September 2006

CoRaL POLICY VALIDATION ENGINE AND POLICIES

ICS-17091-TR-06-001
SRI Project No. 17091
Subcontract No. 2005-11
Prime Contract No. FA8750-05-C-0150

Prepared by

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

Prepared for

Shared Spectrum Company
1595 Spring Hill Road, Suite 110
Vienna, VA 22182-2228

TABLE OF CONTENTS

1. INTRODUCTION

This report documents the CoRaL (*Co*gnitive *Ra*dio *L*anguage) policy reasoner, architecture and implementation, and some sample policies. The reasoner described here is a Prolog-based version, which gives yes/no answers to transmission requests. We refer to this early prototype as the Candidate Validation Engine (CVE). We will soon extend it with a Candidate Search Engine (CSE) capability, which returns information on possible transmission opportunities. Both functionalities together make up the CoRaL Policy Reasoner. In this document, it is generally not necessary to distinguish between the CVE and CSE, in which case we refer to the current CVE reasoner as just "the Policy Reasoner" (PR).

The report is divided into the following sections: Section 2 gives a quick overview of the CoRaL language. The authoritative reference to CoRaL is the Language RFC [1]. Section 3 describes the XG system architecture of which the CoRaL Policy Reasoner is one component. In particular, it describes the details of the API, which is used to communicate with the PR. The XG architecture is one of many possible architectures for using the CoRaL language. The authoritative guide to the XG Architecture is the XG Architecture RFC [2]. Section 4 describes the current implementtation of this architecture and language.

Section 5 serves as a "user guide," providing more practical details on using the system. It provides the hardware and software requirements of the system, installation instructions, and a guide to the *Policy Console* – a useful command-line tool for testing policies and requests. Three appendices give the CoRaL grammar, the "standard" ontologies, and sample policies.

2. LANGUAGE

CoRaL is a *declarative* language based on a *typed* version of *classical first-order logic*. The following three sections describe the language in informal terms, with examples. Many examples of ontologies and policies written in CoRaL can be found in the appendices. We describe the language in a bottom-up fashion, starting with the smaller building blocks, and ending with entire policies.

2.1 TYPES

CoRaL has a static type system. This allows many errors to be detected during development time, instead of later, when corrections can be significantly more costly. Besides the syntax of type expressions, there is a system of typing rules, which can be found in the Language RFC [1]. These rules have been implemented in the PR, and executing these rules is referred to as "type checking". If a policy is not "type correct" (i.e., it fails type checking), it is considered meaningless. In other words, the semantics of CoRaL applies only to *type-correct* (or "well-typed") policies.

A CoRaL type represents a *set*. The members of the type are the elements in the set. For example, the type **Int** represents the set of all integers; the type **Int->Int** represents the set of all functions from integers to integers, and so on.

Besides the types introduced using declarations (see Section 2.4 Statements), CoRaL has the following *built-in types*:

- **Float**, representing arbitrary-size floating point numbers
- **Int**, representing arbitrary-size integers
- **Bool**, representing the truth values true and false

From the built-in types, and declared types, one can then construct types using the following type operators:

- **[]** : Lists[1]. For example, **[Int]** is the type of lists of integers.
- **{}** : Sets. Sets can be specified only for **Int** and **Float**, and their elements are always continuous. For example, **{Int}** is the type of sets of continuous integers. One member of the type **{Int}** is **{3,4,5}**.
- **->** : Functions. For example **Int -> Int** is the type of functions from **Int** to **Int**.
- **Pred** : Predicates. For example, **Pred(Int,Int)** is the type of binary predicates, where both arguments are Ints.
- **()** : Tuples. For example, **(Int,Int)** is the type of pairs of two **Int**s.

These operators can be nested in certain ways. The only restriction is that predicates and functions can occur only at the top level. So, **[{(Int,Int)}]** is a valid type (of lists of sets of tuples of two **Int**s), but **[Int->Int]** is not.

2.2 TERMS

Terms are entities representing *values*. They can be

- Floating point or integer numerals, such as **17** or **3.14**
- Arithmetic expressions, such as **17 + 3.14 – (11 / 3)**
- Variables, such as **?x, ?f** (variables are prefixed by '?')
- Function applications, such as **factorial(5)**[2]
- Constants
- Lists, such as **[1,2,3]**
- Tuples, such as **(1,2)**
- Sets, such as **{1..5}**. The dots represents all values between the two end points. It is possible to define sets with an infinite number of elements, e.g., **{0.0 .. 1.0}**.

For arithmetic expressions, parentheses can be used to define precedence. If no parentheses are present, the connectives bind in the following order, from strongest to weakest binding: unary **-**, ***** and /, + and binary **-**.

Functions are applied to terms, so terms can be arbitrarily nested; for example, **factorial(max(x,5))** takes the factorial of the maximum of the variable x and 5. **max(x,5)** is itself

[1] Note that while many languages specify lists using the language itself, in CoRaL they are built in, because our type system is not polymorphic.

[2] In fact, arithmetic operators, like + and **-**, are also (infix) functions. Arithmetic expressions are thus really function applications as well. For example, **17 + 3.14 – (11/3)** is the (nested) function application **add(17,sub(3.14,div(11,3)))**

a (function application) term, as are **x** and **5**.

Note that the last three kinds of terms have the same syntax as the corresponding type expression (**[1,2,3]** is of the type **[Int]**, **(1,2)** is of the type **(Int,Int)**, and **{1..5}** is of the type **{Int}**). Just like type expression can be nested, so can these terms. For example, **[(1,2),(3,4)]** is the list containing the two tuples **(1,2)** and **(3,4)**.

Constants are introduced by constant declarations (see Section 2.4 Statements).

2.3 FORMULAS

Formulas are built up from atomic formulas, *boolean connectives*, and quantifiers. There are three kinds of atomic formulas:

- "Standard" atomic formulas, consisting of a predicate constant, and terms for all its arguments. For example, given that we have defined a constant **const p : Int**, then **p(17)** is an atomic formula.
- Constraint formulas. The usual inequalities are supported: <, >, =<, and >=. These are written in infix form. So, **x < 17** is a constraint formula. There is also a constraint operator **in**, which serves several purposes:
  - **17 in [17,23,32]** : Returns true iff the lhs term is in the list on the rhs (in this case true).
  - **17 in {10 .. 20}** : Same thing but with a set on the rhs (again, true in this example).

  The **in** operator works for **Float**s and **Int**s.
- Equalities. Also in infix form, e.g., **x = 17**.

Note that the (in)equalities are really predicates in infix form (similar to how the arithmetic operators are really functions in infix form). We do not have any way for the user to introduce new infix symbols, but we do provide the common ones given above, to enhance the readability of policies.

Atomic formulas can be combined into bigger formulas using boolean connectives and quantifiers. The connectives are

- **and** : Conjunction. For example, **p(x) and x < 17**.
- **or** : Disjunction. For example, **p(x) or x < 17**.
- **not** : Negation. For example, **not p(x)**.
- **implies** : Implication. For example, **p(x) implies x < 17**.

These all have their usual meaning. For example, the last one is true iff **p(x)** is false or **x < 17** is true.

Formulas can be nested arbitrarily, as in for example **p(x) or q(x) implies r(x)**. As with arithmetic expressions, parentheses can be used to define precedence. If no parentheses are present, the connectives bind in the following order, from strongest to weakest binding: **not**, **and**, **or**, **implies**.

Quantifiers are a bit more complicated. The type of each variable must be given, and there is also an optional '**in**' constraint formula on each variable.

- **exists**: Existential quantification. For example, (**exists ?x,?y : Float, ?z : Int in {1 .. 10}) ?x + ?y < ?z.**

- **forall**: Universal quantification. For example, (**forall ?x,?y : Float, ?z : Int in {1 .. 10}) ?x + ?y < ?z.**

The scope of the quantified formula extends as far to the right as possible. For example,

```
(forall ?x:Int)
 (exists ?y:Int)
  ?y > ?x
```

is a valid formula. If a more limited scope is intended, use parentheses, as in

```
((exists ?se : SignalEvidence) p(?se)) or
((exists ?te : TimeEvidence) q(?te))
```

Without the extra parentheses, we would have

```
(exists ?se : SignalEvidence) p(?se) or
(exists ?te : TimeEvidence) q(?te)
```

which is equivalent to

```
(exists ?se : SignalEvidence) (p(?se) or (exists ?te : TimeEvidence) q(?te))
```

which clearly has a different meaning (in the first version, there does not have to exist a **SignalEvidence**).

