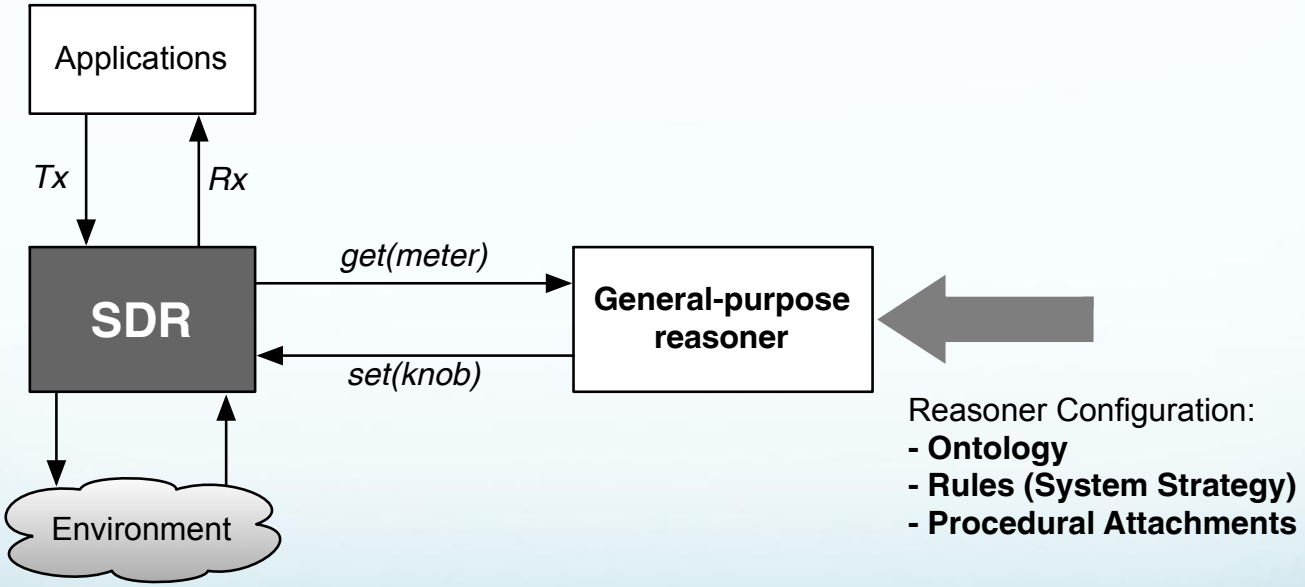
Interfacing a Reasoner with an SDR using a Thin, Generic API:

A GNU Radio Example

Jakub Moskal
Mieczyslaw Kokar
Shujun Rachel Li

Introduction

- Our focus: Cognitive Radios that utilize ontologies, rules (policies) and an inference engine (reasoner)



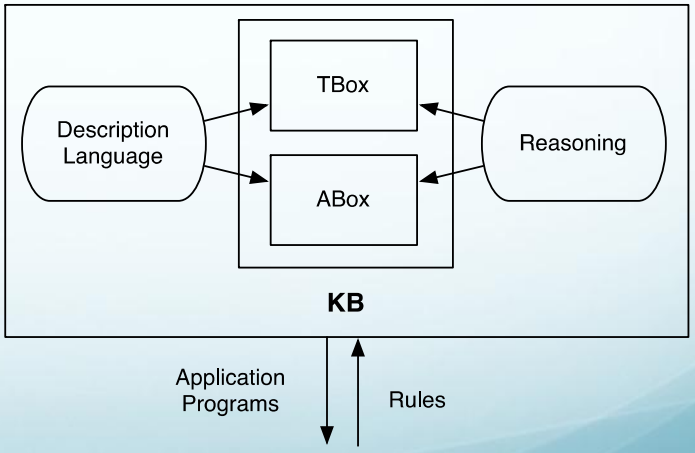
Using Ontology

- **Ontology** – a formal, explicit specification of a set of concepts in a specific domain and the relationship between these concepts
- Benefits of Ontology-based approach to CR:
  - Ontology captures domain knowledge that is abstract and easy to exchange between radios
  - Ontology is not part of the architecture, it is provided dynamically to the reasoner
  - Rules written in ontological terms are abstract and independent of implementation details of the SDR software or hardware layers

