Interact :
a general model of contract
to guarantee the validity of
components and services assemblies

Alain Ozanne
a.ozanne@free.fr
http://alain.ozanne.free.fr

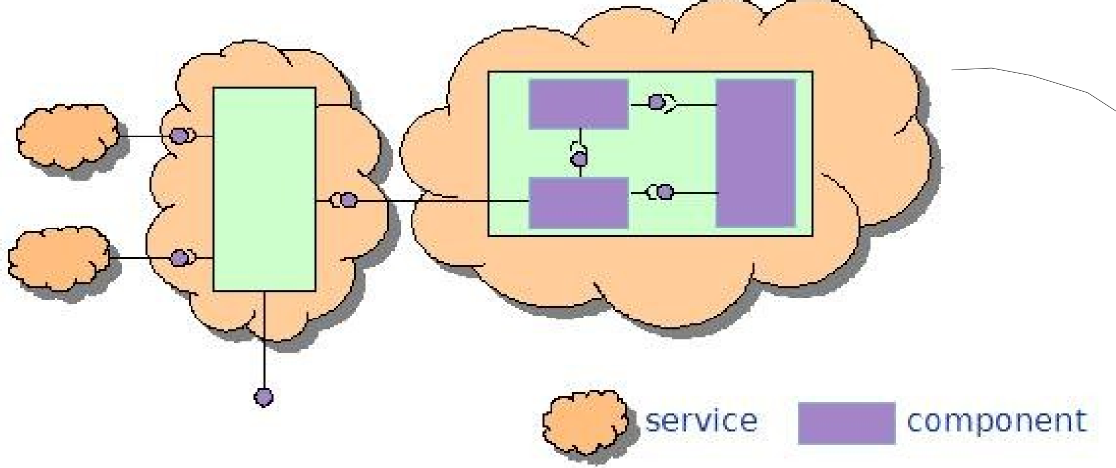
An implementation of Interact is available by France Telecom R&D (send me a mail). This implementation consists in :

- the kernel
- an architecture adapter for the Fractal Component Framework,
- a formalism adapter for the CCLJ formalism,

do not hesitate to mail me any question,

Context : Composite Applications

Components / services : J2ee, OSGI, SCA, Fractal ...



- New requirements : modularity, adaptability, flexibility ...
- Dependability : assemblies validity,
- Robustness : diagnosis, repair,

<<

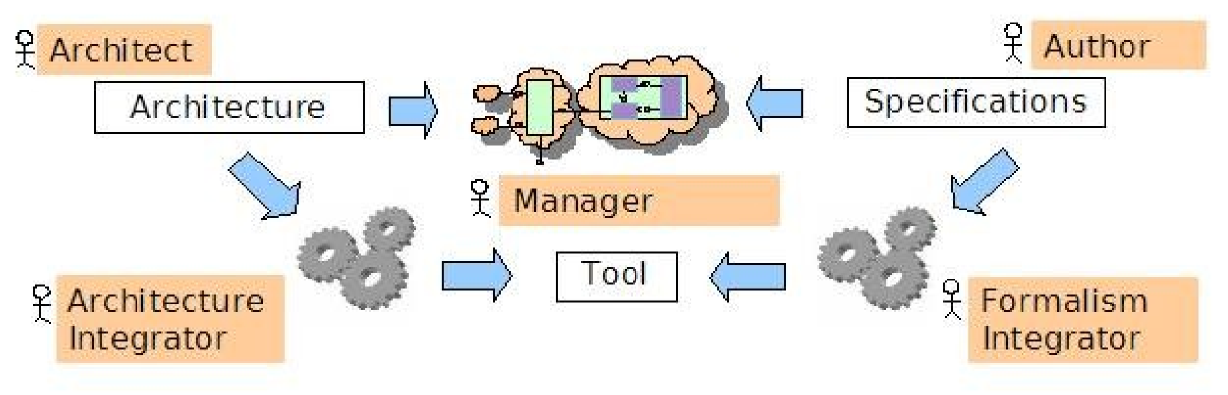
The interact contract model is dedicated to the composite applications. That is : applications made of compositions of objects, components or services. I've made a little sketch of one of these application made of components and services.

Indeed components and services frameworks appeared to address the new requirements of applications such as modularity, adaptability, flexibility etc. But they did not so well consider the requirements of dependability, that is the validity of assemblies, and the one of robustness, that is the ability to determine a diagnosis in case of failure of the assembly ... So Interact is meant to propose a solution for these two requirements.

>>

The stakes

- To Guarantee the validity of assemblies :
  - Validate the collaboration of elements considering :
    - Various properties [Beugnard et al.],
    - Unanticipated configurations.
  - Propose a tool that integrates and organises specifications checking in an architecture.
- To Take in account various actors :



<<

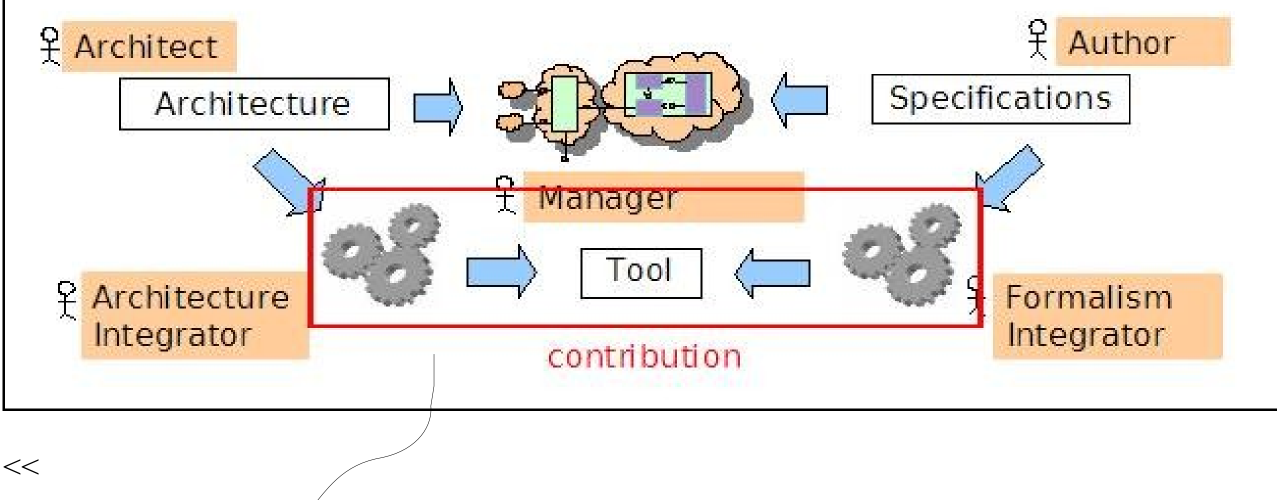
There were two stakes for Interact :

- the first one was to guarantee the validity of assemblies by validating the collaboration of their elements. This was to be done while considering various properties of these elements. Indeed Antoine Beugnard and its colleague have clearly shown on an article that various kinds of properties were to be considered to analyse a component. An other point to be considered was that more and more assemblies are unanticipated and therefore may not be pre-calculated.

- the second stake was to consider the various actors around a software system : of course there are the architect, providing the system configuration, and the author of the specifications, describing this system elements. The manager is then in charge of running the system. I wanted that my tool enables him to make sure that the system was all-right, and to repair it if necessary.

As the manager may not be a specialist of the architecture implementation, the tool has to provide him generic informations about the system.

On the other hand, even if the tool has to provide the manager with generic informations, it must also interpret the concrete architecture and specifications. To do so it uses mechanisms provided by the « architecture integrator » and « formalism integrator », in order to interpret into generic terms a concrete implementation of architecture, and specific formalisms of specification.
>>



<<

I put a red rectangle around the “Interact” system. It consists of a generic « Tool » and the description of the mechanisms it requires to apply to concrete configuration and specifications.

>>

Contracts

- A part of components definition [Szyperski]

  A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only

- Used to describe, validate, ... :
  - Describe an element / environment,
  - Validate each element, express the agreement, ...
- In every day life :
  - Organize properties to guarantee the collaboration :
    - Independently from the properties nature,
    - Independently of the organisation.
  - Make the responsibilities explicit,

<<

Szyperski expressed that contracts were parts of the definition of components.

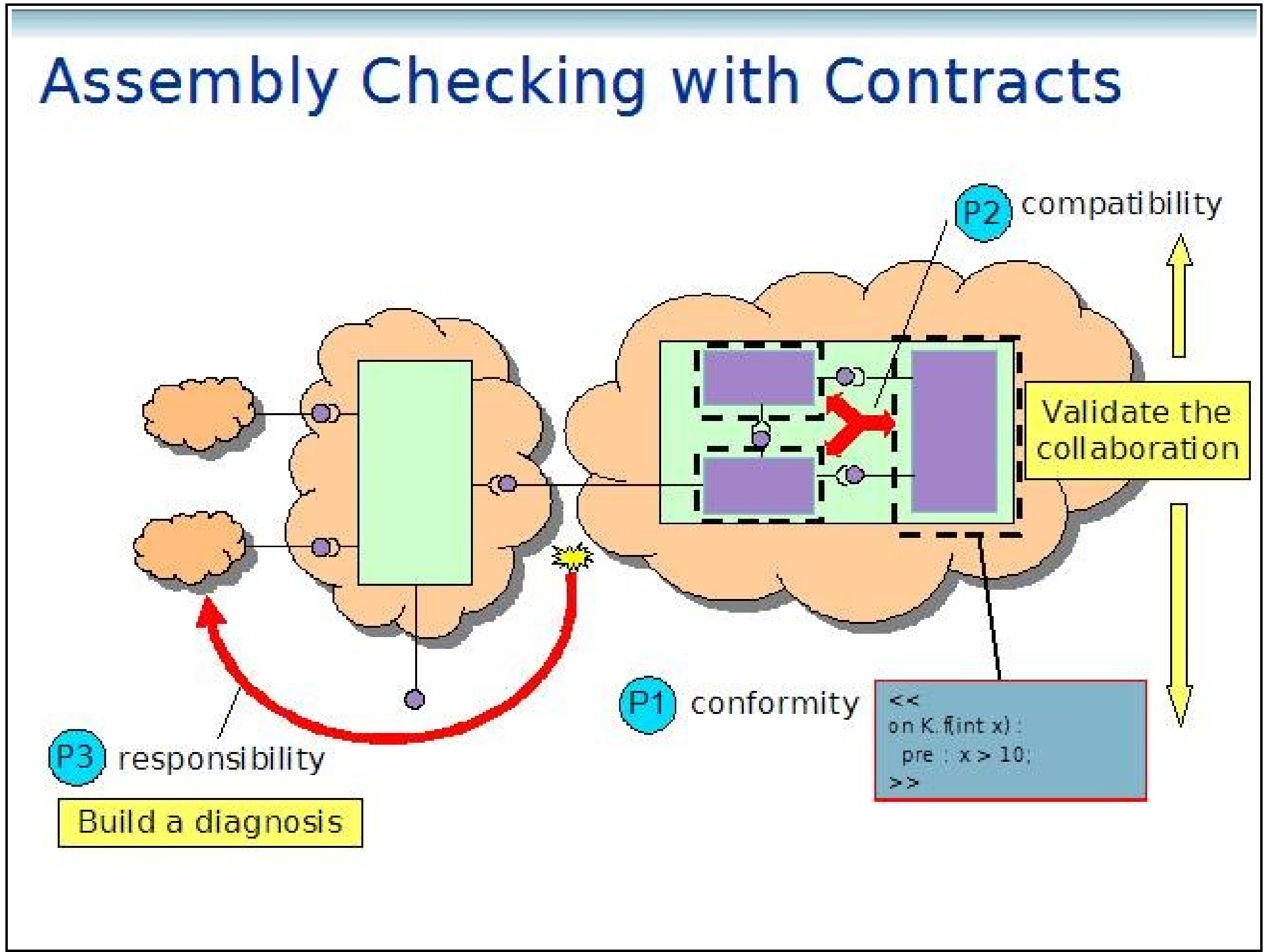
More broadly they are used in software engineering to describe a software element with respect to its environment, or to check that an element implementation fits to its definition, or express an agreement between entities etc

But what interested me is the every day life contract definition : indeed : the every day life contract is a tool that organizes the properties of its participants to guarantee the validity of their collaboration. It does this independently of the nature of the properties, and independently of the configuration of the participants.

Moreover this every day life contract makes the responsibilities of its participants explicit, then enable them to correct a dysfunction in their collaboration.

That's why I choose the metaphor of the every day life contract to describe my tool for the guarantee of assembly based on the validity of their collaboration.

>>



<<

Then what does it mean for a contract to guarantee the validity of an assembly ?

That means for me that this contract should reify and therefore enable to reason about three fundamental properties :

1. if the black dashed rectangle are the specifications of the components, then the contract should reify the fact that each participant should conform to its specification : that's the P1 property

2. the contract should also reify the fact these specifications must be compatible, that's common sense but unavoidable for the assembly to be valid (property P2)

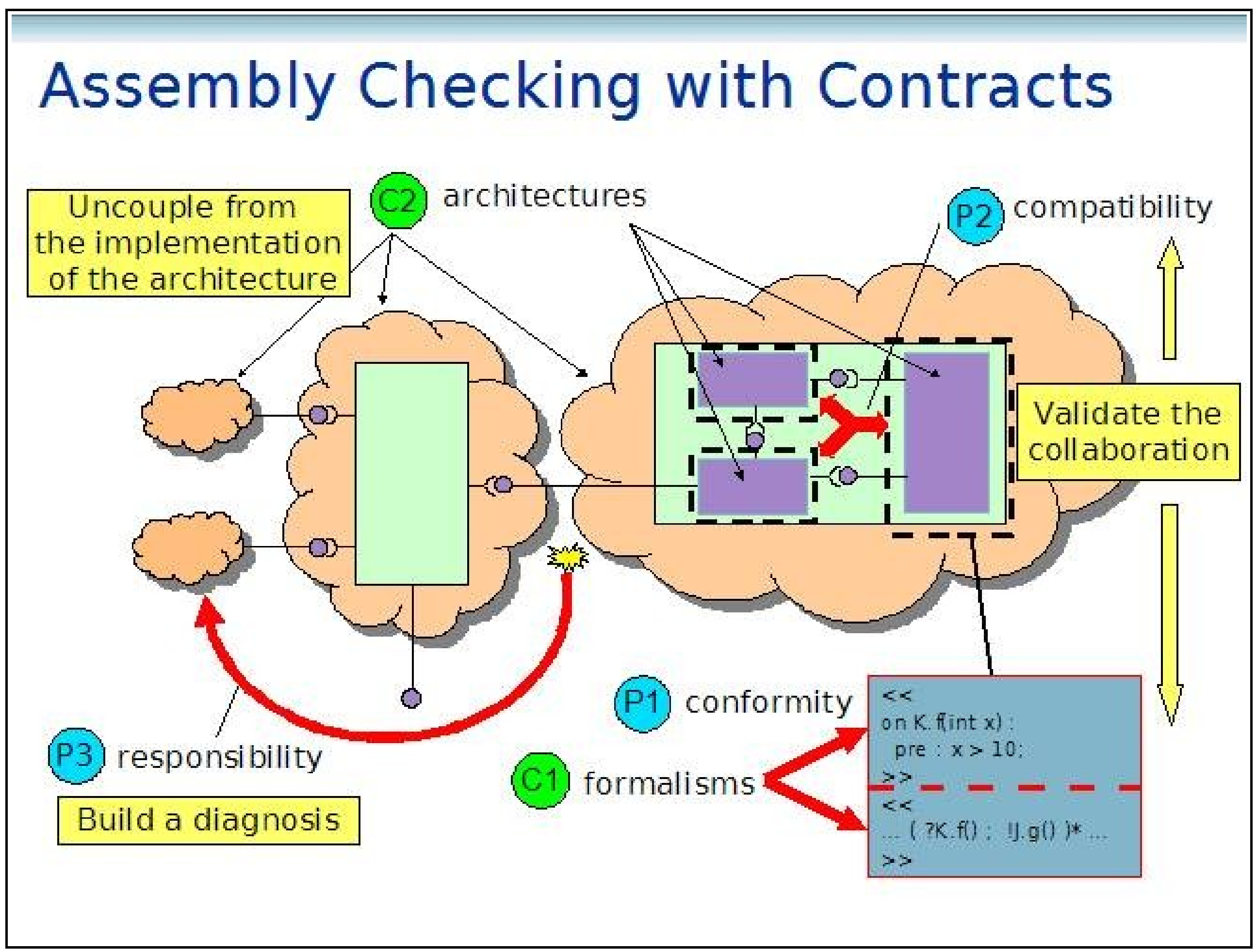
3. finally, the contract should reify the responsibilities of each of its participants, to help building the diagnosis (property P3),

a contract that reifies this 3 properties :

enables to reason on the validity of an assembly : an assembly being valid as long as P1 : conformity and P2 : compatibility are satisfied,

and enables to diagnose a failure through P3 : responsibility

>>



<<

I want Interact to deal and organize different types of properties, in order to do so :

the contract has to accept various formalisms : so it has to reify P1, P2, P3 while having the characteristics “C1 : accepting different formalisms”. Moreover it will allow the contract to handle specifications in a generic way (what I wanted for my “system manager” of the page 4)

I want Interact to handle architecture in a generic way, so the contract has to be independent of the architecture implementation :

the contract should have a second characteristic “C2 : being uncoupled from the architecture implementation”.

This two characteristics of the contract will enable the “system manager” to understand the validity state of an assembly without being a specialist of the architecture implementation, and of the formalisms of the specifications.

>>

Motivation

- To overtake pre and post conditions,
- To Reify in one platform notions recurrent in others,
- Through a model and a framework:
  - Reifying P1 to P3, with C1 and C2,
  - Taking in account the potential weakenings of some formalisms (DbC ...).

<<

Clearly the pre and post conditions are not enough to deal with the collaboration which is the base of new systems : they most often only deal with conformity (with reification),

I wanted also to gather in one platform, some notions as conformity, compatibility, responsibility that appeared separated in some others,

But I also wanted to take into account that some formalisms implied some weakenings in the assessment of the validity of the assemblies : for example : the assertion of the design by contract don't allow to make proof to validate conformity or compatibility, their assessment is based on the observation of the system exection. So Interact has to accept the checking of the properties P1, P2, P3 on the base of observations of the contracted system, not only on proof and static checking.

>>

Plan

- Introduction
- Existing works
  - Architectures and formalisms
  - Contracts
- Contribution
- Implementation
- Validation
- Conclusion

Architectures and formalisms

- Objects, components and services (J2EE, Fractal, WS ...) :
  - Work based on collaboration,
  - Collaboration based on exchanges : + and + explicit, guided by architecture,

- Various specifications formalisms (assertions, protocols, ...) can be used :
  - Express conformity et compatibility,
  - Have checking tools,

P1 P2 C1

- The assume-guarantee approach formalize commitments :
  - Formalize and provide a semantic of commitment and responsibility,
  - Applied to composition:
    - Mainly formal,
    - The architectural configuration is not explicitely considered,

<<

In order to see how Interact may apply indifferently to various architectures, I looked at objects, components and services architectures. What appeared clearly is that their working is based on the collaboration of their elements. Moreover this collaboration is more and more clearly guided by the architecture : the collaboration is something like an image of the architecture.

This confirmed that the approach that consists in checking the collaboration to validate the architecture was valid and could be applied to various kinds of systems, objects components and services. So the characteristic C2 (uncoupling of architecture) could be met.

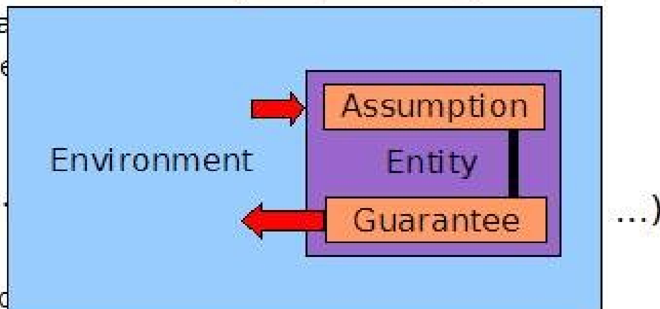
In order to see if various kinds of specification properties could be handled by a contract which was based on the reification of conformity and compatibility, I watched several formalisms, functionnal and non-functionnal.

It appeared then that many formalisms could express conformity and compatibility. Moreover they come with tools to concretely check the satisfaction of these properties. So an assessable contract, based on conformity and compatibility, could be applied to various kinds of properties. The characteristic C1 could be met.

>>

Architectures and formalisms

- Objects, components and services (J2EE, Fractal, WS ...) :
  - Work based on collabora
  - Collaboration based on e architecture,



- Various specifications ...)
  can be used :
  - Express conformity et co
  - Have checking tools,

P1 P2 C1

- The assume-guarantee approach formalize commitments :
  - Formalize and provide a semantic of commitment and responsibility,
  - Applied to composition:
    - Mainly formal,
    - The architectural configuration is not explicitely considered,

<<

Then I considered the problem of expressing and reifing the responsibilities. A very classical approach of specification deals with the responsibilities, that is the assume-guarantee approach. I briefly remind the assume-guarantee approach :

if I consider an entity and its environment, and for this entity a couple of specifications. I will name "assumption" the specification that constrains what the environment provides to the entity, I will name "guarantee" the specification that constrains what the entity provides to its environment. In the assume-guarantee approach, the entity is responsible for the satisfaction of the guarantee as long as the environment satisfies to the assumption.

>>

Architectures and formalisms

- Objects, components and services (J2EE, Fractal, WS ...) :
  - Work based on collabora
  - Collaboration based on e architecture,

C2

Environment

Assumption

Entity

Guarantee

- Various specifications ...) can be used :
  - Express conformity et co
  - Have checking tools,

P1 P2 C1

- The assume-guarantee approach formalize commitments :
  - Formalize and provide a semantic of commitment and responsibility,
  - Applied to composition:
    - Mainly formal,
    - The architectural configuration is not explicitely considered,

P3

<<

So I saw that if the contract could interpret the specifications in assume-guarantee terms, then it would make the responsibilities explicit, as the failure of a specification has in this approach an explicit, clearly pointed, responsible.

Moreover the assume-guarantee approach has often been formalized, providing a semantic to the responsibility as show by several formal works and theorems. An other noticeable point is that the assume-guarantee approach has been often considered for the analysis of compositions of elements.

As a drawback, the results of this approach are most often purely formal, and if the fact that specified entities belong to a same composition is taken in account, their architectural configuration is not considered in a explicit way : the specified entities are considered as gathered but the organization of their relations is ignored.

Nevertheless this assume-guarantee approach appeared to me as the key to the modeling of the responsibilities.

>>

Examples of contract models :

Considering conformity (P1) and compatibility (P2) :

- **Objects** :
  - Conformity : OCL modeling [Richers], collaboration contract [Helm], coordination contract [Andrade], notion of responsability [Petre]...,
  - Compatibility/Responsibility : class/child class[Findler].
  - Java, Eiffel, ... : DbC integration,
- **Components** :
  - Conformity : QCCS [Sassen]...,
  - Compatibility : QoSCL [Jezequel], GCCL [Malenfant], R/A Contracts [Rausch]....
  - J2EE, .Net : DbC integration,
- **Services** :
  - Conformity : SLA model [Keller], WSLA [Ludwig],...
  - Compatibility : WS-Agreement [Andrieux], Composition [Milanovic]...
  - Webservices : SLA integration,

<<

There are a lot of contract models, for the objects, components, services ...

I looked at many of them to see if the properties I was looking for (conformity, compatibility, responsibility) were realized and gathered. I watched these models under the angle of conformity and compatibility :

- for the objects : many models reify the conformity, but far less deal with the compatibility, and for their concrete implementations I found mainly implementations of the design by contract. Responsibilities are almost never explicit.

- for the components : again many models deal with the conformity of the components to their specifications. More often than for the objects, the compatibility of components is considered but often the model of contract is dedicated to one formalism or kind of properties (behavior or qos...)... Responsibilities are sometimes explicit. Looking at the implementation, again the design by contract is most often encountered.

- for the services : there are many models considering the conformity, some other consider also the compatibility, but are often implemented for a service or workflow engine implementation. The responsibilities are often explicit.

>>

Analysis

- Analysis of the state of the art :
  - Implicit Responsibilities */yet/ the assume-guarantee approach makes them explicit,* ≠ P3
  - The expression of compatibility : P2
    - Often coupled with an implementation */yet/ architectures are all collaborative,* ≠ C2
    - Often coupled with a formalism */yet/ various formalism can express and are needed,* ≠ C1
    - Absence of explicit link with the architecture */yet/ architectures + and + explicit,*
  - The properties from P1 to C2 don't appear jointly.

<<

Considering the contract models I've looked, several strange points appeared :

- responsibilities are often implicit, yet the assume-guarantee, which is a very classical approach, makes them explicit,
- the expression/implementation of compatibility :
  - is often coupled with a type or kind of properties, yet it has been shown that to validate a system various kinds of properties should be considered (behavioral, qos, synchronization, ...)
  - is often coupled with the implementation of an architecture, yet all the architecture of objects, components, services rely on the same principle of collaboration,
  - finally : I notice the absence of an explicit link between the compatibility expression and the architectural configuration, yet the architectures are more and more explicit and driving the collaboration of elements,

Then it appears that the three properties I was looking for did not appear jointly, and independently of the formalisms and architecture implementation.

>>

Plan

- Introduction
- Existing works
- Contribution
  - Conception
  - Contract Model
- Implementation
- Validation
- Conclusion

Design principles

- The contract is an assessable first class object :
  - State of the contract = validity of the assembly,
  - Validity : based on mutual commitments of exchange,
  - Exchange : based on the sharing.

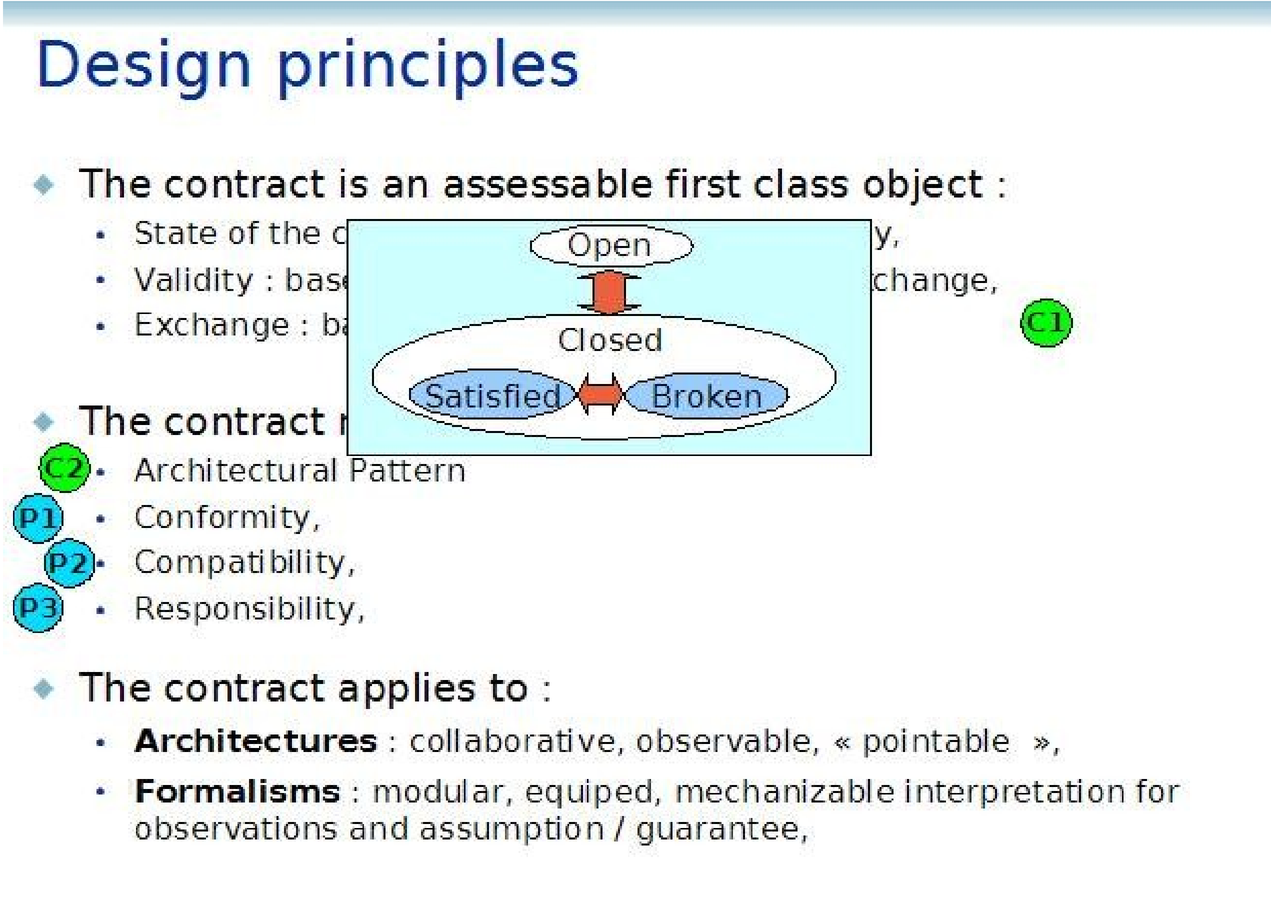
- The contract reifies, in a generic way :
  - C2 Architectural Pattern
  - P1 Conformity,
  - P2 Compatibility,
  - P3 Responsibility,
- The contract applies to :
  - **Architectures** : collaborative, observable, « pointable »,
  - **Formalisms** : modular, equiped, mechanizable interpretation for observations and assumption / guarantee,

<<

So I designed a contract model that follow some principles :

- first : the contract is first class entity, with an assessable state. This state represents the validity of the assembly on which is applied the contract.

