Bridging the Gap Between the Cognitive Engine and the SDR

Jakub Moskal

Mieczysław Kokar

Shujun Li

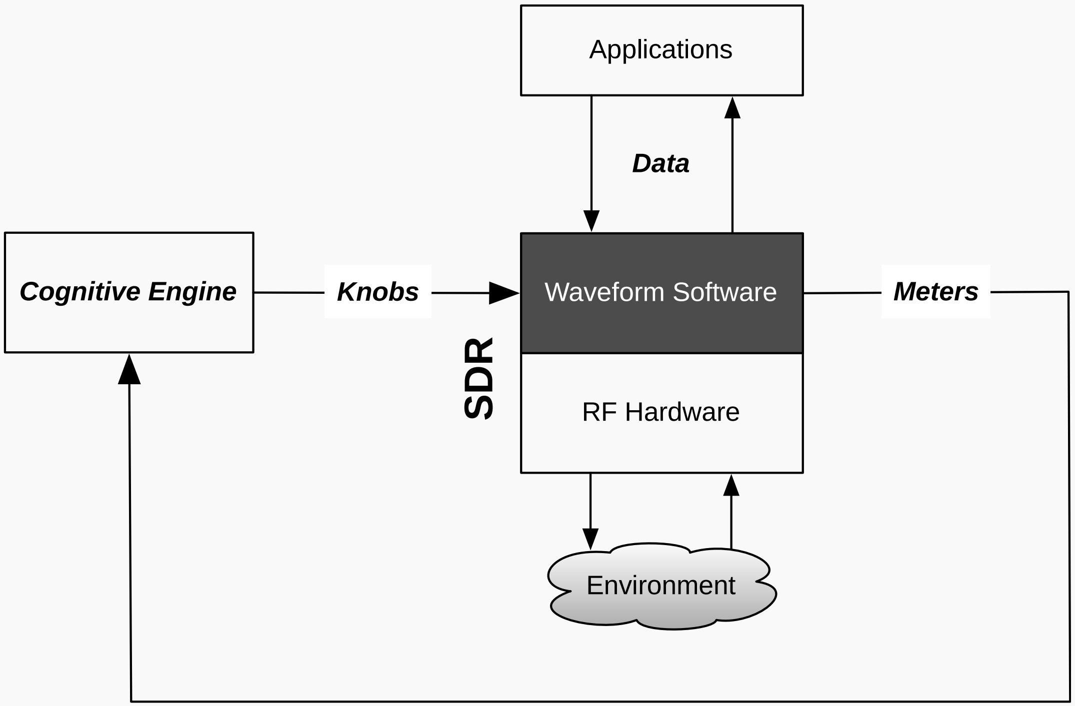
CR Architecture

Cognitive Engine:

- Genetic algorithms
- Case-Based Reasoning
- Knowledge-Based reasoning

CR operational behavior can be altered by modifying its **parameters**:

- observable Meters, perceptions, .e.g.:
  - bit error rate
  - Doppler spread
  - noise power
- controllable Knobs, actions, e.g.:
  - transmitter power
  - modulation type
  - bandwidth
  - carrier frequency



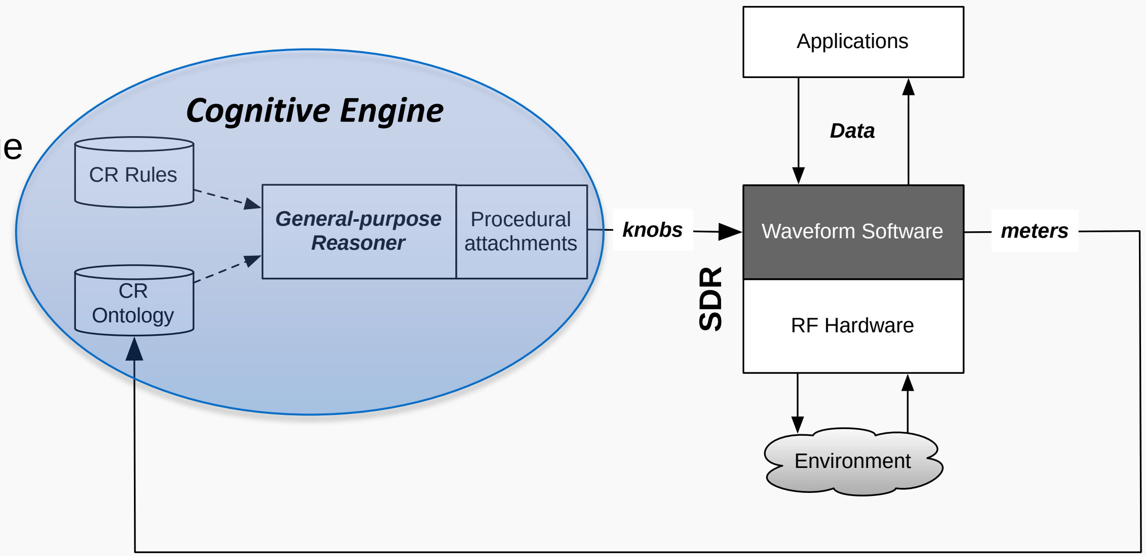
Knowledge-based CR

<u>Main components</u>:

1. General-purpose **reasoner** (inference engine)

2. **Ontology** - domain knowledge described with common terms and concepts

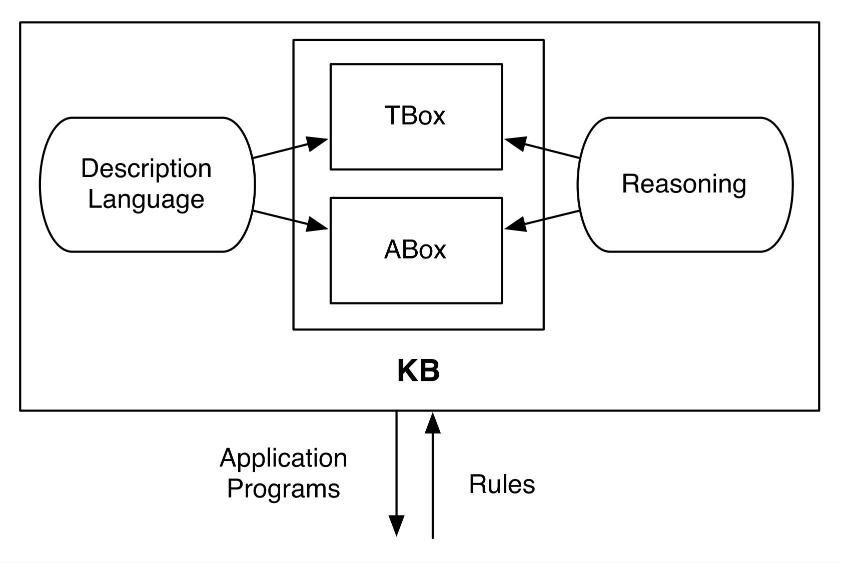
3. **Rules**
   - declarative form
   - out of order execution
   - extended with **procedural attachments** – imperative functions (used for accessing knobs and meters)



Knowledge Representation: OWL

- Web Ontology Language (OWL)
  - TBox
  - Abox

- OWL and CR:
  - TBox – axioms shared by all radios
  - ABox – axioms pertaining to particular individual radios

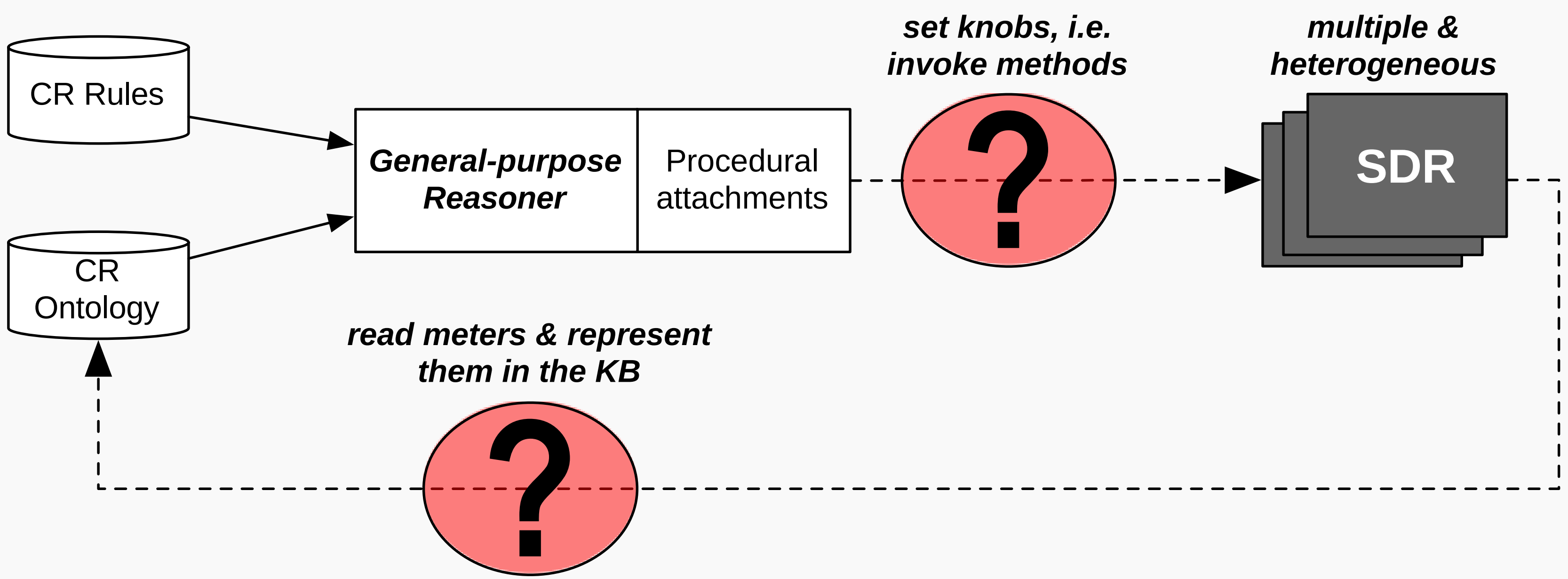


Knowledge-based CR: Benefits

- Domain experts are not required to know the SDR implementation details (programming language, architecture) to write rules
- Rules are declarative, not executed in a prescribed order – they can be modified without the need to recompile
- Easier certification and accreditation – once rules and reasoner are accredited, rules (policies) can be reused

Problem Formulation

- Different radios provide different Knobs & Meters (K&M) that need to be accessed by the reasoner
- Lack of standard SDR Application Programming Interface (API)
- Lack of standard CR architecture