OWL – Web Ontology Language

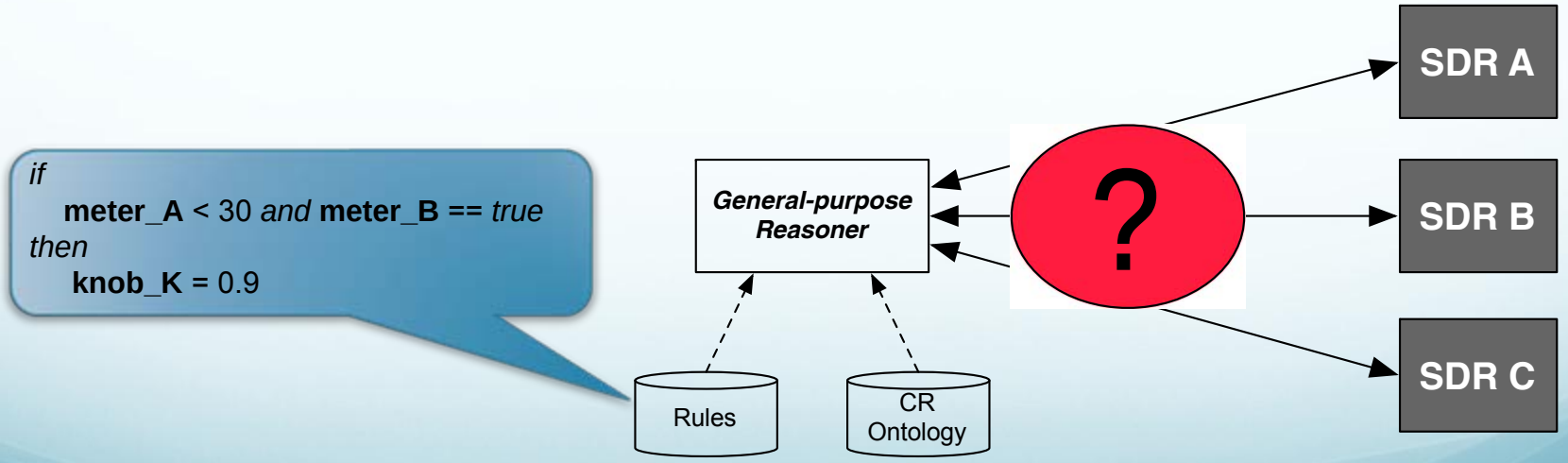
- OWL captures ontological knowledge in a format suitable for automatic interpretation (reasoning)
- KB consists of:
  - **TBox** – axioms about the domain
  - **Abox** – facts pertaining to a particular radio

Radios:
•**share** common Tbox
•**exchange** ABox – facts about themselves
•**infer** implicit facts from the KB



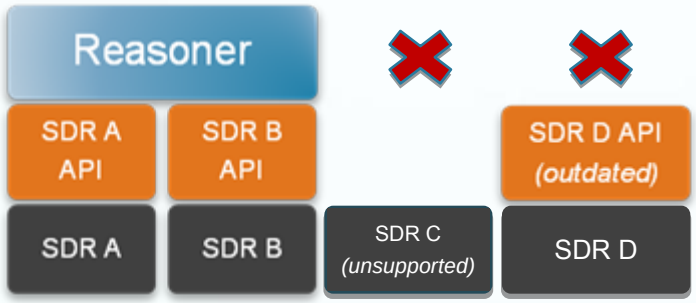
Objective

- Reasoner needs access to SDR's Parameters
  - Knobs and Meters (K&M) need to be represented in the Knowledge Base (ontology)
  - Knobs and Meters need to be accessible from within the rules (via procedural attachments)
- **Design a universal interface for accessing SDR's K&M**



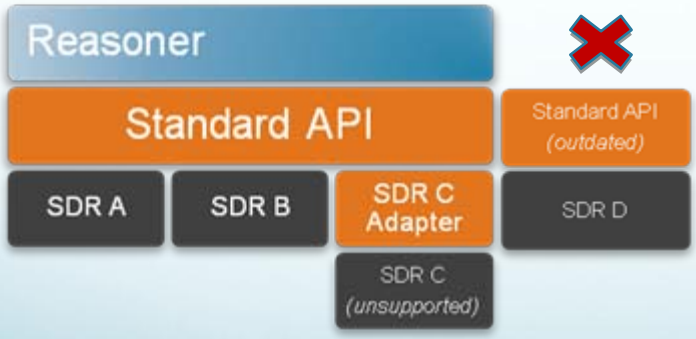
Traditional approaches – Domain-specific API