>>



<<

The state of the contract can have the following values, as shown on the slide :

- the contract can be “open” : meaning that it can not be evaluated : when for example, participants are missing, or their specifications etc
- the contract can be “closed” : when it can be evaluated (all participants are present, with their specifications) : then the contract can be :
  - satisfied : the assembly is valid
  - broken : the assembly is not valid

>>

Design principles

- The contract is an assessable first class object :
  - State of the contract = validity of the assembly,
  - Validity : based on mutual commitments of exchange,
  - Exchange : based on the sharing.

- The contract reifies, in a generic way :
  - Architectural Pattern
  - Conformity,
  - Compatibility,
  - Responsibility,
- The contract applies to :
  - **Architectures** : collaborative, observable, « pointable »,
  - **Formalisms** : modular, equiped, mechanizable interpretation for observations and assumption / guarantee,

<<

for me, an assembly is valid when each of its participants provides what they need to the others and reciprocally,

so the validity of the assembly relies on the one of the exchanges between participants (as an exchange is valid when what is provided corresponds to what is required),

then to evaluate if what is provided corresponds to what is required, one should understand that an exchange is first a sharing : a exchanged data is a data shared by the entities exchanging it, that means that the exchange is valid if entities have compatible descriptions of the exchanged thing

that's for me the base of the validity of an assembly

>>

Design principles

- The contract is an assessable first class object :
  - State of the contract = validity of the assembly,
  - Validity : based on mutual commitments of exchange,
  - Exchange : based on the sharing.

- The contract reifies, in a generic way :
  - Architectural Pattern
  - Conformity,
  - Compatibility,
  - Responsibility,

- The contract applies to :
  - **Architectures** : collaborative, observable, « pointable »,
  - **Formalisms** : modular, equiped, mechanizable interpretation for observations and assumption / guarantee,

<<

So in order to allow to reason about the validity of an assembly the contract reifies a set of principle :

– the first is the “architectural pattern” : this pattern, that is an abstract architectural configuration, is used by the contract to distinguish its participants among the entities of the architecture. *Each contract type is associated to a given predefined pattern*. Examples of patterns can be seen in Design Patterns, Architectural Styles etc

  using the pattern the contract can then discover the relations between its participants, then it can see what they exchange, and therefore point at the properties of its participants that it should consider,

>>

Design principles

- The contract is an assessable first class object :
  - State of the contract = validity of the assembly,
  - Validity : based on mutual commitments of exchange,
  - Exchange : based on the sharing.

- The contract reifies, in a generic way :

  - P1 Conformity, — the implementation of an element fulfills its specification
  - P2 Compatibility,
  - P3 Responsibility,
- The contract applies to :
  - **Architectures** : collaborative, observable, « pointable »,
  - **Formalisms** : modular, equiped, mechanizable interpretation for observations and assumption / guarantee,

<<

the contract reifies also, as required, the conformity of the implementations of its participants to their specifications : that is if their implementations fulfills their specifications,

>>

Design principles

- **The contract is an assessable first class object :**
  - State of the contract = validity of the assembly,
  - Validity : based on mutual commitments of exchange,
  - Exchange : based on the sharing.

- **The contract reifies, in a generic way :**

  - Architectural Pattern
  - P1 Conformity,
  - P2 Compatibility,
  - P3 Responsibility,

Non-contradiction between specifications that, describing distinct elements, constraints the same exchanged data.

- **The contract applies to :**
  - **Architectures** : collaborative, observable, « pointable »,
  - **Formalisms** : modular, equiped, mechanizable interpretation for observations and assumption / guarantee,

<<

the contract reifies the compatibility of its participants : that is the non contradiction between specifications, that describing distinct participants, constrains the same exchanged data.

>>

Design principles

- **The contract is an assessable first class object :**
  - State of the contract = validity of the assembly,
  - Validity : based on mutual commitments of exchange,
  - Exchange : based on the sharing.

- **The contract reifies, in a generic way :**
  - C2 Architectural Pattern
  - P1 Conformity,
  - P2 Compatibility,
  - P3 Responsibility, — Origin of the non-conformity or the incompatibility, among elements involved in the result
- **The contract applies to :**
  - **Architectures** : collaborative, observable, « pointable »,
  - **Formalisms** : modular, equiped, mechanizable interpretation for observations and assumption / guarantee,

<<

finally the contract reifies the responsibilities of its participants : it makes explicit the origin of a non-conformity or incompatibility among its participants involved in this result.

>>

Design principles

- **The contract is an assessable first class object :**
  - State of the contract = validity of the assembly,
  - Validity : based on mutual commitments of exchange,
  - Exchange : based on the sharing.

- **The contract reifies, in a generic way :**
  - Architectural Pattern
  - Conformity,
  - Compatibility,
  - Responsibility,
- **The contract applies to :**
  - **Architectures** : collaborative, observable, « pointable »,
  - **Formalisms** : modular, equiped, mechanizable interpretation for observations and assumption / guarantee,

<<

an other point that should considered for this contract is that it is generic, but to a well defined and explicit extent :

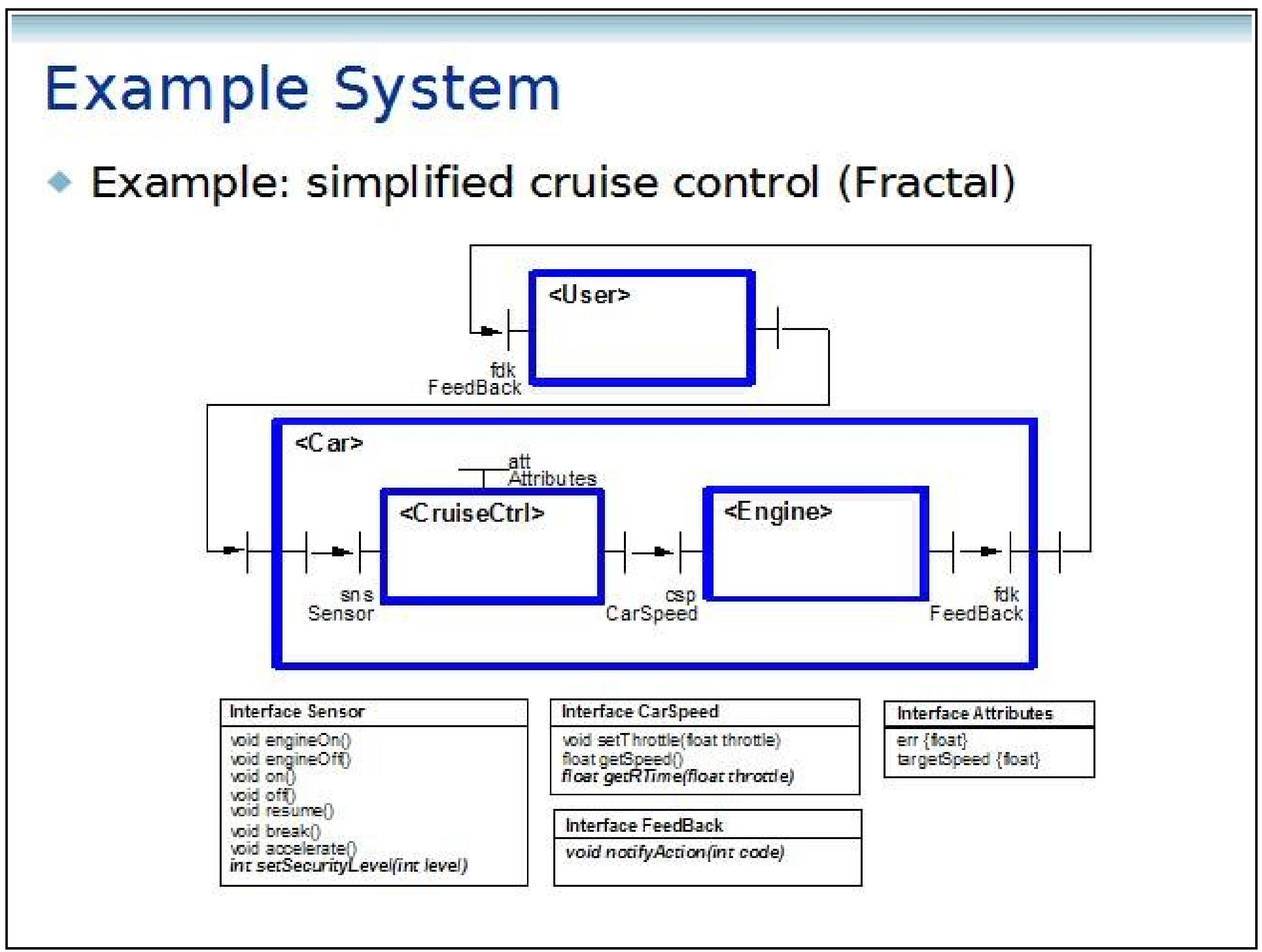
the contract does apply to various architectures : provided that these architectures :

- rely on collaboration of their parts
- can be observed : as well for their configuration as for their working
- and finally the elements of these architecture should be “pointable”, by a name or a path ...

the contract accepts various formalisms : provided that these formalisms :

- are modular
- come with the tools enabling the checking of conformity and compatibility
- have a syntax and semantic such as the interpretation of specifications into assumption and guarantee can be mechanized
- have a syntax and semantic such as it is possible to mechanically extract from the specifications the observations of the system possibly required for the evaluation of conformity and compatibility

>>



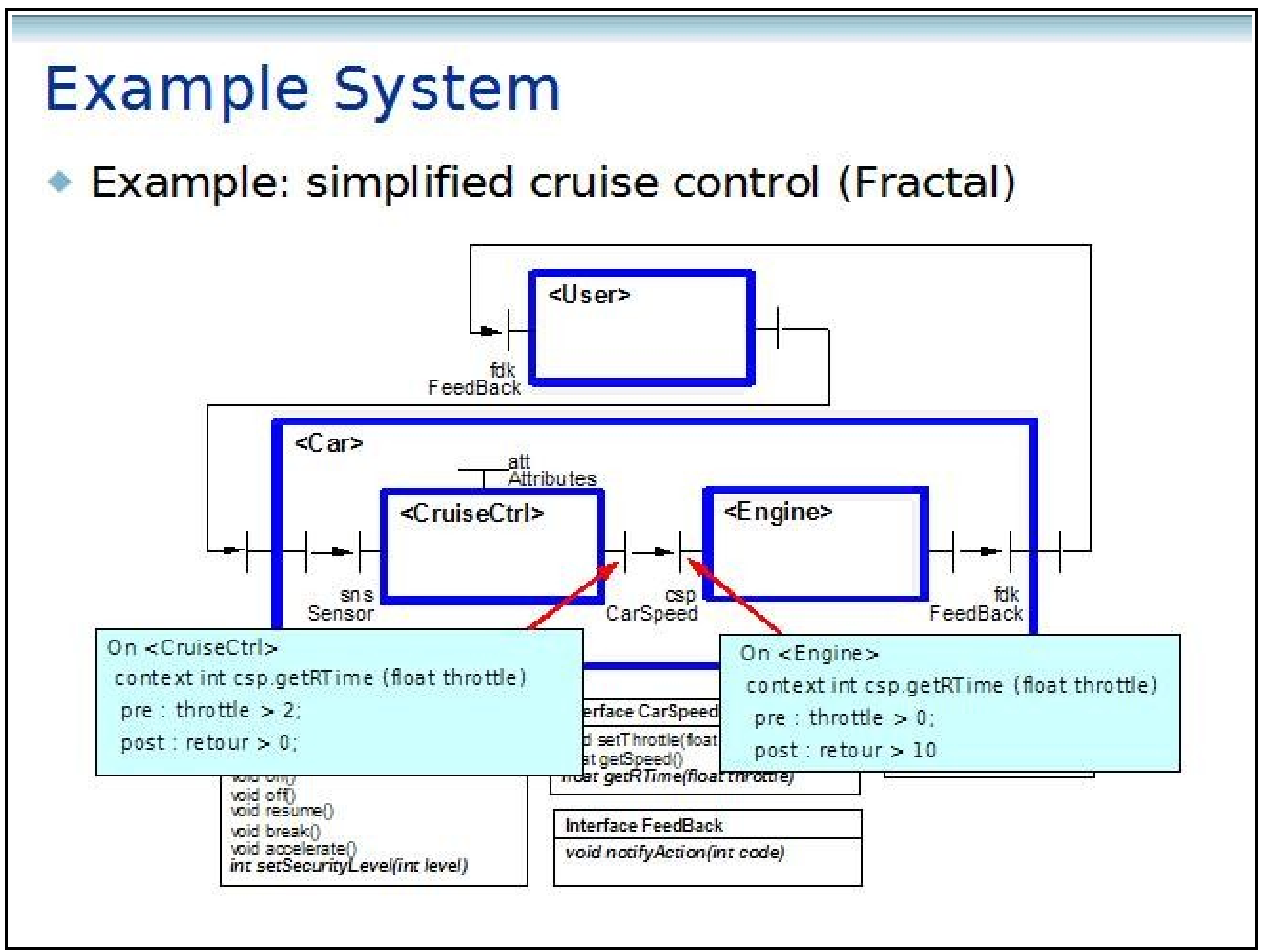
<<

to illustrate the following of the presentation, I will consider a very basic components system. Components are the blue rectangles, they are bound through interfaces drawn as horizontal “T”. Server interfaces are on the left of the component client ones are on their right side. I take a hierarchical component model, such as the “Fractal Component Model” (developed at France Telecom R&D).

This system represents a car (“<Car>”) and its driver (“<User>”). The car contains its engine (“<Engine>”) which is controlled by a cruise control device (“<CruiseCtrl>”) in charge, when engaged, of maintaining a certain speed chosen by the driver.

In the Fractal representation, server interfaces are on the left of components, and client ones on their right.

>>



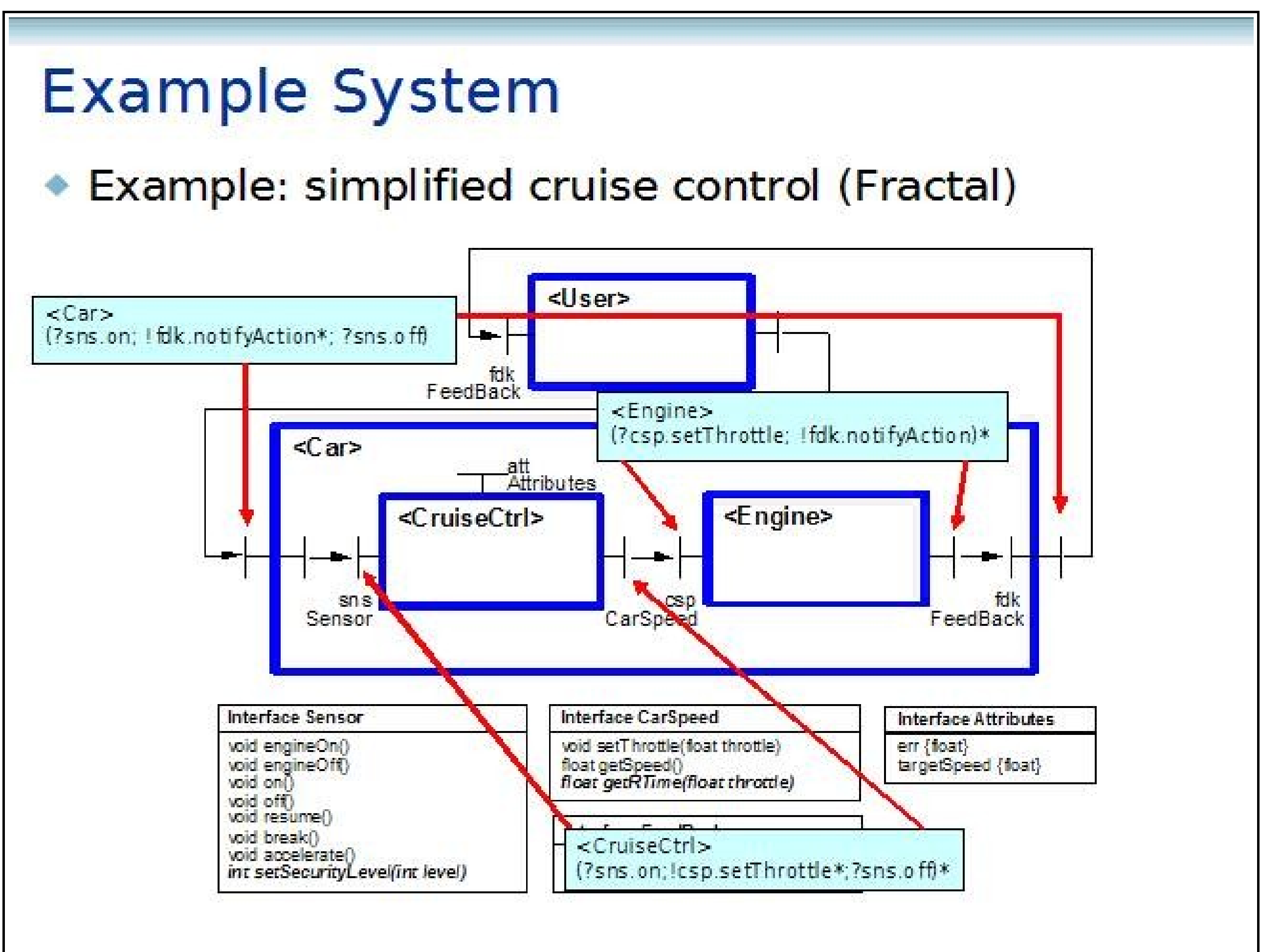
<<

On this system, I can consider various specifications.

The first example I take, is specification expressed in the "CCLJ" formalism which is like OCL but adapted to components with the syntax of java.

So I can consider the specifications of two connected interfaces "<CruiseCtrl>.csp" and "<Engine>.csp". I will need the contract to check that these specifications are respected and that they are compatible, to say the connection between the interface is valid.

>>



<<

but I can also consider an other kind of formalism. So I take the "behavior protocol" formalism developed initially for the SOFA component model and then integrated to the Fractal component model.

Each behavior protocol is a kind of regular expression that describe for a component (or group of components) the allowable sequences of incoming and outgoing messages through their interfaces.

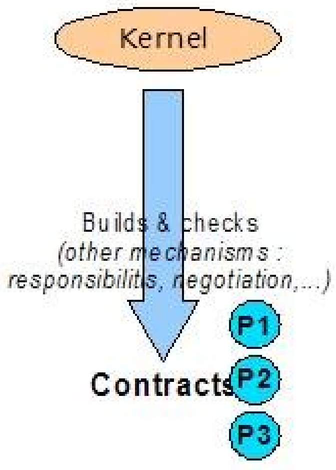
For example, on the <CruisCtrl> component : "(?sns.on;!csp.setThrottle*;?sns.off)*" expresses that this component waits for a call on the method "sns.on", then it will emit an undefined number of times a call on the "csp.setThrottle" method, until it receives a call on the method "sns.off".

Here again, the contract between the car, the cruise and the engine shall show that each component is conform to its specification and that these specifications are compatible to show that the assembly is valid.

>>

Framework design :

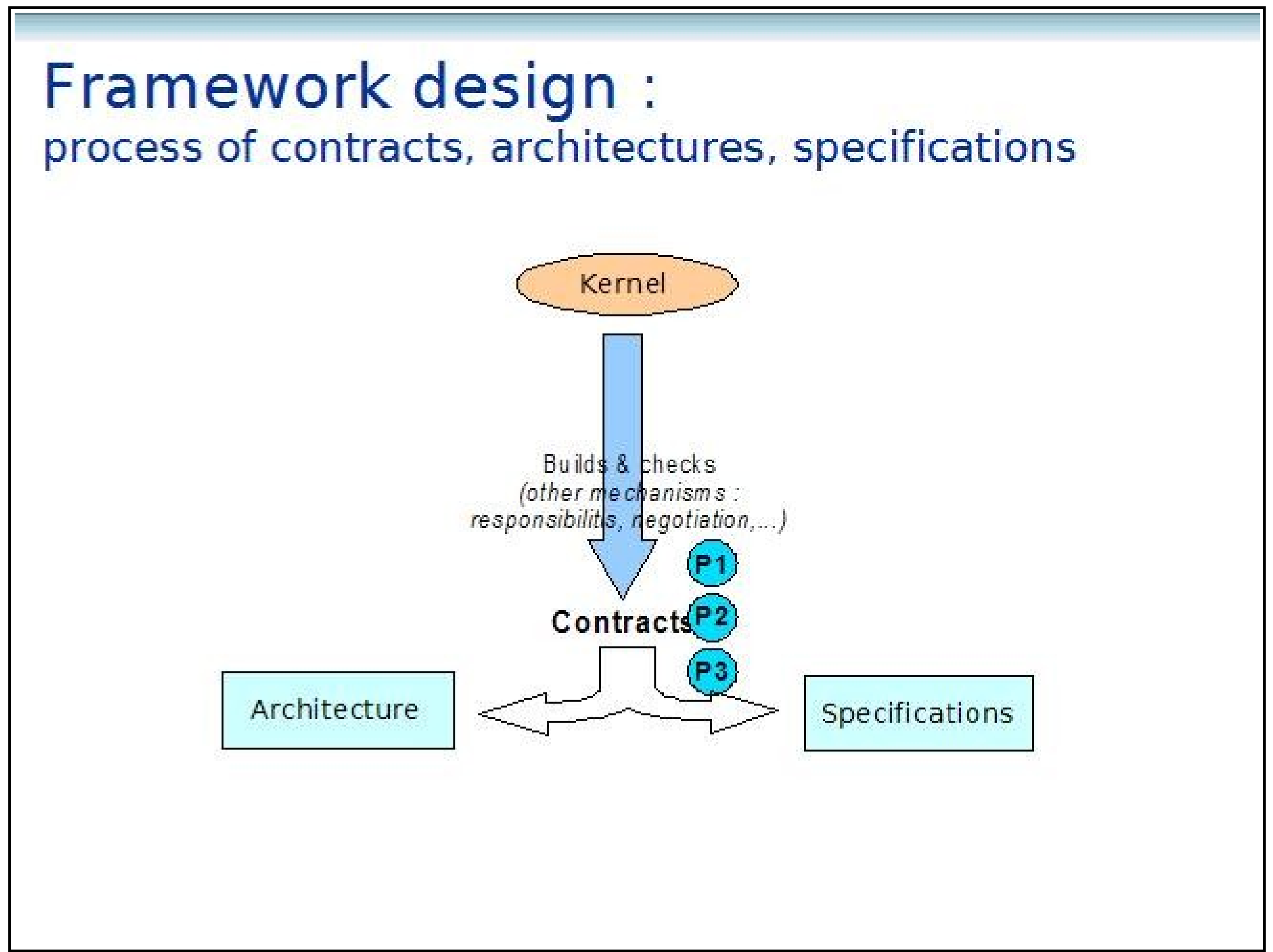
process of contracts, architectures, specifications



<<

the framework that implements the model is based on a kernel which models and provides the generic mechanisms to build and check contracts that reify the properties I am looking for (conformity, compatibility, responsibility)