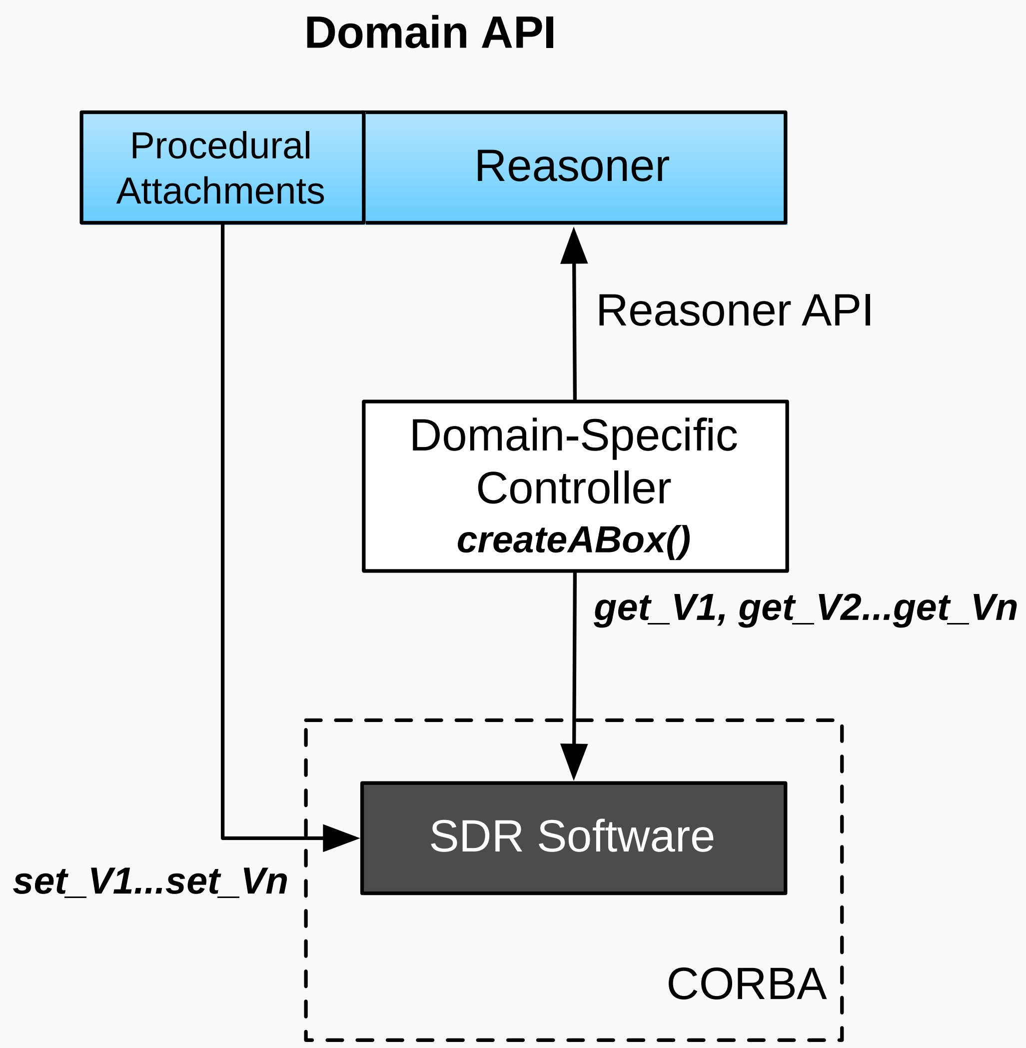


Domain-specific API

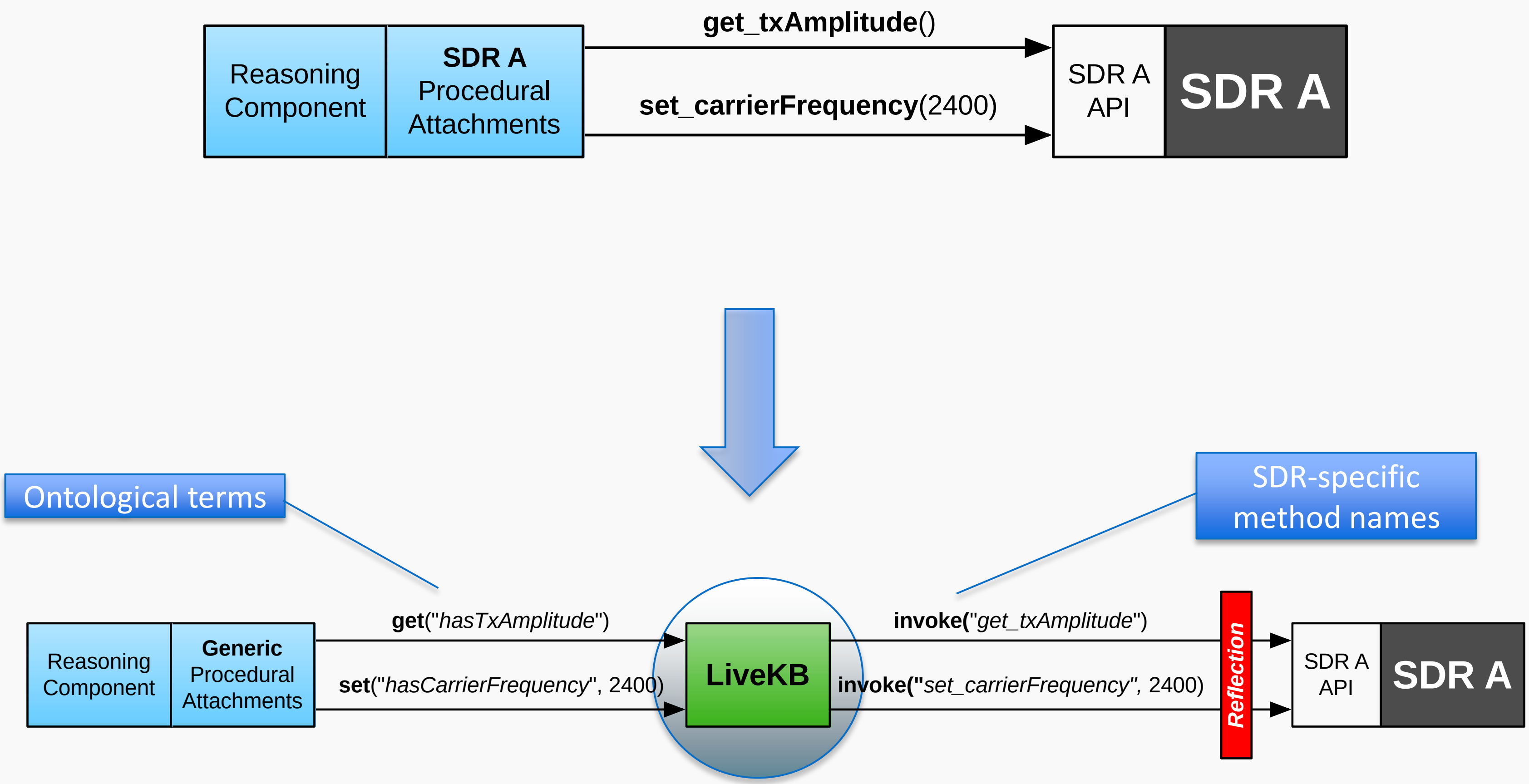
Current CR designs interface SDR via specific APIs: *set_V1, set_V2… get_V1, get_V2…*