- SDR API



Drawbacks:
- Limited to a finite set of APIs known at design time (SDR C is not supported)
- Requires dedicated code and maintenance for each SDR
- Limited to common functionality
- Potentially language/platform dependent

- Standard API



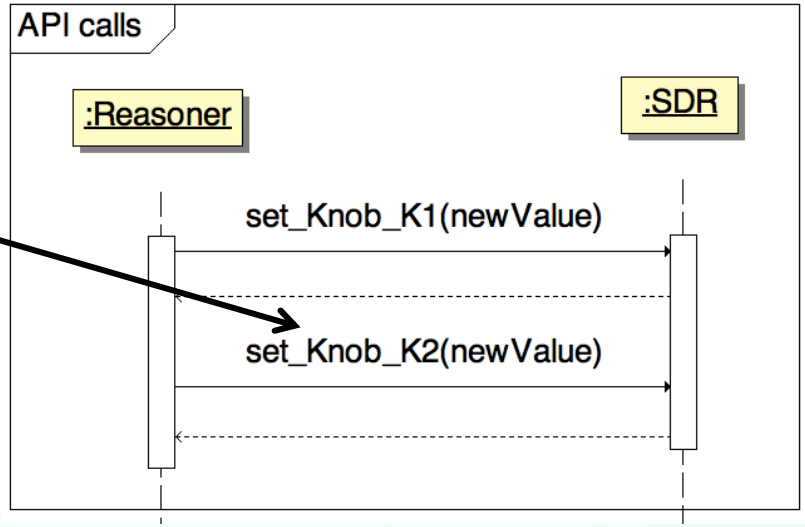
Drawbacks:
- Requires maintenance of the standard API-dedicated code
- Requires adapters for non-compliant software
- Assumes community agreed on a standard
- API may become a bottleneck
- Frequent updates unlikely

SCA CR API would most likely fall in this category

API-dedicated code

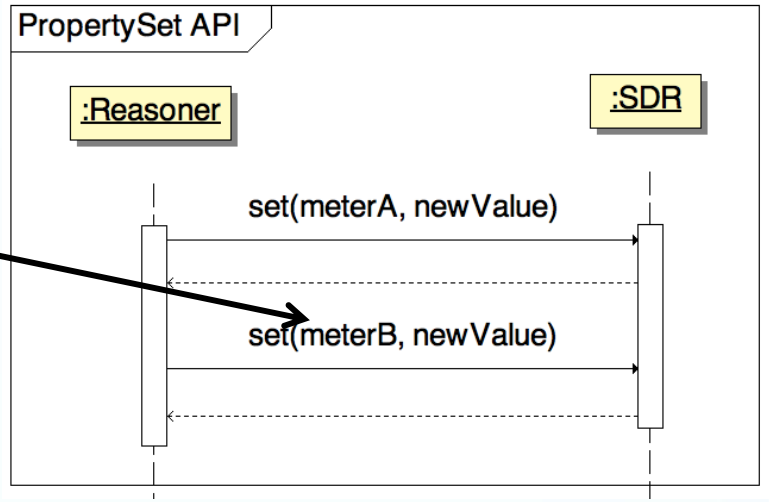
Setting a knob with domain-specific API

- API known at design time
- API becomes a **hard-coded** part of CR implementation
- API changes require recoding, recompiling, redeploying…
- **Not suitable for heterogeneous inter-radio requests for K&M**
- Lack of standard



Alternative: PropertySet API

- Two methods: *set* and *get*
- Parameter names passed as arguments
- Parameter names are **fixed**, meaningful only for radios that use exactly the same names
- **Not suitable for inter-radio requests**



txPower ≠ transmissionPower

scanDuration ≠ detectionDuration

Reflection

- *Reflection* allows programs to **observe** and **modify** their own structure and behavior dynamically, **at runtime**