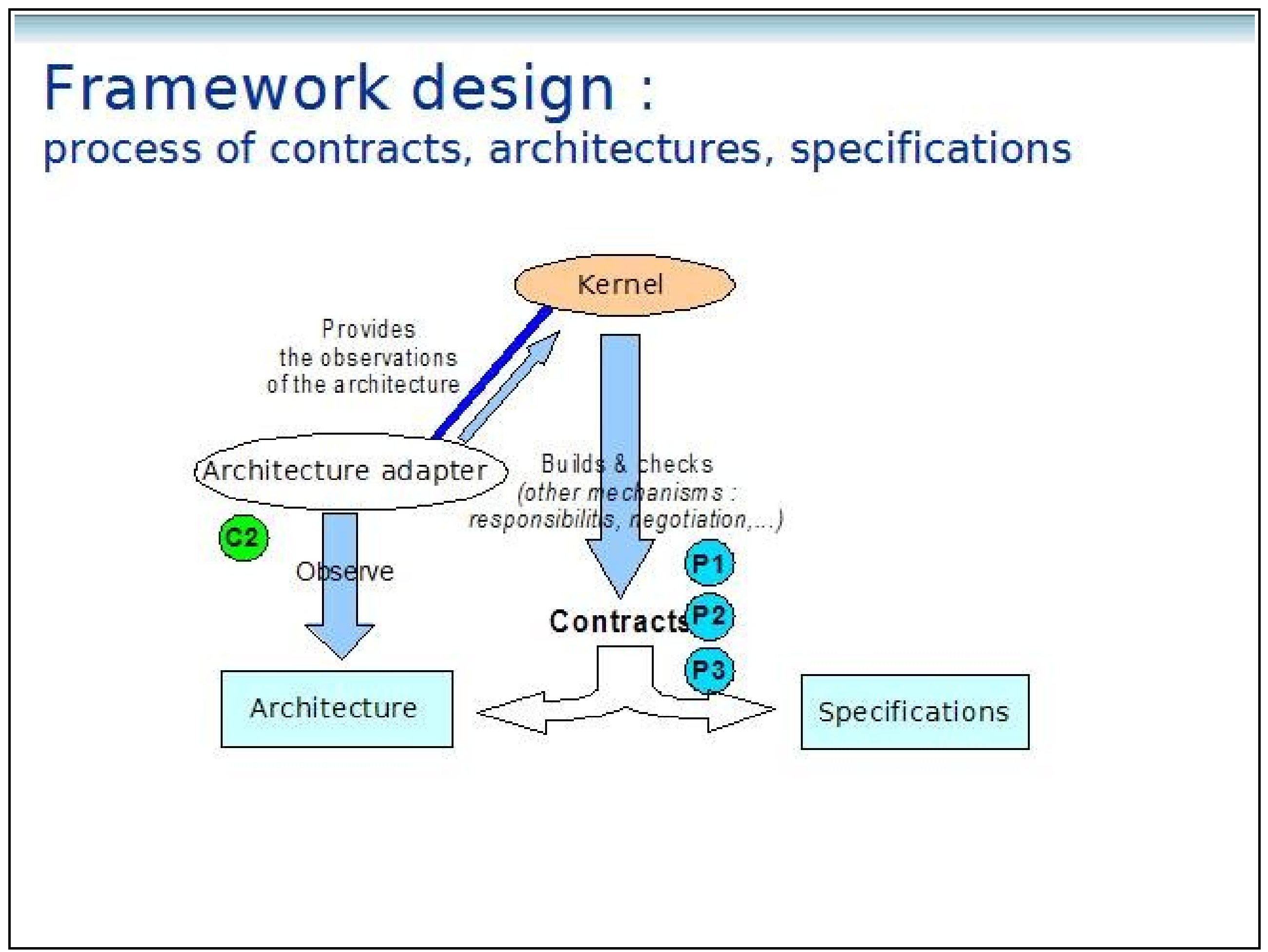
>>



<<

but the contract has to apply to concrete architecture and specifications,

>>

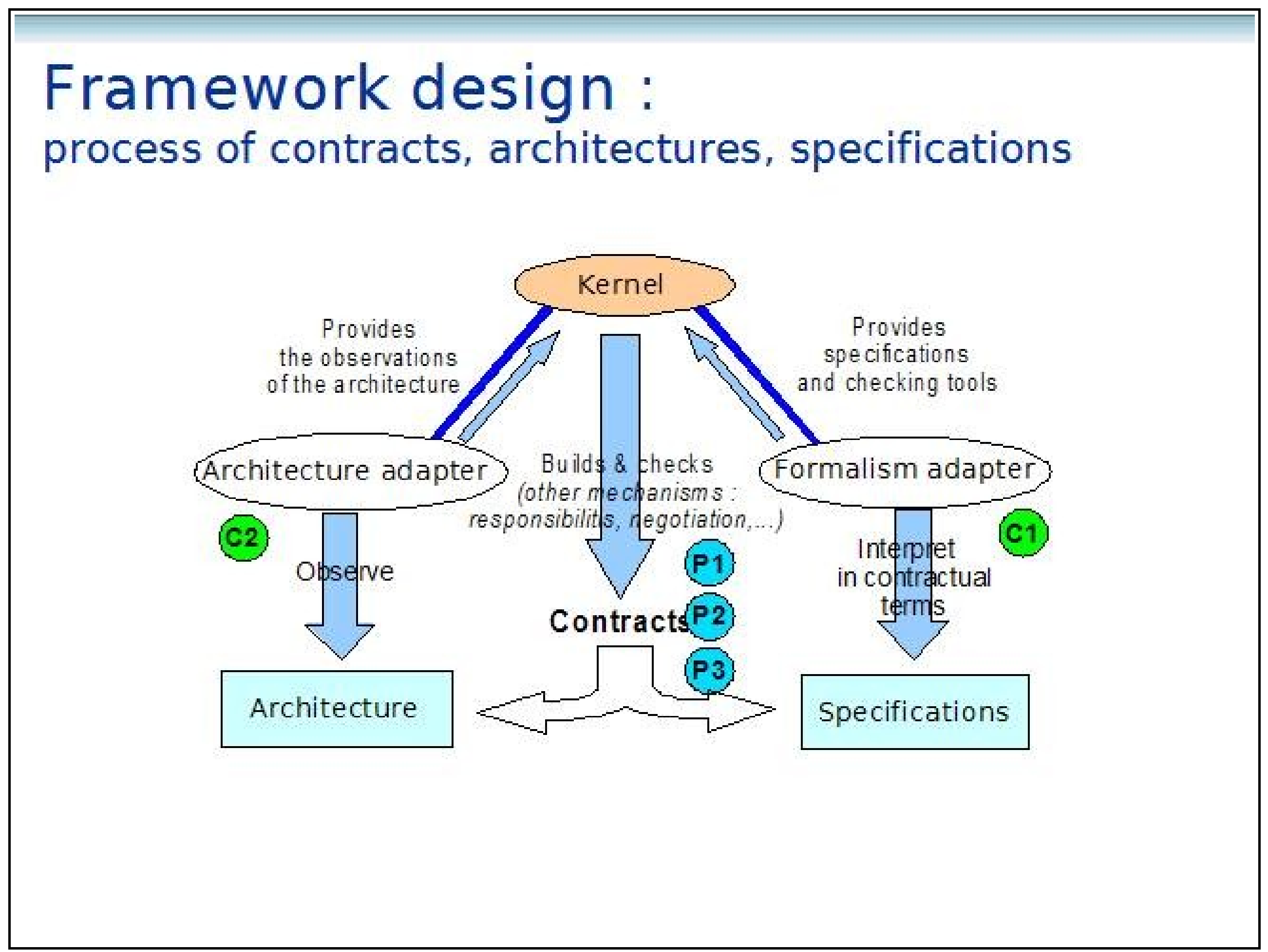


<<

so in order to build contracts that apply to a given architecture, the kernel which is generic, uses an “architecture adapter” that observes the architecture and describes it in the generic terms defined by the kernel.

The generic terms, for the description of an architecture, defined by the kernel are not meant to be an abstraction of all possible architectures, but they reify the architectural concepts required by the contract.

>>



<<

in order to build contracts that accept various kind of concrete specifications, the kernel which is generic, uses one “formalism adapter” for each formalism.

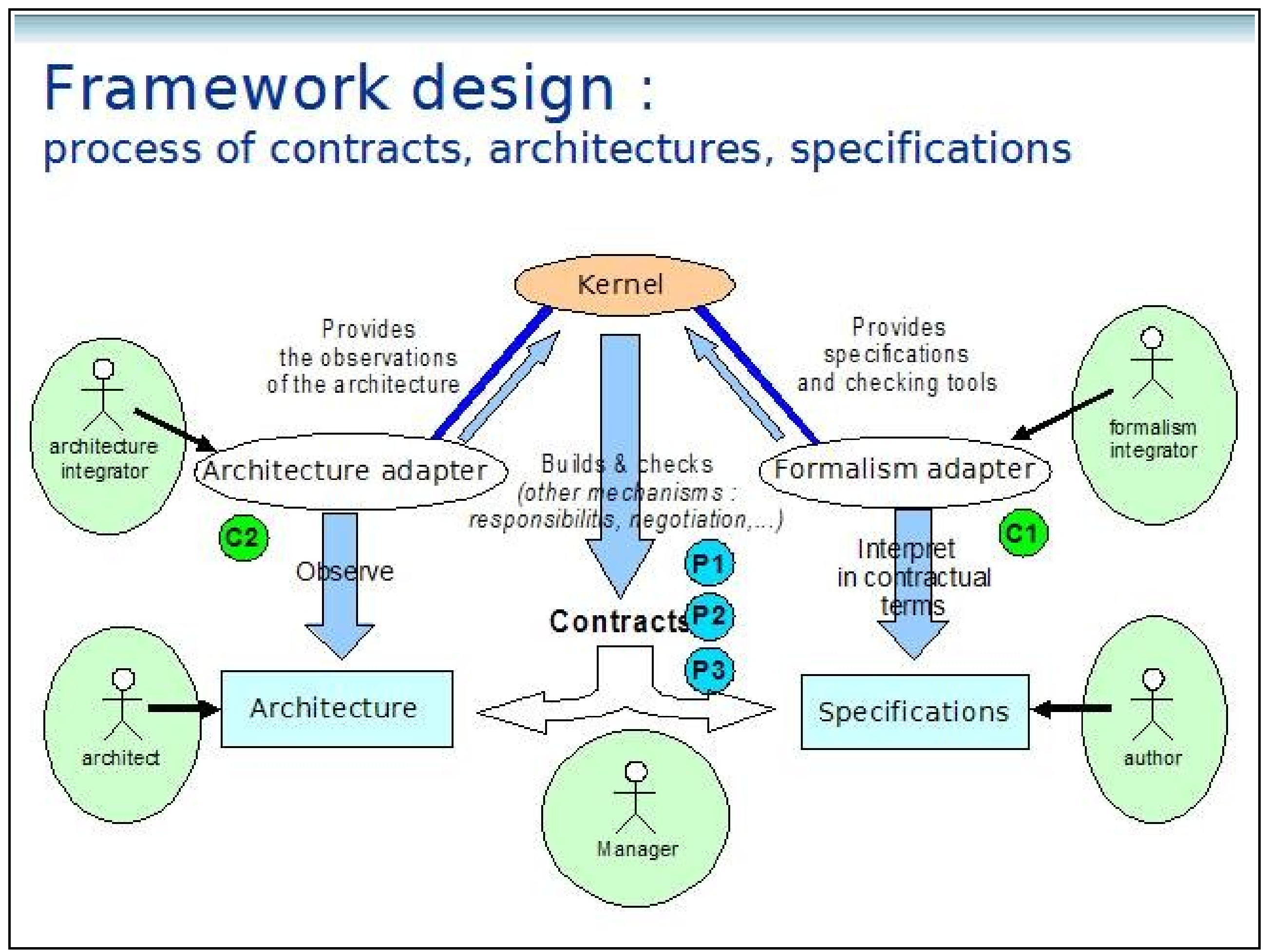
The adapter interprets and translates each specification, in its associated formalism, into assume-guarantee rules described with the generic terms defined by the kernel.

The adapter provides also the kernel with the tools for assessing the compatibility and conformity of the specifications in its associated formalism.

As for the architecture, the generic terms defined by the kernel to model the specification are not meant to capture the common points of all the formalism. These terms only reify the abstract principles of specifications that the contract needs to handle : is the specification an assumption, a guarantee, does it require system observations for its evaluation, etc...

Concretely both formalism and architecture adapter specializes some of the generic concepts defined in the kernel (some objects and mechanisms) for its concrete formalism and concrete architecture.

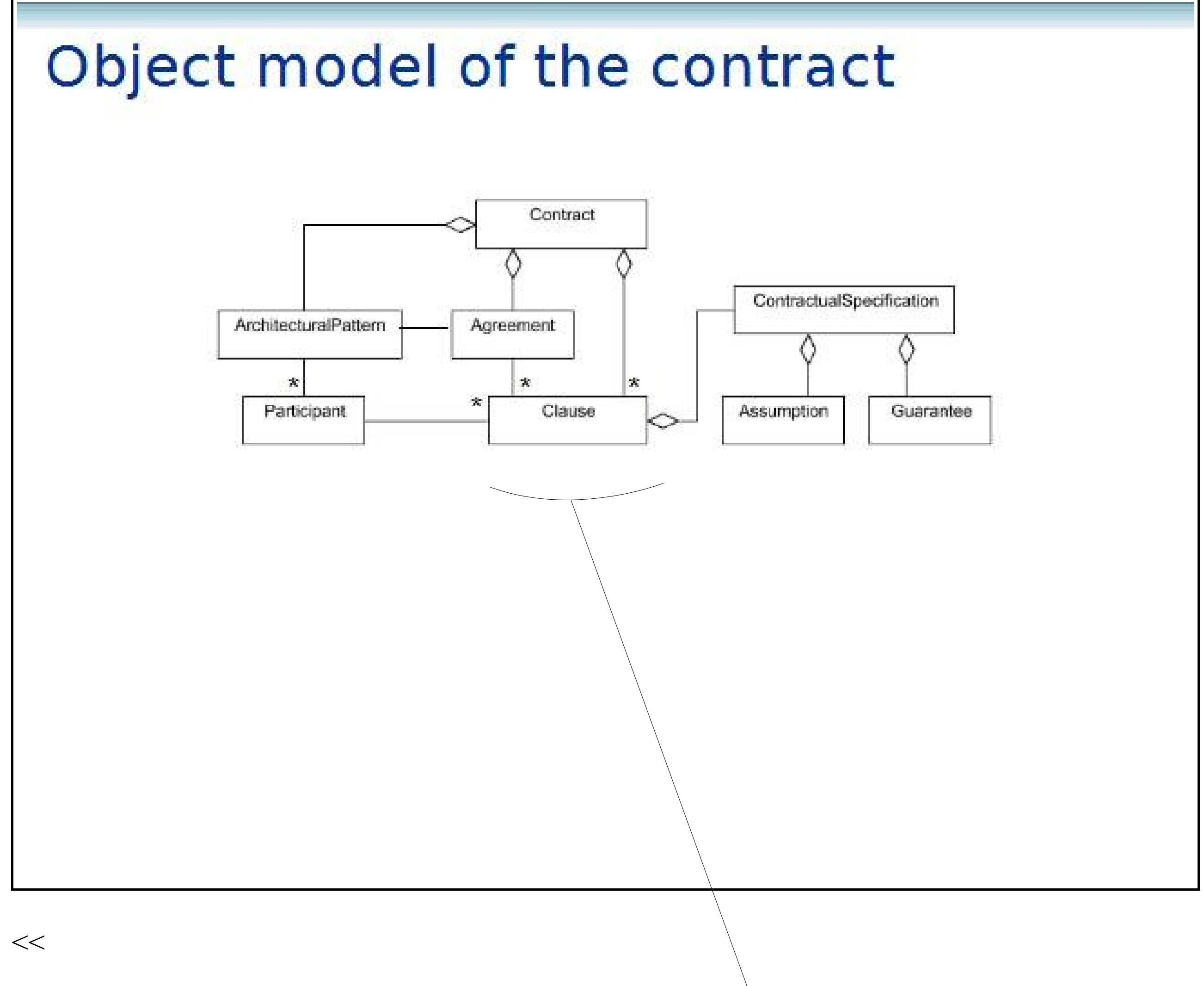
>>



<<

Here we can see around the kernel, the various actors that are involved in the system implementation and working, and that we met at the beginning of this presentation.

>>



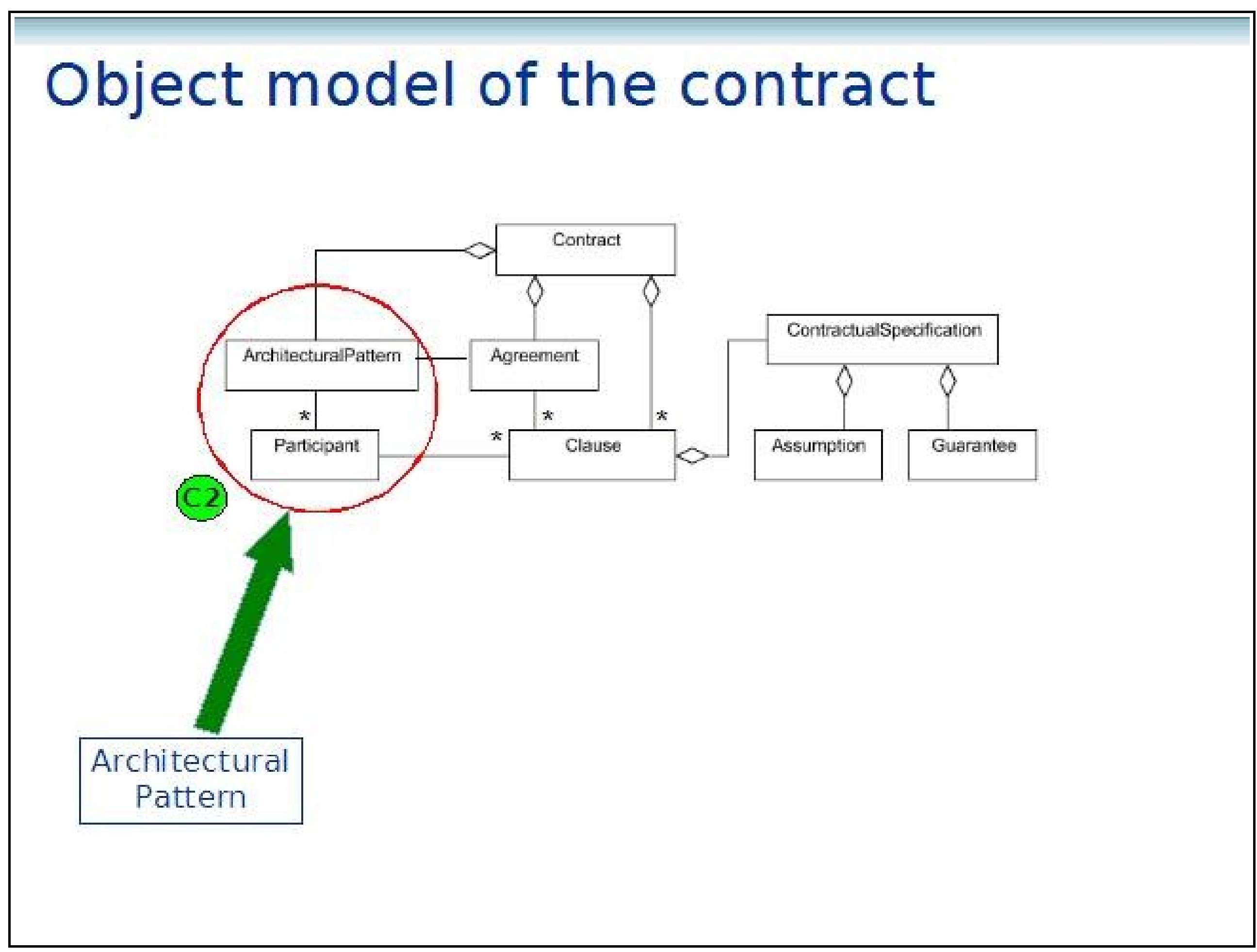
<<

Here a simplified class diagram of the contract. In the middle column of the diagram are the main elements of the contract : its clauses and the agreement.

The states of the clauses represent the conformity of the contract participants to their specifications.

The state of the agreement represent the compatibility of the clauses, that is the compatibility of the specifications of the participants.

>>



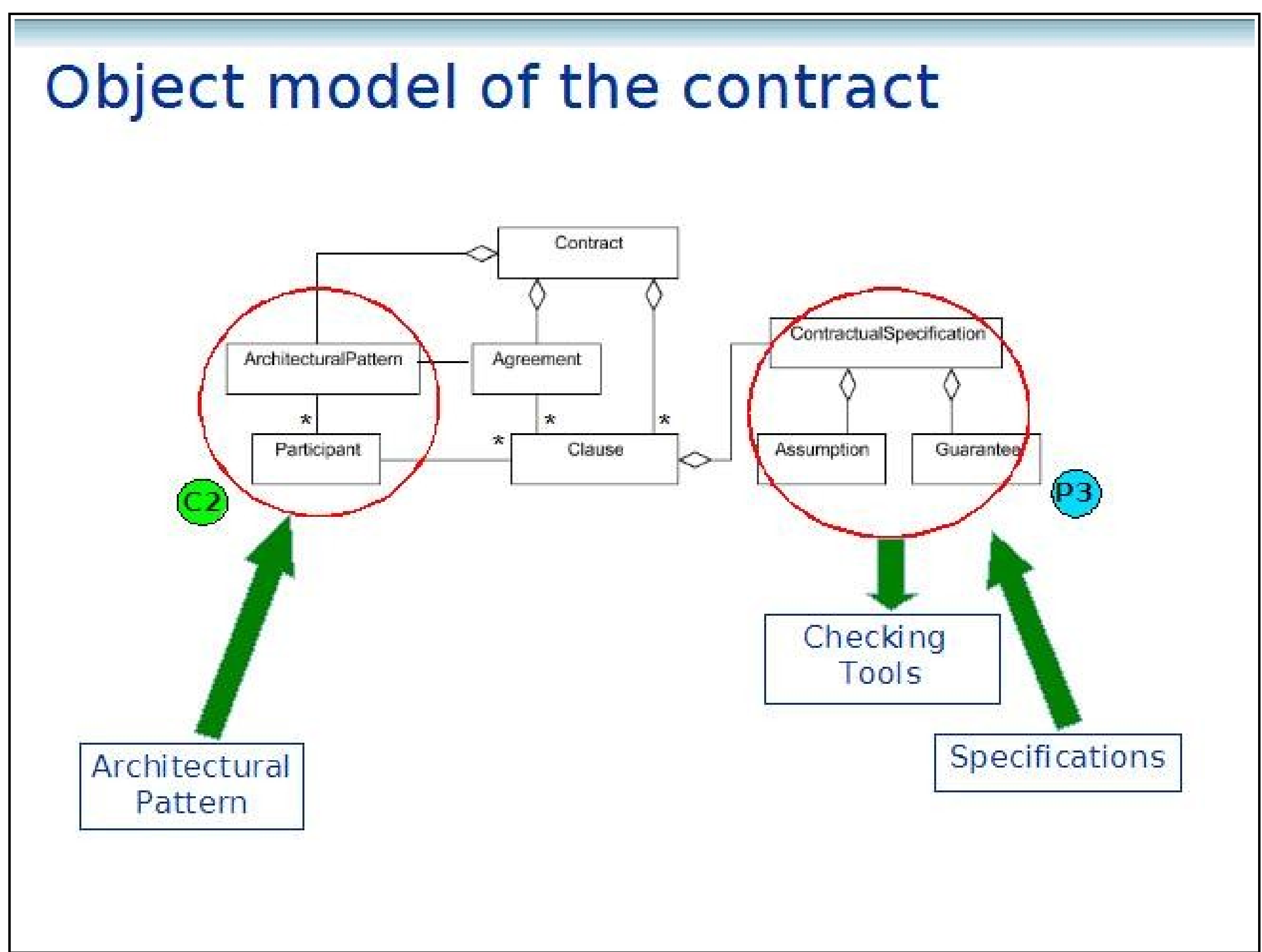
<<

On the left side of the diagram is the generic modeling of the architecture to which is applied the contract :

the "ArchitecturalPattern" (*defined by the contract type*) hold the generic architectural configuration to which the contract applies, but also points through the "Participant" to the concrete architecture entities involved in the contract.

both "ArchitecturalPattern" and "Participant" are independent of the architecture, so this contract principle model is uncoupled from the concrete architecture (characteristic C2).

>>



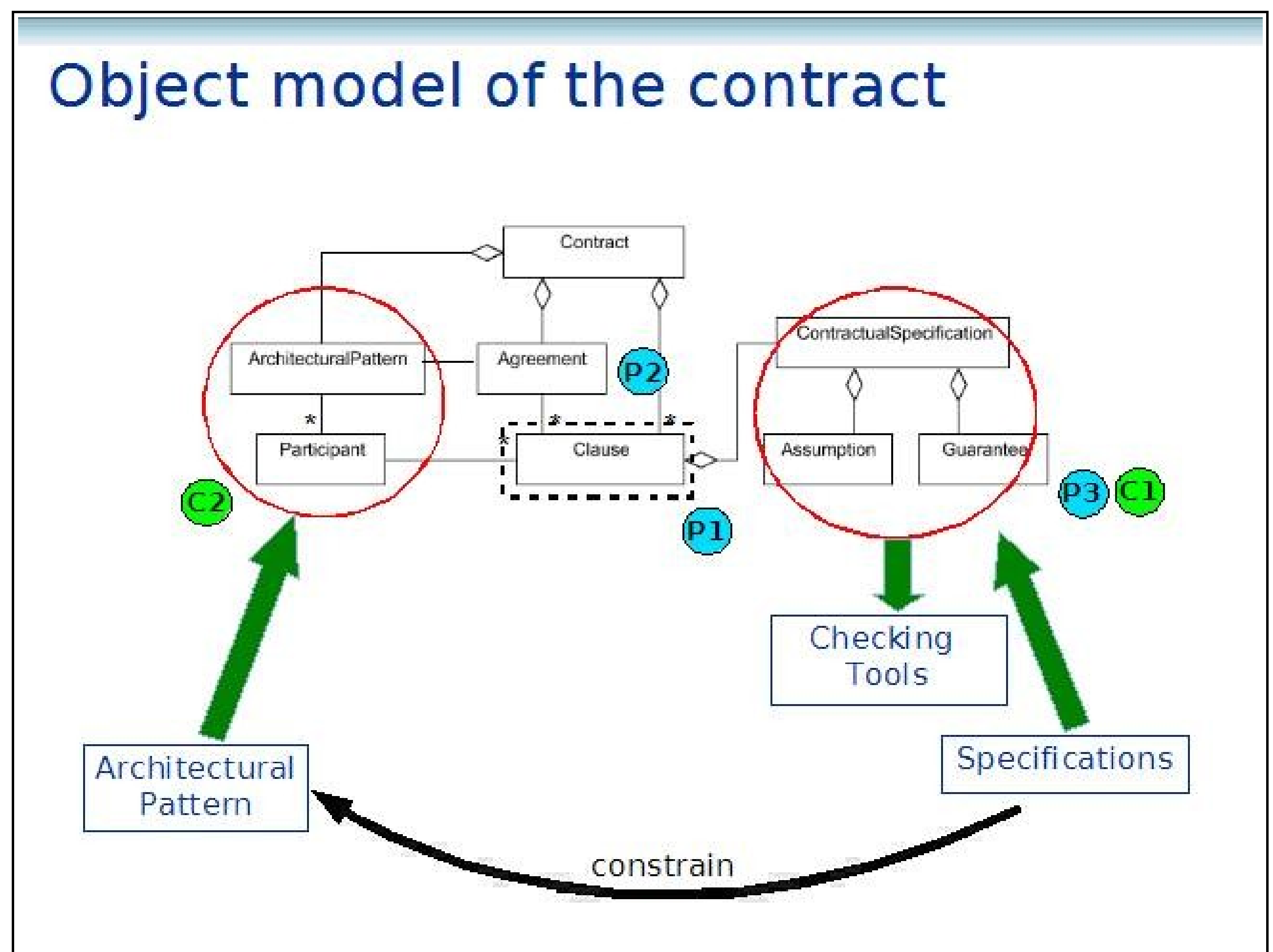
<<

On the right side of the diagram is the generic modeling of the specification interpreted in assume-guarantee terms.

Each specification is interpreted and translated into a “ContractualSpecification” by the formalism adapter that also provides its checking tools.

This reification of the “Guarantee” drives to the one of the responsibilities, as we are going to see with the “Clause”. Also important is the fact that this approach of specifications is independent of their concrete formalism and of the kind of property they describe (functionnal, non functionnal ...).

>>



<<

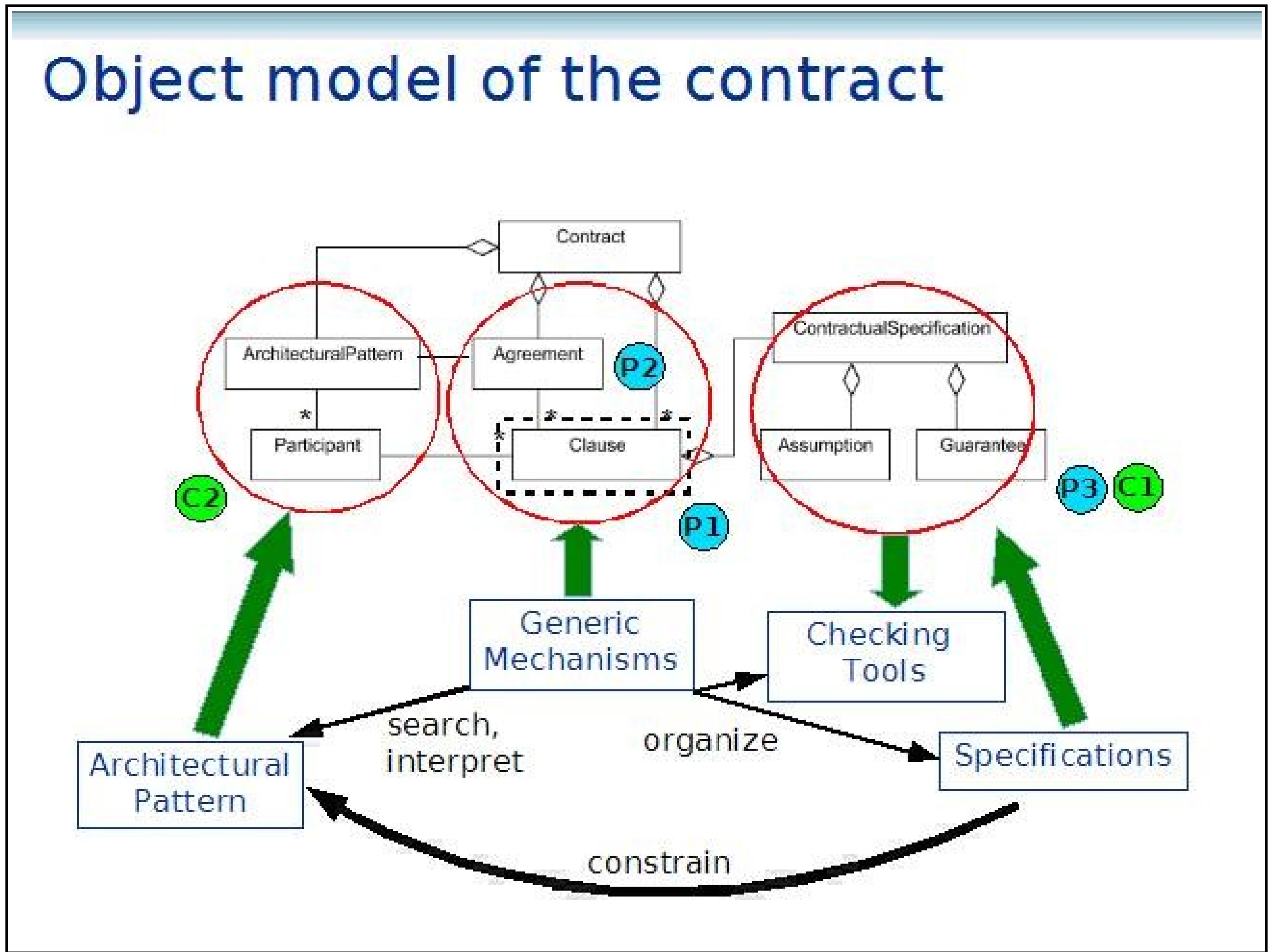
As shown on the drawing, each “Clause” is there to make a link between a “ContractualSpecification” and the “Participant” it constrains. The participant then is responsible for the satisfaction of the “Guarantee” as long as the “Assumption” is satisfied.

