April 2007

XG POLICY ARCHITECTURE

ICS-16763-TR-07-108
SRI Project No. 16763
Contract No. FA8750-05-C-0230

Prepared by

Daniel Elenius, Computer Scientist
Grit Denker, Sr. Computer Scientist
David Wilkins, Sr. Computer Scientist
Information and Computing Sciences Division

Prepared for

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

ACKNOWLEDGMENT OF SUPPORT AND DISCLAIMER:

This material is based upon work supported by the Defense Advanced Projects Agency and the United State Air Force under Contract Number FA8750-05-C-0230.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency and the United States Air Force.

TABLE OF CONTENTS

PURPOSE

This document is one of two RFCs (Requests For Comments) produced by SRI International (SRI) for the DARPA XG (neXt Generation) program. The other RFC focuses on the XG Policy Language CoRaL (Cognitive Policy Radio Language) proposed by SRI International.

This RFC describes the overall architecture in which XG radios operate, as well as the specifics of the interface between the *System Strategy Reasoner (SSR)* and the *Policy Reasoner (PR)*, and the language used in this interface. The SSR and the PR are software components running on each XG radio. The SSR sends transmission requests to the PR, and the PR returns decisions about the policy conformance of those requests. (These concepts are explained in more detail below.)

Conceptually, this document treats the SSR and the PR as black boxes, whose input-output behavior we are constraining to ensure successful communication between them.

This document has several groups of people as its intended audience. *Radio Software Designers* (people who write SSRs) will need to refer to the architecture and interface to develop XG-compliant radios. *Policy Reasoner* developers (SRI, for now) will similarly need to implement the other side of the interface.

This document does not discuss radio hardware and does not assume any previous knowledge about radio hardware. Readers should first read SRI's XG Policy Language RFC. That document includes useful background information on the XG project, as well as a description of the XGPL-2 Policy Language. The latter is useful because the language of the SSR-PR interface is a subset of XGPL-2.

Low-level hardware and communication details are not in the scope of this document. This document does not discuss how the SSR and PR are physically hooked up, what kind of hardware they are running on, whether they are running on the same or different CPUs and memory, or what communication protocol stacks are available. We envision that different radio builders will choose different solutions. This document does describe the syntax and meaning of the messages that are passed between these components.

1 XG ARCHITECTURE OVERVIEW

At the highest level of abstraction, an *XG radio* has four main components:

- **Sensors**. XG radios need sensors in order to discover available spectrum and transmission opportunities.
- **RF**. The XG radio obviously needs an RF component to transmit and receive.
- **SSR**. The SSR performs the main control of the radio's transmissions. It builds transmission requests based on sensor data received from the sensors and its current strategies. It also processes the replies to its transmission requests from the PR.
- **PR**. The PR accepts transmission requests from the SSR and checks policy conformance. It replies with yes, no, or no with constraints to be satisfied. The PR has a policy base with all *active* policies.

The hardware components, that is, the sensors and RF, do not concern us here. Our focus is the interface between the SSR and the PR.

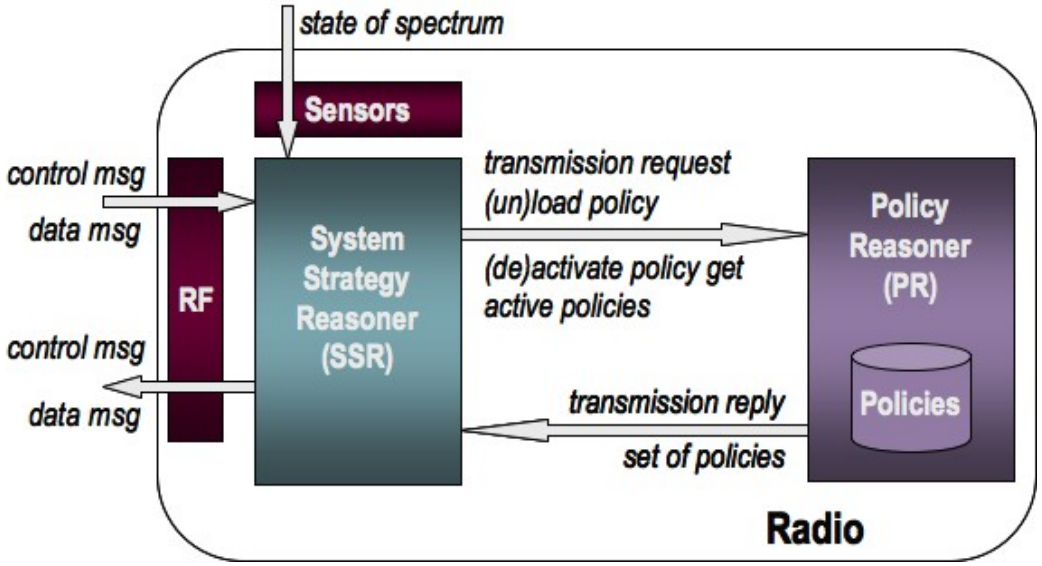

Figure 1. XG architecture. The red boxes are hardware components. The Policy DB is simply a set of policies – they probably will not be stored in a database system within the PR.