Regarding the constraints for quantified variables, it might appear at first glance that, e.g., **(forall x : Int in {1 .. 10})** is the same as **(forall x : Int) x in {1 .. 10}**. Semantically, they *are* the same. The reason for having the first form is that the universal quantifier can be eliminated, by trying all possible values for **x**. This is much more difficult with the second form, where the constraint may be hidden inside some bigger formula.

2.4 STATEMENTS

On the top-level of a policy, ontology, or request (see below), the valid kinds of statements are

- Type declarations, e.g., **type Radio;** These introduce new types.
- Type definitions, e.g., **deftype Frequency = Float**; These introduce new names for existing types. The rhs can be any type of expression.
- Subtype declarations, e.g., subtype **SignalDetector < Detector;**
- Constant declarations, e.g., **const r : Radio;**
- Constant definitions, e.g., **defconst frequencies : [Frequency] = [5000, 5500, 6000];**
  The righthand side can be any term of the given type.

All of the above can be prefixed with a **public** keyword, which makes the declared or defined entity available to ontologies or policies that **use** the current one (see below).

- Use statements, e.g., **use evidence;**
  These specify ontologies that are used by the current ontology or policy. (Policies cannot currently **use** other policies, although this is something being considered for future versions of CoRaL). **use** is transitive. That is, if A uses B, and B uses C, there is no need for A to use C – all the **public** statements in C are already visible in A.
- Rules. Rules make up the bulk of most policies. Rules have a head, which must be an atomic

formula, optionally followed by an **if** keyword, followed by a body, which can be an arbitrary formula. If the head contains variables, the rule must be prefixed by a **forall** quantifier for those variables. The meaning of a rule is an implication from the body to the head. There are two kinds of rules:

- "Standard" rules. These have a "standard" atomic formula as their head. For example, **(forall x:Int) p(x) if q(x) or r(x);**
- Equational rules. These have an equation as their head. For example, **(forall x,y:Int) f(x) = y if y > 5;**

Note that there is no kind of rule that has a "constraint formula" in its head. Also, rules are used for *ground facts,* which can be as simple as **frequency = 5000;** (an equational rule with no body or variables) or **p(10);** (a standard rule with no body or variables).

Note that all statements end with a semicolon (;).

2.5 POLICIES, ONTOLOGIES AND REQUESTS

There are three kinds of CoRaL documents:

Policies. Policies contain statements. They have the form

policy P1 is
statements
end

Among the statements must be at least one **allow** and/or **disallow** rule. **allow** and **disallow** are built-in 0-ary predicates that can be used only in rule heads. The existence of these rules affects the overall request resolution:

A transmission request is approved if there is at least one allow and no disallow.

Ontologies. Ontologies contain any statements *except **allow** and **disallow** rules*. Ontologies have the form

ontology o1 is
statements
end

The other difference between ontologies and policies is that ontologies can be **use**d, whereas policies cannot.

Requests. Requests specify parameters of the radio, in order to get transmission approval (this is described in more detail in Section 3). Requests have the form

request r1 is
statements
end

The kinds of statements allowed in requests are limited to 1) ground atomic formulas, and 2) constant declarations.

3. ARCHITECTURE

A full description of the XG Architecture can be found in the XG Architecture RFC [2]. We repeat some of the material from that document here.

3.1 ARCHITECTURE OVERVIEW

At the highest level of abstraction, an *XG radio* has four main components as shown in **Figure 1**:

- **Sensors**. XG radios need sensors in order to discover available spectrum and transmission opportunities.
- **RF**. The RF component transmits and receives.
- **System Strategy Reasoner (SSR)**. The SSR controls 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, which may affect strategy.
- **PR**. The PR accepts transmission requests from the SSR and checks policy conformance. It replies with yes, no, or (with CSE capability) no with constraints to be satisfied. The PR has a policy base with all *active* policies.

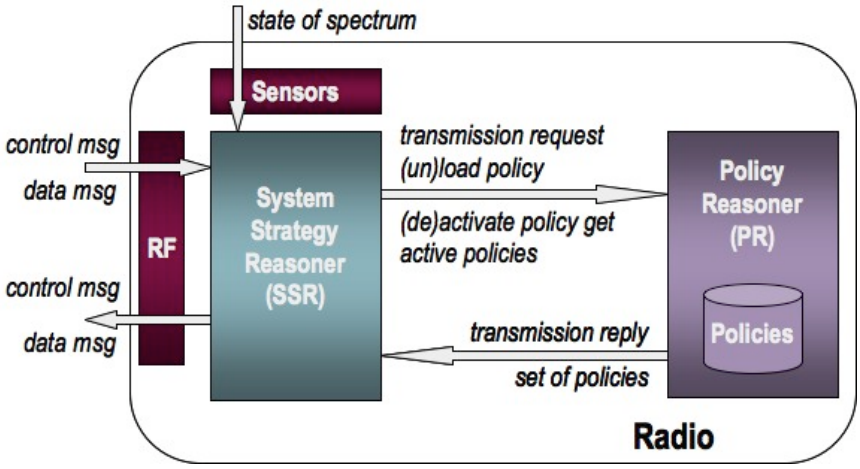

Figure 1. XG Architecture.
The red boxes are hardware components, all other boxes are software components.

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

3.2 TYPES OF MESSAGES

As **Figure 1** indicates, there are a number of different types of messages in the XG Architecture.

- **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 (acknowledgment of policy updates, requests

for new control channels, etc.) 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. The analysis of sensor data, sensor data aggregation, signal detection, and other such processing 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.

3.3 SSR-PR INTERFACE

The system described in this document is the PR. All communication with the PR originates in the SSR. 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 (see Section 5.4.1 Abstract Syntax Representation in Java). Currently, our CVE is implemented in Java and Prolog. The next section gives more details on the implementation.

- **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.

4. IMPLEMENTATION

This section details the implementation of the language, the Policy Reasoner, and the SSR-PR interface outlined in the previous sections. **Figure 2** shows an overview of the main modules of the Policy Reasoner, and some of the relationships between them. This implementation architecture is not part of the XG Architecture. The PR could have been implemented differently.

The larger ovals show the implementation languages. Most of the modules are implemented in Java, but the reasoning is done in Prolog. The SSR can be implemented in any language.[3] We (SRI) have not developed an SSR—it is merely shown in order to illustrate where PR inputs are processed.

The motivations for using Prolog as the reasoning engine were

- Prolog supports efficient execution of a large fragment of the CoRaL language.
- Prolog is a mature technology.
- Prolog interfaces well with popular programming languages, such as C and Java.

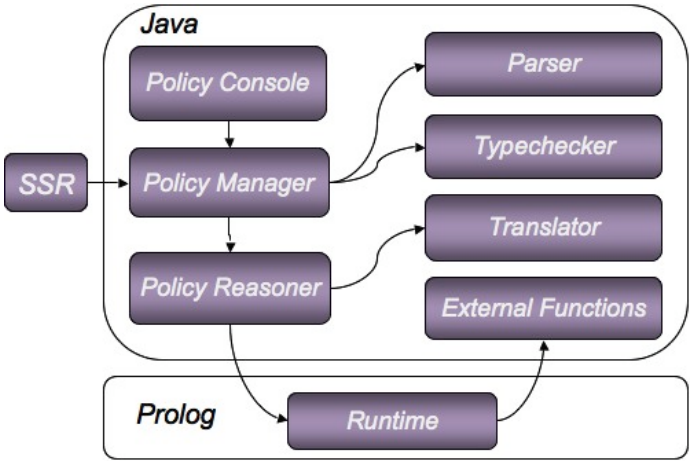

Figure 2. Architecture of the CVE Prolog implementation.
An arrow from A to B roughly means that A *uses* B.

The reasons for choosing Java for the supporting infrastructure were

- *Reliability.* Java programs do not suffer memory leaks as easily as C or C++ programs.
- *Ease of development.* Java code is faster to write than C++ code, mostly because the developer does not have to worry about memory management, and because runtime exceptions make most bugs trivial to find (as opposed to an opaque "Segmentation Fault" in C++).
- *Java interfaces well with Prolog.* Although C and C++ also work with Prolog, the interface is lower level, and requires more work to communicate parameters between the Prolog side and the C/C++ side.
- *Maturity.* An enormous collection of libraries is freely available for the Java platform, which is useful for writing CoRaL external functions in Java. For example, we used a "geotransform" Java library developed at SRI to support geometric operations like finding the distance between two points on the surface of the earth.

[3] Currently, the PR can be controlled only through Java method calls, so the SSR has to be coded in Java. However, it would be easy to add a socket or SOAP interface to allow communication in other languages.

4.1 MODULES

This section discusses the different modules in **Figure 2**. From the perspective of the SSR, the Policy Reasoner can be viewed as a black box, with a certain input/output behavior defined by the SSR-PR interface described in Section 3.3 SSR-PR Interface.