So the “Clause” is the key element that reifies the responsibility through the link it represents (property P3).

Moreover the “Clause”, through its state represents the conformity of a “Participant” to its “ContractualSpecification”, so it reifies the property P1 : conformity.

The “Agreement” hold a process (provided by the formalism adapter) that with respect to the “ArchitecturalPattern” assesses the compatibility of the “Clause”s. The “Agreement” state represents the compatibility of the clauses. It then reifies the property P2 : compatibility of the specifications.

>>

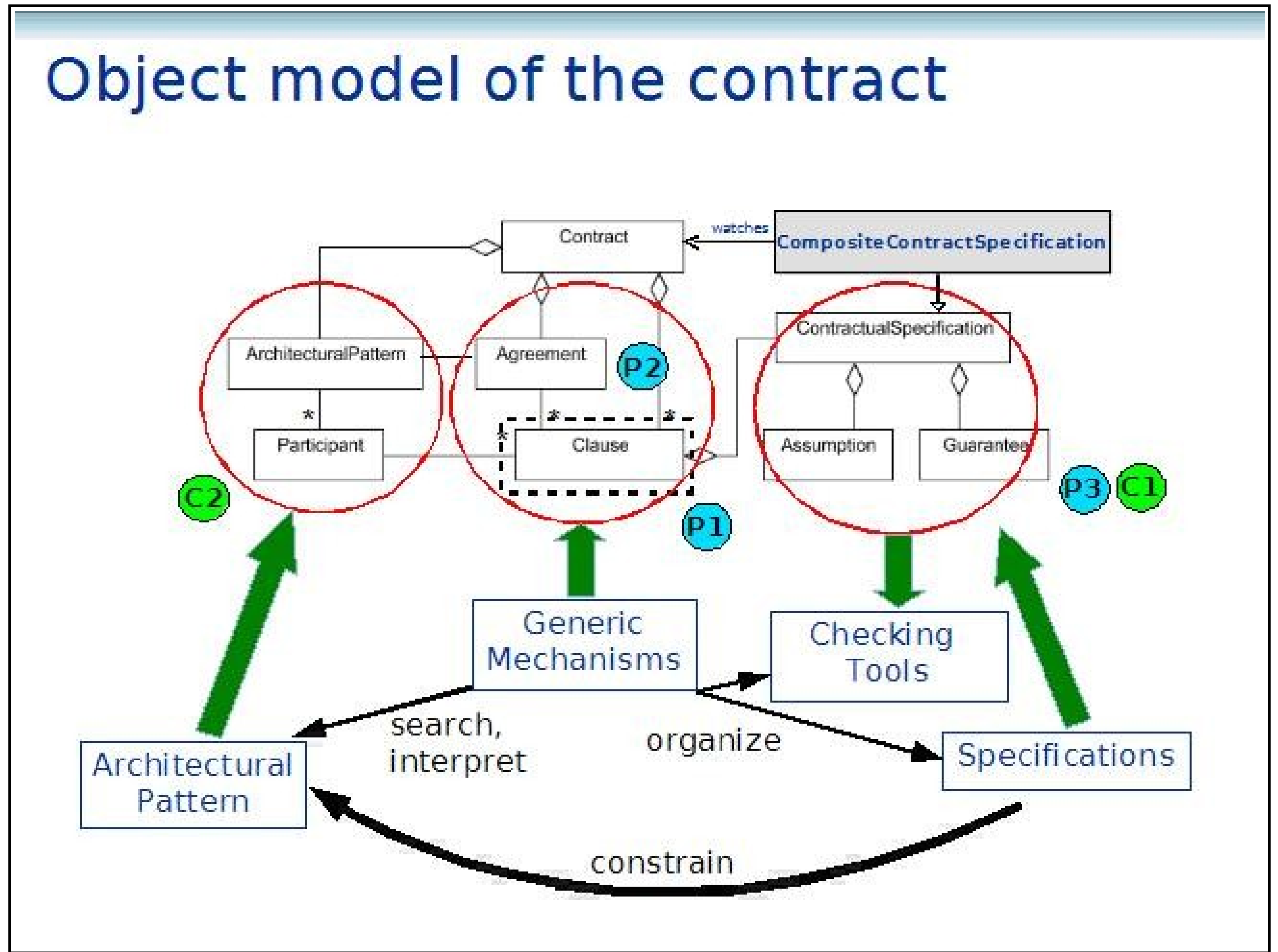


<<

the kernel provides the contract with generic mechanisms that :

- search the application architecture for architectural pattern,
- analyzes a pattern found in terms of participants and relations and data exchanged,
- search the specifications list for the one involve in the exchanges discovered in the pattern,
- organize this specifications in clauses and agreement,
- organize the evaluation of clauses and agreement,

>>



<<

There is one other type of contractual specification named “CompositeContractSpecification” which inherits from the “ContractualSpecification”. This contractual specification refers in addition to a contract, that's why it's “CompositeContract” : it enables a contract to depend of another one.

the state of a clause containing a “CompositeContractSpecification” depends on :

- the conformity of the system to the assume-guarantee specification,
- the state of the contract to which the “CompositeContractSpecification” refers to,

we'll in the following how this contractual specification can be used.

>>

Assertions based Contract

Dynamical assessment of the agreement

<User>
<Car>
<CruiseCtrl>
<Engine>

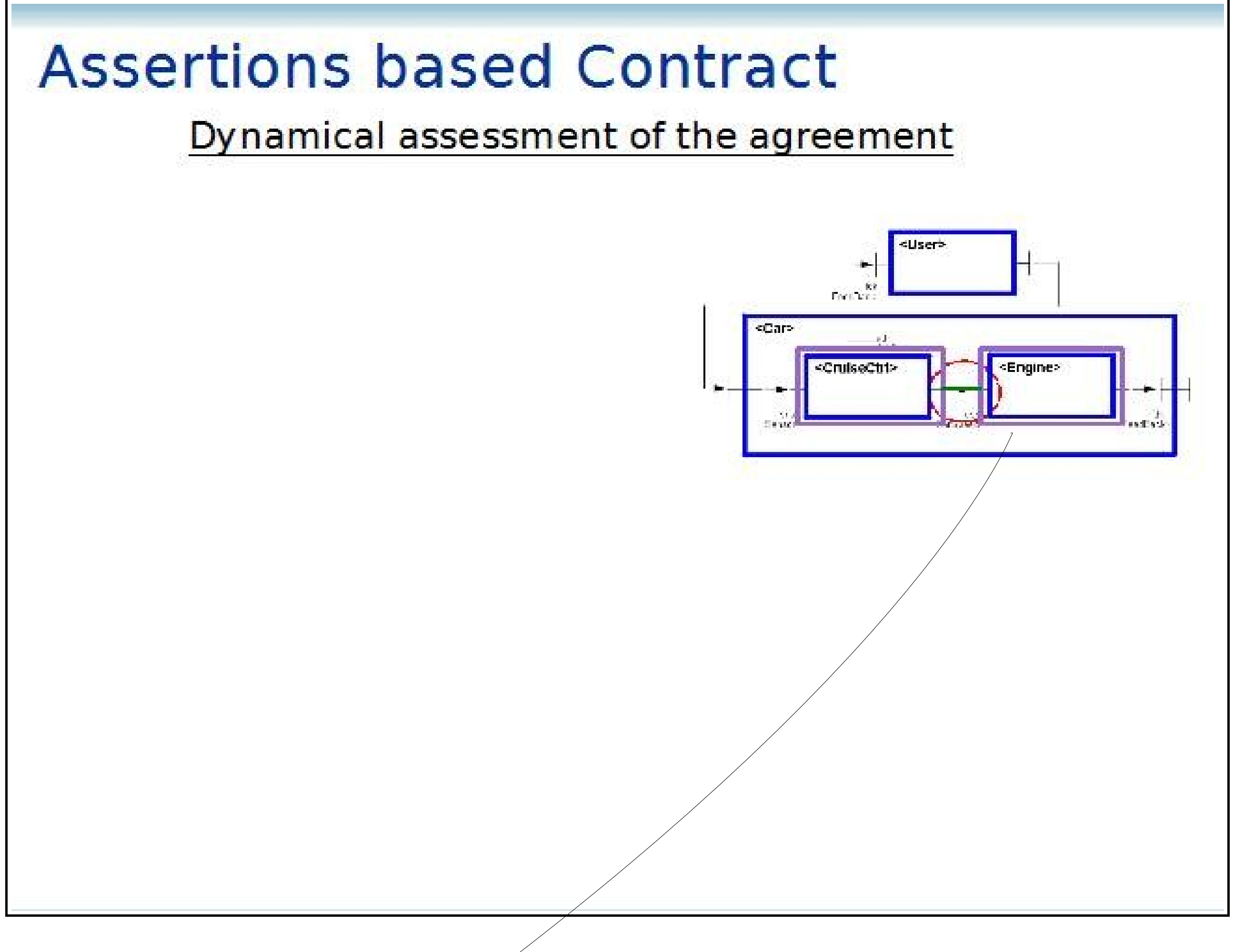
<<

Now I'm going to give you an example of a simple contract applied the “cruisectrl system” for specifications made of assertions.

This contract will guarantee the connection between the CruiseCtrl and Engine components.

Here we can notice that the clauses and agreement built on assertions will not be statically assessed but their state will dynamically evaluated on the base of the observations of the system execution.

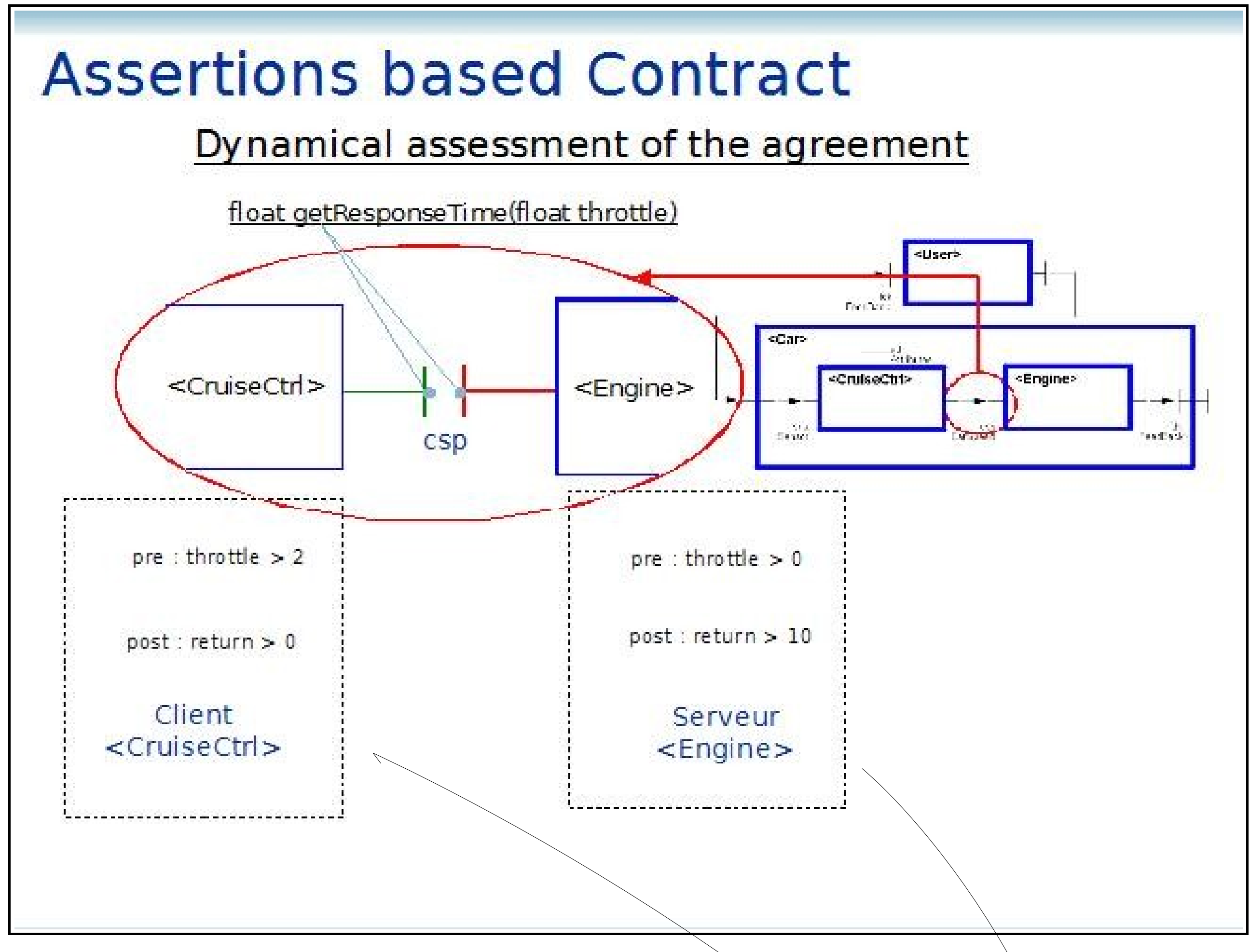
>>



<<

The contract that guarantees the connection between CruiseCtrl and Engine components relies on a most simple architectural pattern, made of two components and a connection between them.

>>



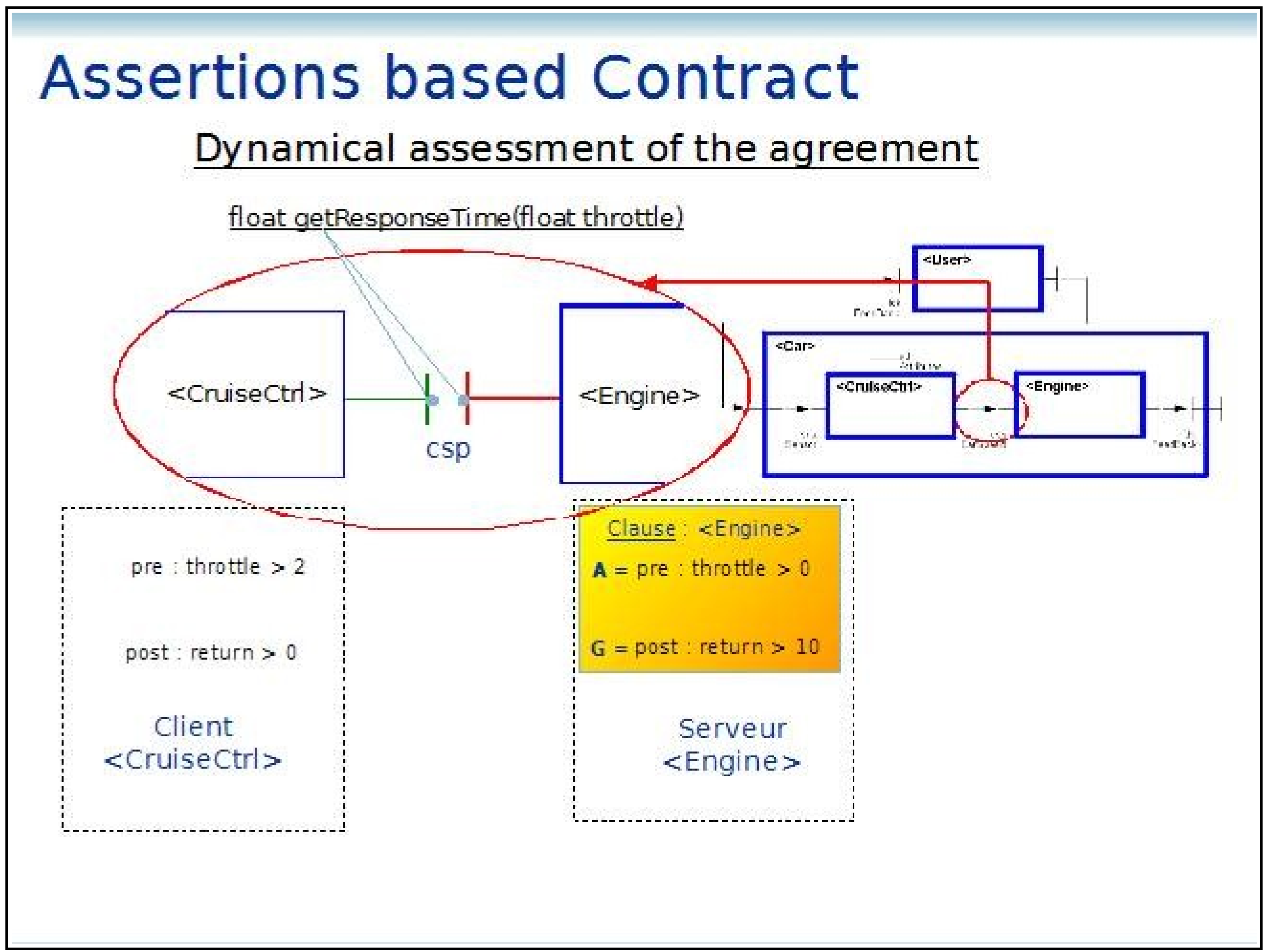
<<

Considering the relation between Engine and CruiseCtrl, the contract system will look for the data exchanged between them.

To keep simple, I will focus on the connection between the client and server methods "getResponseTime" provided by the Engine component, used by CruiseCtrl in order to known the time needed by the Engine to respond to an order given by the CruiseCtrl.

The contract system discovers the specifications on the two methods (client and server), from a specifications repository (handled by the formalism adapter).

>>



<<

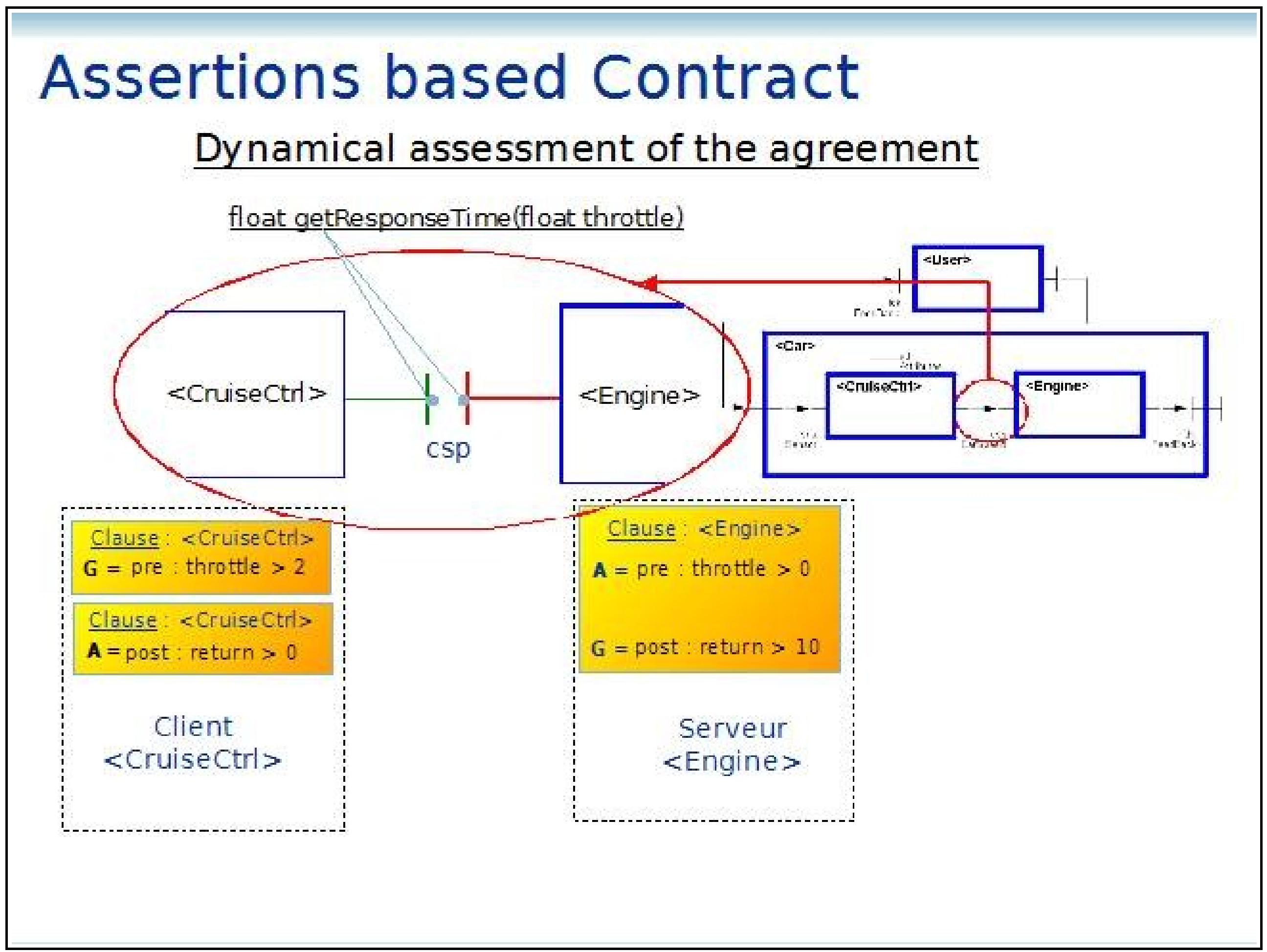
When the system has the specifications of client and the server, it must then interpret them in contractual terms, that is in assume-guarantee :

for the server side, the process is very classical :

the precondition is an assumption that the server makes on its environment, and the postcondition is what the server guarantees to its client provided that the precondition is satisfied,

so the clause of which the server is very classical.

>>



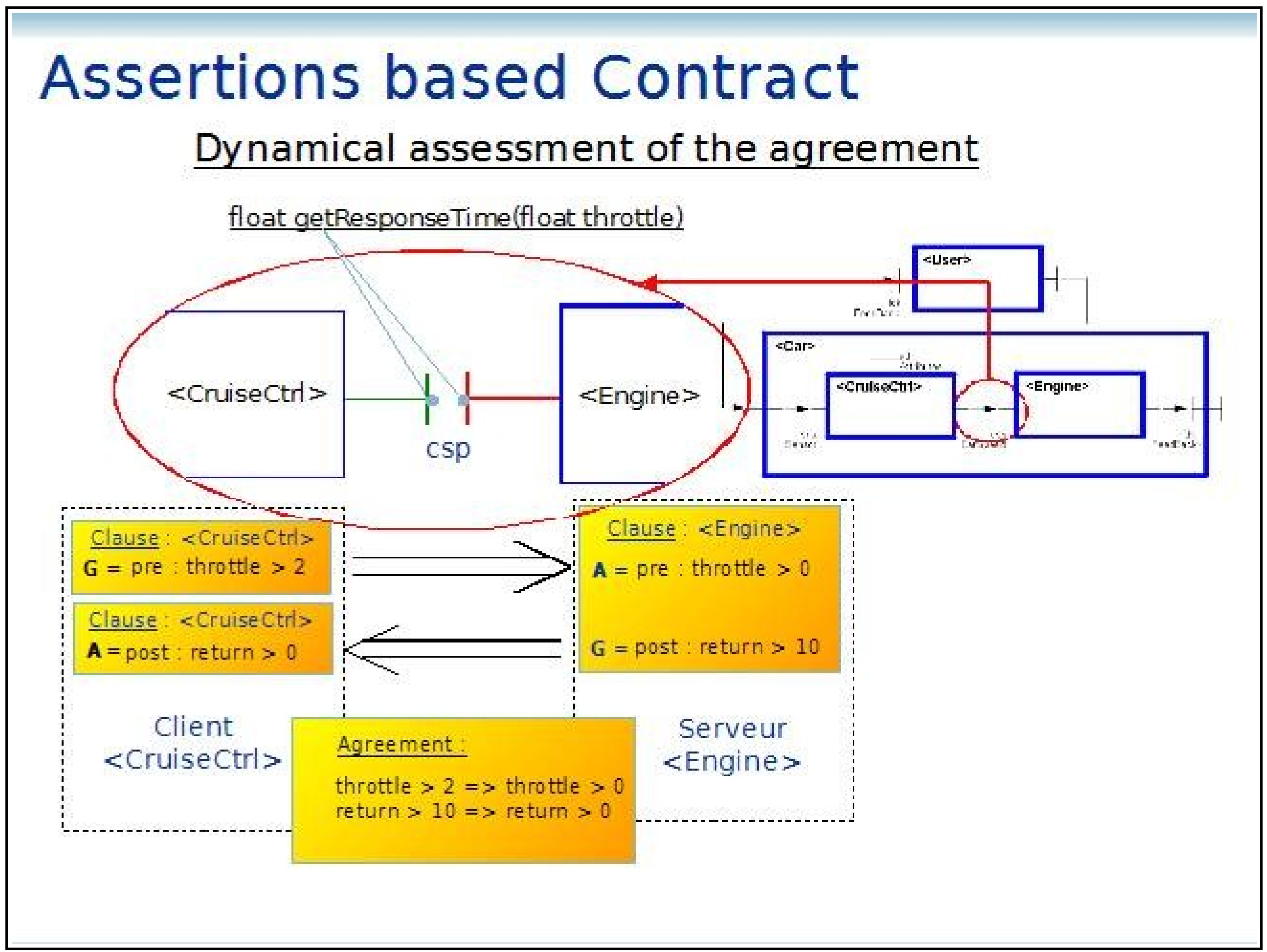
<<

On the client side, the things are slightly different :

the precondition is a guarantee that the client provides to the server, without any condition so there is no assumption. So the client has a first clause containing only a guarantee, the precondition.

the postcondition is an assumption that the client makes on the server, but this assumption has no guarantee counterpart from the client. Then the client has a second clause made of single assumption without guarantee.