Consequences:

- *(get)* Design-time knowledge about the ontology is required to produce appropriate Abox
- *(set)* Reasoner must be extended with API-specific procedural attachments
- The same functionality must be coded for each radio API
- API-dedicated code must be maintained as API changes
- API may become a bottleneck to support compatibility with legacy components

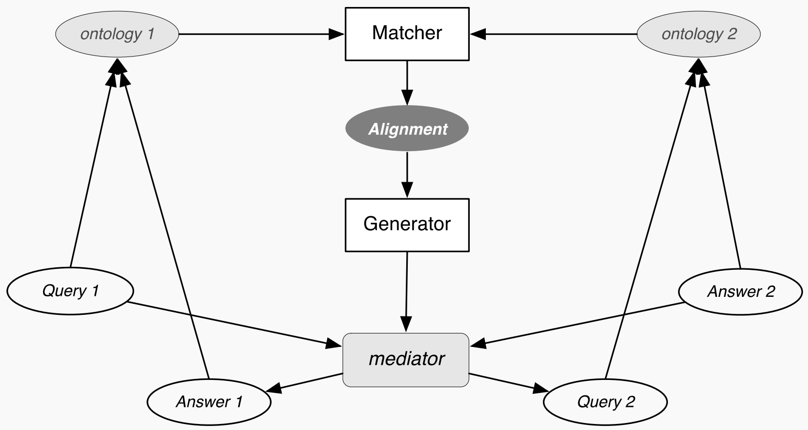


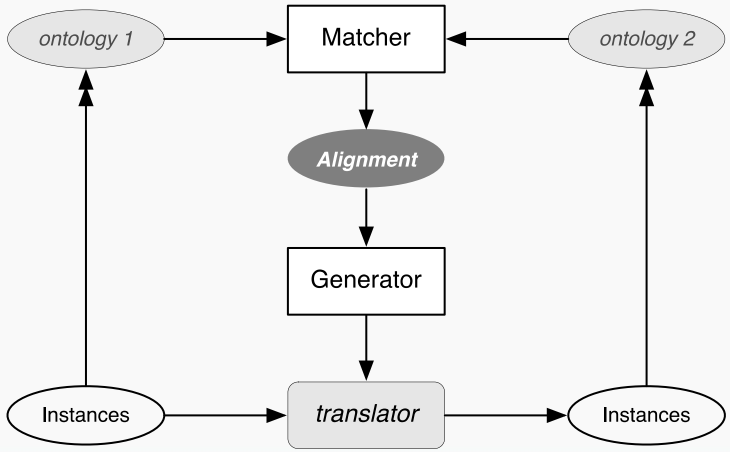
LiveKB - Motivation



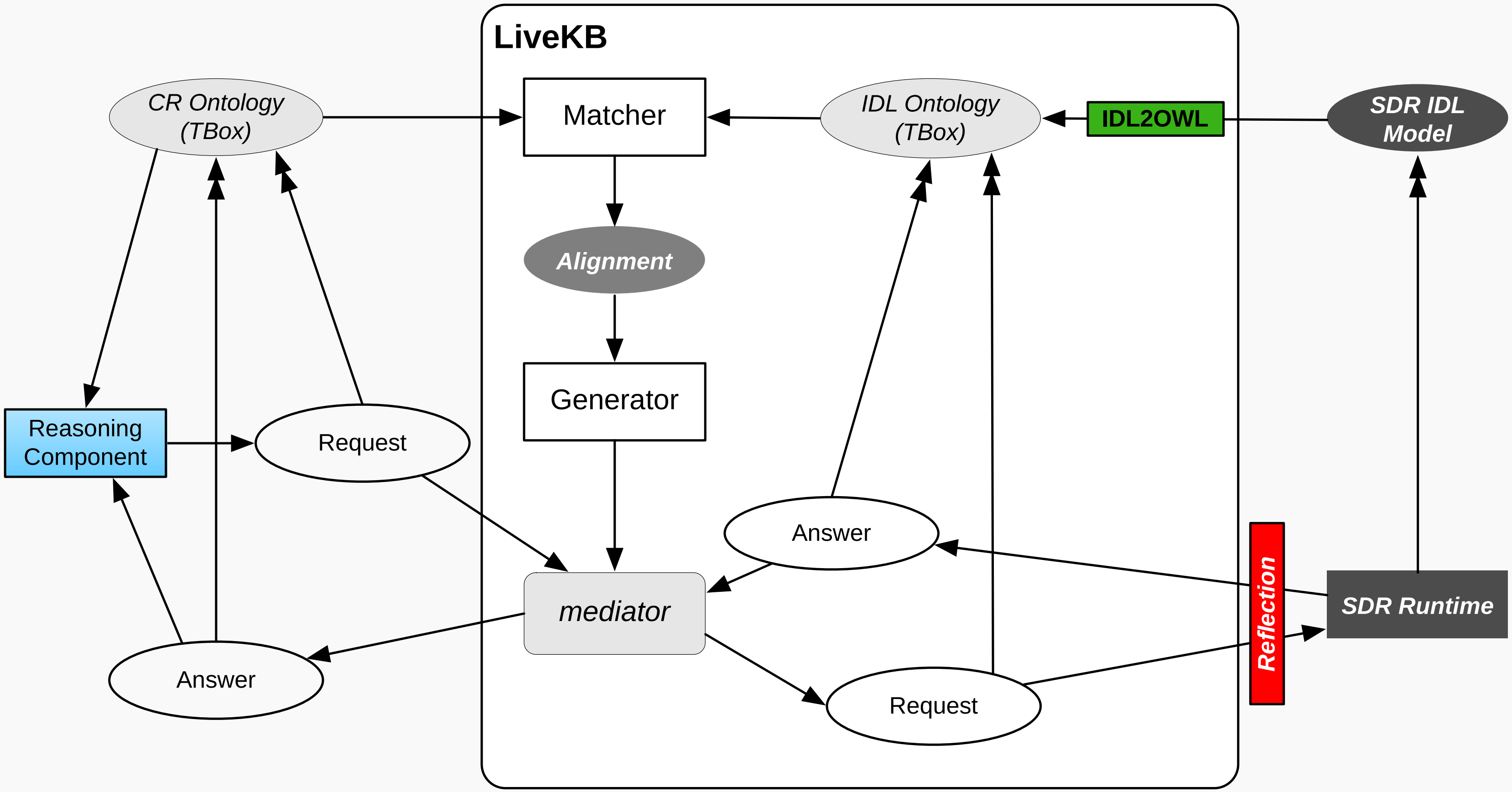
Ontology Matching

- **Ontology Matching** - the process of finding relationships between entities of different ontologies
- **Alignment** – result of matching, includes statements like entity equivalence, sub-super relationship, class intersection, inverse relation, etc.
- Numerous applications, e.g. data integration, semantic web services
- Different ontology *heterogeneity*: syntactic, terminological, conceptual, semiotic
- Alignment representation: **EDOAL**, manipulation: **Alignment API**
- <u>Fully automated only for rather simple correspondences</u>