SSR. All messages to the PR originate from and are initiated by the SSR. The PR never initiates communication itself.

Policy Console. This module is described in detail in Section 5.3 Policy Console.

Policy Manager. This module is the object (a class with static methods) that controls all operations of the PR. All the methods in the SSR-PR interface are implemented as static methods of the PolicyManager Java class. The Policy Manager delegates functionality to various other modules, as can be seen in the figure. This module keeps lists of all currently loaded/active policies (i.e., the "Policy Base" in **Figure 1** is inside this module).

Policy Reasoner. This module is a thin Java wrapper around the module that does the actual reasoning. There is a Reasoner Java interface with only a few methods, and one implementation, PrologReasoner. This means that we could replace the reasoner back end without changing anything else in the implementation. The main functionality in the PrologReasoner class is

- Translating policies, ontologies, and requests from abstract syntax to Prolog
- Loading and unloading these modules to/from Prolog
- Asking Prolog to perform the actual queries, and returning the results as a Java object (of the Result class)

Parser. This module takes a CoRaL policy, ontology, or request as input and produces abstract syntax, in the form of Java objects, as output. The Java code for the parser is generated automatically at build time from a grammar specification, using the JavaCC parser generator.

Typechecker. Type checking is done on the abstract syntax, i.e., on Java objects. After being type checked, the abstract syntax is supplemented with some type information, which is used in the translation to Prolog, among other things.

Translator. This module translates policies, ontologies, and requests from abstract syntax to Prolog (see Section 4.2 CoRaL – Prolog Translation).

External Functions. Some of the functionality needed by CoRaL policies cannot (or should not, due to efficiency concerns) be implemented in CoRaL itself. These functions have been implemented in Java. So far, the external functions fall into the following categories:

- Powermask operations
- Time operations
- Geometric operations

Such functions mean that Java calls Prolog, which in turn calls Java (but there the cycle ends). We have not found the Java-Prolog interactions to be an efficiency bottleneck[4].

[4] Except that the *first* time a certain external Java function is called, it is very slow. See Section 5.3.2 Performing Requests for an example.

Prolog Runtime. This module is the only part that is implemented in Prolog. It consists of

- The translated policies, ontologies, and requests.
- A file called pre.pl[5]. This file contains a number of predicates, which can be called at runtime. More precisely, it contains
  - Code for equality and function evaluation, which is not normally supported in Prolog, and therefore has to be encoded (see Section 4.2 CoRaL – Prolog Translation).
  - Code for the built-in predicates and functions (<, +, mod, etc.).
- The file xg_java.pl, which contains code for marshaling parameters between Prolog and the external functions in Java.

4.2 CORAL – PROLOG TRANSLATION

Prolog was chosen as the reasoning engine because it already supports a large fragment of the CoRaL language. When implementing a language (the *object language*) using another language (the *meta-language*), one can use two main approaches:

1. The object language is *translated* to the meta-language.
2. The object language is *encoded* and *evaluated* in the meta-language.

The first approach is generally more efficient, because there is no overhead involved in evaluation. The object language is essentially executed by executing the meta-language. However, one is limited to the operational behavior of the meta-language. With the second approach (encoding) one has more flexibility, since one can implement the desired operational behavior in the meta-language.

Mixes of the two approaches are also possible. The CVE described in this report uses such a hybrid approach. The following parts of the language are *translated*:

- CoRaL predicates -> Prolog predicates
- CoRaL "standard" rules -> Prolog rules
- CoRaL policies, ontologies, and requests -> Prolog modules.
  There were some differences between how Prolog handles modules and CoRaL handles them, so this was not a completely straightforward translation.
- CoRaL abstract data types -> Prolog unary predicates

However, Prolog's native operational semantics does not support equality constraints, user-defined equations, or function evaluation. The only way to achieve those features in Prolog is to use the encoding approach. The encoding is rather shallow:

- Equality constraints are represented by a Prolog predicate eq/2, defined in the pre.pl file. There are Prolog rules for the transitivity and reflexivity of equality.

[5] "pre" is short for "prelude," a term used in many languages for the "built-in" features of the language that are always available.

- User-defined equations are represented by a rewrite/2 predicate, and are always interpreted in a directional left-to-right way.[6] These equations are taken into account when equality constraints are evaluated.
- Functions are represented as compound Prolog terms. They are evaluated, when the arguments are provided, by applying rewrite rules, in a call-by-value (eager) evaluation fashion.

There are many other details to the translation and encoding. The authoritative documentation is really the translation code itself (in the /src/com/sri/xgpl/prolog directory). As is so often the case, the devil is in the details.

Figure 3 gives an overview of the process of going from a policy in concrete syntax to the Prolog encoding, showing some of the software modules involved. We now give an example of the translation/encoding of a CoRaL policy to Prolog.

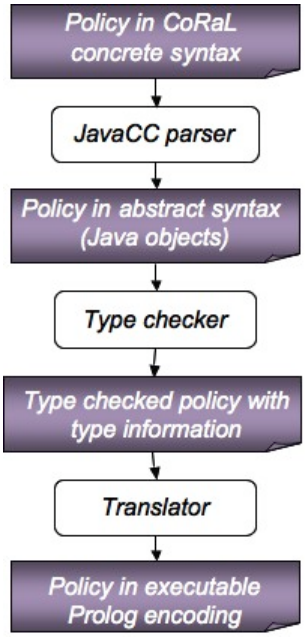

Figure 3. Loading and activating a policy. This workflow depicts the process of going from the CoRaL syntax to the Prolog encoding.

The example policy is the "tv1" policy (in the /policies/SSC-new directory, and also reprinted in Appendix C.20 TV1).

```
policy tv1 is
    use ssc_params;
            defconst F : Frequency = centerFrequency(req_transmission);
    allow if
      F in {450.0 .. 600.0} and
      timeDurationLessThanOrEqual(maxOnTime(req_transmission),
                    td(0,0,0,1,0)) = true and
      bandwidth(req_transmission) =< 6.0 and
      timeDurationLongerThanOrEqual(minOffTime(req_transmission),
                                td(0,0,0,0,150)) = true and
      meanEIRP(req_transmission) =< -53.0 and
      (exists ?se:SignalEvidence)
        req_evidence(?se) and
        peakRxPower(?se) =< 100.0 and
        {F-3.0 .. F+3.0} in scannedFrequencies(?se) and
        (exists ?d:PeriodicSignalDetector)
          detectedBy(?se) = ?d and
          sampleRate(?d) >= 0.2 and
          dutyCycle(?d) >= 0.5;
end
```

[6] This is of course inspired by rewrite logic.

The translation/encoding is

```
%--- Header ---
:-module(tv1,[]).
:-style_check(-singleton).
:-style_check(-discontiguous).
:-dynamic allow_l/1.
user:allow(X) :-
     context_module(M),
    debug:debug(runtime,debug,"~n~w:allow",[M]),allow_l(X).
rewrite_l(if(F,M1,M2),X) :- call(F) -> X=M1 ; X=M2.

%--- Rules ---
allow_l(tv1) :-
     in(centerFrequency(req_transmission),range(450.0,600.0)),
     eq(timeDurationLessThanOrEqual(maxOnTime(req_transmission),
        td(0,0,0,1,0)),true),
     lte(bandwidth(req_transmission),6.0),
     eq(timeDurationLongerThanOrEqual(minOffTime(req_transmission),
                          td(0,0,0,0,150)),true),
     lte(meanEIRP(req_transmission),uminus(53.0)),
     ('SignalEvidence'(Se),
      req_evidence(Se),
      lte(peakRxPower(Se),100.0),
      in(range(sub(centerFrequency(req_transmission),3.0),
                add(centerFrequency(req_transmission),3.0)),
         scannedFrequencies(Se)),
      ('PeriodicSignalDetector'(D),
       eq(detectedBy(Se),D),
       gte(sampleRate(D),0.2),
       gte(dutyCycle(D),0.5))).
```

In these policies, we can observe the shallow encoding of function applications as compound Prolog terms, the representation of built-in predicates (e.g., eq for equality, lte for less than or equal), and so forth. We also see that the defined constant F has been substituted, because Prolog does not natively support definitions. So “centerFrequency(req_transmission)” occurs everywhere in place of the defined constant “F” (which has the same behavior). We can also see how some special syntactic forms of CoRaL have been translated, for example, “{450.0 .. 600.0}” became “range(450.0,600.0)”.