>>



<<

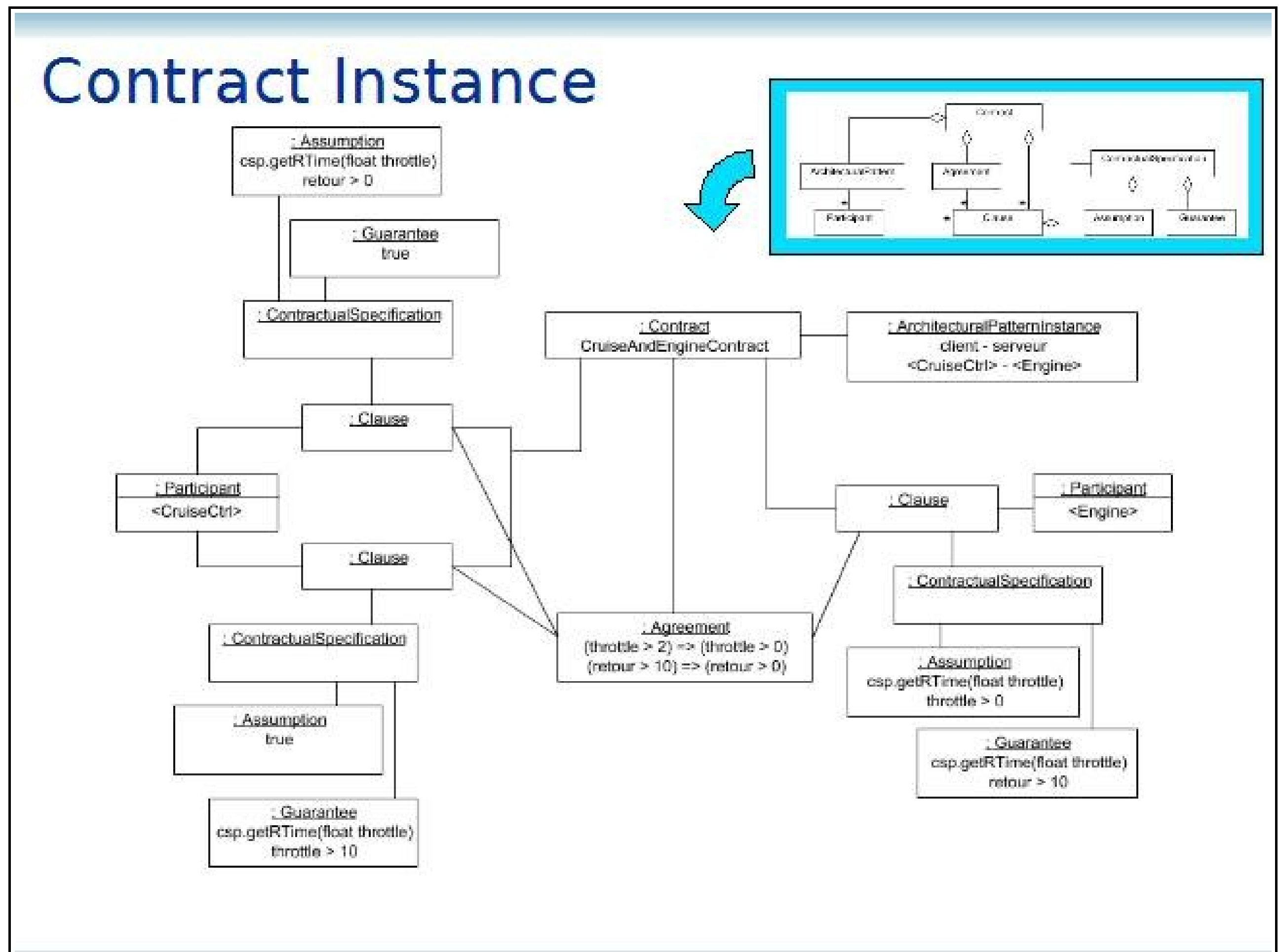
so we can see that as long as the client and the server conform to their clause, they conform to their specifications : but this conformity will be checked on the base of observations of the execution of the system.

Now we have to consider the compatibility of the specifications :
for the assertions,
if we consider an exchanged value, “throttle” for example,
and if we consider two specifications (assertions) constraining it on its emitter and receiver : then one specification is an assumption, and the other a guarantee :
then these two specifications are compatible if the guarantee fulfills the assumption : more concretely if the guarantee implies the assumption.

So the agreement, which is made of compatibility expressions between clauses, contains, in this example, two implications.

These implications are very simple in this example and could be statically proved, but in the general case this is not possible, so as for the clauses, the agreement assessment relies on the observations of the system execution.

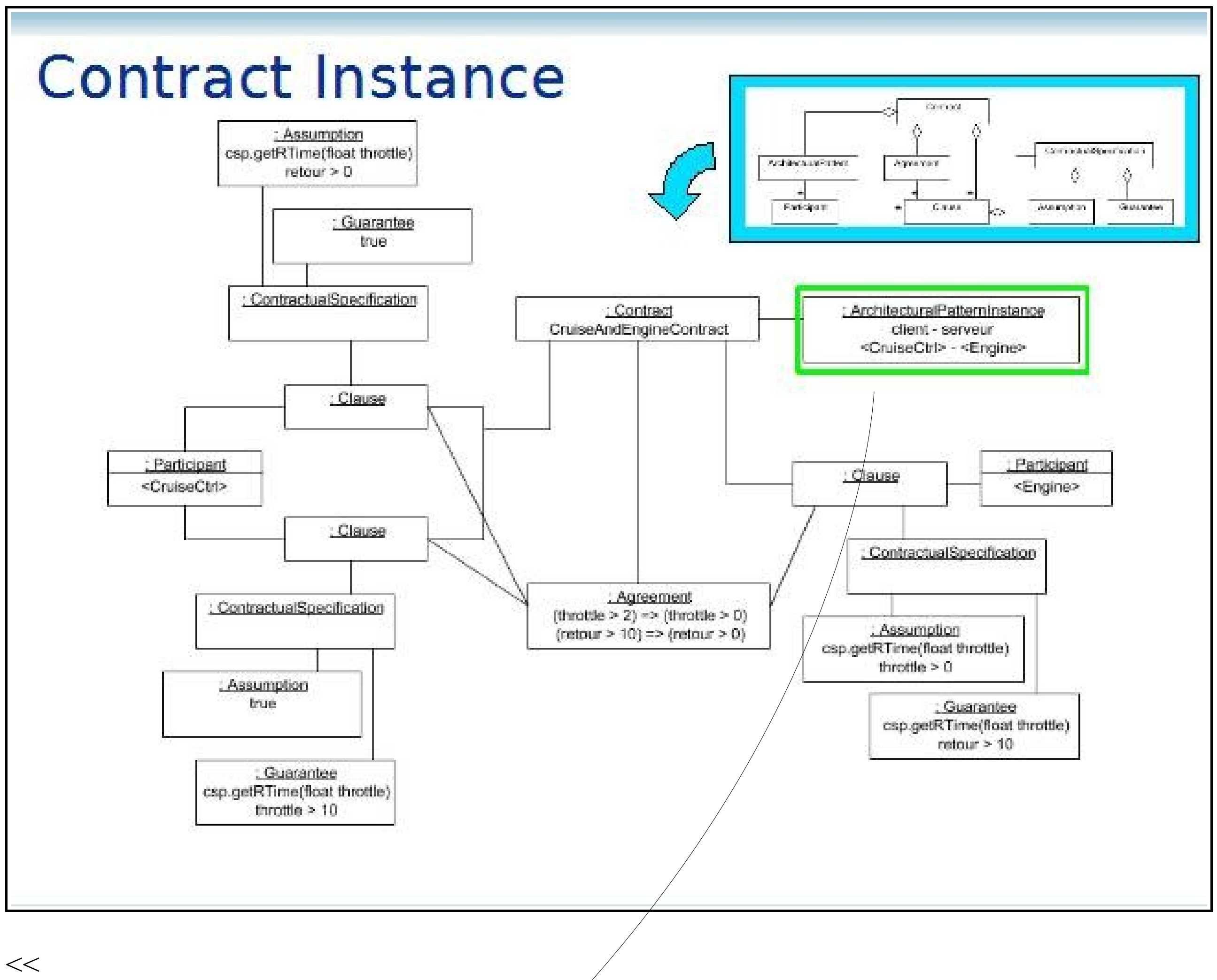
>>



<<

Here is an instance diagram of the contract model.

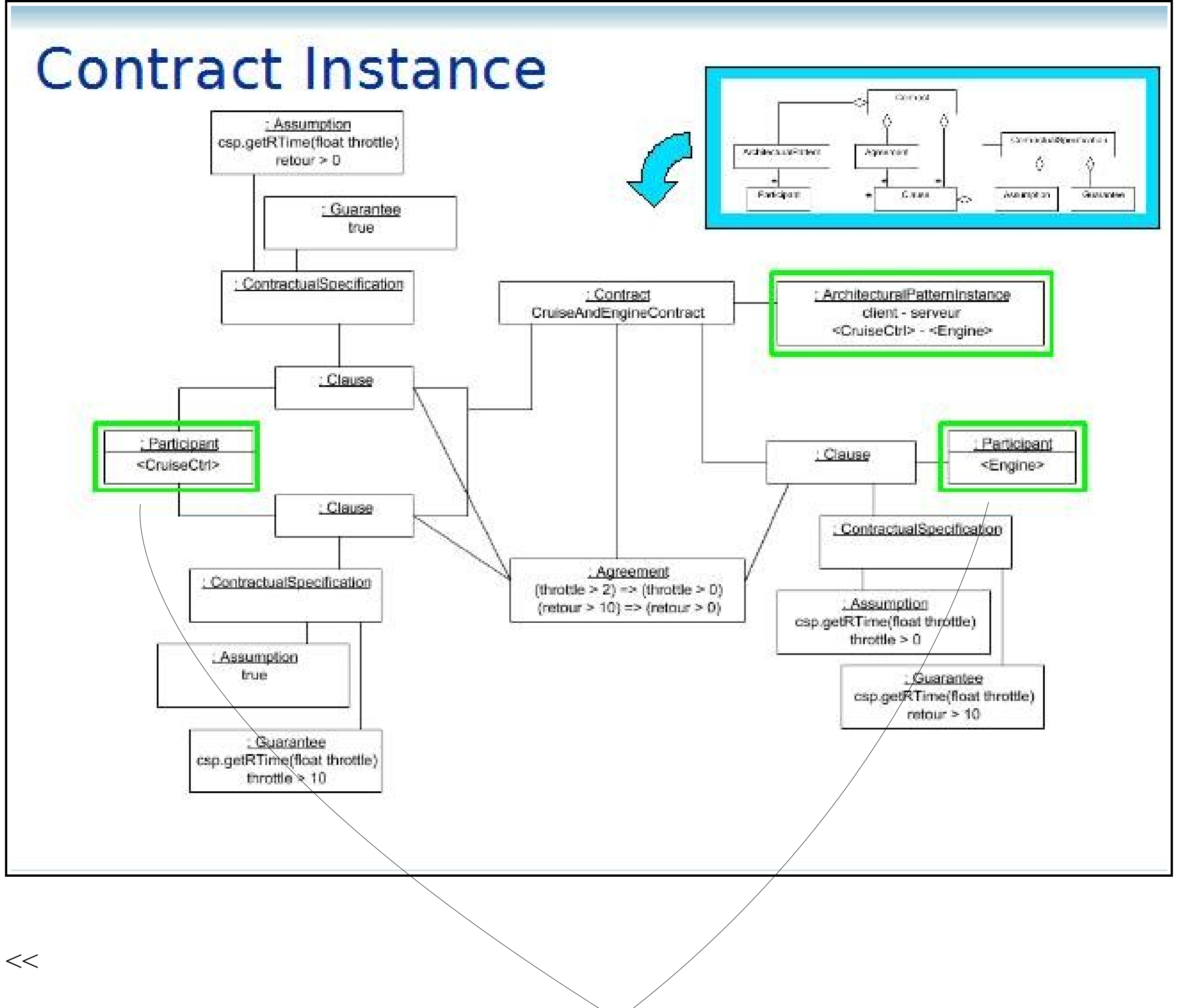
>>



<<

the contract is built from the architectural pattern

>>



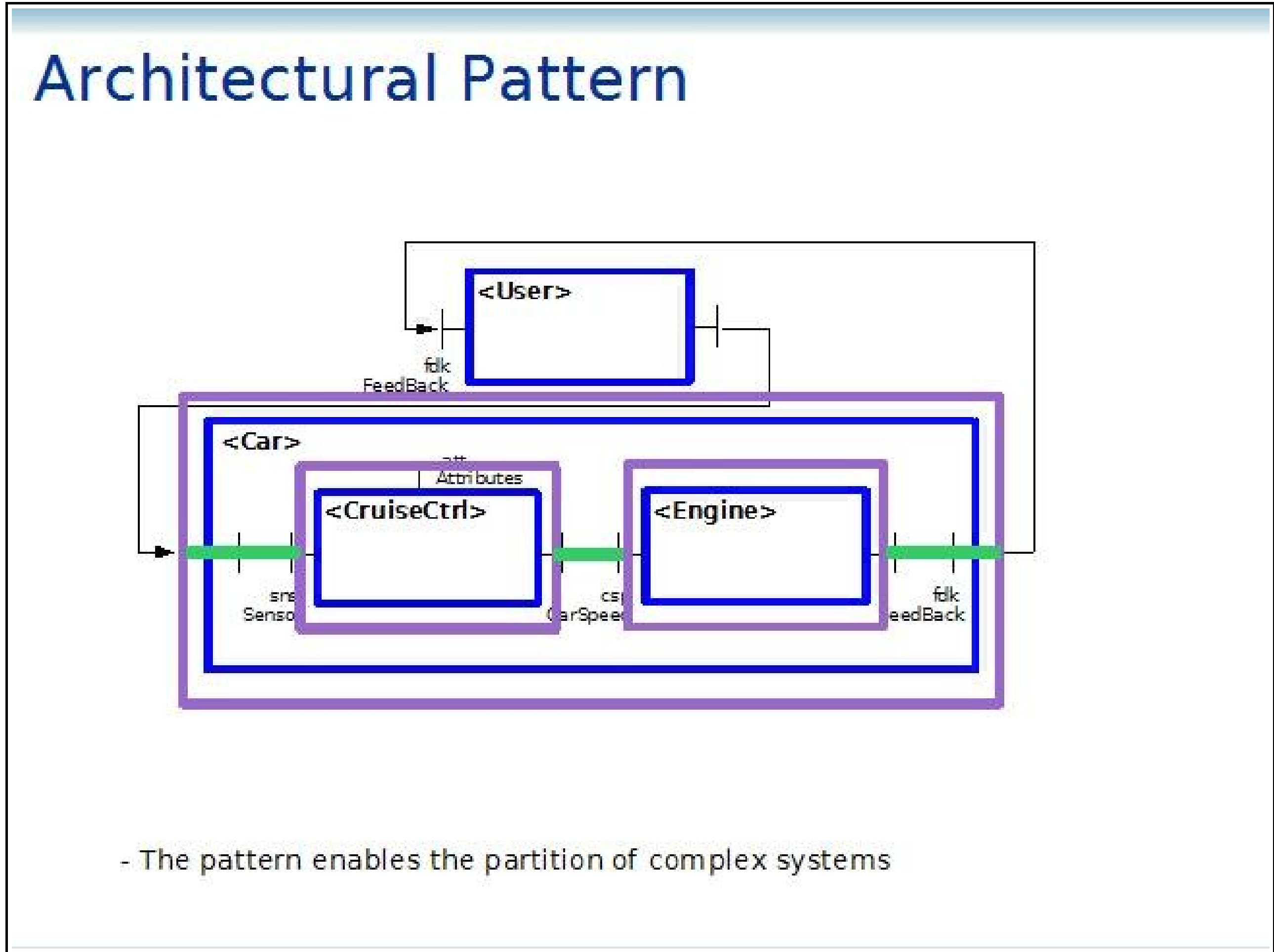
<<

the architectural pattern is used to distinguish the participants of the contract,
then the relation between the participants, pointed by the pattern, is used to retrieve the specifications that constrain what is exchanged on it.

From these specifications are built the clauses, and the agreement.

An important point with the agreement is that each of its predicates applies to an exchange, and then to a relation of the architectural pattern. Here we have only one relation, but in the case of a pattern with several relations, the interest is that with a detailed knowledge of the agreement failure, the reason of incompatibility can be precisely pointed in the architecture.

>>

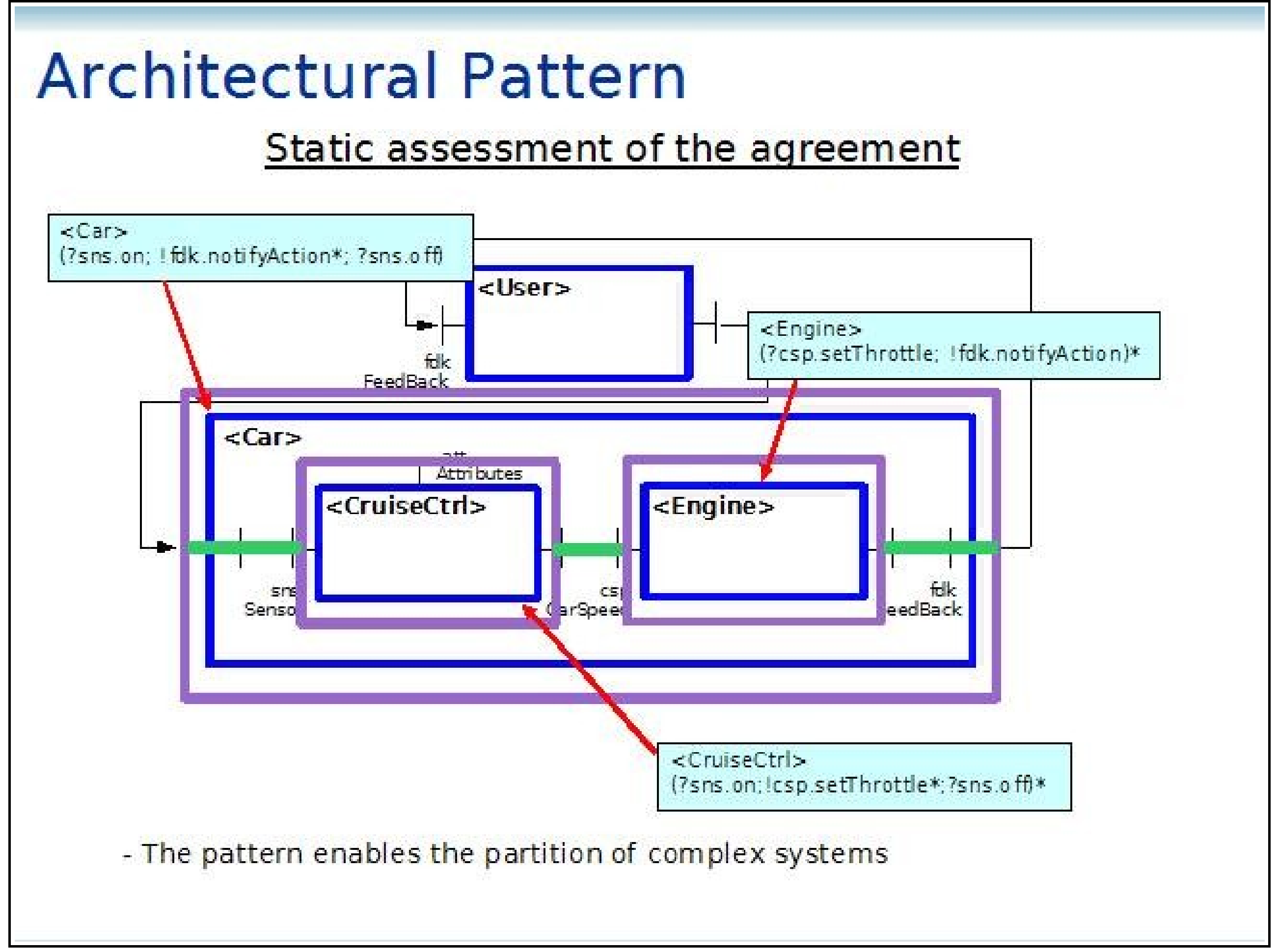


<<

I will now consider a very slightly more complex architectural pattern : it gathers three components linked by three relations : a composite including two subcomponents, two by two interconnected.

This pattern applies to the cruise control system, so a contract relying on this pattern can be built.

>>



<<

For this contract I will consider the behavior protocol formalism, and the specifications in this formalism I presented with the cruise system first description.

I'll just remind you that each specification denote the possible sequences of incoming and outgoing events for each component. “?” denotes an awaited event, “!” denotes an emitted event, the rest of the syntax is very near from the regular expression one.

In the case of the behavior protocol, the compatibility between the specifications can be assessed statically with the help of a model checker provided by the formalism designers.

So the agreement of the contract which participants are the Car, the CruiseCtrl and the Engine can be statically checked.

>>

Architectural Pattern

Static assessment of the agreement

- The pattern enables the partition of complex systems

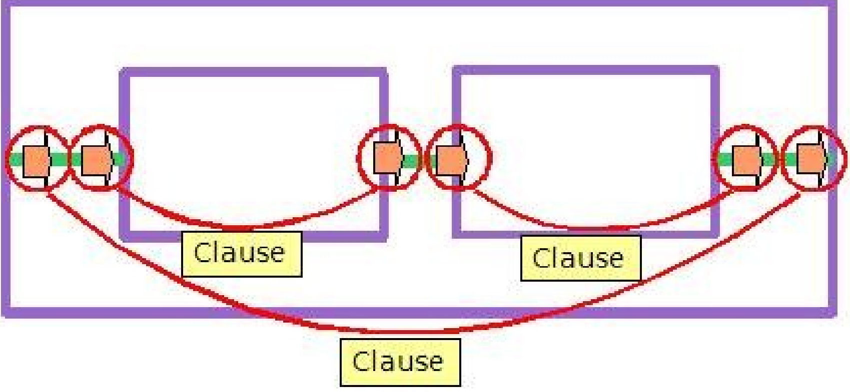
<<

so the contract system will consider the relations pointed by the architectural pattern, then discover the message they carry and get through the formalism adapter the specifications that constrain them for each of the participant of the contract.

>>

Architectural Pattern

Static assessment of the agreement



- The pattern enables the partition of complex systems

<<

On the base of the specifications the system retrieve for each of its participant, it builds the clauses of the contract : for each participant each clause constrains its emitted messages and received ones : the guarantee and the assumption, with in addition the fact that if awaited messages don't come then the component is no more responsible for emitted ones.

We can see that these clauses can be easily deduced from the behavior protocol specifications of the components, as each behavior protocol specification constrains the order of received and emitted messages, and then can be interpreted as : some messages are to be emitted in a certain order, some are to be received in a certain order :
in addition one can see an emitted message as causally linked with a previously received one, so the behavior protocol can be interpreted as an assume guarantee specification : indeed if an awaited message don't come then the conformity with the behavior protocol is broken and the behavior of the component is no more described by the protocol.

In the case of behavior protocols the satisfaction of the clauses can be runtime checked by observing the exchanged messages or if the participants code is available it can statically proved.

>>

Architectural Pattern

Static assessment of the agreement

Agreement

Clause

Clause

Clause

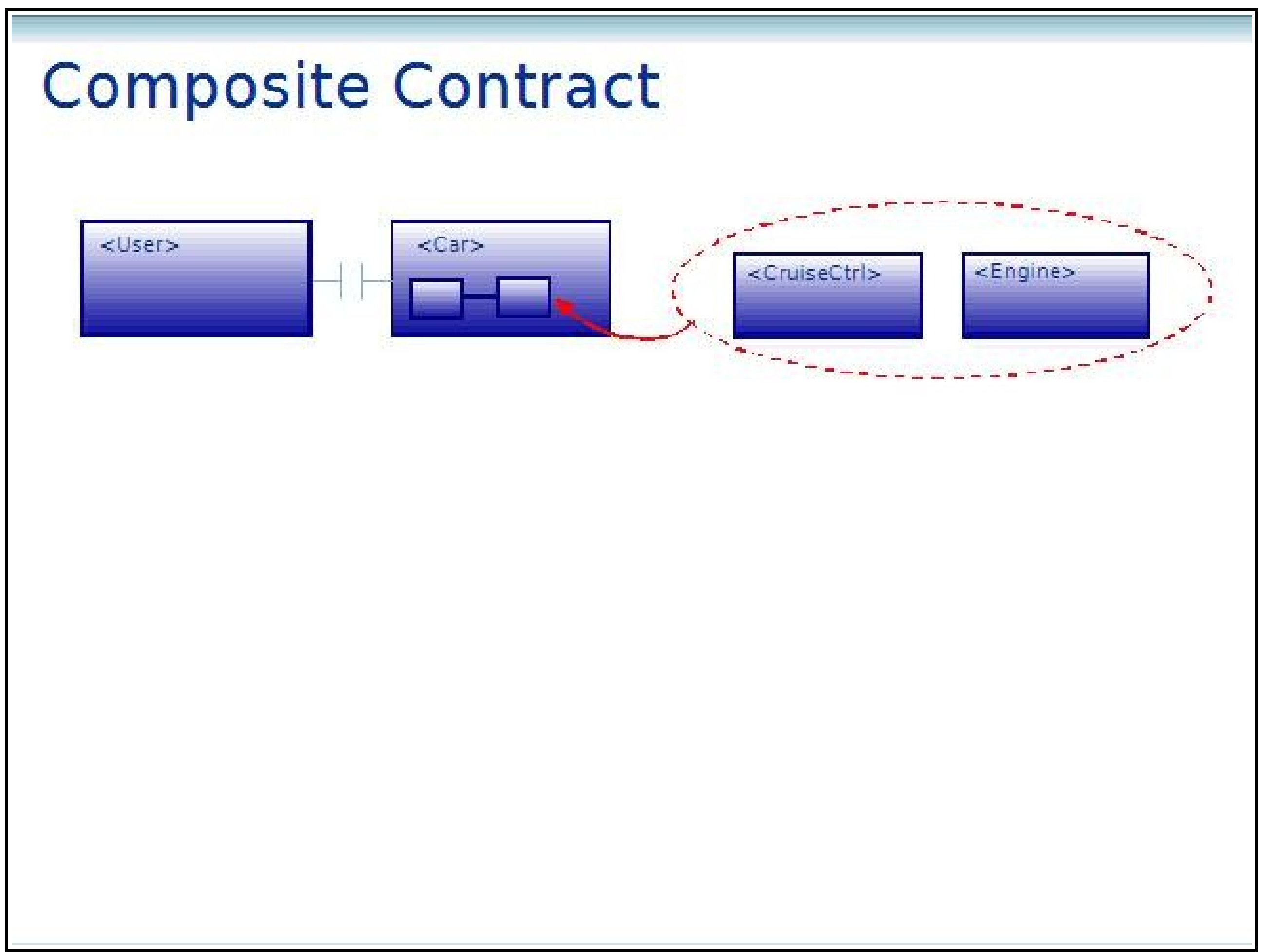
- The pattern enables the partition of complex systems

<<

The agreement of the contract reifies the compatibility of the clauses : that is : on each link, each emitted message by a component (according to its clause) corresponds to a message awaited by one other component (according to its clause).

As I said this can be model checked in the case of the behavior protocols.

>>



<<

I'm going now to show you how a contract can depend on an other contract :

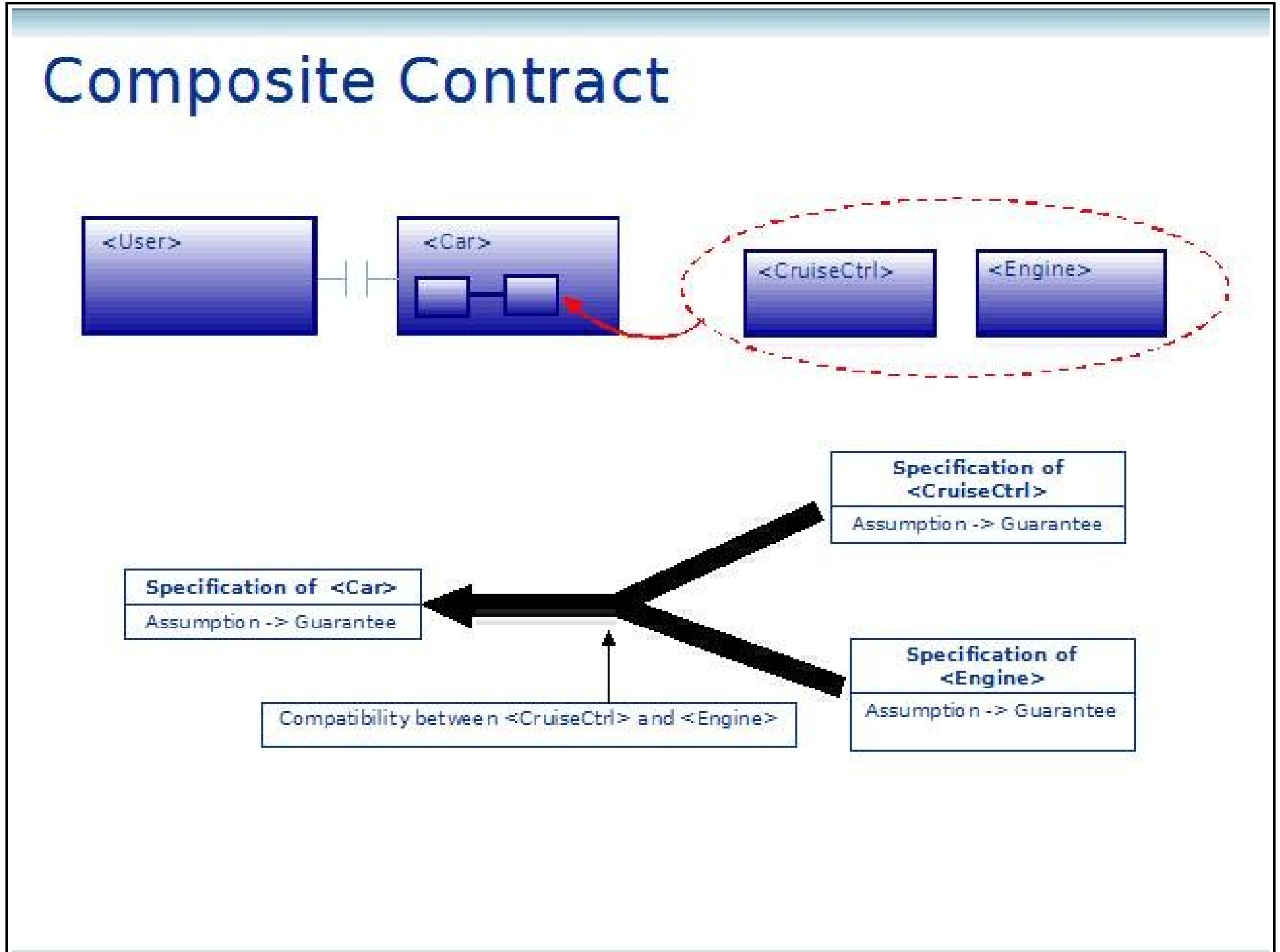
The <Car> components contains the <CruiseCtrl> and <Engine> ones.