SDR_A
getTxAmplitude() float

```
Class sdrClass = getClass("SDR_A");
Object sdrObject = sdrClass.newInstance();
Method method = findMethod("getTxAmplitude");
Object txAmp = method.invoke(sdrObject);
```

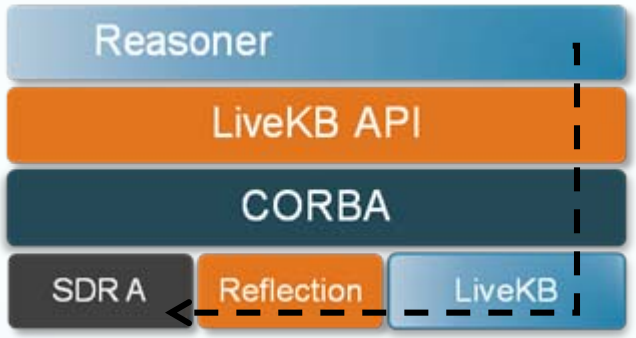
Pass different **arguments**

SDR_B
getTxAmpl() float

```
Class sdrClass = getClass("SDR_B");
Object sdrObject = sdrClass.newInstance();
Method method = findMethod("getTxAmp");
Object txAmp = method.invoke(sdrObject);
```

LiveKB API

- SDR-independent, thin and **generic** API

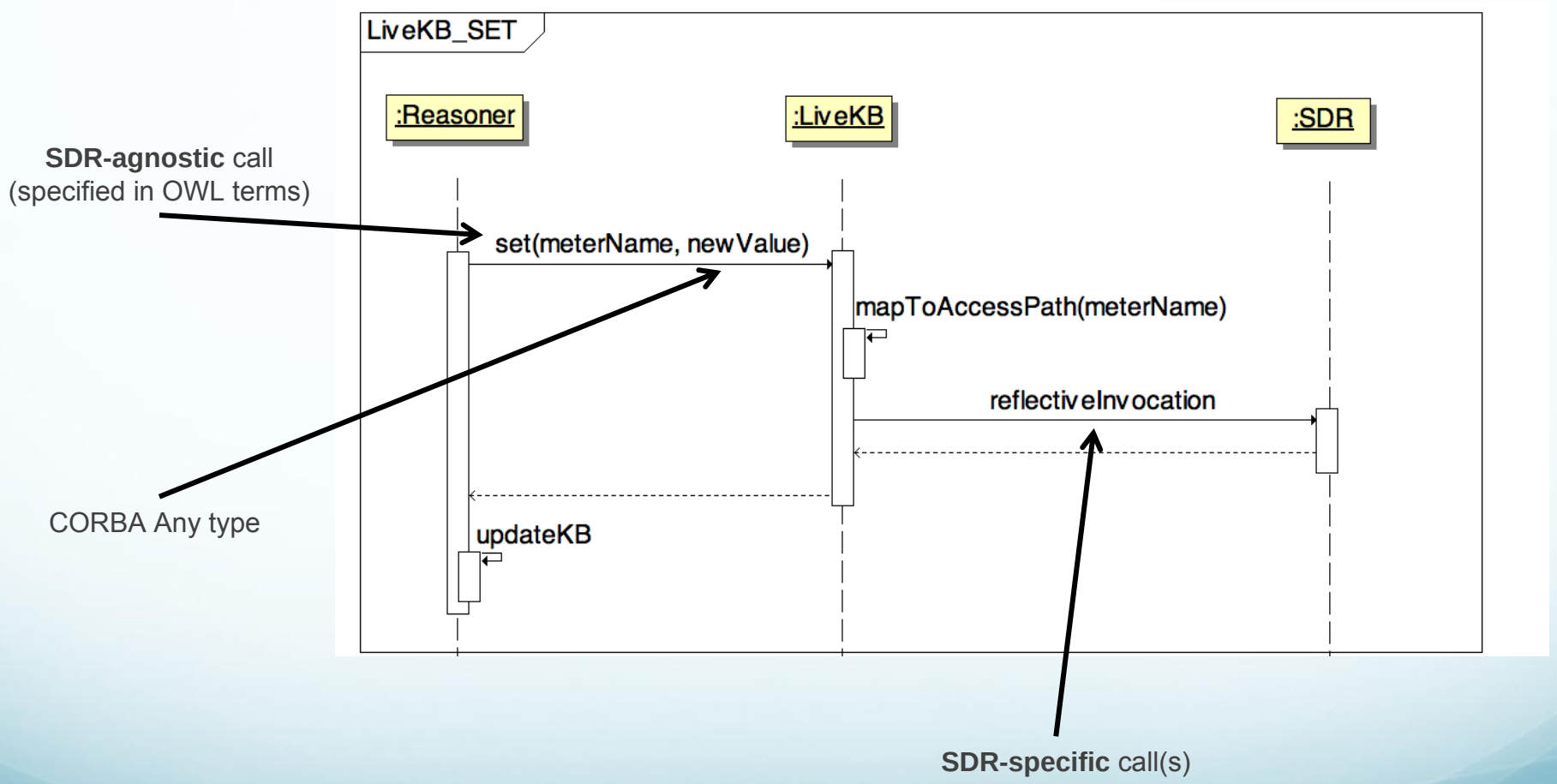


Characteristics:
•Not specific to any SDR
•Platform-independent via CORBA
•Uses **reflection** to invoke SDR-specific invocations
•Requires three components:
- Domain ontology
- SDR's IDL
- Mapping from OWL to IDL

LiveKB API:
•any **get**(string owlProperty)
•void **set**(string owlProperty, any value)
•string **getAll**()

LiveKBFactory API:
•LiveKB **getInstance**(string IDL, string OWL, string mapping)

Setting a knob with LiveKB



LiveKB Mapping

Mapping:

```
<sdro:Radio resourse="radioA:Radio_A">
  <sdro:participatesIn>
    <sdro:hasParticipant>
      <sdro:txAmplitude
            get="GNURadio.getTransmitter.getTxAmpl"
            set="GNURadio getTransmitter.setTxAmp" />
   </sdro:hasParticipant>
  </sdro:participatesIn>
</sdro:Radio>
```

Sequence of invocations

Name of CORBA NamingService object

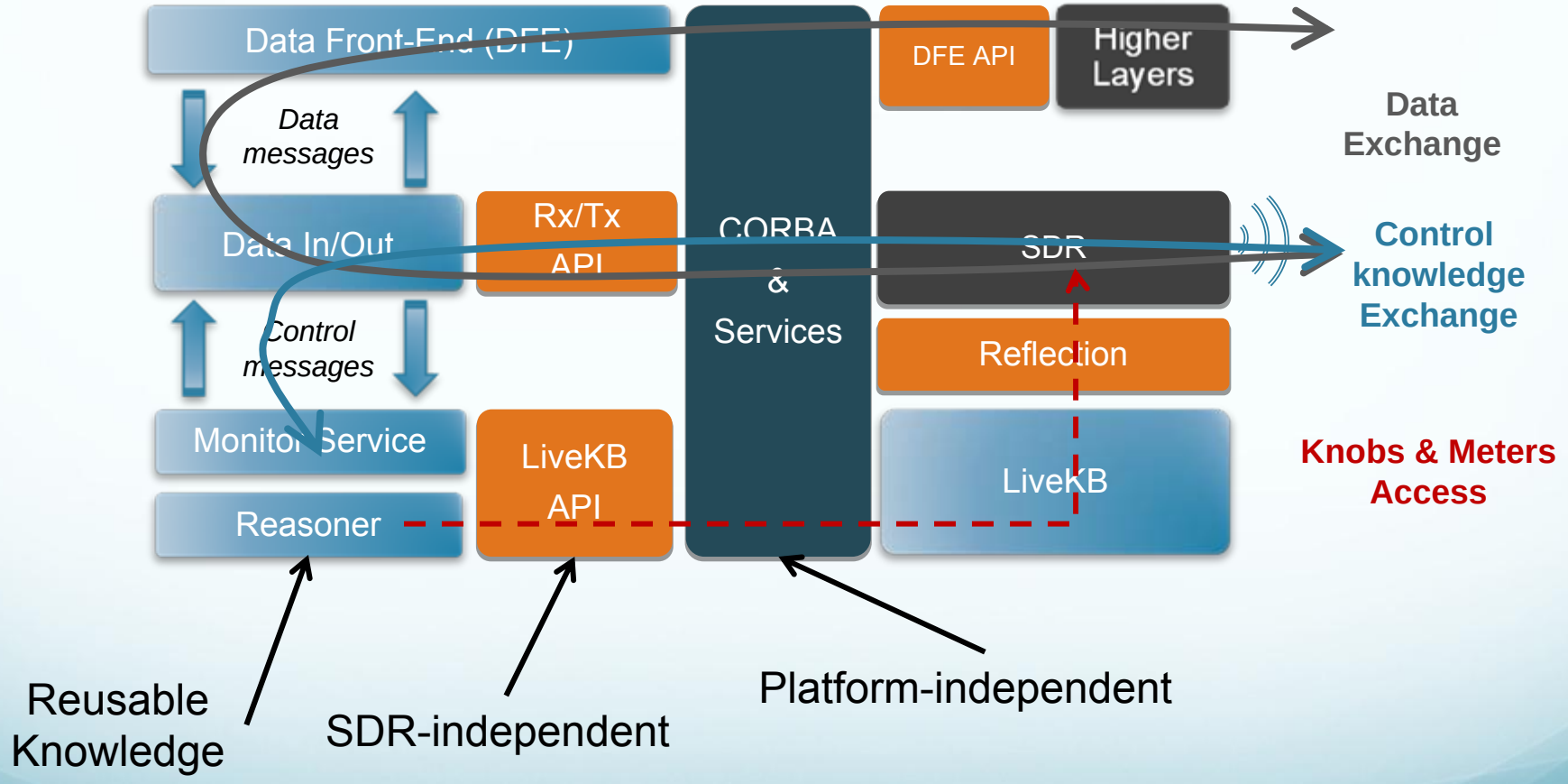
In Rules:

```
<set>
  <param>/sdro:Radio/sdro:participatesIn/sdro:hasParticipant/sdro:txAmplitude</param>
  <param datatype="xsd:float">0.9</param>
</set>
```