1.1 TYPES OF MESSAGES

As Figure 1 indicates, the XG Architecture has different types of messages.

- **RF-SSR**. All incoming messages to the XG radio arrive at the RF unit, and end up in the SSR. These messages can be *control* messages, such as updates to system strategies, updates to policies, or messages controlling the coordination with other radios. Similarly, all messages going out from the XG radio originate in the SSR and are passed through the RF component. Outgoing messages can also be control messages (e.g., acknowledgment of policy updates, requests for new control channels) or data messages.

- **Sensors-SSR**. The details of this interface will be determined by the radio designer. We assume that the sensors send their received data (or conclusions drawn from it) to the SSR. Analysis of sensor data, sensor data aggregation, signal detection, and so on could happen in the sensor component(s), in the SSR, or in a dedicated component (not shown). The SSR may send control messages to the sensor components.
- **SSR-PR**. There are several types of messages in the interface between the SSR and the PR. Before an XG radio can send a transmission, it needs to get approval from the PR. The SSR builds a transmission request, and sends it to the PR. The PR looks at the request and the active policies, and responds by sending one of three replies back to the SSR: (1) The transmission is allowed. The SSR must not transmit unless it has received a message of this type. (2) The transmission is not allowed. (3) The PR returns constraints that must be satisfied (CSE only, the request is underspecified). Given acceptable values of the underspecified request parameters, the transmission will be allowed. The SSR can also send *policy-update* messages to the PR, in order to add or remove policies to and from the PR's policy base and to activate or deactivate policies. (Only activated policies are used in the reasoning process.) Finally, the SSR can request information from the PR regarding which policies are loaded or active.

This document describes the SSR-PR interface in detail. We will not be further concerned with the RF-SSR and sensor-SSR interfaces.

The rest of this document is organized as follows. First, we will describe the SSR-PR Interface API. Following that, we describe transmission requests in detail, as they are key during radio usage. We first describe the *transmission request parameters*. These are the building blocks of the SSR-PR communication. Then, we will describe the request and response message types in more detail. We give the meaning and examples of complete SSR-PR interactions. Finally, we describe the policy-update message type.

2 SSR-PR INTERFACE API

The API presented here specifically addresses the SSR-PR interface. Additional PR API calls may be added later, to support authoring and management tools. The interface for this communication was developed jointly by SRI and Shared Spectrum Company (SSC).

The interface is described here in the form of an abstract API to the PR. When a policy, ontology, or request has been parsed, it is stored internally in a so-called abstract-syntax representation..

- **load** : policy or ontology -> nil. Loads a policy or ontology in the form of a string, a stream, a file pathname, or the abstract-syntax representation. The **use** closure is also loaded (i.e., dependencies defined by **use** relations on ontologies are automatically handled). When a policy or ontology is loaded, it is parsed (unless it is already in abstract syntax form) and type checked, and stored in an internal representation.
- **get loaded** : -> List of all loaded policies.
- **unload** : module -> nil. Unloads the given module.
- **unload all** : -> nil. Unloads all modules.
- **activate** : module -> nil. Activates the given module (and loads it first, if necessary).

Activating a module ensures its use in reasoning until the module is deactivated.

- **get active** : -> List of all active policies.
- **deactivate** : module -> nil. Deactivates one policy. It will still be loaded, and can be activated again without reloading.
- **deactivate all** : -> nil. Deactivates all policies.
- **request** : transmission request -> Result. Processes the supplied request (in the form of a string, a stream, or a file pathname) and returns the result, along with information on which policies allowed and disallowed the request, and some timing information.

We note that there is some complexity in doing unload and deactivate, given that some policies import ("use") other policies (see the XG Language RFC). Loading a policy implies that all imported policies will also be loaded. Thus, the PR must keep a dependency graph that tracks all the reasons a particular policy was loaded or activated. Policy X can be unloaded or deactivated only when all policies that included Policy X have been deactivated or unloaded.