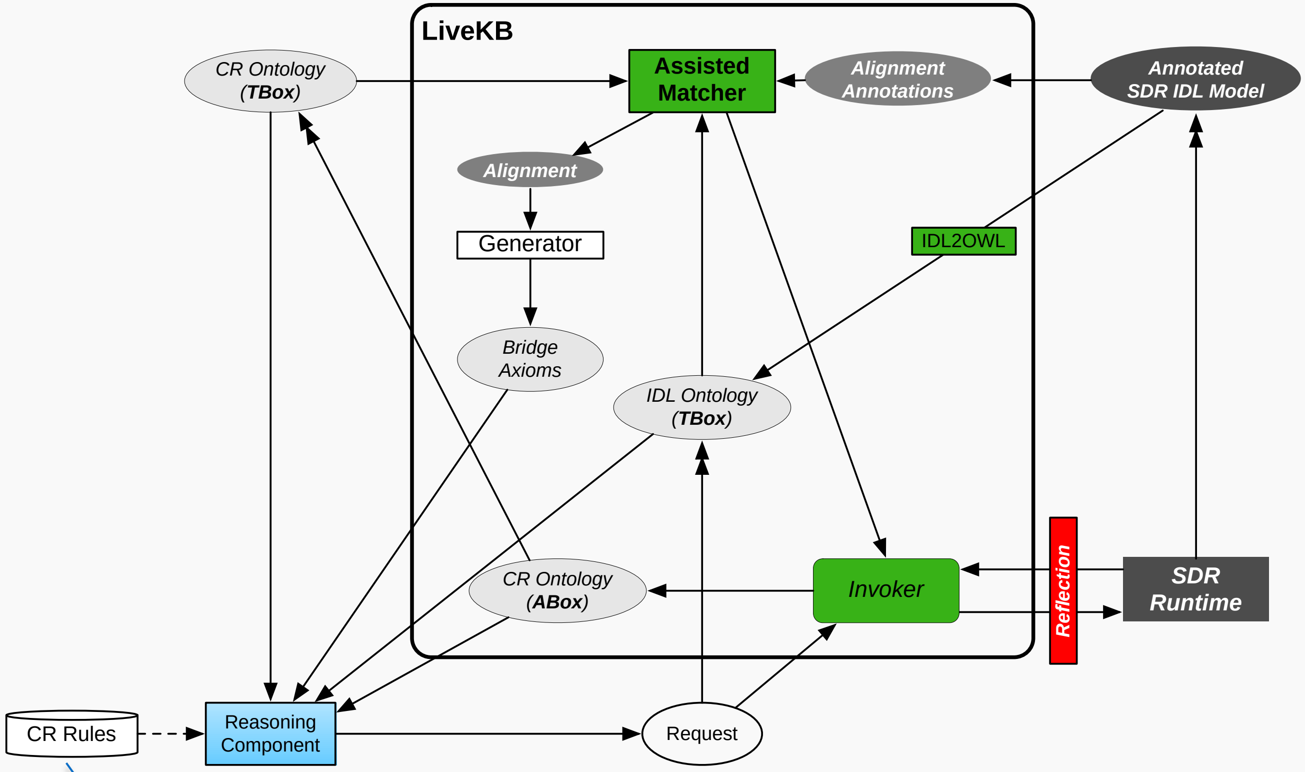




LiveKB – Ideal Design



LiveKB – Feasible Design



condition →
find a **setter** property in the IDL ontology that is **equivalent** to a **knob** in the CR ontology
→
invoke(**setter**, newValue)

Generating IDL Ontology

```
module api {
        interface SignalDetector {
                attribute float sampleRate;
        };
        interface Transmitter {
                float getNominalRFPower();
                long getTransmitCycle();
                void setTansmitCycle(in long
newTransmitCycle);
        };
        interface TestRadio {
                readonly attribute Transmitter transmitter;
                readonly attribute SignalDetector
signalDetector;
                float getTxAmplitude();
        };
};
```

idl:SignalDetector	
idl:SignalDetector@sampleRate	Float

idl:Transmitter	
idl:Transmitter@getNominalRFPower	Float
idl:Transmitter@getTransmitCycle	Integer
idl:Transmitter@setTransmitCycle	Integer

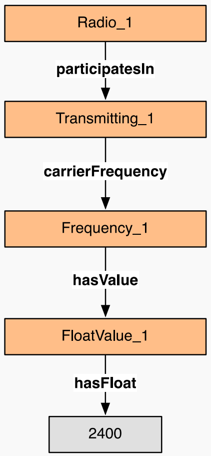
idl:TestRadio	
idl:TestRadio@getTxAmplitude	Float

- topDataProperty
  - getterMethod
    - SignalDetector@sampleRate
    - TestRadio@getTxAmplitude
    - Transmitter@getNominalRFPower
    - Transmitter@getTransmitCycle