There are several formal results about assume-guarantee formalisms that show how to deduce mechanically the assume-guarantee specification of a composite component from the assume-guarantee specifications of its subcomponents.

I have looked at 4 or 5 theorems, and seen that each time the specification of the composite given by the theorem was valid as long as a certain compatibility expression between the subcomponents was satisfied.

But in my approach of the agreement, I did only specified that it has to hold a compatibility expression of the participants of the contract : so it is possible to consider a contract, between the subcomponents, whose agreement expression is the one of the theorem.

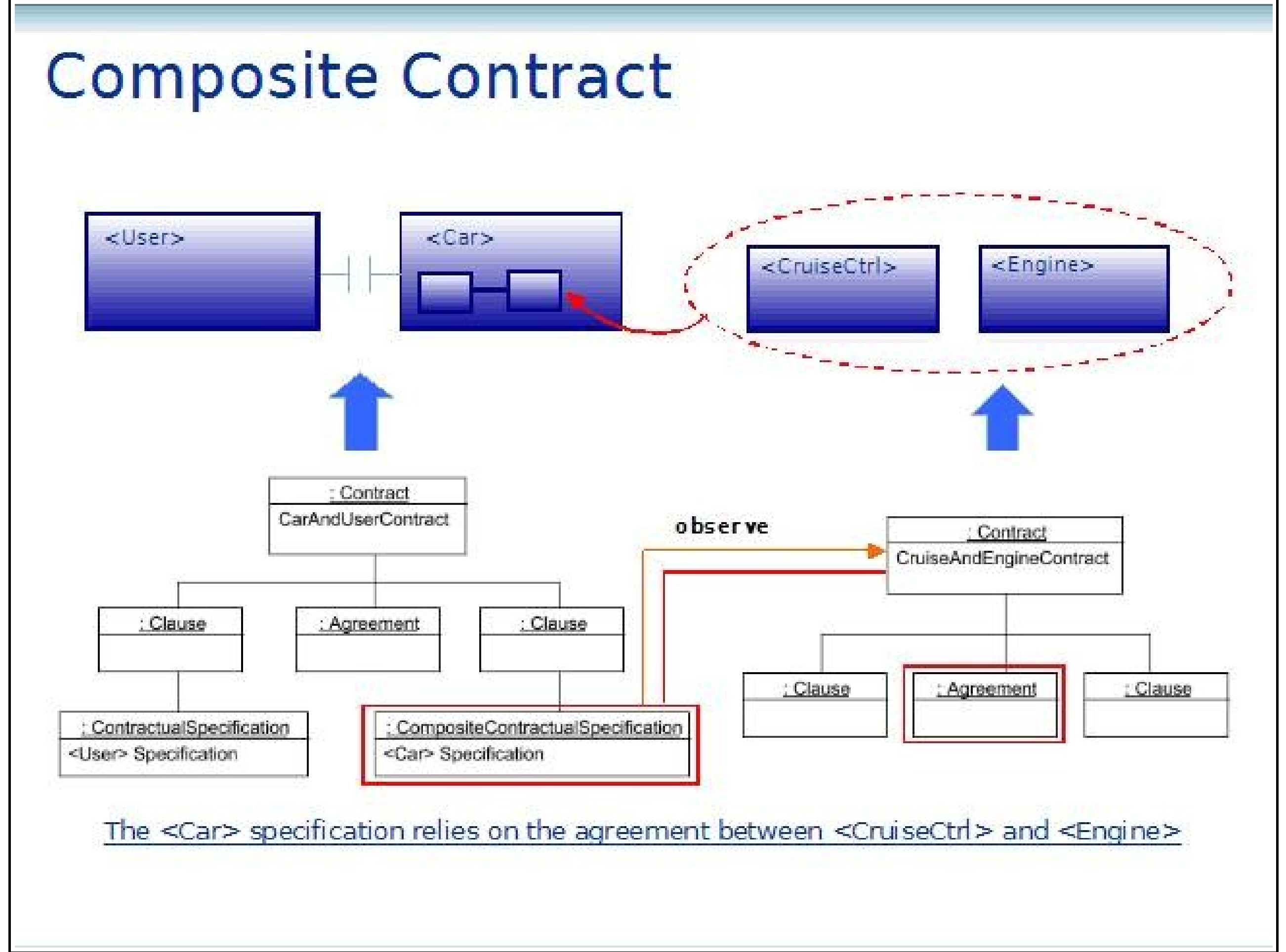
>>



<<

In this case the specification expression of the composite component (Car) is mechanically deduced from the ones of the subcomponents (Engine and CruiseCtrl), and is valid as long as the agreement of the contract between the subcomponents holds.

>>



<<

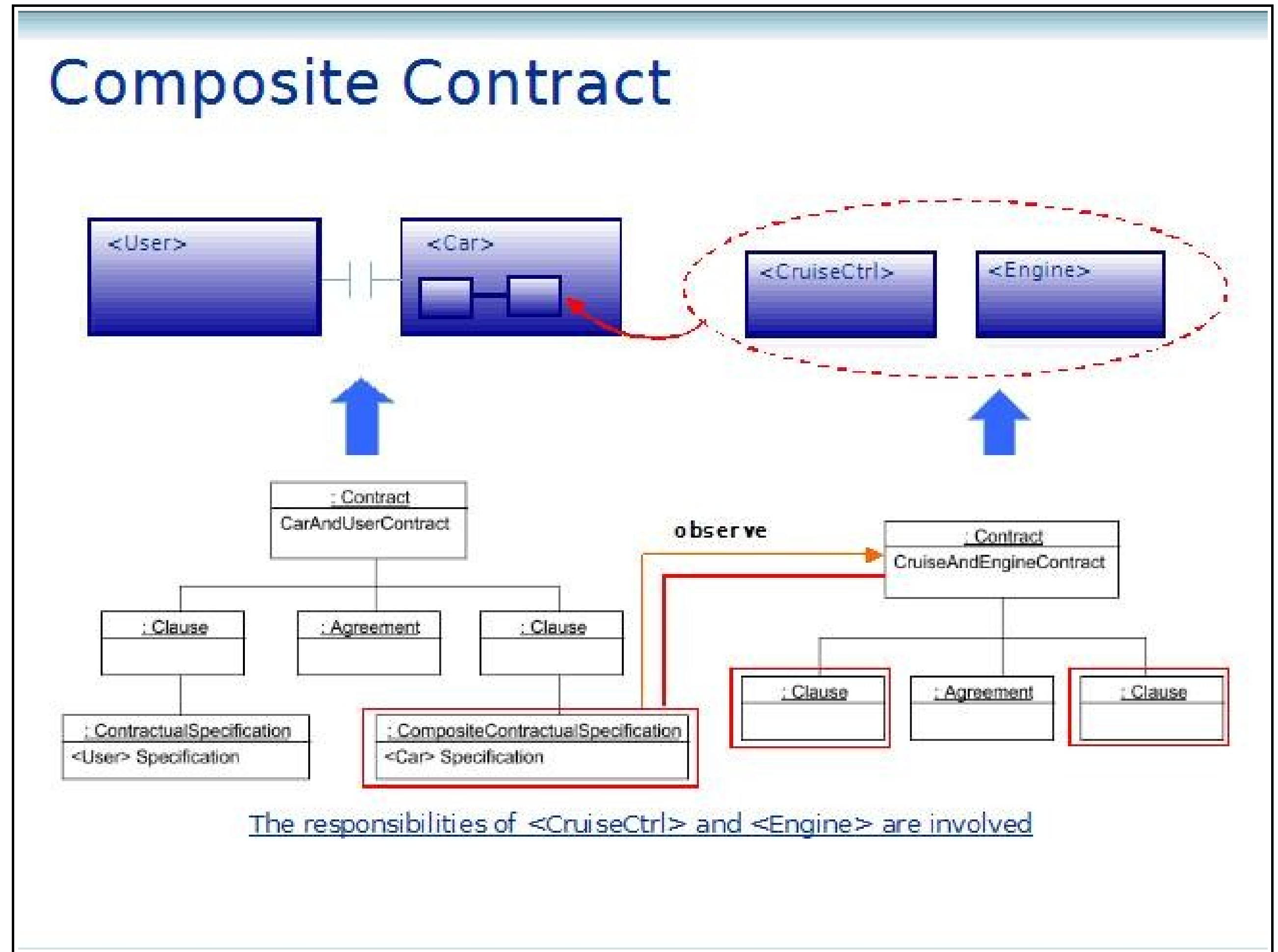
Then the Car specification satisfaction is a "CompositeContractSpecification" because it depends on :

- the conformity of Car to the specification, as a classical specification,
- the agreement between CruiseCtrl and Engine, for the validity of its expression,

the contract between Car and User can fail because :

- Car doesn't respect its specification
- the Car specification expression is no longer valid

>>



<<

An other interesting point in using a theorem enabling the mechanical deduction of the Car specification from the CruiseCtrl and Engine ones, is that many of these theorems show that :

if the compatibility expression, the agreement, between Engine and CruiseCtrl (that is the subcomponents) is satisfied, then :

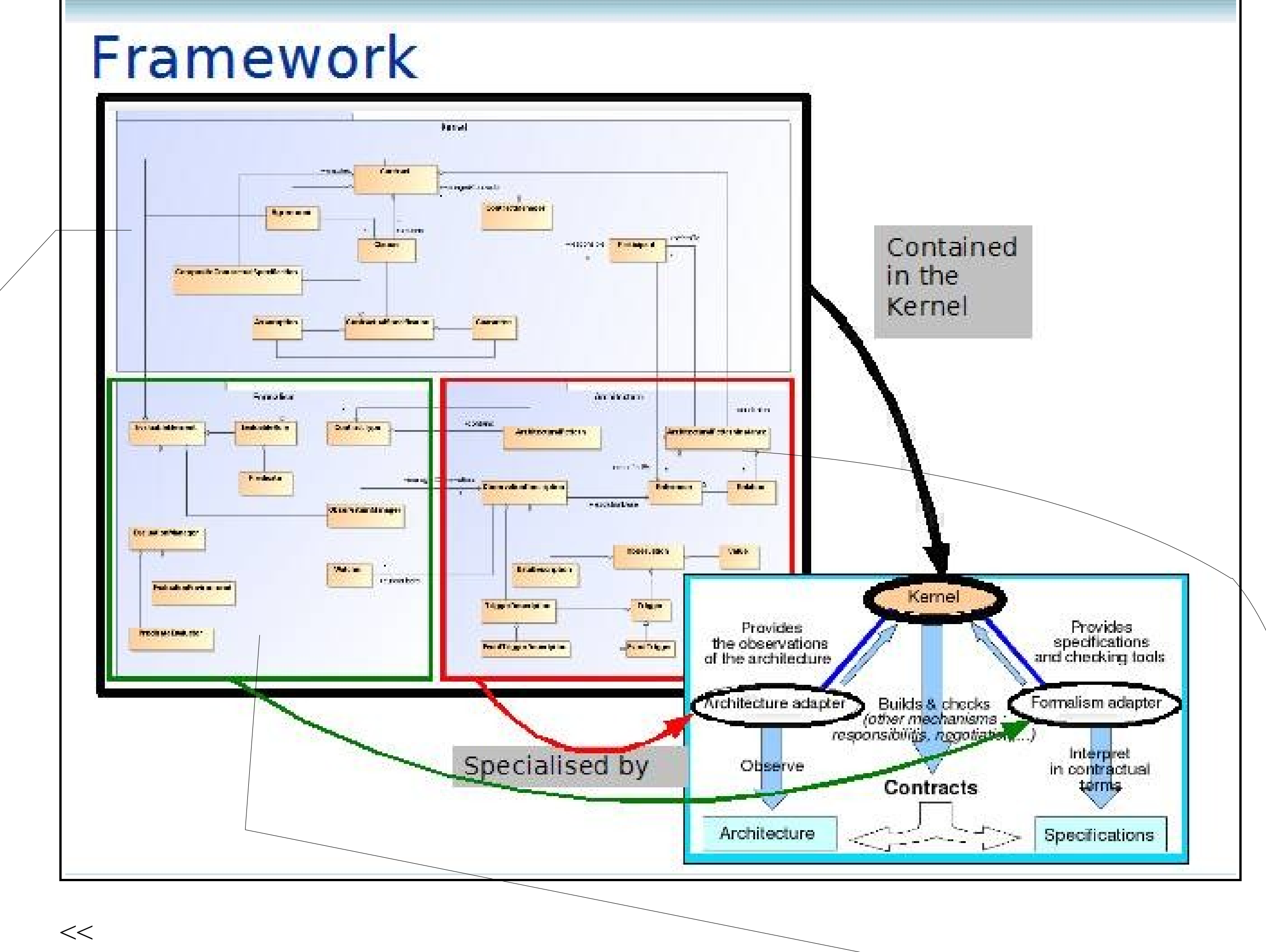
if the Car does not conform to its specification that is because one of the subcomponents, CruiseCtrl or Engine, does not conform to its specification.

So it is formally proved in this case that the responsibility of a conformity failure at a given level of architecture can be searched at its underneath level.

>>

Plan

- Introduction
- Existing works
- Contribution
- Implementation
  - Framework,
  - Architecture and observations modeling,
  - Modeling of conformity, compatibility, responsibility
- Validation
- Conclusion



<<

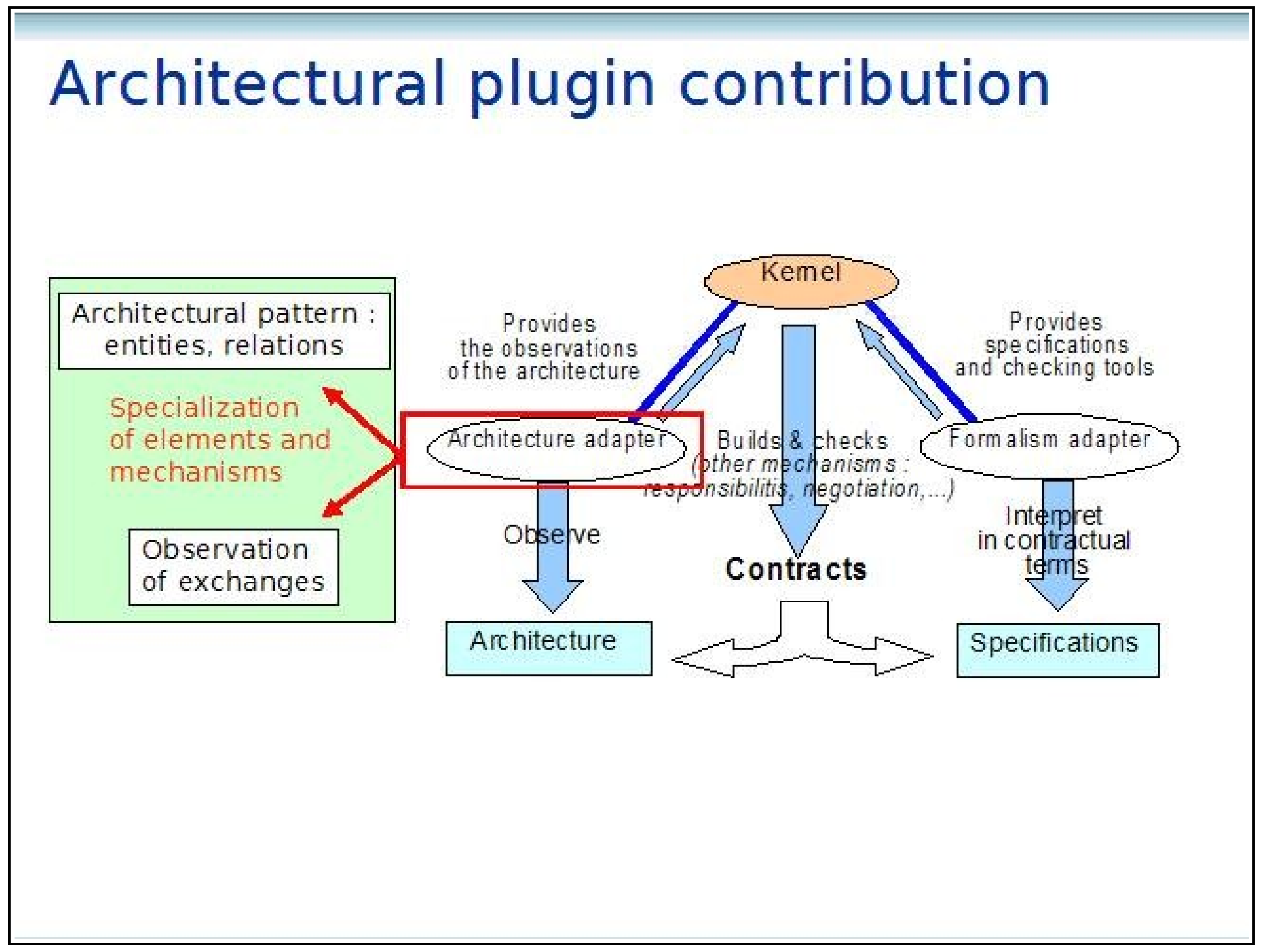
The kernel of the framework is divided in three parts :

the first contains the main elements of contract modeling : Contract, Clause, Agreement ...

the second contains the generic elements for modeling the specifications, specialized by the formalism adapter,

the third contains generic elements for modeling the architecture modeling, specialized by the architecture adapter,

>>

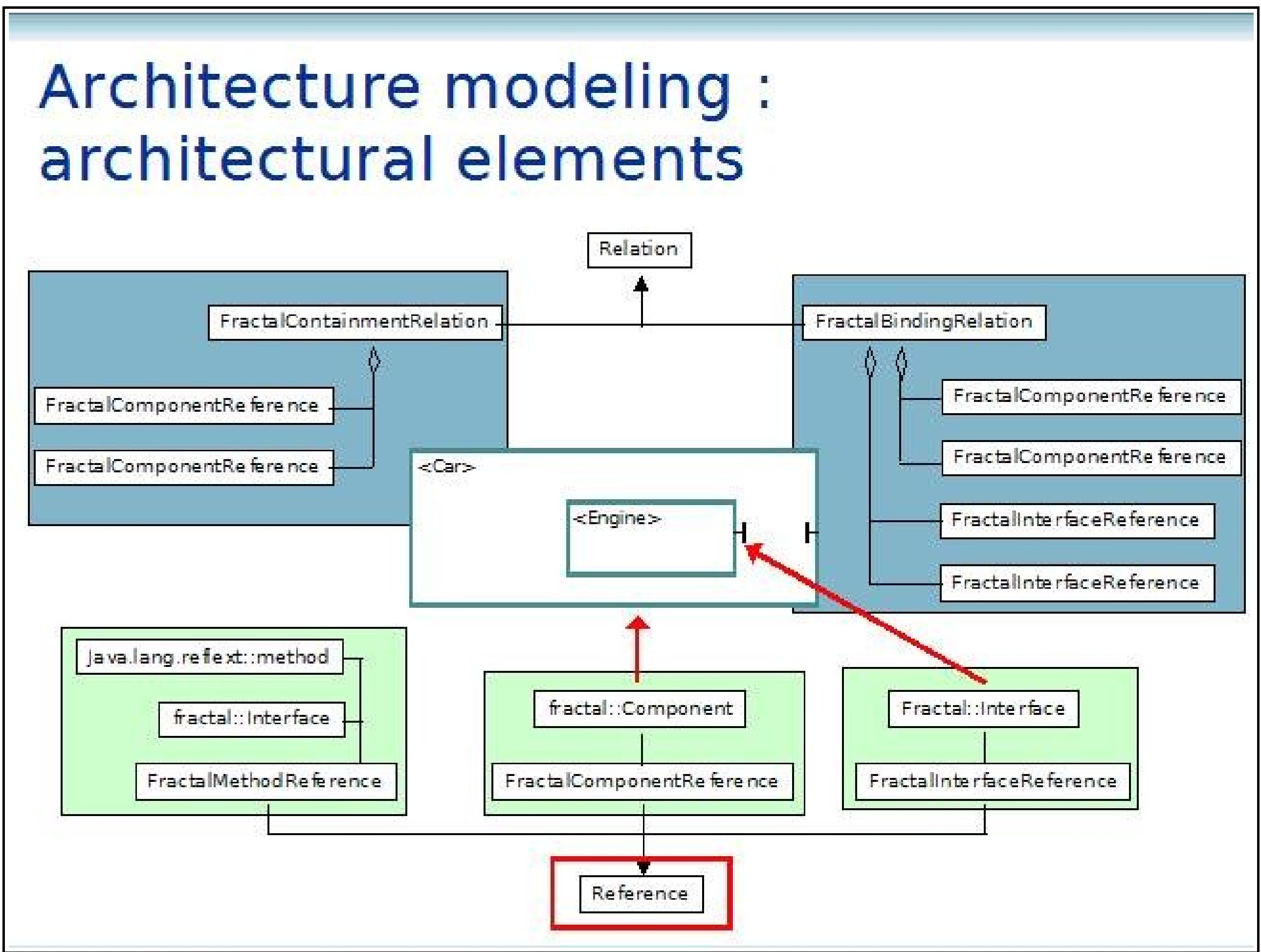


<<

The architectural adapter (or plugin for its implementation) provides the contract system with :

- the specialization of kernel elements for a given concrete architecture : architectural entities and relations are implemented to describe the elements of a concrete architecture,
- the specialization of kernel generic mechanisms for the concrete architecture : for example : the navigation in the architecture, the observation of the behavior of the system, etc

>>



<<

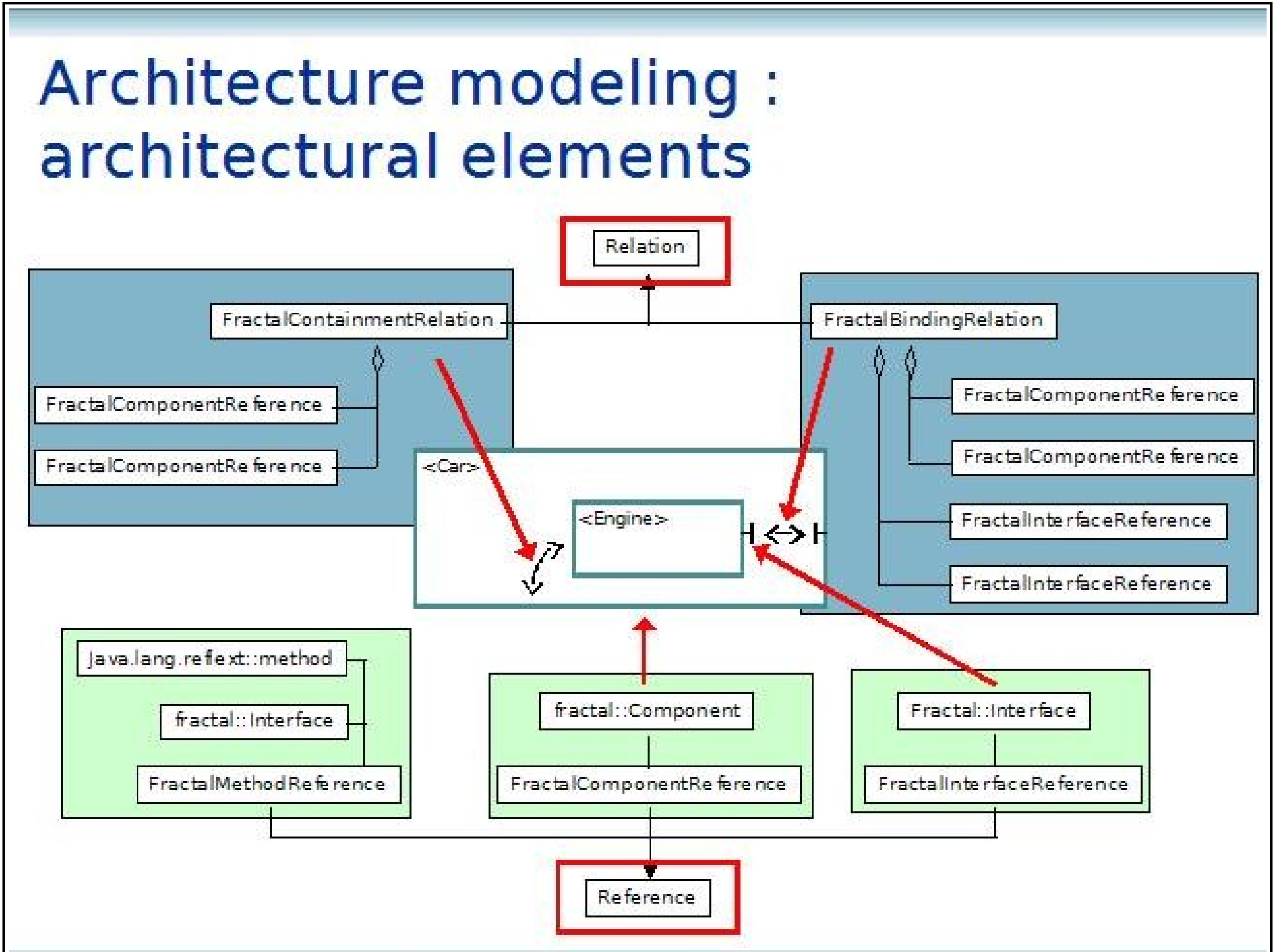
The first thing the contract system needs is elements to model the architecture it should apply to. Then the architecture adapter provides it the specialization of Reference and Relation :

here I show a specialization of Reference for the different elements of the Fractal component model :

- the components
- the interfaces of the components
- the methods of the interfaces

...