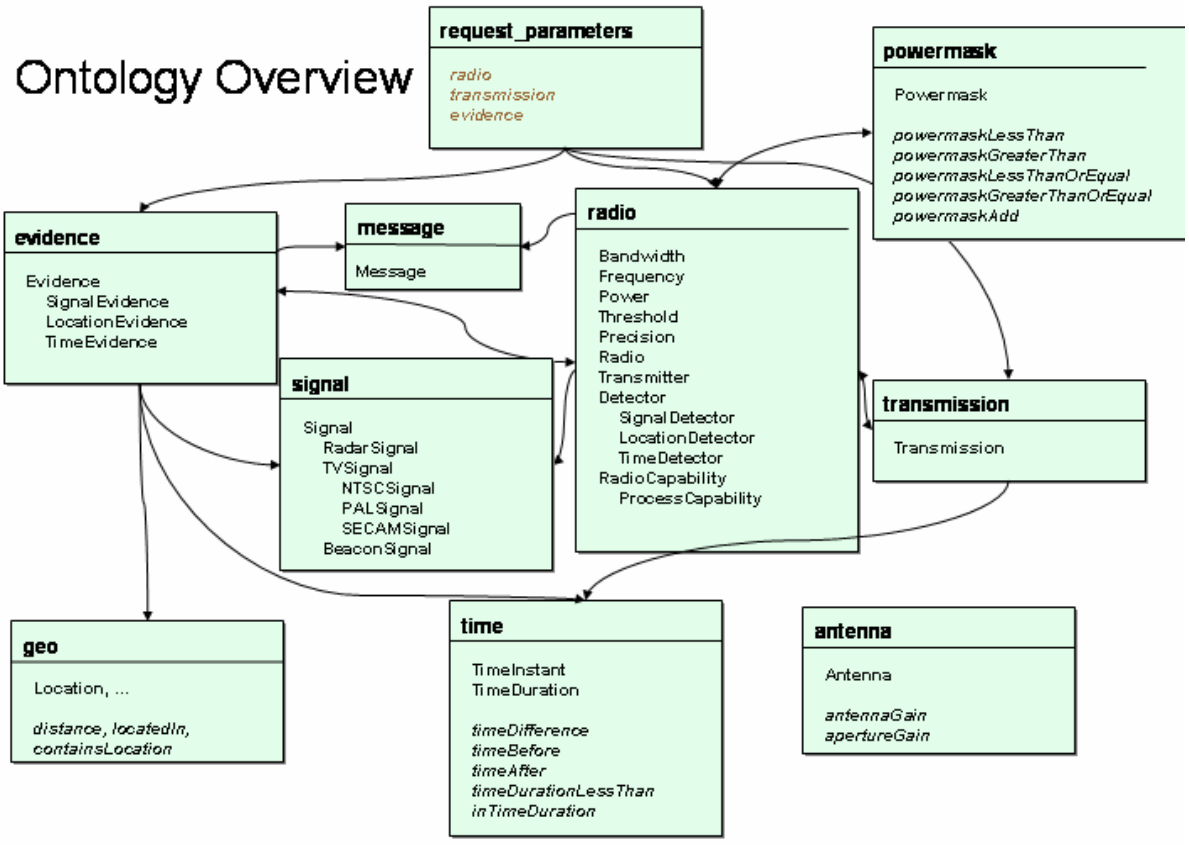

Figure 2. Basic XG ontologies. See Appendix A.1-A.8 for full details.

3 TRANSMISSION REQUEST PARAMETERS

The SSR and the PR share knowledge of the ontology of domain concepts (e.g., *Frequency*, *Power* – see Figure 2), and request parameters. The *XG Language RFC* describes the domain concepts of the standard ontologies of the XG Policy Language. This section describes the predefined request parameters, which are assumed to be recognized by every XG-enabled device. These are three special predicates, defined in *request_params.xg* (see Appendix A.1):

public const evidence : Pred(Evidence);

evidence(e) means that e is an evidence object that the SSR presents to the PR. See the evidence.xg file (Appendix A.2). This is a predicate, which means that the SSR can present multiple evidences to the PR.

public const transmission : Transmission;

transmission(t) means that t is the transmission that the SSR wants to send. See the transmission.xg file (Appendix A.3).

public const radio : Radio;

radio(r) means that r is the radio requesting to transmit. See the radio.xg file (Appendix A.4).

Note that the value of these three predicates can be different for each request. This sets them apart from all other entities in the language.

3.1 ADDING NEW REQUEST PARAMETERS

New request parameters can be defined. We cannot anticipate all possible future uses of XG radios and the XG Policy Language, and some users and/or regulators may have specific needs that could not, and should not, be part of the common knowledge encoded in the basic ontology of request parameters. Any request parameters used in transmission requests or replies must be declared. Syntactically, declaring request parameters is no different from declaring other constants. The difference is that request parameters have a special meaning when used in requests. The XG Language RFC gives the syntax for defining new request parameters.

3.2 SUPPORTING REQUEST PARAMETERS

Each request parameter has certain extra-logical conditions stating its *intended meaning*. For an SSR to *support* (or understand) a request parameter, it must be able to evaluate those conditions. For example, supporting the *evidence* parameter requires the SSR to know what an evidence is, and under what conditions it is allowed to present evidences to the PR. The notion of supporting request parameters is not relevant for the PR – the PR can express constraints on any request parameters that have been defined, and the constraints will, by definition, have the intended meaning.