So couldn’t we have done this rather similarly in any programming language? Most of it, yes. However, there is one feature that is rather specific to Prolog that we make heavy use of. We mentioned that abstract data types are translated to unary Prolog predicates. We can see two examples above: 'SignalEvidence' and ‘PeriodicSignalDetector’ (these are in single quotes because Prolog would otherwise interpret them as variables). Let us look at the existential quantifiers in the policy above. We have

```
(exists ?se:SignalEvidence) …
```

In the Prolog translation, this becomes

```
'SignalEvidence'(Se)
```

?se has become Se because variables have to start with an uppercase letter in Prolog. This translation has the correct behavior, because of Prolog's *unification* and *backtracking* features. Prolog will see 'SignalEvidence'(Se) as a goal that must be satisfied. This will happen if there is a rule that unifies with this goal. For example, if we have, in a request, a line

```
sigev : SignalEvidence;
```

this is translated to the Prolog version:

```
'SignalEvidence'(sigev)
```

This will unify with 'SignalEvidence'(Se), with the obvious substitution Se = sigev. Prolog can now continue to the next goal, keeping this substitution for whenever the Se variable next appears. If some further goal fails, Prolog will *backtrack* to the same goal again, and try another value for Se, if there is one. Therefore, the existentially quantified formula will succeed if and only if the Prolog version succeeds[7].

Of course, this kind of execution can be very slow if there are many unifiers, and only a few of them succeed.

4.3 LANGUAGE LIMITATIONS

The Prolog encoding of policies works essentially by "executing" them. This is usually very efficient, but is also has serious drawbacks.

Universal quantifiers. We cannot translate all kinds of universal quantification to native Prolog. Some kinds are not problematic. For example, the universal quantifiers on the outside of rules are ok, because they are already implicitly in Prolog rules. Also, some kinds of *finite* universal quantifiers are handled by trying all possibilities. For example,

```
(forall x:Int in [1,2,3]) p(x)
```

is translated to

```
forall(X,(member(X,[1,2,3]), p(X)))
```

Prolog's *forall* operator does what we just said: It tries all possibilities, i.e. all unifiers. So, in this case, X will be unified first with 1, and the p(1) goal tried. If this succeeds, X is unified with 2, and p(2) tried, and then the same for p(3). Only if all three goals succeed does the whole forall statement succeed.

Our comments on the efficiency of using unification for existential quantification are true for universal quantification and result in even worse efficiency. In the existential case, we can at least quit when we find a unifier that makes the existential statement succeed. In the universal case, we always have to try all substitutions.

Constraints. For the CSE capability, we want the Policy Reasoner to be able to return constraints when a policy fails due to being underspecified. For example, given a request for certain frequencies, we might get back a constraint on the maximum power on those frequencies.

[7] We make a closed-world assumption; see the Language RFC. We currently have no formal proof of the correctness of the Prolog translation.

While Prolog does have the so-called clp/r constraint solving capability, we have found this to be inadequate for several reasons[8], and it does not work well with all the other features of the language.

Having mentioned these limitations, it should be said that so far we have been able to encode and execute all the policies that were desired in our work on the XG project.

5. USING THE SYSTEM

Using the CoRaL system described in this document is rather straightforward once the policies have been written and the system has been installed. This section gives some additional notes on using the system, including a walk-through of the Policy Console, a useful tool for testing policies.

5.1 REQUIREMENTS

The current implementation of the CoRaL system runs on Java 1.5 and SWI Prolog (tested with version 5.6.3 and newer versions, compiled with multithreaded support). Any environment that supports these two languages should be sufficient to run the system. Both Java and SWI Prolog can be downloaded and used for free.[9]

The *ant* tool is needed to build the system from source code (which is how it is distributed). Ant is also free software.[10] The Geotransform package is free and open source.

Startup scripts for the Policy Console are provided for Linux, Mac OSX, and Windows, and the system has been tested on these setups. It should be straightforward to modify the scripts for any other UNIX-like environment.

5.2 INSTALLATION

The system is distributed as a zip archive, with a name indicating the date of the source code version contained therein, e.g., "XGPL_Prolog_PR 2006-07-31.zip."[11] When unpacked, it contains the following directory structure. We also describe some files and directories of specific interest:

```
XGPL_Prolog_PR/
lib/
  geotransform.jar  -- geo library developed at SRI
  javacc.jar – JavaCC, a Java parser generator
  jpl.jar – The jpl library for Java<->Prolog interaction
policies/
  Ontology/
```

[8] It does not work well with negations and quantifiers, and we have to decide in advance exactly which variables can be constrained.

[9] At http://www.swi-prolog.org/ and http://www.java.sun.com/, respectively.

[10] Available at http://ant.apache.org/

[11] The name "xgpl" occurs throughout the source code and in file names, as a leftover from the previous name of the language. XGPL stands for XG Policy Language and was used as a generic acronym for the language before it was named CoRaL.

all the ontology files, see Appendix B: Ontologies
SSC-new/
all the policies formalized for SSC, see Appendix C: Policies
prolog/
debug.pl -- Prolog code used at runtime to print debug messages in the Policy Console
loader.pl – Prolog code to load and unload policies and ontologies
pre.pl – Prolog implementation of equations and function evaluation
xg_java.pl – Prolog code to communicate parameters between Prolog and Java formats
xg_prolog_reasoner.pl – A file which just loads the other files above
src/com/sri/xgpl/
builtin/
built-in functions for geo, time, and powermask ontologies
console/
code for the Policy Console
exception/
parser/
ast/
classes representing all the syntactical constructs of CoRaL
XGPLParser.jj – JavaCC specification of the CoRaL grammar
prolog/
source code for Prolog-based policy reasoner, and translator from abstract syntax to Prolog
typechecker/
Java implementation of CoRaL type checker
types/
Java versions of types used by external (Java) functions, e.g. TimeInstant, Powermask
test/
Scripts to run the Policy Console
build.xml -- ant build **file**

After unpacking the zip archive, connect to the XGPL_Prolog_PR directory, and execute the command “ant jar” (without the quotes). If this doesn’t work, the ant tool is probably not on the PATH.

Next, enter the test/ directory, and try out the Policy Console, described in the next section.

5.3 POLICY CONSOLE

The policy console is a text-based utility, which is used to test (and demo) policies and requests. It can be started from the /test subdirectory, using one of the scripts

run_console_linux.sh

run_console_windows.bat

run_console_mac.sh

depending on your system architecture. You may need to edit some file paths in the script for your architecture, so that the libraries, Java runtime, and SWI Prolog runtime can all be found. Also make sure the script has execute permission (e.g., on Linux, execute

`chmod +x run_console_linux.sh`), and remember to run "`ant jar`" before trying this (see Section 5.2 Installation).

Once it starts, you should see a help screen, giving all the commands that can be used at the console prompt:

```
-------------------------------------
Welcome to the XG Policy Console
Daniel Elenius <elenius@csl.sri.com>
-------------------------------------
Commands:
  batch <file>
    Executes all the commands in <file>. Each command must be on a separate
line.
  load <file>
    Loads all policies and ontologies contained in the file.
    If the file is already loaded, and the file has changed, this is
equivalent to unload followed by load.
  unload <file> [all]
    Unloads all policies and ontologies contained in a given file loaded
previously.
    This may cause reloading and reactivation of ontologies and policies that
depend on ontologies/policies in the file.
    Use "unload all" to unload all policies and ontologies.
  activate <policy>
    Activates a loaded policy and its dependencies.
  deactivate <policy> [all]
    Deactivates a loaded policy and its dependencies.
    Use "deactivate all" to deactivate all policies.
  loaded
    Returns all loaded files, and the policies and ontologies they contain.
  active
    Returns all active policies.
  request <file> [why]
    Performs a transmission request, and shows the first allowing/disallowing
policies found.
    If there are neither allowing nor disallowing policies, and the 'why'
parameter is used, some explanations will be displayed.
  requestall <file> [why]
    Performs a transmission request, and shows all the allowing/disallowing
policies found.
    If there are neither allowing nor disallowing policies, and the 'why'
parameter is used, some explanations will be displayed.
  debuglevel <module> <level>
    The only supported <module> as of now is 'runtime'.
    <level> is one of error, warning, info, debug
  help
    Prints this help.
  quit
    Quits the XG Policy Console.
```

This is mostly self-explanatory. Your paths in the start-up scripts should be set up so that the policy console will find policies and requests in the /policies/Ontology and /policies/SSC-new directories (see Section 5.2 Installation). It will also find them if they are in the current directory, i.e. /test. This is a useful place to put temporary test policies.

Note that the commands available here are a superset of the SSR-PR interface described in Section 3.3 SSR-PR Interface. The only additional commands are “batch”, “debuglevel”, “help”, and “quit”. One can view the Policy Console as letting its operator play the part of the SSR, and the computer that of the PR.