>>

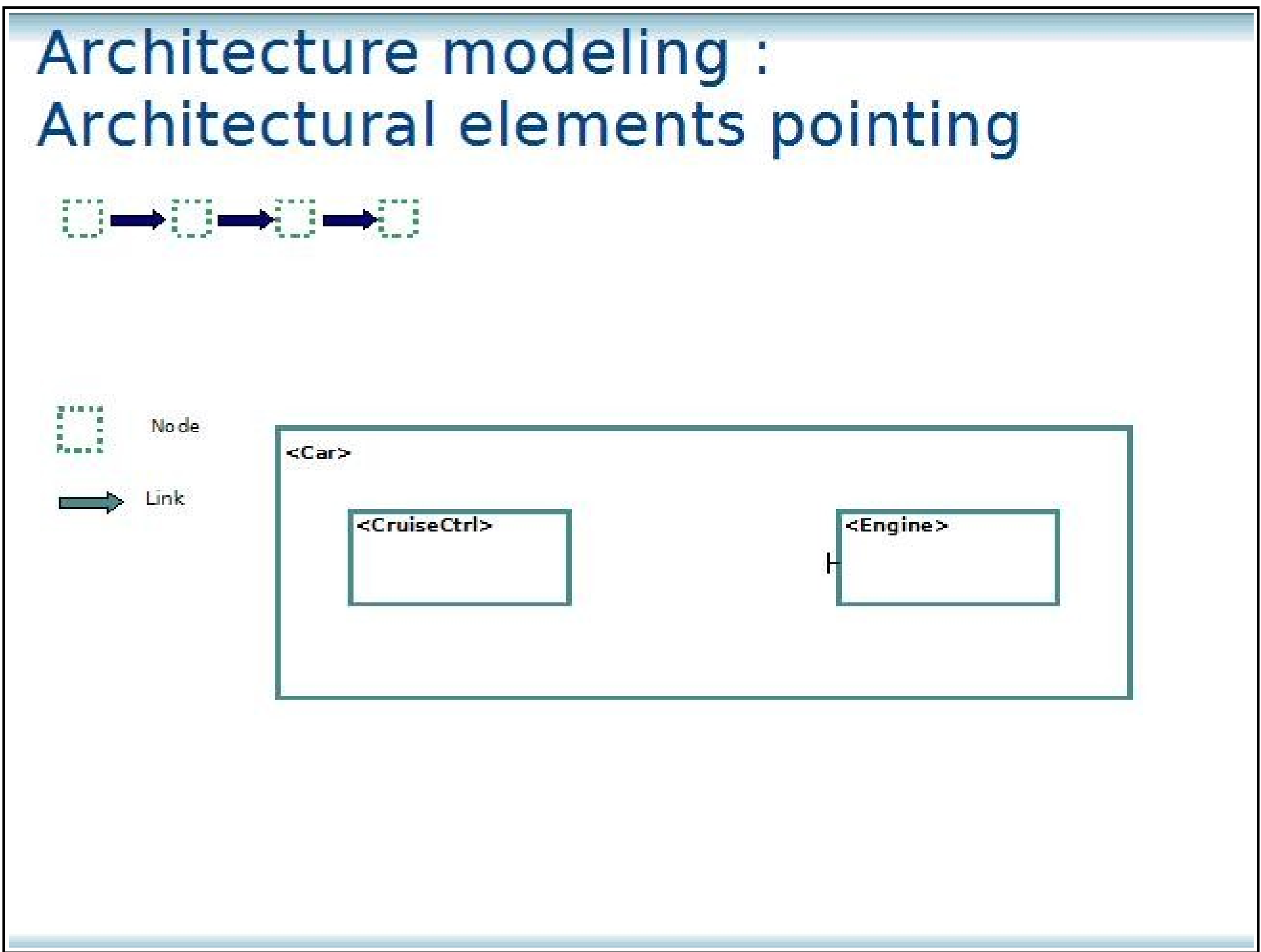


<<

Here is the specialization of the Relation class for the Fractal model :

- a relation of containment between a composite and a subcomponent
- a relation of binding between two interfaces

>>



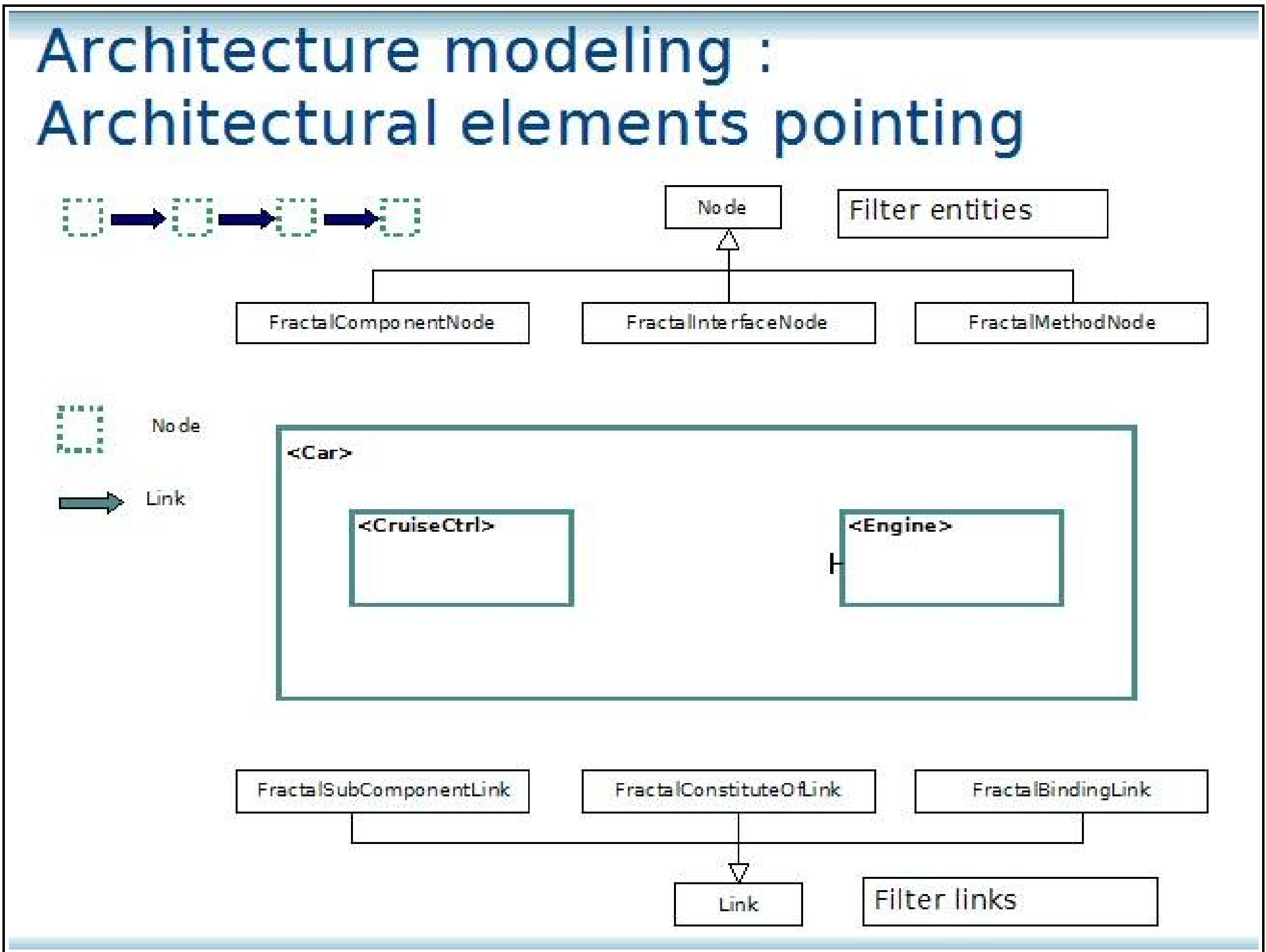
<<

An other thing very important to the contract system is the ability to point at an element of the architecture. To do so it defines a very classical and generic mechanism of path.

A path is made of a sequence alternating nodes and links :

- a node filter the elements of the architecture, among the ones pointed by its preceding link,
- a link filter the relations of the architecture, among the one starting from the elements pointed by its preceding node,

>>

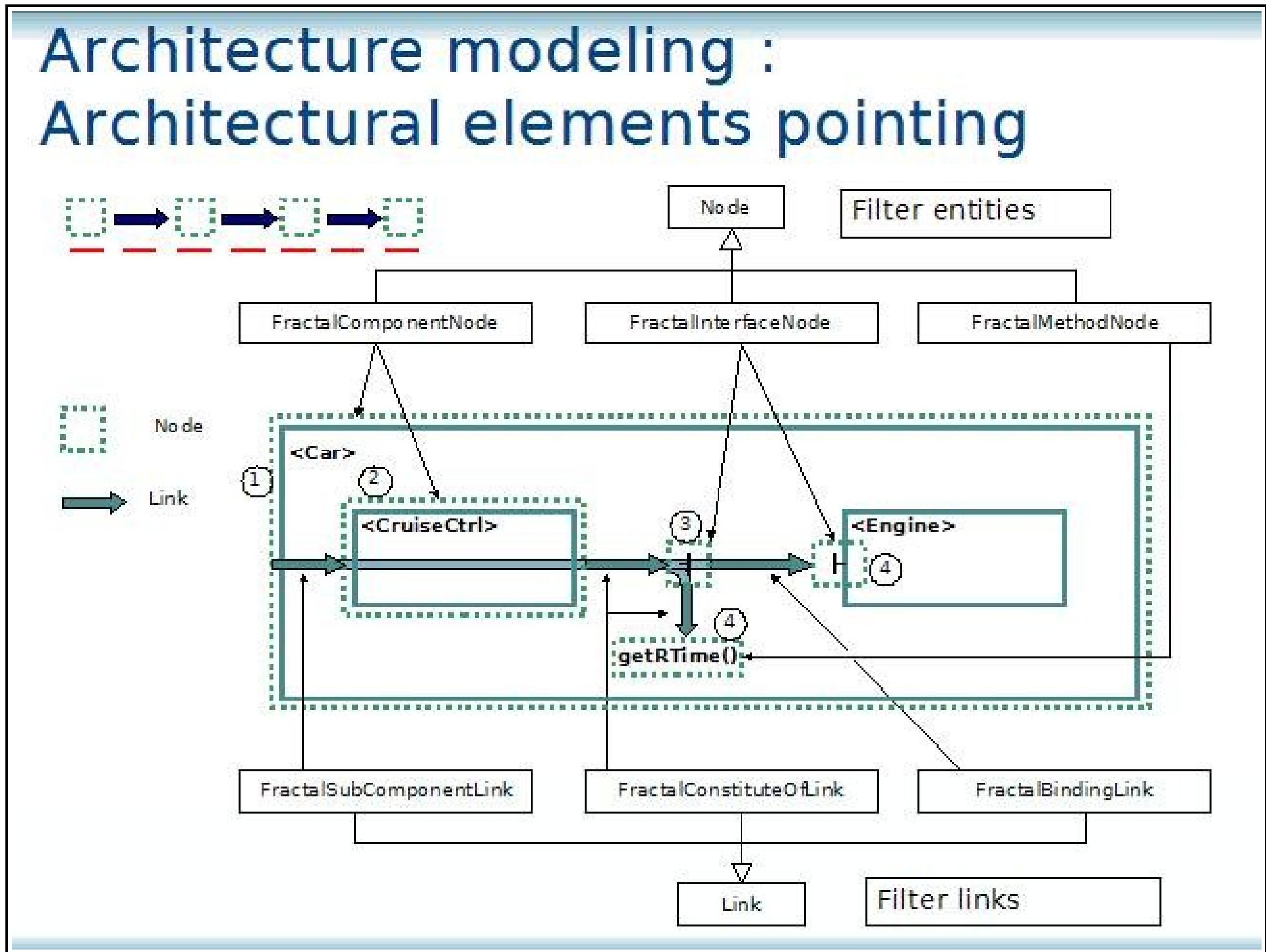


<<

I give here an example of specialization of the Node and the Link for the Fractal component model :

- a node : can point at a component (FractalComponentNode), an interface (FractalInterfaceNode), a method (FractalMethodNode)
- a link : can refer to composite-subcomponent relation (FractalSubComponentLink), to a component-interface or interface-method relation (FractalConstitueOfLink), a relation between two connected interfaces (FractalBindingLink)

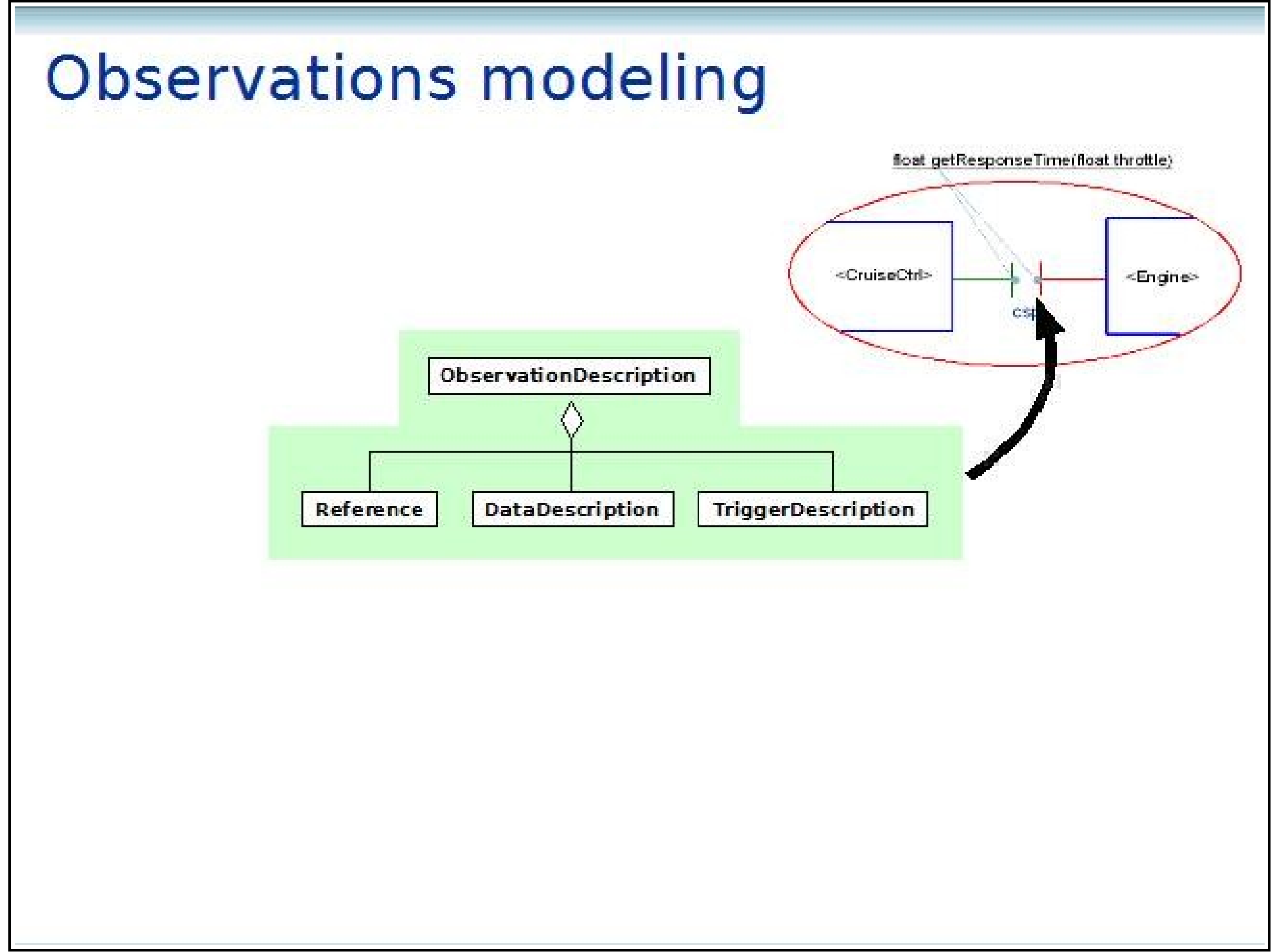
>>



<<

This is an example of two paths : both starting at the Car component and for one finishing on an interface of Engine component, for the other on the method getRTime of an interface of the CruiseCtrl.

>>



<<

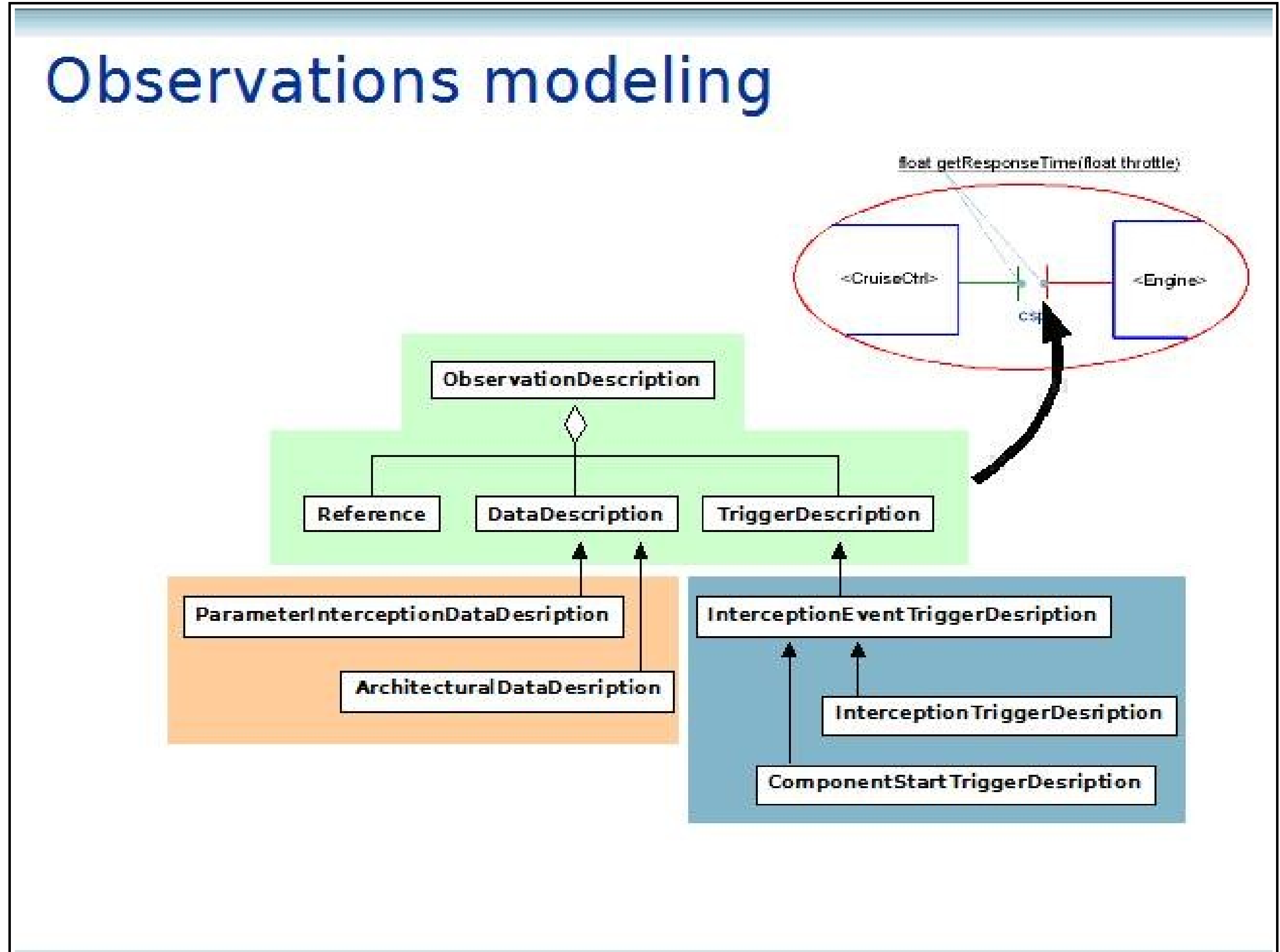
The last generic mechanism the architecture adapter specializes is the observation one.

I take as an illustration the observation of the getResponseTime method of the interface csp between CruiseCtrl and Engine components.

If I want to observe this method, I will have to specialize the generic observation which is first based on its description : ObservationDescription. This class describes the observation the system will have to make. It contains ;

- the description of the data that is to be retrieved : DataDescription
- the description of the instant at which the data should be retrieved : TriggerDescription
- a reference to an element of the system from which resolve the paths potentially contained in the data and instant descriptions,

>>



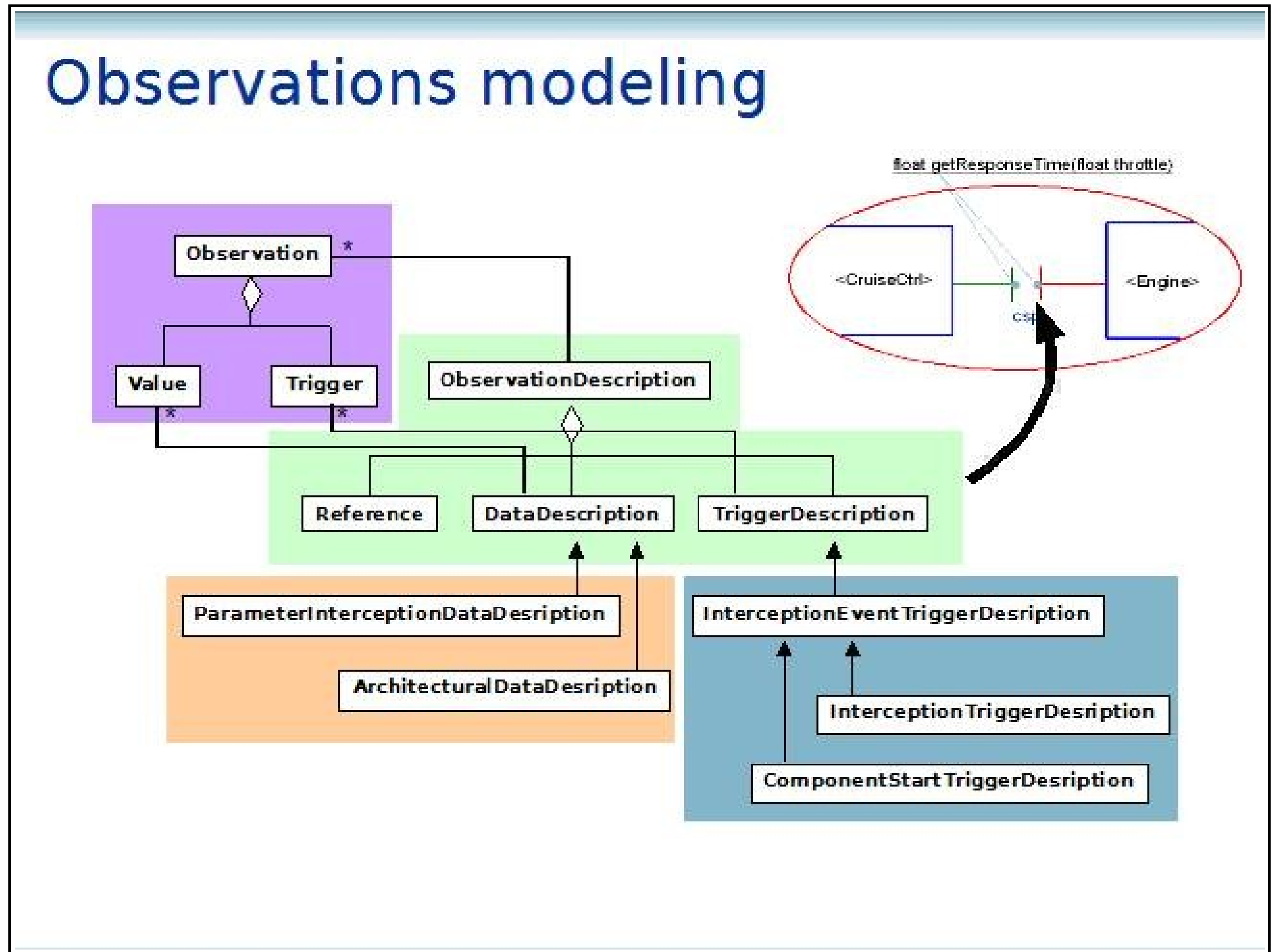
<<

Here some specializations of the observation description for the Fractal component model :

- the ParameterInterceptionDataDescription :describes a parameter value of the method observed
- the ArchitecturalDataDescription : describes an architectural data to be read on the system configuration, e.g. the number of subcomponents of a composite etc
- the InterceptionTriggerDescription : describes the execution interception on which read the parameter

....

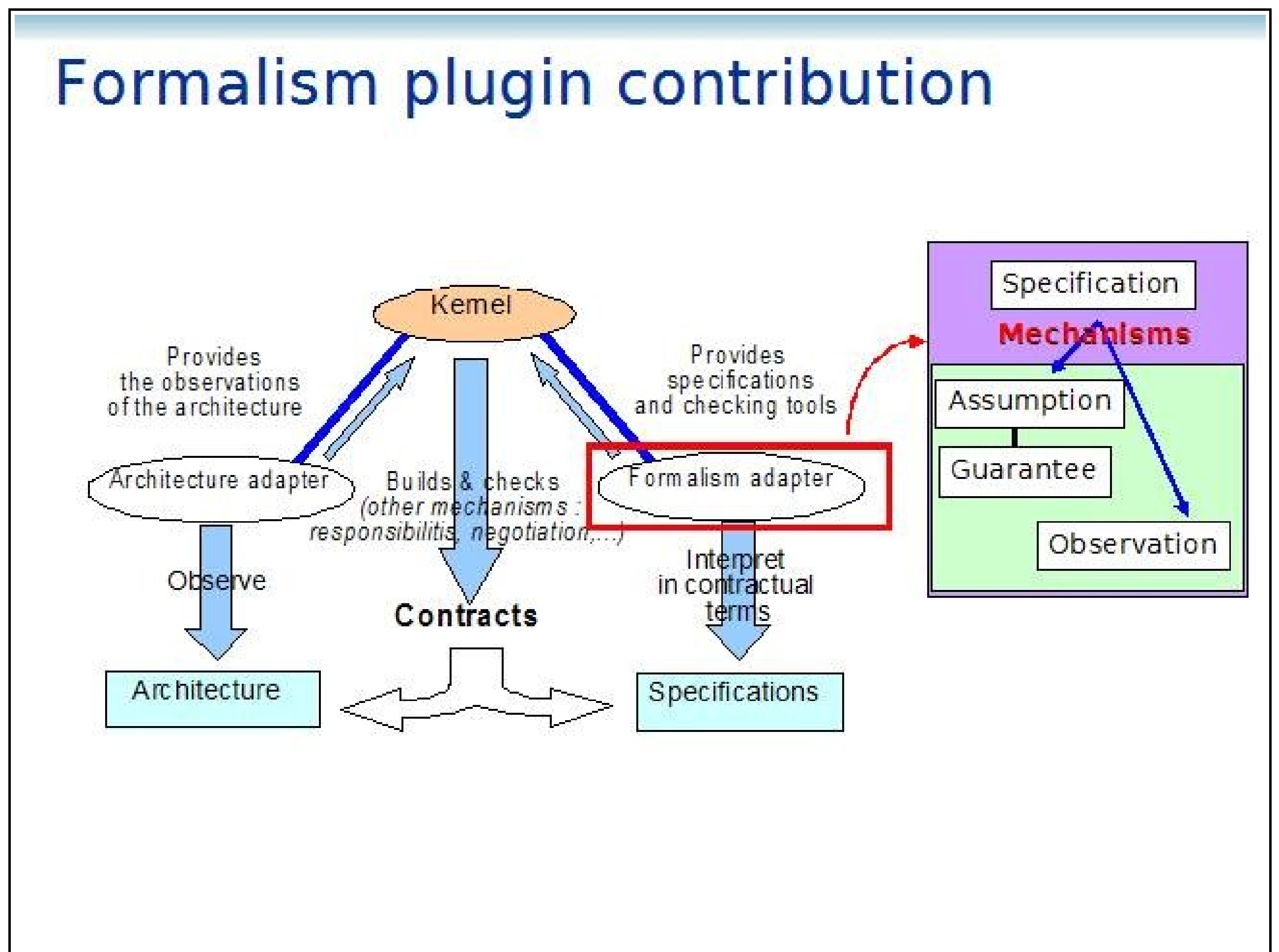
>>



<<

So the observation mechanism specialized by the architecture adapter then provides, for a given ObservationDescription, an Observation containing the value of the data, and the concrete values concerning the triggering of the observation.