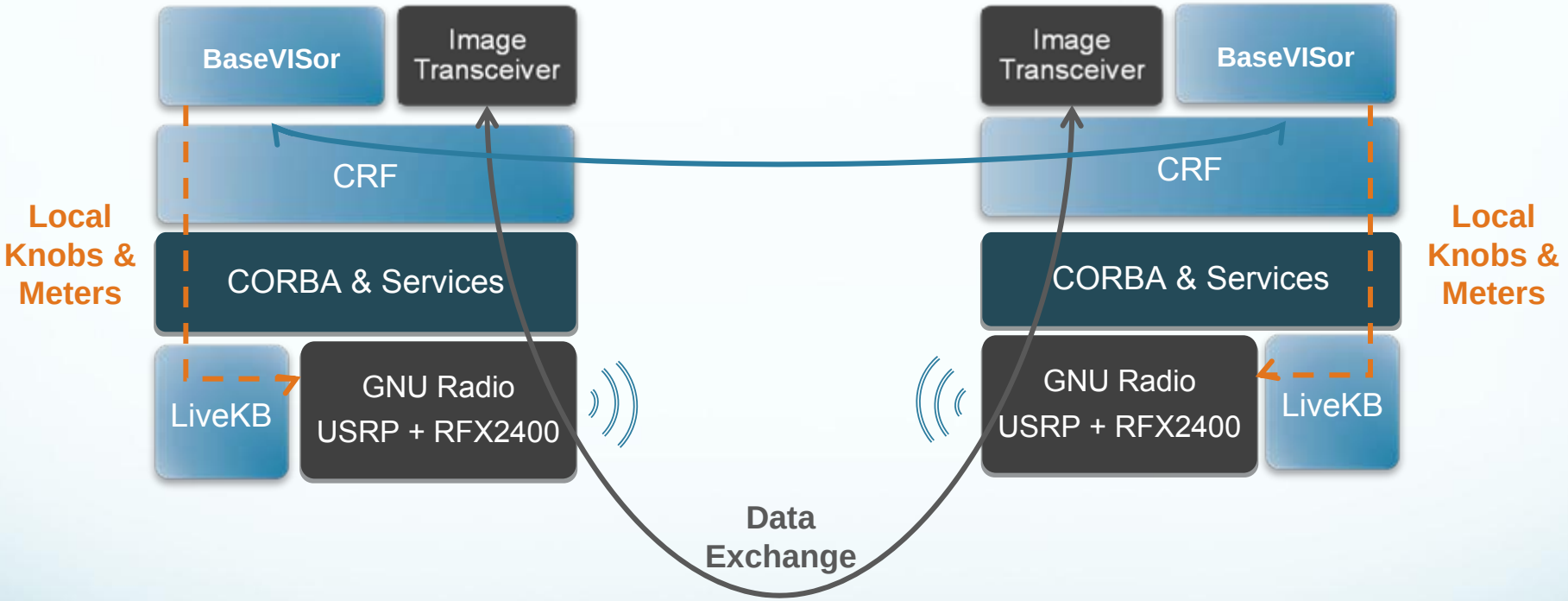
Domain-specific vs. LiveKB

Feature	Domain-specific API	LiveKB API
Style of invocation	Tied to a specific API	Defined in abstract terms from the ontology shared by all radios (**WINNF Standard**)
Access to remote knobs and meters	Limited to homogenous radios	Available via exchange of control messages
Platform-independence	It depends …	Inherent in the API design
Consequence of API change	Recode, recompile, redeploy, etc.	Adjust mapping
Required resources	None	Additional middleware – CORBA, LiveKB

Cognitive Radio Framework



Demo setup



Control Knowledge Exchange
Remote K&B

Improving power efficiency