5.3.1 Parsing and Type Checking

One use of the policy console is to make sure policies and ontologies are syntactically well formed and correctly typed. If the “load” command succeeds, then the policy and its used ontologies are correct in this regard. As an example, we have prepared a policy with some errors, called buggy.xg, located in the /test directory and in Appendix C: Policies . Here’s what happens when we try to load it:

```
> load buggy
com.sri.xgpl.exception.XGPLException: com.sri.xgpl.parser.gen.ParseException:
Encountered "tpc" at line 12, column 5.
Was expecting one of:
    "if" ...
    "(" ...
    ";" ...
    "*" ...
    "/" ...
    "mod" ...
    "quo" ...
    "+" ...
    "-" ...
    "=" ...
```

The parse tells us where the error occurred (line 12, column 5), what it encountered (“tpc”), and what it expected to encounter. In this case, we forgot to put “if” after “allow” in the top-level rule. After fixing this, we try again:

```
> load buggy
com.sri.xgpl.exception.XGPLException: com.sri.xgpl.parser.gen.ParseException:
Encountered "=" at line 14, column 33.
Was expecting one of:
    "true" ...
    "false" ...
    "(" ...
    "{" ...
    "[" ...
    "-" ...
    <INTEGER_LITERAL> ...
    <FLOATING_POINT_LITERAL> ...
    <VAR> ...
    <IDENTIFIER> ...
```

In line 14, where the error occurs, we find

```
meanEIRP(req_transmission) <= 30.0 and
```

The problem is that we've written '<=', whereas it should be '=<'[12]. After fixing this, we try again:

```
> load buggy
Parsed buggy
-------------------------------------
Type error in module buggy line 30 column 30 : sensingThreshold(se)
No function type matching function application
-------------------------------------
```

Note the message "Parsed buggy". This means that the policy is syntactically well formed: The system was able to parse it and generate an internal representation. However, it is not correctly typed. In this case, we wrote "se" instead of "?se", which would to refer to the quantified variable. We try again:

```
> load buggy
Loaded buggy
Loaded policies: [buggy]
Loaded ontologies: [signal, powermask, request_params, transmission, time,
message, geo, basic_types, radio, evidence, ssc_params]
```

This time it succeeded! After loading a policy or ontology, the system prints out all the loaded ontologies and policies. This policy has a statement "use ssc_params", to import the "ssc_params" ontology. That ontology, in turn, imports other ontologies. The whole import closure gets loaded automatically, as we can see.

5.3.2 Performing Requests

The other major use of the Policy Console is to test policies by performing requests against them. Here is an example run. First we load the lbt4 policy (located in /policies/SSC-new/lbt4.xg and in Appendix D: Requests):

```
> load lbt4
Loaded lbt4
Loaded policies: [lbt4]
Loaded ontologies: [signal, powermask, request_params, transmission, time,
message, geo, basic_types, radio, evidence, ssc_params]
>
```

Then we need to remember to *activate* the policy:

```
> activate lbt4
Activating lbt4
Activating ssc_params
Activating request_params
Activating radio
Activating evidence
Activating geo
```

[12] We chose the latter form for "less than or equal" because the former looks like a left implication. Similarly, we have '>=' rather than '=>' for "greater than or equal".

```
Activating time
Activating signal
Activating message
Activating basic_types
Activating transmission
Activating powermask
Active policies: [lbt4]
Active ontologies: [signal, powermask, request_params, transmission, time,
message, geo, basic_types, radio, evidence, ssc_params]
>
```

All the supporting ontologies are activated automatically. Next, we try an example request called request3 (from the file /policies/SSC-new/request3.xg), shown below in CoRaL syntax:

```
request request3 is
   defconst IBL : Powermask = pm(linear,[(1.0,1.50),(2.0,2.50)]);
   defconst OBL : Powermask = pm(linear,[(1.0,1.50),(2.0,2.50)]);

   tpc(req_transmission) = true;
   //tpc(req_transmission) = false;

   centerFrequency(req_transmission) = 5550.0;
   meanEIRP(req_transmission) = 25.0;
   inBandLeakage(transmittedBy(req_transmission)) = IBL;
   outOfBandLeakage(transmittedBy(req_transmission)) = OBL;
   bandwidth(req_transmission) = 20.0;

   public const se : SignalEvidence;
   req_evidence(se);
   public const te : TimeEvidence;
   req_evidence(te);

   timeStamp(te) = ti(2006,5,5,12,0,0,0);
   lastCompleteEmptyScanTime(se) = ti(2006,5,5,7,0,0,0);
   lastCompleteEmptyScanDuration(se) = td(0,0,2,0,0);

end
```

Request3 was written specifically to test this particular policy (lbt4), so it should give us a positive result:

```
> request request3
Permitted!
Allowed by: [lbt4]
Disallowed by: []
Time: 389 ms (parsing 67 ms, translating 1 ms, reasoning 321 ms)
```

The output contains four pieces of information:

1. Was the request approved or not? The output will say "Permitted!" or "Not permitted!" as applicable. A request is permitted if at least one policy *allows* it, and *no* policy *disallows* it, as explained previously.
2. A list of all policies allowing the request; in this case, the lbt4 policy.
3. A list of all policies disallowing the request; in this case, none.
4. Time measurements for the request, broken up into different tasks. The total time should be the sum of the times of the three tasks within the parentheses, but it may differ slightly because of rounding-off errors).

The time for this request is rather high. However, if we run the same request again, we get

```
> request request3
Permitted!
Allowed by: [lbt4]
Disallowed by: []
Time: 49 ms (parsing 2 ms, translating 1 ms, reasoning 46 ms)
```

The reason for this seems to be that the first time the request is done, the Java HotSpot compiler will compile the external functions implemented in Java (in this case the Powermask and Time functions). On subsequent runs, this compiled version is used, resulting in much faster execution times.

Let's try another request (from the file /policies/SSC-new/request1.xg; see also Appendix D: Requests):

```
> request request1
Not Permitted!
Allowed by: []
Disallowed by: []
Time: 25 ms (parsing 2 ms, translating 1 ms, reasoning 22 ms)
```

Here, the request was neither allowed nor disallowed by any policy. When this happens, we can try to figure out why, by using the "why" parameter after the request:

```
> request request1 why
Not Permitted!
Possibly due to missing parameters in request:
lbt4 : [lastCompleteEmptyScanTime, sensingThreshold, bandwidth,
inBandLeakage, lastDetected, transmittedBy, lastCompleteEmptyScanDuration,
outOfBandLeakage]
Allowed by: []
Disallowed by: []
Time: 47 ms (parsing 25 ms, translating 5 ms, reasoning 2 ms)
```

When we do this, the result will include a list of "missing parameters" for each active policy. In this case, lbt4 is the only active policy. Apparently, lbt4 needs eight parameters not specified by the request, and therefore it cannot approve it.

We will leave it to the user to perform further experiments. We recommend looking at request1 to request7, and different policies in /policies/SSC-new, as a starting point. The policies are also

in Appendix C: Policies and the requests are in Appendix D: Requests for convenience.

5.4 USING THE SYSTEM PROGRAMMATICALLY

To use the PR from another program (e.g., an SSR), it will be helpful to examine the source code of the Policy Console, located in /src/com/sri/xgpl/console. This code exposes the whole API. Requests can be sent to the PR by several means:

- Strings representing file paths
- Java streams
- Java Strings containing the request
- Java objects representing the abstract syntax

Using the String representation is recommended, as it will avoid unnecessary I/O.

Another thing worth noting is that in many cases requests will be parametric, that is, there will be several requests that are the same except for the values of some parameters. In this case, it is helpful to write a simple method that creates the request String based on the parameters. For example:

```
public String createRequest(float frequency, int i){
  return "request  r" + i + " is centerFrequency = " + frequency + "; end";
}
```

Note that a different name should be used for each request, which is why we have included the "int i" parameter. The method above might be called from a loop with the parameter "i" ranging from 0 to some limit.

5.4.1 Abstract Syntax Representation in Java

In some cases, it might be useful to use the abstract-syntax representation of requests. When a policy, ontology, or request has been parsed, it is stored internally in a so-called abstract-syntax representation. This representation is in the form of Java objects of the classes in /src/com/sri/xgpl/parser/ast/.

For example, a policy is represented by a Policy object, which points to a list of Statement objects for the statements in the policy. The Statement class has subclasses for Rule, Defconst, etc. A rule has fields for its body, head, and quantified variables.

Programs do not have to rely on the PR's loading mechanism to create the abstract-syntax representation. These objects can also be created directly. This is useful for applications that need to manipulate CoRaL objects, for example, graphical editing tools.