>>

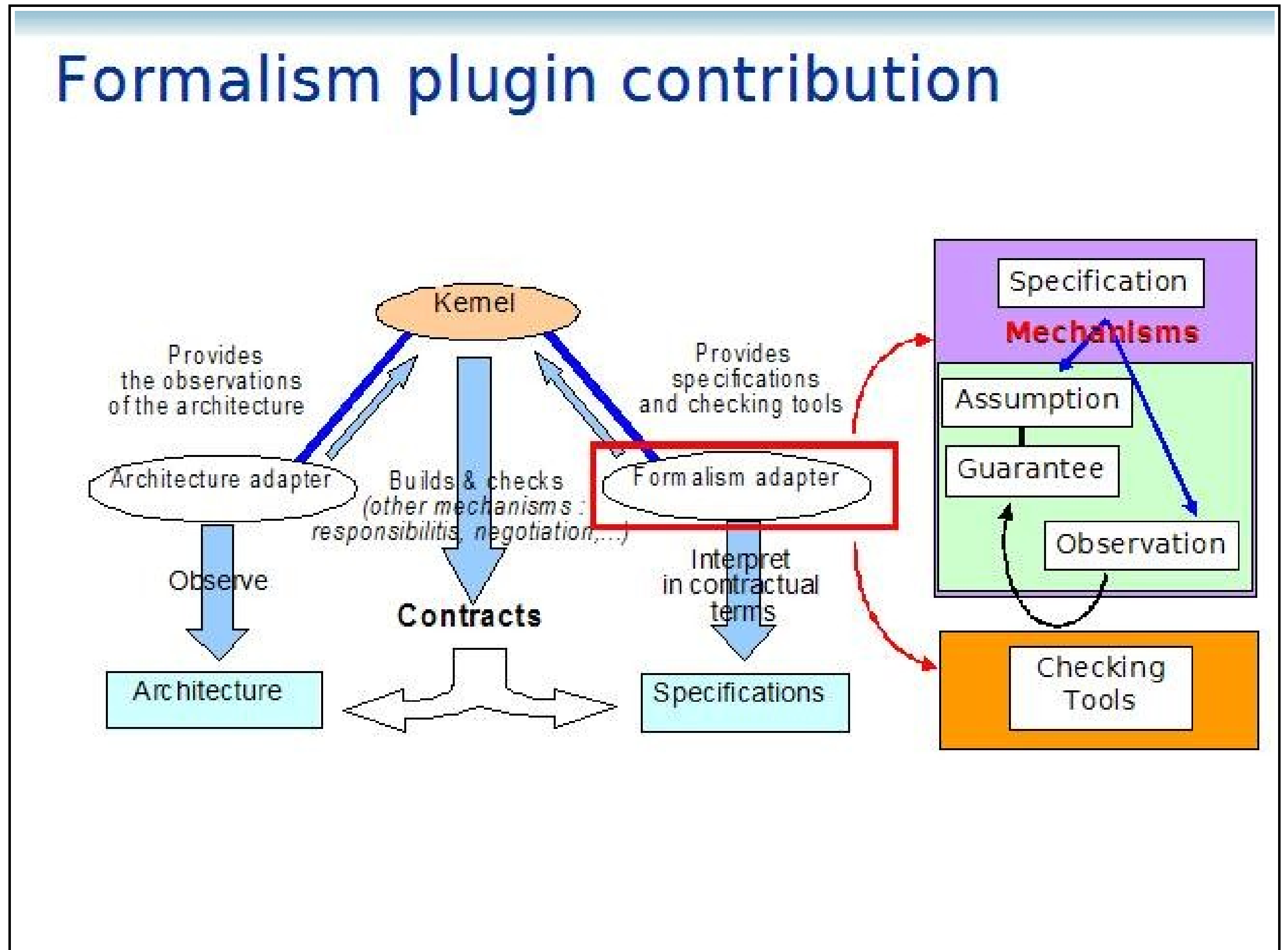


<<

The formalism adapter (plugin for its implementation) provides the contract system with :

- the specialization of mechanisms :
  - that translate the specifications into assume-guarantee expressions,
  - that extract the observations, potentially necessary for the evaluation of these expressions, from the specifications,
  - that build the agreement expression on the base of the specifications ones,
- the architectural patterns on which collaborations constrained by this formalisms can be contractualized,

>>



<<

- the formalism adapter also provides : the checking tools :
  - for the specifications conformity
  - for the specifications compatibility

>>

Specifications

Mechanically modeled (API) :

```
Rule :
On < a component >
  Observe {
        (val : <some value>
         at : <some times>; )+
  }
  Verify <some properties>
```

Contractual specification :

```
Predicate : Rule("assumption") -> Rule ("guarantee")
```

<<

In order to handle specifications, the contract system defines a generic description of them. Each specification is considered as a rule.

I give here a literal representation of a specification through a rule, this is not a formalism to infer properties but only an alternative way from diagram to represent the rule.

Each rule is made of three blocks :

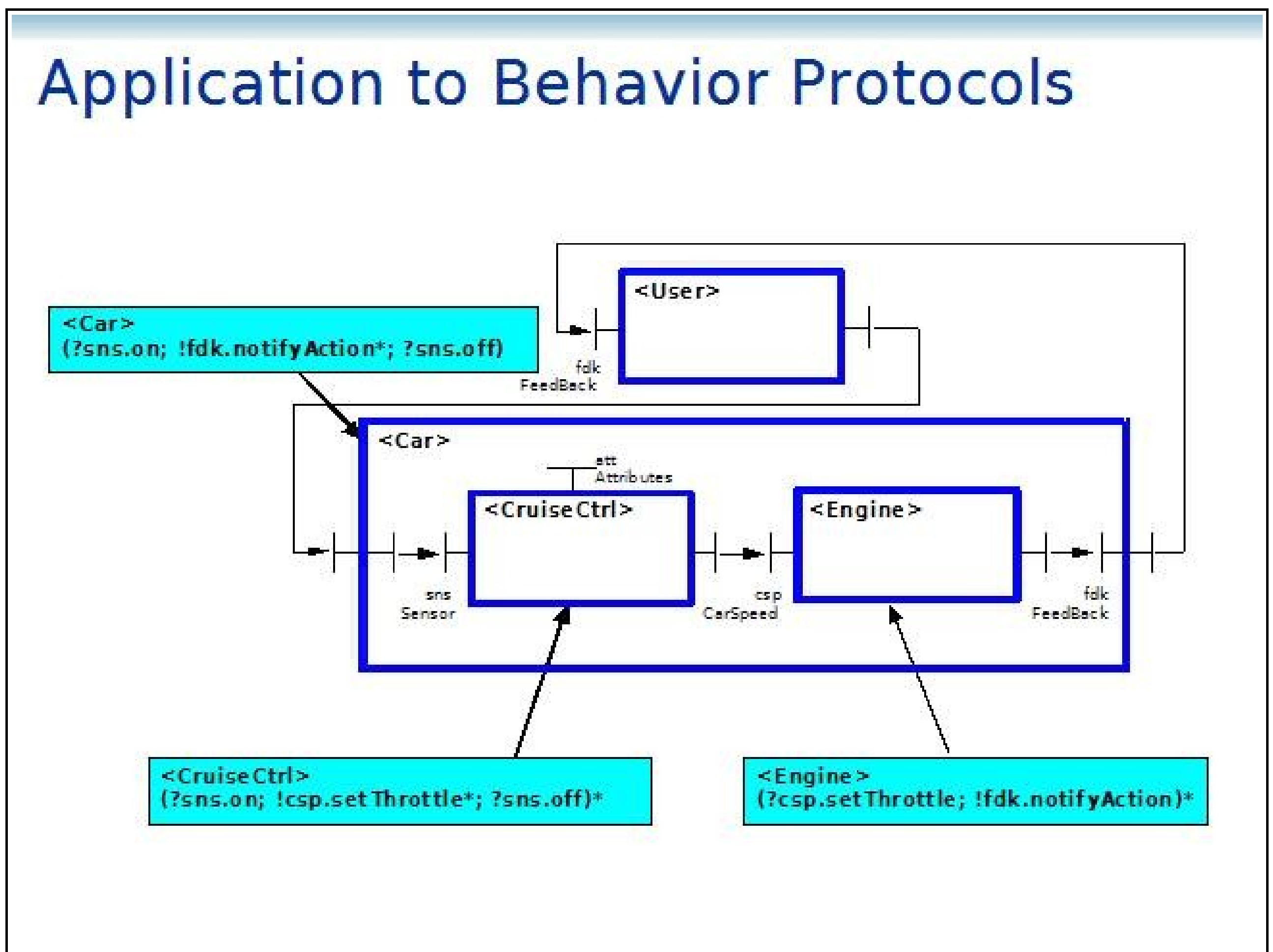
- the On block : that points at the architecture element which is constrained by the rule
- the Verify block : contains the constraint on this element
- the Observe block : contains the observations of the element that are constrained by the Verify block

With this system of rule, a contractual specification is represented as a predicate containing two rules :

- a assumption rule
- a guarantee rule

the predicate assessing that the guarantee rule is true as long as the assumption rule is true (the "as long as" is there of course a delicate point which has to be carefully handled in the use of the observations provided by the architecture adapter)

>>



<<

I'm going to illustrate how the formalism adapter models specifications and help to build the contract between the Car, the CruiseCtrl and the Engine, with the behavior protocols formalism.

>>

Resulting contract (1/2)

- Clauses based on Frame Protocols :

```
Contract :
  Participants : <Car>, <CruiseCtrl>, <Engine> ;
  ...
  Clause :
   responsible : <CruiseCtrl>
   assumption :
            On <CruiseCtrl>
            Observe : val : evt.* at entry sns.*;
            Verify : runtimeCheck(cruiseCtrlFP);
    guarantee :
            On <CruiseCtrl>
             Observe : val : evt.setThrottle at entry csp.*
             Verify : runtimeCheck(cruiseCtrlFP);
  ...
```

sns

csp

<CruiseCtrl>

cruiseCtrlFP = (?sns.on; !csp.setThrottle*; ?sns.off)*

<<

I give here a literal expression of the contract, as for the rule representing the specification, this is not a formalism meant to be use to deduce properties but simply a representation of the contract.

So we have a contract with three participants :

We are going to first see how the clauses are built on the base of the specifications.

I will consider a clause applying to the CruiseCtrl component. So this clause is deduced from the specification of this component.

>>

Resulting contract (1/2)

- Clauses based on Frame Protocols :

```
Contract :
  Participants : <Car>, <CruiseCtrl>, <Engine> ;
  ...
  Clause :
   responsible : <CruiseCtrl>
   assumption :
            On <CruiseCtrl>
            Observe : val : evt.* at entry sns.*;
            Verify : runtimeCheck(cruiseCtrlFP);
   guarantee :
            On <CruiseCtrl>
             Observe : val : evt.setThrottle at entry csp.*
             Verify : runtimeCheck(cruiseCtrlFP);
  ...
```

sns csp

<CruiseCtrl>

cruiseCtrlFP = (?sns.on;
!csp.setThrottle*; ?sns.off)*

<<

The assumption of the clause constrains what the component receives from its environment :

so it constrains messages entering through the server sns interface (in Fractal representation server interfaces are on the left of components),

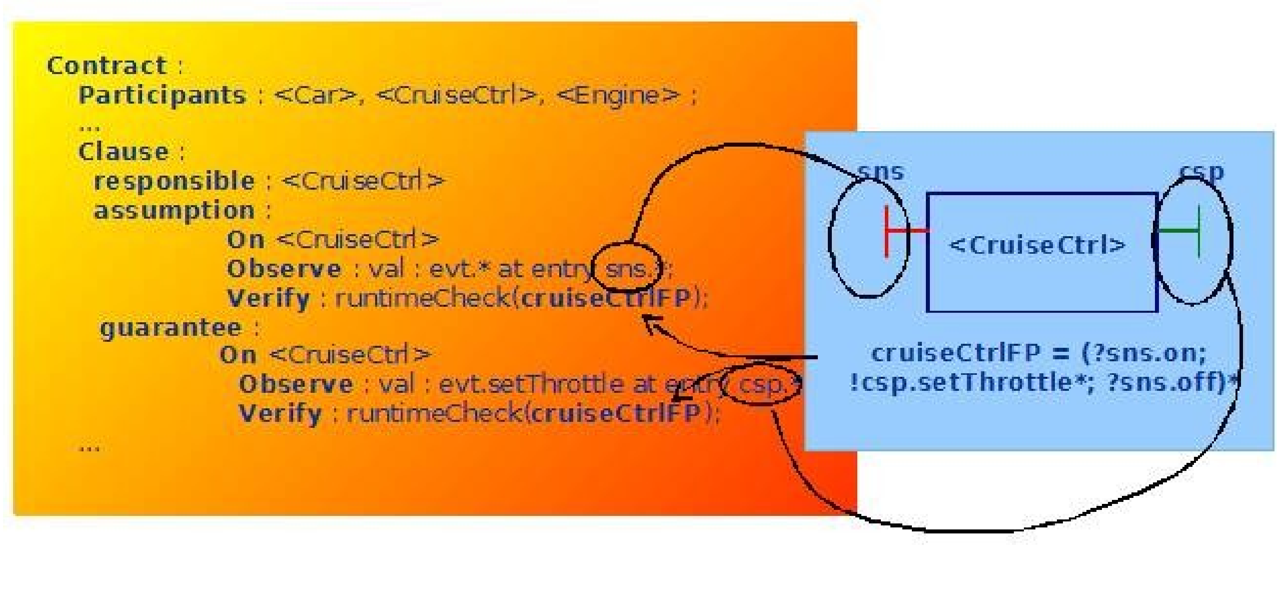
so the rule of the assumption will observe all the method of the sns interface : “sns.*”

and the messages have to income accordingly with the behavior protocol, so the Verify block runtime checks the protocol for these incoming messages on sns interface.

That gives the expression of the rule of the assumption.

If this runtime checking fails that means that an incoming message was received at a bad time accordingly to the behavior protocol, and that is the environment which is responsible for sending a message to the component at a bad time. Here is the meaning of the assumption.

>>



<<

as the guarantee of the clause :

it constrains what the component provides to its environment, so its outgoing message

so the guarantee rule constrains messages emitted on the client csp interface (in Fractal representation client interfaces are on the right of components) : it observes then all the methods of this interface “csp.*”

as for the assumption, emitted messages should be outgoing on the right times according to the behavior protocol, so the Verify block of the rule runtime checks the outgoing messages against the behavior protocol.

If this runtime checking fails, then that is that the message is not sent by the component on the right time. The component is then responsible. Here is the meaning of the guarantee.

Moreover : if the assumption fails we consider that the component is no more responsible for a message emitted at a bad time, as :
- this component has not be given the right information to work correctly,
- the sequence of messages incoming and outgoing has anyway stopped to follow the protocol : and that the first failure that we consider,

>>

Resulting contract (2/2)

- Agreement :

```
Contract :
  Participants : <Car>, <CruiseCtrl>, <Engine> ;
  ...
  Agreement :
    On <Car>
    Observe : val : evt.start at : entry <Car>.start
    Verify : verticalCheck (carFP, parallelCheck (cruiseCtrlFP, engineFP))
```

<Car>

<CruiseCtrl>

<Engine>

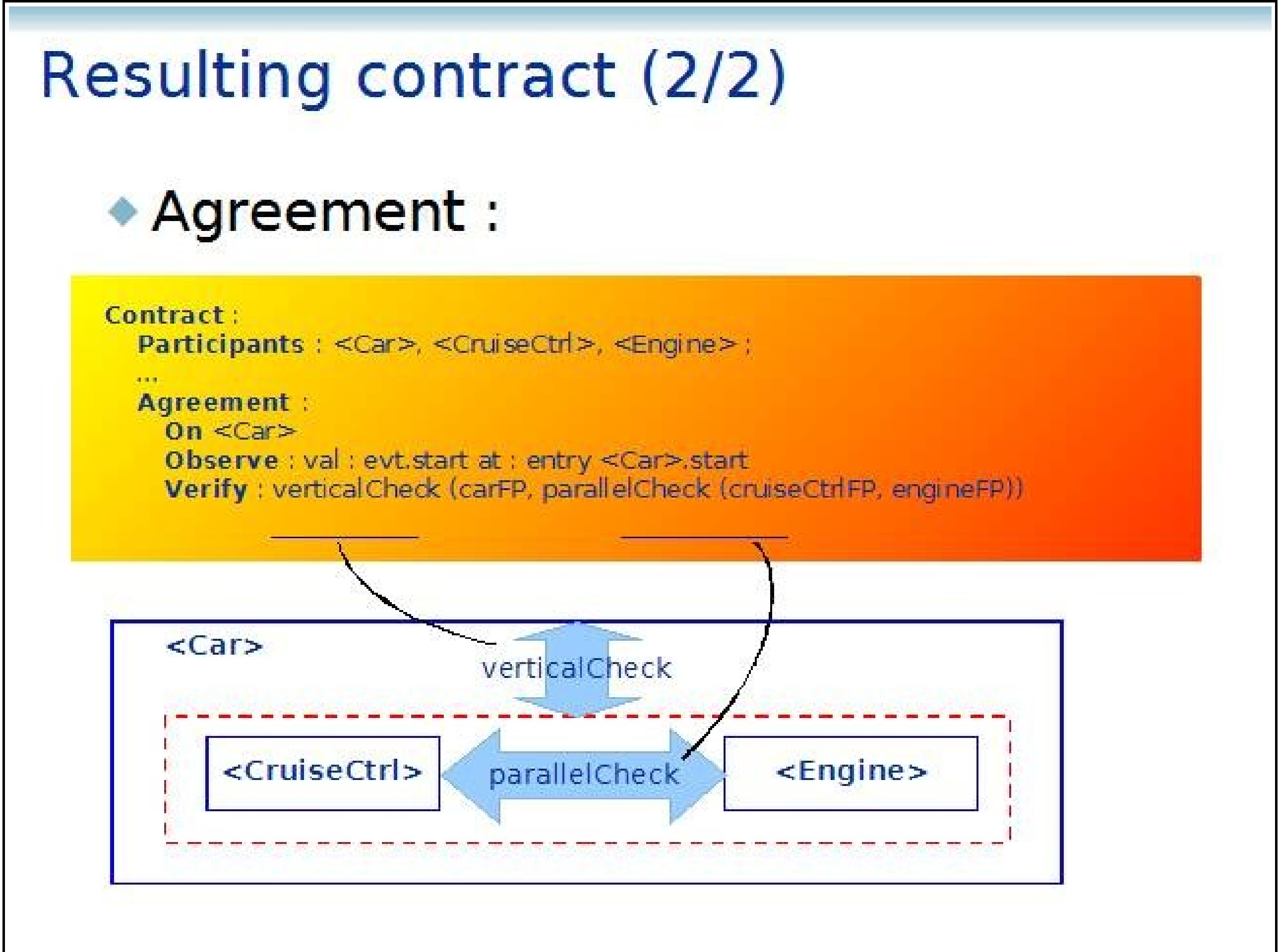
<<

For the agreement : considering the compatibility of the clause is equivalent to considering the compatibility of the specifications.

The behavior protocol are provided with a tool that can modelcheck the compatibility between many components at the same level of composition (subcomponents), and between a composite and a group of subcomponent.

So the formalism adapter, interpreting the configuration of the architectural pattern by grouping subcomponents and then comparing them to the composite, provides the expression of the agreement.

>>



<<

So the expression of the agreement is composed of an horizontal modelcheck of components at the same level of composition, and a vertical modelcheck between the group of subcomponents and the composite.

An interesting point in keeping a parallel between the architectural configuration and the expression of the compatibility expression is that details about the failure in checking the expression can be interpreted in the architecture configuration.

If the parallel check fails then that's because subcomponents are not compatible together, on the other hand if the failure of agreement comes from the vertical check then the problem is between composite and subcomponents.

>>

Plan

- Introduction
- Existing works
- Contribution
- Implementation
- Validation
  - Example of application
- Conclusion

Illustrative application

centers of interest

visitor

- Benefit of contract use :
  - Detect internal malfunctions before they become external / various properties
  - Provide explicit and generic informations to the manager

<<

As an illustration of the use of the contract system, I will consider an application deployed for a big car show, for example the Paris International Car Show, in order to provide visitors with informations :

indeed the car shows are most of the time very vast, there are a lot of stands and visitors may have difficulty to find what interest them

so with this application, a visitor can register his centers of interest (taken in a predefined set) associated to his mobile phone number at an interactive terminal, and then he will be notified, on his mobile, of relevant informations according to his centers of interest.

>>

Illustrative application

visitor

centers of interest

Center of interest

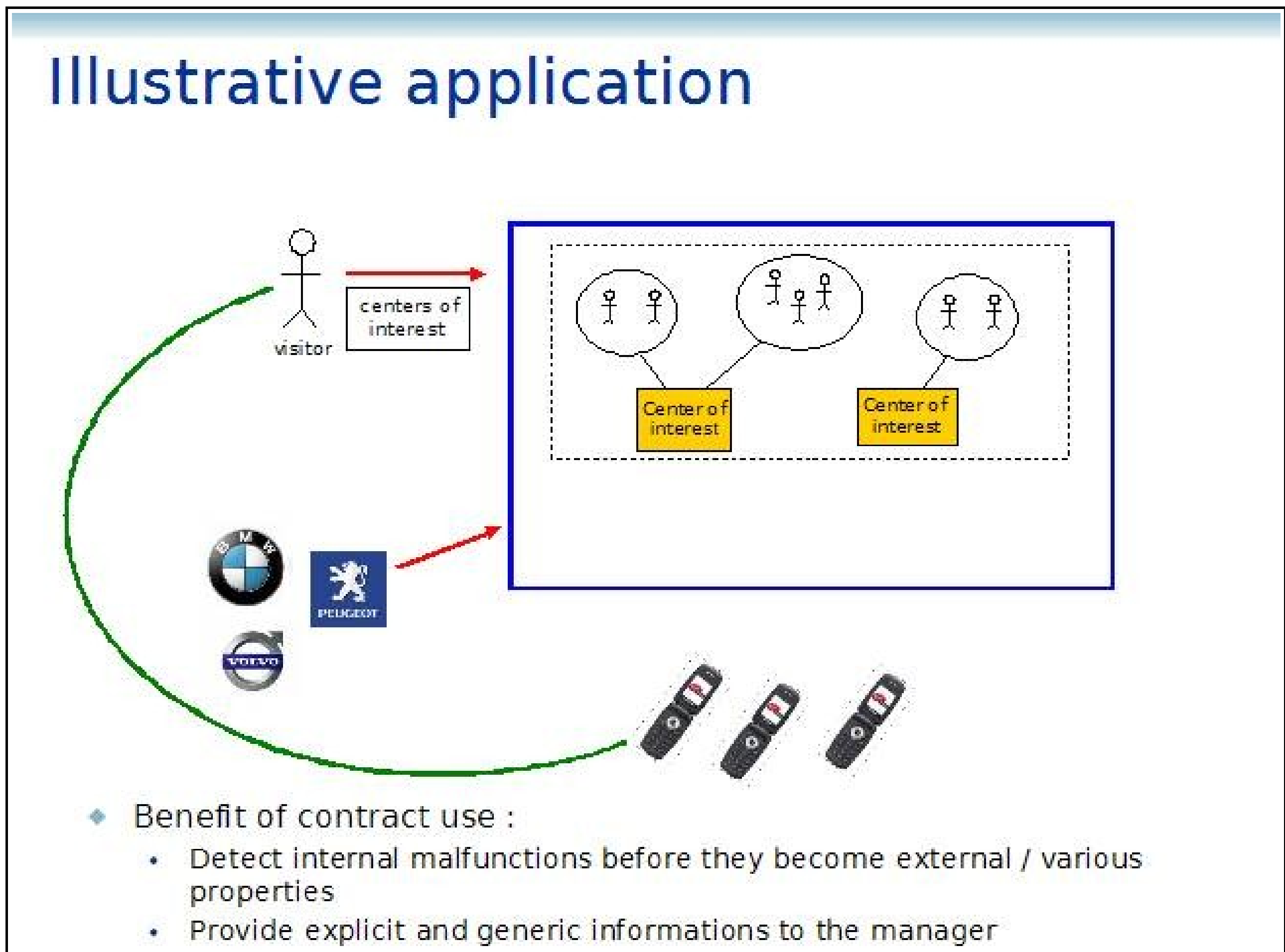
Center of interest

- Benefit of contract use :
  - Detect internal malfunctions before they become external / various properties
  - Provide explicit and generic informations to the manager

<<

The application simply builds group of visitors on the base of their centers of interest.

>>



<<

the exhibitors (the car brands) have also previously registered by the application.

>>

Illustrative application

- Benefit of contract use :
  - Detect internal malfunctions before they become external / various properties
  - Provide explicit and generic informations to the manager

<<

They have associated different kinds of notification to the centers of interest :
- advertisement
- small contest
- geographical information about their stand
- ...

of course they also specified a notification schedule etc,
some services may be interactive : for example a visitor can send a sms with the number of the stand on which he is and then receives the nearby stands corresponding to its centers of interest etc

>>

Illustrative application

visitor

centers of interest

Center of interest

Center of interest

adv

contest

stand

adv

adv

- Benefit of contract use :
  - Detect internal malfunctions before they become external / various properties
  - Provide explicit and generic informations to the manager

<<

Then the application notifies the mobile phones of the visitors with the information of the exhibitors

>>

Illustrative application

visitor
centers of interest
Interface with users
Management of Groups and Centers of Interest
Notifications Management

- Benefit of contract use :
  - Detect internal malfunctions before they become external / various properties
  - Provide explicit and generic informations to the manager

<<

This application can be seen as made of three components :
- the users interface
- the management of groups and centers of interest
- the management of notifications

The fact is that the specification of this application, from the outside, may well be as complex as the application itself : so, as the components should have been specified, the contract system that checks the interaction between the components is far more economical and pragmatic,

An other interesting point is that the “interact” contract detects a malfunction while it is still internal, before it goes out of the application,

Then the third point is that, the contract system provides the system manager with generic informations : indeed it will not be possible to have a specialist of the architecture, and an other one of the formalisms of specification every day of the car show, but the manager should anyway be able to understand what happens in the component system : the contract system is able to tell him which component or assembly is faulty, so the manager can choose to switch off a service, or to change a component, ...

>>

Plan

- Introduction
- Existing works
- Contribution
- Implementation
- Validation
- <u>Conclusion</u>

Conclusion (1/2)

- **Problem :**
  - Validity and diagnosis,
  - Joint applications of formalisms,
  - Uniform application of a formalism to various architectures.
- **Approach :**
  - Contract as a first class entity, that reifies conformity, compatibility and responsibility.
- **Result :**
  - A model and generic mechanisms gathered in a framework,

<<

So I tried to answer to several problems :

- the definition of the validity of an assembly, and the ability to build a diagnosis in case of failure
- the joint application of various formalisms to a composite system,
- the application of one given formalism to various concrete systems,

To do so, I defined a contract, which is a first class entity, and that reifies the conformity of its participants to their specifications, the compatibility of these specifications and the responsibilities of the participant for the conformity and compatibility

This resulted in a model and framework of contract based on generic mechanisms independent from the formalism of specification and implementation of the architecture constrained,

>>

Conclusion (2/2)

- **With respect to the state of the art :**
  - Explicit validity of the assembly, uncoupled from formalisms and architectures, but considering the architectural configuration,
- **Limits :**
  - Range (accepted formalisms and architectures), methodology,
- **Future prospects :**
  - Development of plugins for other formalisms (QoS) and architectures,
  - Negotiation, other demonstrator,
  - Evolved dependancies between obligations (second order clauses), extended sharing (admission control...),

<<

With respect to the state of the art : I gave an explicit definition of the validity of an assembly, independently of the formalism of specification and architecture implementation but based on the architectural configuration,

But my proposal has of course its limits : it accepts only well defined formalisms and architectures (see page 17), an very other important is that this model only checks the validity of an assembly, it does not provide a method to build a valid assembly : that's a very important.

Finally : some points would be interesting as future works :

- the development of adapters for other formalisms and architectures
- the integration of negotiation processes
- more fundamentally, it would be interesting to have second order clauses, that is clauses that constrain other clauses

- an other fundamental point, would be to consider the sharing, which is the base of the contract, not only in the field of data exchange, but in the field of resources sharing ...

>>