  - setterMethod
    - SignalDetector@sampleRate
    - Transmitter@setTransmitCycle

Bridge Axioms

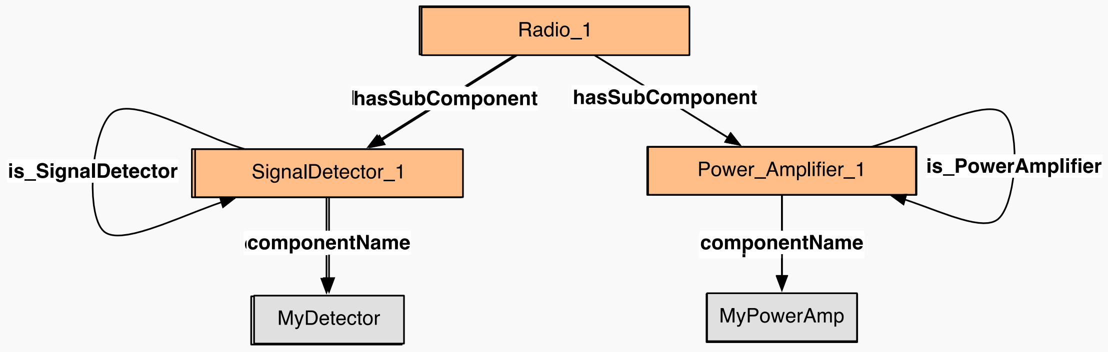
- An IDL ontology property needs to be maped to a chain of CR ontology properties
- Example of a Bridge Axiom:

  participatesIn ∘ carrierFrequency ∘ hasValue ∘ **hasFloat ⊑ Transmitter@carrierFreq**



- Chains can be ambiguous:
  - hasSubComponent ∘ componentName ⊑ IDLProperty1
  - hasSubComponent ∘ componentName ⊑ IDLProperty2

- We add **self-restrictions** to disambiguate chains:
  - hasSubComponent ∘ is_SignalDetector ∘ componentName ⊑ IDLProperty1
  - hasSubComponent ∘ is_PowerAmplifier ∘ componentName ⊑ IDLProperty2



Assisted Matcher – IDL annotations

- Each getter and setter in IDL must be annotated according to the following pattern:
  - ***Class1.(objectProperty.Class)^n.datatypeProperty***
- Annotations explicitly indicate the alignment with the CR ontology
- Assisted Matcher generates self-restrictions and creates bridge axioms

EXAMPLE

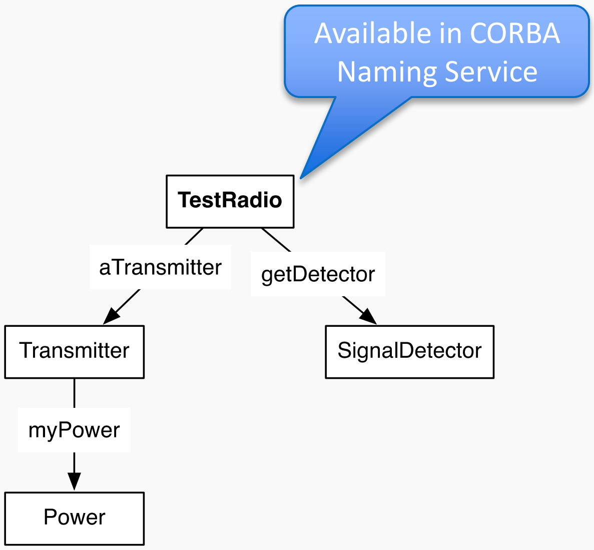
```
module api {
        interface TestRadio {
                // Radio.hasSubComponent.PowerAmplifier.txAmplitude
                float getTxAmplitude();
        };
};
```