As a first example, we show a Java function similar to the one in the previous section. It returns a Request object based on a frequency and an integer identifier:

```
import com.sri.xgpl.parser.ast.*;

public Request createRequest(float frequency, int i){
      EqnRule freqRule =
        new EqnRule(null,          // Variable list
                    new Eqn(new Const(new Id("centerFrequency")),
```

```
                        new Float(freq.toString())),   // Rule head
                  null);         // Rule body

      List<Statement> statements = new ArrayList<Statement>();
      statements.add(freqRule);

      Request r = new Request(new Id("r" + i), statements);
      return r;  }
```

In this case, the String-based version of the previous section is simpler to write. However, the abstract syntax representation allows more flexibility. The following example shows a Java function that takes a Policy, and returns all declared and defined constants in the Policy:

```
public List<Id> getDeclared(Policy pol){
      List<Id> ids = new ArrayList<Id>();
      for (Statement s : pol.getStatements()){
            if (s instanceof Decconst)
                  ids.addAll(((Decconst)s).getIds();
            else if (s instanceof Defconst)
                  ids.add(((Defconst)s).getId());
      }
      return ids; }
```

The API can also be used to *change* the abstract syntax objects. For each *get* method there is a corresponding *set* method. The next example shows a Java function that takes a constant definition and adds a suffix to the name of the constant depending on its type:

```
public void addTypeSuffic(Defconst c){
      Id oldId = c.getId();
      String oldIdString = oldId.getValue();
      Id newId = null;
      Type type = c.getType();

      if (type instanceof List_t)
            newId = new Id(oldIdString + "_list");
      else if (type instanceof Set_t)
            newId = new Id(oldIdString + "_set");
      else if (type instanceof Pred_t)
            newId = new Id(oldIdString + "_pred");
      . . .

      c.setId(newId);         }
```

Using the API should be straightforward from reading the JavaDoc.

6. REFERENCES

[1] SRI XG Team. XG policy language. Request for comments. Technical report, SRI International, to be released, 2006.

[2] SRI XG Team. XG architecture. Request for comments. Technical report, SRI International, to be released, 2006.

APPENDIX A: CORAL GRAMMAR

Java-style comments appear throughout the grammar definition below. This grammar is directly based on the JavaCC grammar from which the CoRaL parser is built.

A.1 TOKENS

Notes:

Regular expressions are used to define tokens. ~ means not, * means zero or more occurrences,

? means one or more occurences, | means chose one or the other, ["a"-"z"] means chose one "between"

"a" and "z".

All the tokens like <IS> could optionally be put directly in the productions as "is".

The reason they are defined tokens is so that they can't be used as identifiers (they are reserved words).

White space may optionally appear between any two tokens.

```
SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
```

Comments are tokens, but are skipped on parsing

```
SPECIAL_TOKEN : /* COMMENTS */
{
 <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < IS: "is" >
| < TRUE_VALUE: "true" >
| < FALSE_VALUE: "false" >
| < TRUE: "True" >
| < FALSE: "False" >
| < INT: "Int" >
| < FLOAT: "Float" >
| < BOOL: "Bool" >
| < PRED: "Pred" >
| < FORALL: "forall" >
| < EXISTS: "exists" >
| < IN: "in" >
```

```
| < TYPE: "type" >
| < DEFTYPE: "deftype" >
| < CONST: "const" >
| < DEFCONST: "defconst" >
| < AND: "and" >
| < OR: "or" >
| < IMPLIES: "implies" >
| < NOT: "not" >
| < IF: "if" >
| < POLICY: "policy" >
| <ONTOLOGY : “ontology”>
| < END: "end" >
| < SUBTYPE: "subtype" >
| < EXTERNAL: "external" >
| < C: "C" >
| < JAVA: "Java" >
| < PROLOG: "Prolog" >
| < CLASS: "class" >
| < METHOD: "method" >
| < FUNCTION: "function" >
| < PROCEDURE: "procedure" >
| < REQUEST: "request" >
| < REPLY: "reply" >
| < PUBLIC: "public" >
| < USE: "use" >
}

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL: <DECIMAL_LITERAL>>
|
  < #DECIMAL_LITERAL: ((["1"-"9"])+ (["0"-"9"])*) | "0" >
|
  < FLOATING_POINT_LITERAL: <DECIMAL_LITERAL> "." (["0"-"9"])+ (<EXPONENT>)? >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}

TOKEN : /* IDENTIFIERS and VARIABLES*/
{
  < VAR: "?" <IDENTIFIER> >
|
  < IDENTIFIER: (<LETTER> (<LETTER>|<DIGIT>)*) | <QUOTED_IDENTIFIER> >
|
  < #QUOTED_IDENTIFIER: """ (~["\n","\r"])* """ >
|
  < #LETTER : ["a"-"z","A"-"Z","_","?"] >
|
  < #DIGIT : ["0"-"9"] >
}

TOKEN : /* SEPARATORS */
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
```

```
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < COLON: ":" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
}

TOKEN : /* OPERATORS */
{
  < FUNC: "->" >
| < RANGE: ".." >
| < LE: "=<" >
| < GE: ">=" >
| < LT: "<" >
| < GT: ">" >
| < EQ: "=" >
| < ADD: "+" >
| < MULT: "*" >
| < SUB: "-" >
| < DIV: "/" >
| < MOD: "mod" >    // modulo
| < QUO: "quo" >    // integer quotient
}
```

A.2 PRODUCTIONS

Note how the tokens defined above are referred to using <X> syntax.

These are defined using normal BNF syntax. In particular, * means zero or more occurrences, | means choice, [X] means X is optional.

```
Ontology :==
  <ONTOLOGY> Identifier <IS>
  StatementList
  <END>
  <EOF>

Policy :==
  <POLICY> Identifier <IS>
  StatementList
  <END>
  <EOF>

Request :==
  <REQUEST> Identifier <IS>
  StatementList
  <END>
  <EOF>

/**** Statements ***/

Statement :==
```

```
   (DectypeStatement
  | DeftypeStatement
  | SubtypeStatement
  | DecconstStatement
  | DefconstStatement
  | UseStatement
  | RuleStatement) <SEMICOLON>

DectypeStatement :==
  [<PUBLIC>] <TYPE> IdentifierList

DeftypeStatement :==
  [<PUBLIC>] <DEFTYPE> Identifier <EQ> Type

SubtypeStatement :==
  [<PUBLIC>] <SUBTYPE> Identifier <LT> Identifier

DecconstStatement :==
  [<PUBLIC>] <CONST> IdentifierList <COLON> Type

DefconstStatement :==
  [<PUBLIC>] <DEFCONST> Identifier <COLON> Type <EQ> Term

UseStatement :==
  <USE> IdentifierList

RuleStatement :==
  EquationalRule | StandardRule

StandardRule :==
  [ ForallVardec ] StandardAtomicFormula [ <IF> Formula ]

EquationalRule :==
  EquationFormula [ <IF> Formula ]

StatementList :==
  Statement ( Statement )*

IdentifierList :==
  Identifier ( <COMMA> Identifier )*


Identifier :==
  <IDENTIFIER>


/*** Types ***/

Type :==
  FunctionType
  | PredicateType
  | SimpleType

SimpleType :==
  AtomicType | ListType | SetType | TupleType
```

```
AtomicType :==
  PrimitiveType | EnumeratedType

PrimitiveType :==
  <INT> | <FLOAT> | <BOOL>

EnumeratedType :==
  Identifier

ListType :==
  <LBRACKET> SimpleType <RBRACKET>

SetType :==
  <LBRACE> SimpleType <RBRACE>

TupleType :==
  <LPAREN> SimpleTypeList <RPAREN>


// N.B. Higher-order arguments disabled here. Change SimpleType to
// FunctionType below if we want it.

PredicateType :==
  <PRED> [ <LPAREN> SimpleTypeList <RPAREN> ]

// N.B.: Higher-order functions disabled here. Changes SimpleType to
// FunctionType below if we want it.

FunctionType :==
   SimpleTypeList <FUNC> SimpleType [ <EXTERNAL> ExternalFunctionParams ]

ExternalFunctionParams :==
  JavaParams | CParams | PrologParams

JavaParams :==
  <JAVA> <CLASS> <EQ> Identifier <METHOD> <EQ> Identifier

CParams :==
  <C> <FUNCTION> <EQ> Identifier

PrologParams :==
  <PROLOG> <PROCEDURE> <EQ> Identifier

SimpleTypeList :==
  SimpleType ( <COMMA> SimpleType )*


/*** Terms ***/

Term :==
  ArithmeticExpression | FunctionAppl | SimpleTerm | ParensTerm

/* Note that for arithmetic expressions, the grammar is ambiguous.
This is disambiguated by imposing the following precedence order on the arithmetic operators:
parens, unary negation, (mult,div,mod,quo), (add, sub) */
```

```
ArithmeticExpression :==
  Term ( <ADD> | <SUB> | <MULT> | <DIV> | <MOD> | <QUO>) Term |
  <SUB> Term

ParensTerm :==
  <LPAREN> Term <RPAREN>

FunctionAppl :==
  Identifier <LPAREN> TermList <RPAREN>

SimpleTerm :==
  VariableTerm | ConstantTerm | Literal | ListTerm | SetTerm | TupleTerm

ConstantTerm :==
  Identifier

VariableTerm :==
  <VAR>

Literal :==
  IntegerLiteral | FloatLiteral | BoolLiteral

IntegerLiteral :==
  <INTEGER_LITERAL>

FloatLiteral :==
  <FLOATING_POINT_LITERAL>

BoolLiteral :==
  <TRUE_VALUE> | <FALSE_VALUE>

ListTerm :==
  <LBRACKET> TermList <RBRACKET>

SetTerm :==
  <LBRACE> Term <RANGE> Term <RBRACE>

TupleTerm :==
  <LPAREN> TermList <RPAREN>

TermList :==
  Term ( <COMMA> Term )*

// These are terms, but they can only be used in EquationFormulas right now.
IfThenElse() :==
    <IF> Formula <THEN> Term <ELSE> Term

/*** Formulas ***/

/* Note that for formulas, the grammar is ambiguous. This is disambiguated by imposing the following
precedence order on the connectives: parens, not, and, or, implies, (exists, forall). */

Formula :==
  Formula ( <IMPLIES> | <OR> | <AND> ) Formula |
  <NOT> Formula |
  ExistsVardec Formula |
```

```
  ForallVardec Formula |
  <LPAREN> Formula <RPAREN> |
  AtomicFormula

AtomicFormula :==
  EquationFormula
  | ConstraintFormula()
  | StandardAtomicFormula()

StandardAtomicFormula :==
  Identifier [ <LPAREN> TermList <RPAREN> ]

EquationFormula :==
  Term <EQ> ( Term | IfThenElse )

ConstraintFormula :==
    Term
    ( <GT> | <LT> | <GE> | <LE> | <IN> )
    Term

ExistsVardec :==
    <LPAREN> <EXISTS> VardecList <RPAREN>

ForallVardec :==
    <LPAREN> <FORALL> VardecList <RPAREN>

VardecList :==
    Vardec ( <COMMA> Vardec )*

Vardec :==
    Vardecf
    | VardecStd

VardecStd :==
   VariableList <COLON> Type

Vardecf :==
   VariableDec <COLON> Type <IN> Term

VariableDec :==
    <VAR>

VariableList :==
    VariableDec ( <COMMA> VariableDec )*
```