3.3 UNRECOGNIZED REQUEST PARAMETERS

There are two types of situations to consider. Both are handled in a manner consistent with our design goal of “do no harm”.

1. The SSR uses a parameter in a request and no active policy makes use of it. This may happen when the SSR introduces a new parameter, or hopes to use policies that happen to be inactive. In this case, no harm is done and the parameter is ignored.
2. The PR uses a request parameter that the SSR does not support. This may happen if the radio is not capable of fully utilizing all opportunities provided by XG policies, or if a new parameter has been defined. Constraints on this parameter will fail, and the PR returns either "no" or additional constraints (see next section). Additional constraints must be interpreted as a "no" until they can be understood and satisfied.

4 TRANSMISSION REQUEST-REPLY INTERACTIONS

Before an XG radio can send a transmission, it must get approval from the PR. The SSR builds a transmission request, and sends it to the PR. The PR looks at the request and the policies loaded into its policy database. There are three possible outcomes:

1. The request is allowed. The PR sends back an ok-to-transmit message in the transmission reply.
2. The request is not allowed. The PR sends back a don't-transmit message in the transmission reply.
3. The request is underspecified. Given acceptable values of the underspecified or unspecified request parameters, the request will be allowed. The PR sends back constraints, which describe the possible ways of getting approval, i.e. the values of the request parameters that will result in an ok-to-transmit reply.

The SSR *must not* transmit unless it has received a message of type 1. Messages of type 3 *do not* license any transmissions. They merely provide information to the SSR on how to improve the request (if the SSR so desires) in order to later receive a message of type 1. Messages of type 3 do not guarantee that the PR will grant a later request, because the set of policies loaded on the PR could change between the type 3 message and the next request.

4.1 TRANSMISSION REQUEST FORMAT

A request is a set of formulas expressing constraints over request parameters.

For example consider the following request:

```
request R1 is
    (exists L:LocationEvidence)
        evidence(L) and
        location(L, some value);

    frequency(transmission)=5500;
    peakTxPower(transmission) =< 30.0;

    (exists T:Transmitter)
        transmitter(radio)=T and
        inBandLeakage(T)=step[(0,-30),(20,-40),(30,-50)];
end
```

```
request IDENTIFIER is

constraint_1 ;
constraint_2 ;
.
.
constraint_n ;

end
```

Figure 1

This request gives concrete values for the *frequency* and *inBandLeakage*. For *peakTxPower*, a looser constraint is given.

Requests cannot include type, const, def, deftype, defconst statements.

4.2 TRANSMISSION REPLY FORMAT

The three types of transmission replies (see above) are expressed in the forms given in Figure 2:

reply IDENTIFIER is request_ok. end	reply IDENTIFIER is request_rejected. end	reply IDENTIFIER is FO_Formula end

Figure 2

The IDENTIFIER of the reply is always the same as the IDENTIFIER in a preceding request message, as defined above. That is, a reply is always a reply to a request. Request and reply identifiers are assumed to be unique within the operation of a specific XG radio.

In the case of a type (3) reply, the listed constraints will not include the constraints given in the preceding request. The set of constraints under which the transmission is allowed is thus the union of the constraints in the preceding request, and those in the reply.

The following example shows a type (3) scenario, where the reply consists of two sets of constraints, either one of which can be satisfied by the SSR to obtain approval for a request.

```
reply R1 is

    (frequency(transmission) in {5000..5500} and
    peakTxPower(transmission) =< 25.0) or

    (frequency(transmission) in {4500..5200} and
    peakTxPower(transmission) =< 30);

end
```