hasSubComponent ∘ is_PowerAmplifier ∘ txAmplitude ⊑ TestRadio@getTxAmplitude

Invoker and Object Tree

- IDL interfaces provided by SDR are assumed to form a tree-like structure:
  - Vertices – implementations of interfaces
  - Edges – interface type attributes or methods with interface return type
- Implementation of the root must be available via CORBA Naming Service
- Could be extended to a forest

```
module api {
        interface SignalDetector { };
        interface Power { };

        interface Transmitter {
                readonly attribute Power myPower;
        };

        interface TestRadio {
                readonly attribute Transmitter aTransmitter;
                 SignalDetector getDetector();
        };
};
```



Choice of middleware

- **CORBA**
  - Robust and reliable technology
  - Already used in SCA-based radios
  - Very efficient (implementations of ORBs in DSPs and FPGAs
- **Alternative: Web Services**
  - IDL → WSDL
  - GIOP → SOAP
  - Naming Service → UDDI
  - <u>Potential</u> problems: additional middleware for SCA radios, serialization of binary data, convincing the SDR community

LiveKB API – Simple & Generic

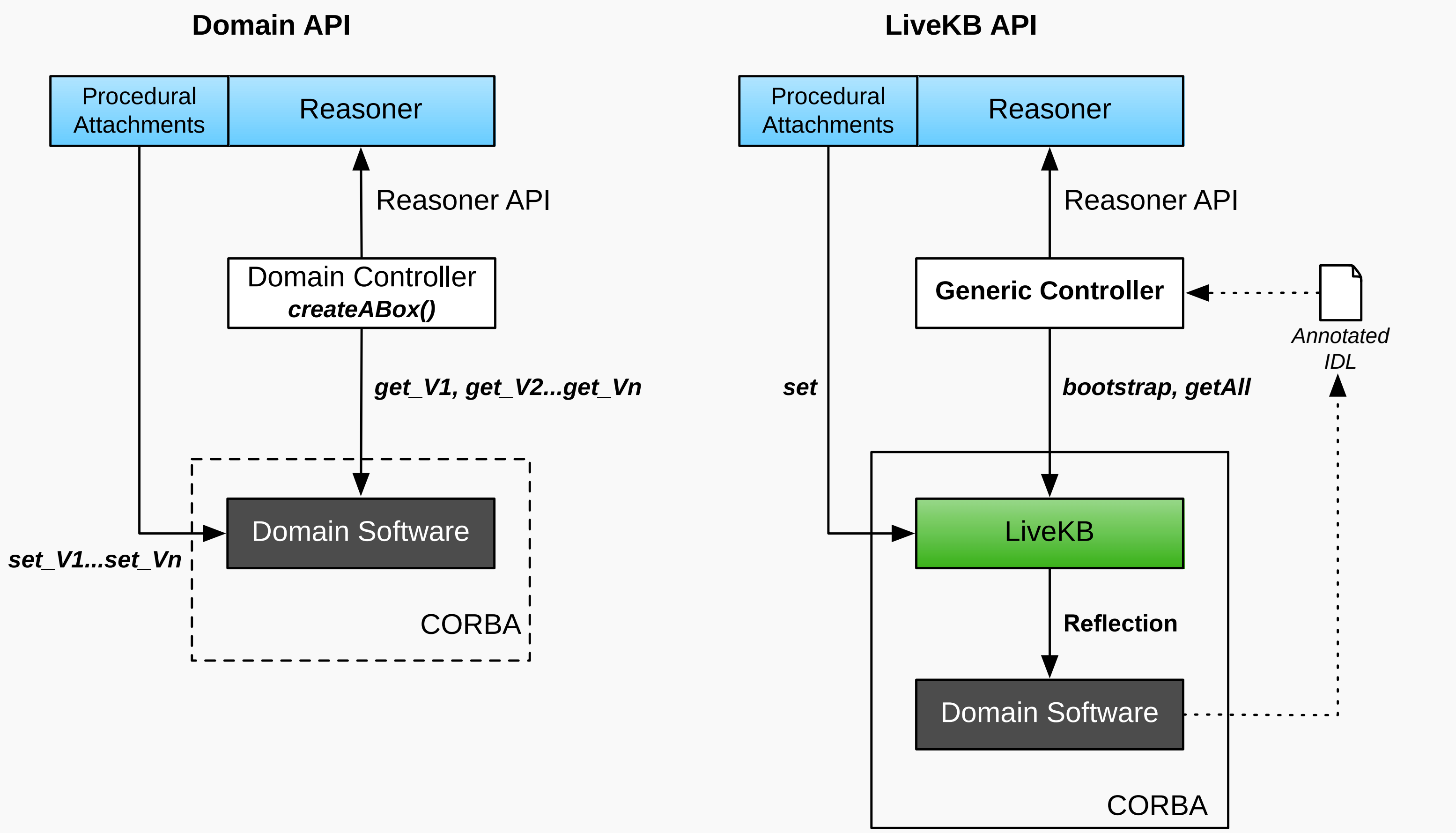
```
module livekb {
    interface LiveKB {
        string getAll();
        any get(in string property);
        void set(in string property, in any value);
    };

    interface LiveKBFactory {
        LiveKB getInstance(in string model,
            in string rootName, in string ontology);
    };
};
```