mSNR reading
(local meter)

Sender's power efficiency
(remote meter)

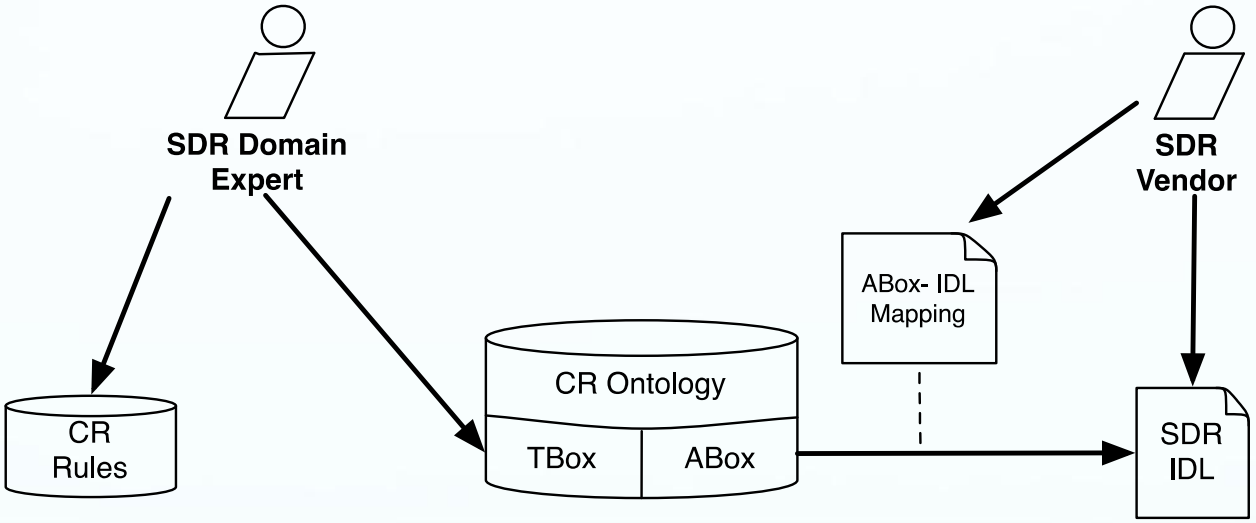
Request to alter sender's
Transmit Amplitude
(remote knob)

Future work

- Establish metrics for a quantitative evaluation of differences between LiveKB and traditional API
- Integrate LiveKB with SCA
- Use LiveKB on a different SDR platform and in a different self-controlling software domain
- Investigate alternatives to CORBA+IDL
- Implement a Protégé OWL Editor plugin for supporting mapping development (optional)

Thank you!

Roles and artifacts



- Establish ontology (WINNF Standard)
- Write rules

- Provide IDL
- Write mapping