4.3 INTERACTION EXAMPLE

We will now look at an example of a complete SSR-PR interaction. Three policies (*P1*,*P2*, *P3*) are loaded onto the PR:

```
policy P1 is
        use radio,transmission;

        defconst maxInBandLeakage : Powermask = linear [(..) ,(..)];

        const frequency_ok : Pred;
        const power_ok : Pred;
        const in_band_leakage_ok : Pred;

        allow if
                frequency_ok and
                power_ok and
```

```
            in_band_leakage_ok;

        frequency_ok if carrierFrequency(transmission) in {5000..5500};
        power_ok if peakTxPower(transmission) =< 25.0;
        in_band_leakage_ok if
            (exists T:Transmitter)
                transmitter(transmission)=T and
                powermaskLessThanOrEqual(inBandLeakage(T),maxInBandLeakage);
end

policy P2 is
        use radio,geo,evidence;

        type TvStation;
        const stationLocation : TvStation -> Location;
        const gradeBContour : TvStation -> GeoBorder;
        type City;
        const cityLocation : City -> Location;

        const station1 : TvStation;
        location(station1)=…;
        gradeBContour(station1)=…;
        const station2 : TvStation;
        location(station1)=…;
        gradeBContour(station1)=…;
        const city1:City;
        cityLocation(city1)=…;
        const city2:City;
        cityLocation(city2)=…;
        defconst restrictedCities : [City] = [city1,city2,…];

        const frequency_ok : Pred;
        const interference_ok : Pred;
        const location_ok : Pred;

        allow if frequency_ok and interference_ok and location_ok;

        frequency_ok if frequency(transmission) in {4000..5200};

        interference_ok if
            (forall S : Signal in [fullPowerAnalogTV, fullPowerDigitalTV, …])
                (exists E:SignalEvidence)
                    evidence(E) and
                    detectedSignal(E,S) and
                    lastDetected(E) >= 100;

        location_ok if
            (exists E:LocationEvidence)
                evidence(E) and
                (forall S : TvStation)
                    distance(location(E), stationLocation(S)) >= 200 and
```

```
                    (forall C : City)
                        C in restrictedCities and
                        distance(location(E), cityLocation(C)) >= 80;
end

policy P3 is
    use radio,transmission;
    disallow if carrierFrequency(transmission) in {4500..5200};
end
```

These three policies are not meant to suggest a certain style of writing policies. Refer to the XG Policy Management RFC for such issues. We will now look at a series of requests and replies, which are presented in the order they occur:

```
request R1 is
    peakTxPower(transmission) >= 20;
end

reply R1 is
    (carrierFrequency(transmission) in {5200..5500} and
    peakTxPower(transmission) =< 25.0 and
    (exists T:Transmitter)
        transmitter(transmission)=T and
        powermaskLessThanOrEqual(inBandLeakage(T), linear[…])) or

    (carrierFrequency(transmission) in {4000..4500} and

    (exists E:SignalEvidence)
        evidence(E) and
        signal(E)=fullPowerAnalogTV and
        lastDetected(E) >= 100 and

    (exists E:SignalEvidence)
        evidence(E) and
        signal(E)=fullPowerDigitalTV and
        lastDetected(E) >= 100 and
    …

    (exists E:LocationEvidence)
        evidence(E) and
        distance(location(E), value of station1's location) >= 200 and
        distance(location(E), value of station2's location) >= 200 and
        ...
        distance(location(E), value of city1's location) >= 80 and
        distance(location(E), value of city2's location) >= 80 and
        …
    )
end
```

The first reply from the PR is a result of

1. Eliminating rules.
2. Expanding finite quantifiers.
3. Simplifying constraints (look at the frequency ranges—the restrictive policy P3 has taken away a chunk from the ranges of P1 and P2).

Note that we do not currently prescribe a certain form (normal form) for the reply. In the current case, we have a disjunction at the top level, with each disjunct corresponding to one policy (the first one to P1 and the second one to P2).

Suppose the SSR does not know about its inBandLeakage. It knows its location, but it does not know how to compute the distance function. It performs some sensing actions (or consults its history of previous sensing results), and then forms the new request:

```
request R2 is
        carrierFrequency(transmission) in {4000..4500} and
        (exists E:SignalEvidence)
                evidence(E) and
                signal(E)=fullPowerAnalogTV and
                lastDetected(E) >= 100 and

        (exists E:SignalEvidence)
                evidence(E) and
                signal(E)=fullPowerDigitalTV and
                lastDetected(E) >= 100 and

        … (and so on for all the signals listed in the policy)

        (exists E:LocationEvidence)
                evidence(E) and
                location(E) = some value;
end
```

Suppose the distance functions evaluates to true. Then the PR will respond with

```
reply R2 is
        request_ok
end
```

Again, it would be useful to have a special category for “transmission parameters” here. After this final reply, the SSR needs to look at only the transmission parameters in its last request, to see the constraints that are relevant to its impending transmission. In this case, only the constraint on carrierFrequency is relevant. The other constraints can be forgotten at this point.

4.4 SSR STRATEGIES

Given the open-ended nature of the SSR-PR interaction, how should the SSR behave? What should it put into its first request? The XG Architecture does not prescribe any answers to these questions. We envision a broad range of XG radio devices, where some may have a very

sophisticated SSR component, and others may have a relatively ‘dumb’ SSR.

A ‘dumb’ SSR might put only the requested carrier frequency and maximum power in the request, and hope for direct approval. If any additional constraints are returned, the dumb SSR could just try another frequency from a built-in table, and so on until the request is approved, or the frequency table is exhausted and the SSR gives up.

A sophisticated (i.e., more ‘cognitive’) SSR might try to get its hands on as much spectrum as possible, by constructing a request that satisfies the additional constraints returned by the PR. This could involve performing sensing actions for certain parameters.

