Huet, Zsolt István, and Zsolt Istv. 2013. "Multi-threaded Active Objects." in COORDINATION 2013.

Multi-active objects: Results / Status

- Implemented multi-active objects above ProActive
- Added dynamic compatibility rules
- Used on case studies (NAS, CAN)
- Specified the semantics of Multi-ASP
- Proved first … that two rec…
- Next steps: …
  - Publish t…
  - Formalis…
  - Use the …
  - Prove st…



LOCAL

$$\frac{(a,\sigma) \rightarrow_{\text{loc}} (a',\sigma')}{\alpha[F;\mathcal{R}_c[a];R,\sigma] \parallel Q \longrightarrow \alpha[F;\mathcal{R}_c[a'];R;\sigma'] \parallel Q}$$

ACTIVE

$$\frac{\gamma \text{ fresh activity name} \qquad f_\varnothing \text{ fresh future} \qquad \sigma_\gamma = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\varnothing,\iota_0)}{\alpha[F;\mathcal{R}_c[\mathit{Active}(\iota)];R;\sigma] \parallel Q \longrightarrow \alpha[F;\mathcal{R}_c[\gamma];R;\sigma] \parallel \gamma[\varnothing;[\iota_0.m_0([]) \mapsto f_\varnothing];\varnothing;\sigma_\gamma] \parallel Q}$$

REQUEST

$$\frac{\sigma_\alpha(\iota)=\beta \qquad \iota'' \notin dom(\sigma_\beta) \qquad f \text{ fresh future} \qquad \sigma'_\beta = \mathit{Copy\&Merge}(\sigma_\alpha,\iota'\,;\,\sigma_\beta,\iota'')}{\alpha[F;\mathcal{R}_c[\iota.m_j(\iota')];R;\sigma_\alpha] \parallel \beta[F';C';R';\sigma_\beta] \parallel Q \longrightarrow \alpha[F;\mathcal{R}_c[f];R;\sigma_\alpha] \parallel \beta[F';C';R'::[m_j;\iota'';f];\sigma'_\beta] \parallel Q}$$

ENDSERVICE

$$\frac{\iota' \notin dom(\sigma) \qquad \sigma' = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\sigma,\iota')}{\alpha[F;\iota \mapsto f::[a_i \mapsto f_i]^{i\in 1..n} \parallel C;R;\sigma] \parallel Q \longrightarrow \alpha[F::f \mapsto \iota';[a_i \mapsto f_i]^{i\in 1..n} \parallel C;R;\sigma'] \parallel Q}$$

REPLY

$$\frac{\sigma_\alpha(\iota)=f \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\beta,\iota_f\,;\,\sigma_\alpha,\iota) \qquad (f \mapsto \iota_f) \in F'}{\alpha[F;C;R;\sigma_\alpha] \parallel \beta[F';C';R';\sigma_\beta] \parallel Q \longrightarrow \alpha[F;C;R;\sigma'_\alpha] \parallel \beta[F';C';R';\sigma_\beta] \parallel Q}$$

SERVE

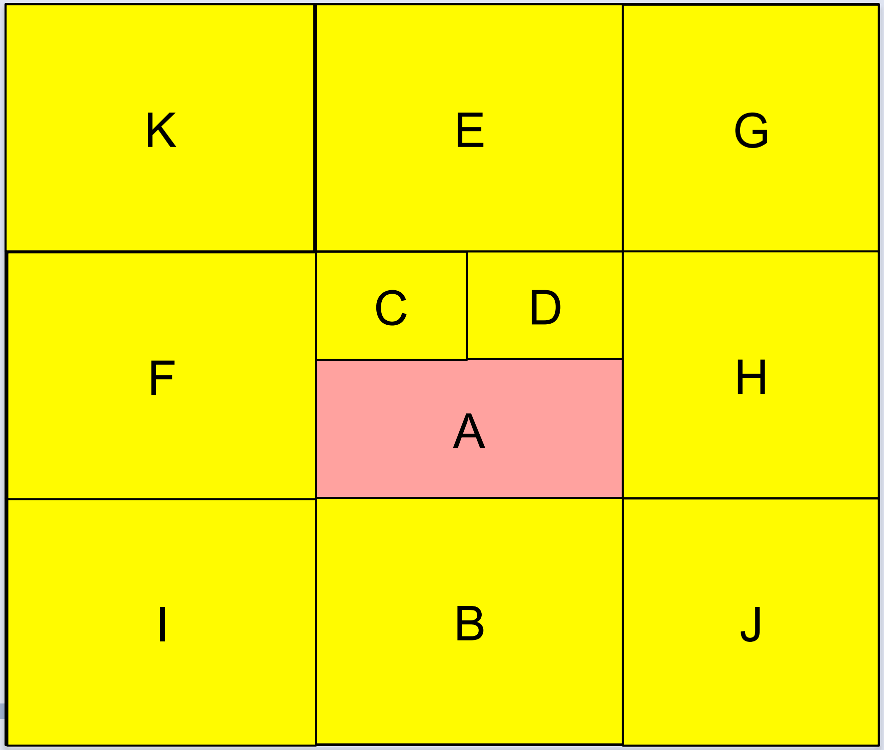
$$\frac{C = [\mathcal{R}[\mathit{Serve}(M)] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \parallel C' \qquad \mathit{SeqSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')}{\alpha[F;C;R;\sigma] \parallel Q \longrightarrow \alpha[F;[\iota_0.m(\iota) \mapsto f]::[\mathcal{R}[[]] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \parallel C';R';\sigma] \parallel Q}$$

PARSERVE

$$\frac{C = [\mathcal{R}[\mathit{Serve}(M)] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \parallel C' \qquad \mathit{ParSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')}{\alpha[F;C;R;\sigma] \parallel Q \longrightarrow \alpha[F;[\iota_0.m(\iota) \mapsto f] \parallel \mathcal{R}[[]] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \parallel C';R';\sigma] \parallel Q}$$

CAN dissemination algorithm
Distributed systems + theorem proving

- CANs are P2P networks organised in a cartesian space of N-dimensions (used for RDF data-storage)
- Objective: disseminate efficiently information (no duplicate)
- Designed an efficient algorithm (tested)
- Proved the existence of an efficient broadcast in Isabelle



Next steps:

- Prove the designed algorithm is efficient
- Conduct large-scale experiments (ONGOING
- Study churns

Bongiovanni, Francesco, and Ludovic Henrio.
“A Mechanized Model for CAN Protocols.” in FASE 2013.

THAK YOU for your attention

Questions ???

http://www-sop.inria.fr/oasis/Ludovic.Henrio/

OASIS Team
INRIA – UNS – I3S – CNRS
Sophia Antipolis