Requirements

- SDR parameters accessible via CORBA
- Run-time objects form a tree-like structure and the root is available via CORBA Naming Service
- The IDL respects the following constraints:
  - Getters have one of the following forms:
    - Operations that have no parameters and return primitive value
    - Operations that have a single parameter of primitive type, return **void** and use **out** passing direction
    - Attributes of primitive types
  - Setters have one of the following forms:
    - Operations that have a single parameter of primitive type, return **void** and use **in** passing direction
    - Attributes of primitive types that are not **readonly**
  - Annotations follow the pattern:

    Class1.(objectProperty.Class[*])^{n}.datatypeProperty
  - All annotations allow the Invoker to generate a *proper* Abox

Domain API vs. LiveKB



Comparison: adaptability

- Four different scenarios:
  1. Ontology has been redesigned – hierarchy changed, available K&M remained the same
  2. Ontology has been augmented to include new parameters
  3. Switch to a new domain – ontology, rules, domain software replaced
  4. Domain software API has changed to a new version, not backwards compatible

Scenario	Domain-API	LiveKB
1	**Rewrite code** that creates Abox	Adjust IDL annotations
2	**Develop** new procedural attachments, add **code** that creates new ABox axioms	Add IDL annotations to the new methods
3	**Implemented** new domain API, **develop** new procedural attachments, **implement** code that creates Abox	Annotate IDL for the domain ontology
4	Either **implement adapter**, or re-**implement** domain API, **update** procedural attachments, **rewrite code** that generates Abox	Move annotations to the new IDL

Comparison: complexity

Operation	Domain API	LiveKB
Bootstrap	*O(1)*	***O(i*m*c)*** **i** – number of IDL interfaces, **m** – number of methods and attributes per interface, **c** – length of the annotation related to the method/attribute
getAll	*Θ(n),* n – number of getters	*Θ(n),* n – number of getters
get	*O(1)*	*O(1)*
set	*O(1)*	*O(1)*

- Using LiveKB bootstrap operation is more complex, because LiveKB generates artifacts specific to the domain software. This operation is performed only once.
- LiveKB also produces additional triples that need to be loaded to reasoner's KB, it is in the order of *O(i*m*c),* where I is the number of IDL interfaces with annotated getters or setters, m is the number of getters and setters per interface, and c is the length of the annotations

Conclusions

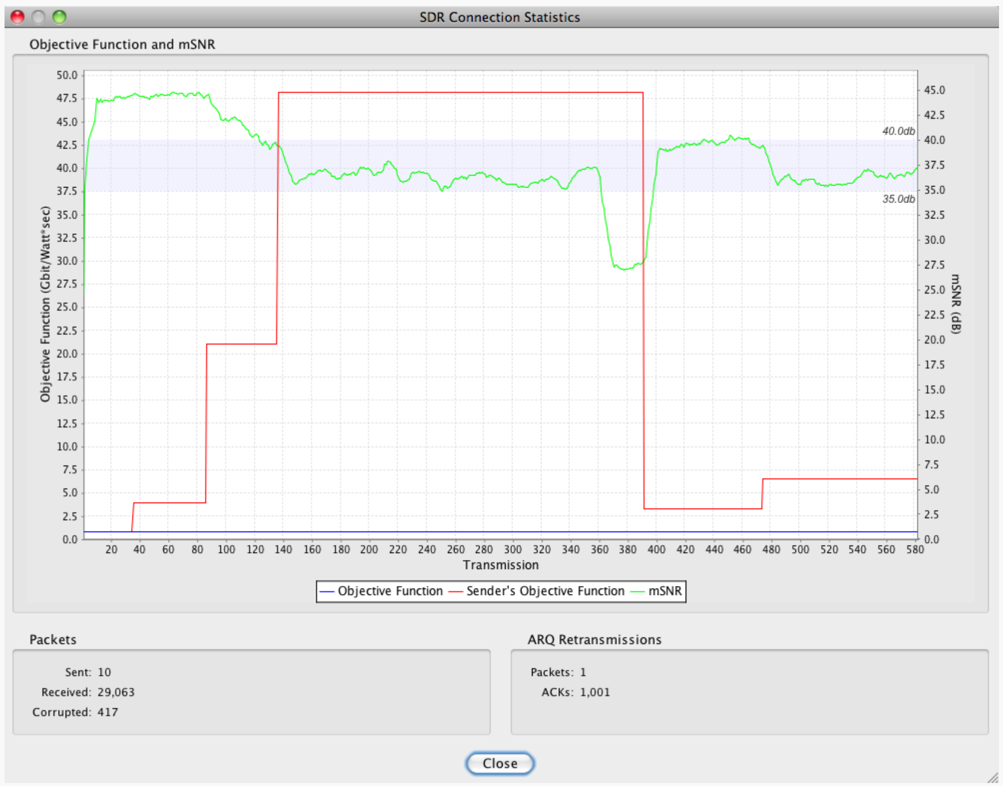
- Benefits of using LiveKB:
  - Support for knowledge reusability and exchange
  - Relatively small effort to adapt to changes
  - Inherent domain and platform-independence

- Drawbacks of using LiveKB:
  - Requirement to use CORBA
  - Increased number of facts in the KB (bridge axioms)
  - Slower bootstrap

- Use of LiveKB is recommended in domains that lack standards, and where changes are likely to happen in the future – **Cognitive Radio is a good match**

SDR'10 Demo

- LiveKB was successfully showcased at the SDR'10 Technical Conference
- An image was sent pixel-by-pixel to generate data traffic
- Radios performed collaborative link optimization, exchanged facts and rules
- Meters were accessed and knobs modified using LiveKB



Thank You