In some conditions, the SSR-PR interactions will consist of just one request and one reply indicating either “okay” or “not-okay” for the requested transmission candidate. This will be the case if the SSR is ‘aware’ of the requirements of the policies loaded onto the PR. The SSR might be aware because it has ‘learned’ from previous interactions, or because it is simply preprogrammed to work with specific policies.

A more common case, however, would be that the PR returns some additional constraints to be satisfied. If the SSR is not aware of the policies, it would need to tell the PR everything it knows, and perform all possible sensing actions, in order to get a direct yes/no reply. This is clearly impossible.

Note that the SSR can send an empty request to the PR. The reply, in effect, will be the whole policy database. Underspecified requests correspond to database queries. This kind of query would result in a very large answer set. A smarter strategy would be to bind or constrain at least some parameters before the query is sent.

APPENDIX A.1

```
policy request_params is
    use radio,evidence,transmission;

    /* evidence(e) means that e is an evidence object that the SSR
       presents to the PR. */
    public const evidence : Pred(Evidence);

    /* transmission is the transmission that the SSR wants to do */
    public const transmission : Transmission;

    /* radio is the device  requesting to transmit */
    public const radio : Radio;
end
```

APPENDIX A.2

```
policy transmission is
    use time,radio;

    type Transmission:
    const carrierFrequency : Transmission -> Frequency;
    const bandwidth : Transmission -> Bandwidth;
    const minOnTime : Transmission -> TimeDuration;
    const minOffTime : Transmission -> TimeDuration;
    const peakTxPower : Transmission -> Power;
    const transmitter : Transmission -> Transmitter;
end
```

APPENDIX A.3

```
policy evidence is
    use time,geo,radio,signal,message;

    public type Evidence;
    public const timeStamp : Evidence -> TimeInstant;
    public const detectedBy : Evidence -> Detector;

    public subtype SignalEvidence < Evidence;
    public const lastDetected : SignalEvidence -> TimeInstant;
    public const lastCompleteEmptyScanTime : SignalEvidence -> TimeInstant;
    public const lastCompleteEmptyScanDuration : SignalEvidence -> TimeDuration;
    public const peakRxPower : SignalEvidence -> Power;
    /* N.B: A SignalDetector always senses for the signals that it can sense, according
    to its can_sense property */
```

```
    public const detectedSignal : Pred(SignalEvidence,Signal);
    public const sensingThreshold : SignalEvidence -> Threshold;
    public const scannedFrequencies : SignalEvidence -> {Frequency};

    public subtype LocationEvidence < Evidence;
    public const location : LocationEvidence -> Location;

    public subtype TimeEvidence < Evidence;

    public subtype MessageEvidence < Evidence;
    public const message : Pred(MessageEvidence,Message);

end
```

APPENDIX A.4

```
policy radio is
    use evidence,transmission,message,signal,powermask;

    /* These could be ADTs if one wants more type safety */
    public deftype Bandwidth = Float;
    public deftype Frequency = Float;
    public deftype Power = Float;
    public deftype Threshold = Float;
    public deftype Precision = Float;

    public type Detector;
    public const threshold : Detector -> Threshold;
    public const precision : Detector -> Precision;
    public const presentsEvidence : Pred(Detector,Evidence);
        /* inverse of evidence.detectedBy */

    public subtype SignalDetector < Detector;
    public const canSense : Pred(SignalDetector,Signal);

    public subtype ContinuousSignalDetector < SignalDetector;
    public subtype PeriodicSignalDetector < SignalDetector;
    public dutyCycle : Pred(PeriodicSignalDetector,Float);   /* ratio of on-off time */
    public sampleRate : Pred(PeriodicSignalDetector,Float);
    /* Note: Much more could be added on sensing, e.g. exponential backoff */

    public subtype TimeDetector < Detector;
    public subtype LocationDetector < Detector;

    public type Transmitter;
    public const inBandLeakage : Transmitter -> Powermask;
    public const outOfBandLeakage : Transmitter -> Powermask;
    public const precision : Transmitter -> Precision;
    public const setupForTransmission : Transmitter -> Transmission;
```

```
        /* inverse of transmission.transmitter */

    public type MessageDetector < Detector;
    public const spuriousEmissions : MessageDetector -> Powermask;
    public const canReceive : MessageDetector -> Message;

    public type RadioCapability;
    public type ProcessCapability;
    public subtype ProcessCapability < RadioCapability;

    public type Radio;
    public const transmitter : Pred(Radio,Transmitter);
    public const receiver : Pred(Radio,Receiver);
    public const detector : Pred(Radio,Detector);
    public const capability : Pred(Radio,RadioCapability);

end
```