APPENDIX B: ONTOLOGIES

SRI developed these ontologies with the assistance of Shared Spectrum Company (SSC), who provided most of the domain knowledge about radios and transmissions.

B.1 BASIC_TYPES

```
ontology basic_types is
  public type Detector;
  public type Transmitter;
  public type Evidence;

  /* The following 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;
end
```

B.2 GEO

```
ontology geo is
  /* Abstract Data Types */
  /* Altitudes are relative to the WGS84 ellipsoid. Note that this can differ
     significantly from the distance above Mean Sea Level. */
  public type Location;
  public const loc : Float, Float, Float -> Location;
  public const latitude : Location -> Float;
  public const longitude : Location -> Float;
  public const altitude : Location -> Float;
  (forall ?lat,?long,?alt:Float)
   latitude(loc(?lat,?long,?alt)) = ?lat;
  (forall ?lat,?long,?alt:Float)
   longitude(loc(?lat,?long,?alt)) = ?long;
  (forall ?lat,?long,?alt:Float)
   altitude(loc(?lat,?long,?alt)) = ?alt;

   /* GeographicArea is defined by its bordering polygon (disregarding altitudes) */
  public type GeographicArea;
  public const border : GeographicArea -> [Location];

  public type Ellipse;
  /* Takes semi-major and semi-minor distances, rotation in degrees, and an origin. */
  public const ell : Float,Float,Float,Location -> Ellipse;
  public const locationInEllipse : Location,Ellipse -> Bool
           external Java class='com.sri.xgpl.builtin.Geo' method=locationInEllipse;

  /* Operations */
  public const locatedIn : Location,GeographicArea -> Bool;
  /* Distance is in meters, and is the shortest straight path between the two points.
     This could mean going through the earth, i.e. we do not calculate distance along the curvature
```

```
    of the earth. */
  public const distance : Location,Location -> Float
         external Java class='com.sri.xgpl.builtin.Geo' method=distance;
end
```

B.3 MESSAGE

```
ontology 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 don't need to talk about instances of
     messages. */
  public type Message;
end
```

B.4 RADIO

```
ontology radio is
  use evidence,transmission,powermask;
  public const threshold : Detector -> Threshold;
  public const detectionPrecision : 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;
  /* Pred or func? Should these be under Evidence? */
  public const dutyCycle : PeriodicSignalDetector->Float;   /* ratio of on-off time */
  public const sampleRate : PeriodicSignalDetector->Float;
  /* Note: Much more could be added on sensing, e.g. exponential backoff */
  public subtype TimeDetector < Detector;
  public subtype LocationDetector < Detector;

  public const inBandLeakage : Transmitter -> Powermask;
  public const outOfBandLeakage : Transmitter -> Powermask;
  public const transmissionPrecision : Transmitter -> Precision;
  public const setupForTransmission : Transmitter -> Transmission;
         /* inverse of transmission.transmitter */

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

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

B.5 EVIDENCE

```
ontology evidence is
  use geo,time,signal,message,basic_types;
  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;
  /* 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 const peakRxPower : SignalEvidence -> Power;

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

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

B.6 MATH

```
/* This can be expanded to include many useful mathematical functions. */
ontology math is
  public const max : Float,Float->Float;
  (forall ?i,?j:Float)
  max(?i,?j) = if ?i >= ?j then ?i else ?j;
  public const min : Float,Float->Float;
  (forall ?i,?j:Float)
  min(?i,?j) = if ?i =< ?j then ?i else ?j;
  public const floor : Float -> Int;
  (forall ?f:Float)
  floor(?f) = if ?f > 1.0 then 1 + floor(?f-1.0) else 0;
  /* Now built-in instead: quotient, modulo  */
end
```

B.7 POWERMASK

```
ontology powermask is
  use basic_types;
  deftype PowermaskValues = [(Frequency,Power)];
  /* ADTs */
  public type MaskShape;
  public const linear : MaskShape;
  public const step : MaskShape;
  public type Powermask;
  public const pm : MaskShape, PowermaskValues -> Powermask;
  public const maskShape : Powermask -> MaskShape;
  public const values : Powermask -> PowermaskValues;

  (forall ?m:MaskShape,?p:PowermaskValues)
  maskShape(pm(?m,?p)) = ?m;

  (forall ?m:MaskShape,?p:PowermaskValues)
  values(pm(?m,?p)) = ?p;
```

```
  /* Operations */
  const powermaskLessThanExt : MaskShape,PowermaskValues,MaskShape,
PowermaskValues -> Bool
         external Java class='com.sri.xgpl.builtin.Powermask' method=lessThan;
  public const powermaskLessThan : Powermask,Powermask -> Bool;
  public const powermaskGreaterThan : Powermask,Powermask -> Bool;
  public const powermaskLessThanOrEqual : Powermask,Powermask -> Bool;
  public const powermaskGreaterThanOrEqual : Powermask,Powermask -> Bool;

  (forall ?p1,?p2:Powermask,?x:Bool)
  powermaskLessThan(?p1,?p2)=?x if
    powermaskLessThanExt(maskShape(?p1),values(?p1),
                  maskShape(?p2),values(?p2))=?x;

  (forall ?p1,?p2:Powermask,?x:Bool)
  powermaskGreaterThan(?p1,?p2)=?x if
    powermaskLessThan(?p2,?p1)=?x;

  (forall ?p1,?p2:Powermask,?x:Bool)
  powermaskLessThanOrEqual(?p1,?p2)=?x if
    not powermaskGreaterThan(?p2,?p1)=?x;

  (forall ?p1,?p2:Powermask,?x:Bool)
  powermaskGreaterThanOrEqual(?p1,?p2)=?x if
    not powermaskLessThan(?p1,?p2)=?x;

 const powermaskAddExt : MaskShape,PowermaskValues,
                   MaskShape,PowermaskValues -> PowermaskValues
          external Java class='com.sri.xgpl.builtin.Powermask' method=add;
  public const powermaskAdd : Powermask,Powermask -> Powermask;

  (forall ?p1,?p2,?p3:Powermask)
  powermaskAdd(?p1,?p2)=?p3 if
    (maskShape(?p1)=linear or maskShape(?p2)=linear) and
    powermaskAddExt(maskShape(?p1),values(?p1),
               maskShape(?p2),values(?p2))=values(?p3) and
    maskShape(?p3)=linear;

  (forall ?p1,?p2,?p3:Powermask)
  powermaskAdd(?p1,?p2)=?p3 if
    maskShape(?p1)=step and
    maskShape(?p2)=step and
    powermaskAddExt(linear,values(?p1),linear,values(?p2))=values(?p3) and
    maskShape(?p3)=step;
end
```