APPENDIX A.5

```
policy geo is

    /* Abstract Data Types */

    public type Location;
    public type LocationCode;
    public type InstallationTypeCode;
    public type CountryCode;
    public type StateCode;
    public type ProvinceCode;
    public type TacticalZone;
    public type PlanCode;
    public type ICAOCode;
    public type ArmyLocationCode;
    public type NavyLocationCode;
    public type GSACityCode;
    public type GSAStateCode;
    public type CountryName;
    public type StateName;
    public type ProvinceName;

    public const locationCode : Location -> LocationCode;
        /* Every location has a unique 4 char code */
    public const installationTypeCode : Location -> InstallationTypeCode;
    public const countryCode : Location -> CountryCode;
    public const stateCode : Location -> StateCode;
    public const countryName : Location -> CountryName;
    public const stateName : Location -> StateName;
    public const provinceCode : Location -> ProvinceCode;
```

```
public const provinceName : Location -> ProvinceName;
public const tacticalZone : Location -> TacticalZone;
public const latitude : Location -> Float;
public const longitude : Location -> Float;
public const altitude : Location -> Float;
public const planCode : Location -> PlanCode;
      /* Army logistics plan code */
public const icaoCode : Location -> ICAOCode;
      /* International Civil Aviation Org Code */
public const armyLocationCode : Location -> ArmyLocationCode;
public const navyLocationCode : Location -> NavyLocationCode;
public const gsaCityCode : Location -> GSACityCode;
public const gsaStateCode : Location -> GSAStateCode;

public type GeographicArea;
public deftype LocationList = [Location];

/* Operations */
public const locatedIn : Location,GeographicArea -> Bool;
public const containLocations : GeographicArea,LocationList -> Bool;
public const distance : Location,Location -> Float;

/* Country codes */
public const ac : CounryCode; /*  Ascension Island
public const ad : CounryCode; /*  Andorra
public const ae : CounryCode; /*  United Arab Emirates
public const af : CounryCode; /*  Afghanistan
public const ag : CounryCode; /*  Antigua and Barbuda
public const ai : CounryCode; /*  Anguilla
public const al : CounryCode; /*  Albania
public const am : CounryCode; /*  Armenia
public const an : CounryCode; /*  Netherlands Antilles
...

/* State Codes */
public const AL : StateCode; /*ALABAMA*/
public const AK : StateCode; /*ALASKA*/
public const AS : StateCode; /*AMERICAN SAMOA*/
public const AZ : StateCode; /*ARIZONA*/
public const AR : StateCode; /*ARKANSAS*/
public const CA : StateCode; /*CALIFORNIA*/
public const CO : StateCode; /*COLORADO*/
public const CT : StateCode; /*CONNECTICUT*/
public const DE : StateCode; /*DELAWARE*/
...

/* Installation Type Codes */
public const AFD : InstallationTypeCode ; /* AIRFIELD */
public const AFS : InstallationTypeCode ; /* AIR-FORCE-STATION */
public const APT : InstallationTypeCode ; /* AIRPORT*/
public const ASN : InstallationTypeCode ; /* AIR-STATION*/
```

```
public const IAP : InstallationTypeCode ;  /* INTERNATIONAL-AIRPORT*/
public const JAP : InstallationTypeCode ; /* JOINT-USE-AIRPORT*/
public const MAP : InstallationTypeCode ;/* MILITARY-AIRPORT*/
public const BAY : InstallationTypeCode ; /* BAY*/
public const CHL : InstallationTypeCode ; /* CHANNEL*/
public const CNL : InstallationTypeCode ; /* CANAL*/
public const CPE : InstallationTypeCode ; /* C APE*/
public const DOC : InstallationTypeCode ;          /* DOCK*/
public const GLF : InstallationTypeCode ; /* GULF*/
public const LKE : InstallationTypeCode ; /* LAKE*/
public const OCN : InstallationTypeCode ;          /* OCEAN*/
public const PRT : InstallationTypeCode ; /* PORT*/
public const PSG : InstallationTypeCode ;/* PASSAGE*/
public const SEA : InstallationTypeCode ; /* SEA*/
public const STR : InstallationTypeCode ; /* STRAIT*/
public const ADM : InstallationTypeCode ;          /* ADMINSTRATION*/
public const ANX : InstallationTypeCode ;/* ANNEX*/
public const ATM : InstallationTypeCode ;/* AIR-TERMINAL*/
public const COM : InstallationTypeCode ;          /* COMMUNICATION*/
public const RRC : InstallationTypeCode ;/* RADAR-RECEIVER*/
public const RRJ : InstallationTypeCode ; /* RAILROAD-JUNCTION*/
public const RRL : InstallationTypeCode ; /* RADIO-RELAY*/
public const RTC : InstallationTypeCode ; /* RESERVE-TRAINING-CENTER*/
public const RTR : InstallationTypeCode ; /* RADAR-INSTALLATION*/
public const MFC : InstallationTypeCode ;/* MAINTENANCE*/
public const SVC : InstallationTypeCode ; /* SERVICE*/
public const TNG : InstallationTypeCode ;/* TRAINING*/
public const WAE : InstallationTypeCode ;          /* WEATHER-STATION*/
public const CAP : InstallationTypeCode ; /* CIVIL-AIR-PATROL*/
public const CGI : InstallationTypeCode ;  /* COAST-GUARD-INSTALLATION*/
public const COC : InstallationTypeCode ;          /* COMMAND-OPERATIONS*/
public const AGS : InstallationTypeCode ;/* AIR-NAT-GUARD-STATION*/
public const AIN : InstallationTypeCode ;  /* ARMY-INSTALLATION*/
public const MBK : InstallationTypeCode ;/* MARINE-BARRACKS*/
public const MCC : InstallationTypeCode ;          /* MARINE-CORPS-CAMP*/
public const MGI : InstallationTypeCode ; /* MARINE-GROUND-INSTALLATION*/
public const NBA : InstallationTypeCode ; /* NAVAL-BASE*/
public const NYI : InstallationTypeCode ;  /* NAVY-INSTALLATION*/
public const AMO : InstallationTypeCode ;          /* AMMUNITION-STORAGE*/
public const DFP : InstallationTypeCode ; /* DEFENSE-FUEL-SUPPORT-POINT*/
public const POL : InstallationTypeCode ; /* POL-RETAIL-DISTRIBUTION-STATION*/
public const STG : InstallationTypeCode ; /* STORAGE*/
public const DEP : InstallationTypeCode ; /* DEPOT*/
public const FHG : InstallationTypeCode ;/* FAMILY-HOUSING*/
public const REC : InstallationTypeCode ;/* RECREATION*/
public const SCH : InstallationTypeCode ;/* SCHOOL*/
public const HSP : InstallationTypeCode ;/* HOSPITAL*/
public const DIS : InstallationTypeCode ;  /* DISPENSARY*/
public const CLN : InstallationTypeCode ; /* CLINIC*/
public const CTY : InstallationTypeCode ; /* CITY*/
public const ISL : InstallationTypeCode ;  /* ISLAND*/
```

```
    public const OPA : InstallationTypeCode ;/* OPERATING AREA*/
    public const RPA : InstallationTypeCode ;/* RURAL POPULATED AREA*/
    public const MSL : InstallationTypeCode ;/* MISSILE*/
    public const NAC : InstallationTypeCode ;/* NAVAL ACTIVITY*/
    public const NAV : InstallationTypeCode ;/* NAVAID*/

end
```

APPENDIX A.6

```
policy time is
    /* Types */
    public type TimeInstant;
    public const year : TimeInstant -> Int;
    public const month : TimeInstant -> Int;
    public const day : TimeInstant -> Int;
    public const hour : TimeInstant -> Int;
    public const minute : TimeInstant -> Int;
    public const second : TimeInstant -> Int;
    public const millisecond : TimeInstant -> Int;

    public type TimePeriod;
    public const startTime : TimePeriod -> TimeInstant;
    public const endTime : TimePeriod -> TimeInstant;

    public type TimeDuration;
    public const years : TimeDuration -> Int;
    public const months : TimeDuration -> Int;
    public const days : TimeDuration -> Int;
    public const hours : TimeDuration -> Int;
    public const minutes : TimeDuration -> Int;
    public const seconds : TimeDuration -> Int;
    public const milliseconds : TimeDuration -> Int;

    /* Operations */
    public const timeDifference : TimeInstant,TimeInstant -> TimeDuration;
    public const timeBefore : TimeInstant,TimeInstant -> Bool;
    public const timeAfter : Pred(TimeInstant,TimeInstant -> Bool;
    public const timeDurationLessThan : TimeDuration,TimeDuration -> Bool;
    public const inTimeDuration : TimeInstant,TimeDuration -> Bool;

end
```

APPENDIX A.7

```
policy signal is
    /* A signal is just an enumerated abstract entity. It is really a waveform, but
    we do not really need to say anything about the waveform, except to identify it */
    public type Signal;

    /* Perhaps these should be consts? */
    public subtype RadarSignal < Signal;
    public subtype TVSignal < Signal;
    public subtype NTSCSignal < TVSignal;
    public subtype PALSignal < TVSignal;
    public subtype SECAMSignal < TVSignal;
    public subtype BeaconSignal < Signal;
end
```

APPENDIX A.8

```
policy message is

    /* This represents types of messages. For example, an instance might be
    okToTransmit from DFS. This would be the type of all okToTransmit messages.
    We do not need to talk about instances of messages.

    Not entirely clear what the relationship between messages and signals is.
    A message is always conveyed by a signal? In that case, we could put properties
    Signal conveys Message, Message conveyedBy Signal. But then, do we need
    MessageDetectors? */

    public type Message;
end
```