B.8 REQUEST_PARAMS

```
ontology request_params is
  use radio;
  /* evidence(e) means that e is an evidence object that the SSR presents to the PR. */
  public const req_evidence : Pred(Evidence);
  /* transmission(t) means that t is the transmission that the SSR wants to do */
  public const req_transmission : Transmission;
  /* radio(r) means that r is the radio requesting to transmit */
  public const req_radio : Radio;
```

```
end
```

B.9 SIGNAL

```
ontology signal is
  public type Signal;
  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
```

B.10 TIME

```
ontology time is
  /* --- Types & Constructors ---*/
  //public type Month;
  //public const Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec : Month;
 /* TimeInstants are in UTC plus milliseconds. However, the operations onTimeInstant (see below)
    do not take leap seconds into account, for simplicity.
    See also http://en.wikipedia.org/wiki/Unix_time */
  public type TimeInstant;
  public const ti : Int,Int,Int,Int,Int,Int,Int -> 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;

  (forall ?y,?m,?d,?h,?min,?s,?ms : Int)
  year(ti(?y,?m,?d,?h,?min,?s,?ms)) = ?y;
  (forall ?y,?m,?d,?h,?min,?s,?ms : Int)
  month(ti(?y,?m,?d,?h,?min,?s,?ms)) = ?m;
  (forall ?y,?m,?d,?h,?min,?s,?ms : Int)
  day(ti(?y,?m,?d,?h,?min,?s,?ms)) = ?d;
  (forall ?y,?m,?d,?h,?min,?s,?ms : Int)
  hour(ti(?y,?m,?d,?h,?min,?s,?ms)) = ?h;
  (forall ?y,?m,?d,?h,?min,?s,?ms : Int)
  minute(ti(?y,?m,?d,?h,?min,?s,?ms)) = ?min;
  (forall ?y,?m,?d,?h,?min,?s,?ms : Int)
  second(ti(?y,?m,?d,?h,?min,?s,?ms)) = ?s;
  (forall ?y,?m,?d,?h,?min,?s,?ms : Int)
  millisecond(ti(?y,?m,?d,?h,?min,?s,?ms)) = ?ms;

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

  (forall ?ps,?pe:TimeInstant)
  startTime(tp(?ps,?pe)) = ?ps;
  (forall ?ps,?pe:TimeInstant)
  endTime(tp(?ps,?pe)) = ?pe;
```

```
  public type TimeDuration;
  public const td : Int,Int,Int,Int,Int -> TimeDuration;
  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;

  (forall ?d,?h,?min,?s,?ms : Int)
  days(td(?d,?h,?min,?s,?ms)) = ?d;
  (forall ?d,?h,?min,?s,?ms : Int)
  hours(td(?d,?h,?min,?s,?ms)) = ?h;
  (forall ?d,?h,?min,?s,?ms : Int)
  minutes(td(?d,?h,?min,?s,?ms)) = ?min;
  (forall ?d,?h,?min,?s,?ms : Int)
  seconds(td(?d,?h,?min,?s,?ms)) = ?s;
  (forall ?d,?h,?min,?s,?ms : Int)
  milliseconds(td(?d,?h,?min,?s,?ms)) = ?ms;

 /* --- Operations --- */
  public const subtractTime : TimeInstant, TimeDuration -> TimeInstant
    external Java class='com.sri.xgpl.builtin.Time' method=subtractTime;
  public const addTime : TimeInstant, TimeDuration -> TimeInstant
    external Java class='com.sri.xgpl.builtin.Time' method=addTime;
  public const timeBetween : TimeInstant, TimeInstant -> TimeDuration
    external Java class='com.sri.xgpl.builtin.Time' method=timeBetween;
  public const timeBefore : TimeInstant,TimeInstant -> Bool
    external Java class='com.sri.xgpl.builtin.Time' method=timeBefore;
  public const timeAfter : TimeInstant,TimeInstant -> Bool
    external Java class='com.sri.xgpl.builtin.Time' method=timeAfter;
  public const timeBeforeOrSame : TimeInstant,TimeInstant -> Bool
    external Java class='com.sri.xgpl.builtin.Time' method=timeBeforeOrSame;
  public const timeAfterOrSame : TimeInstant,TimeInstant -> Bool
    external Java class='com.sri.xgpl.builtin.Time' method=timeAfterOrSame;
  public const timeDurationLessThan : TimeDuration,TimeDuration -> Bool
    external Java class='com.sri.xgpl.builtin.Time'
            method=timeDurationLessThan;
  public const timeDurationLongerThan : TimeDuration,TimeDuration -> Bool
    external Java class='com.sri.xgpl.builtin.Time'
            method=timeDurationLongerThan;
  public const timeDurationLessThanOrEqual : TimeDuration,TimeDuration -> Bool
    external Java class='com.sri.xgpl.builtin.Time'
            method=timeDurationLessThanOrEqual;
  public const timeDurationLongerThanOrEqual :
    TimeDuration,TimeDuration -> Bool
    external Java class='com.sri.xgpl.builtin.Time'
             method=timeDurationLongerThanOrEqual;
  public const inTimePeriod : TimeInstant,TimePeriod -> Bool;

  /*--- Equations for public operators ---*/
  (forall ?ti : TimeInstant, ?tp : TimePeriod)
   inTimePeriod(?ti,?tp) = true if
   timeAfterOrSame(?ti,startTime(?tp)) = true and
   timeBeforeOrSame(?ti,endTime(?tp)) = true;
end
```

B.11 TRANSMISSION

```
ontology transmission is
  use time,basic_types;
  public type Transmission;
  public const centerFrequency : Transmission -> Frequency;
  public const bandwidth : Transmission -> Bandwidth;
  public const maxOnTime : Transmission -> TimeDuration;
  public const minOffTime : Transmission -> TimeDuration;
  public const meanEIRP : Transmission -> Power;
  public const transmittedBy : Transmission -> Transmitter;
end
```

APPENDIX C: POLICIES

C.1 BEACON1

```
/* Beacon policy 1
    " A policy allows XG radios to transmit if it can hear a permit-use beacon operating at 300 MHz
broadcasting a beacon signal continuously for duration of 100 milliseconds every one second."*/

policy beacon1 is
  use ssc_params;
  allow if
    (exists ?se:SignalEvidence, ?t:TimeInstant)
      req_evidence(?se) and
      detectedSignal(?se,permitUse) and
      300.0 in scannedFrequencies(?se) and
      lastCompleteEmptyScanDuration(?se) = td(0,0,0,0,100) and
      timeStamp(?se) = ?t and
      timeDurationLongerThan(timeBetween(?t,lastDetected(?se)),
                    td(0,0,0,1,0)) = true;
end
```

C.2 CONFIG1

```
/* Device configuration policy 1
   "A policy allows XG radios to transmit in the band 5180 MHz to 5250 MHz."*/
policy config1 is
  use request_params;
  allow if
    centerFrequency(req_transmission) in {5180.0 .. 5250.0};
end
```

C.3 DEVICE1

```
/* Device capability policy 1
   "A policy allows XG radios to transmit only if they are certified to be XG version 5 compliant."*/
policy device1 is
  use ssc_params;
  allow if
    capability(req_radio,xg_v_5);
end
```

C.4 DEVICE2

```
/* Device capability policy 2
   "A policy allows XG radios to transmit only if they support a parameter P."*/
policy device2 is
  use ssc_params;
  allow if
   capability(req_radio,p);
end
```

C.5 DEVICE3

```
/* Device capability policy 3
   "A policy allows XG radios to transmit only if they support a process type P."*/
policy device3 is
```

```
  use ssc_params;
  allow if
    capability(req_radio,pr);
end
```

C.6 DISTRIBUTED1

```
/* Distributed Control policy 1
   "A policy expires on 2006-12-30T23:59:59."*/
policy distributed1 is
  use ssc_params;
  allow if
  (exists ?te:TimeEvidence,?t:TimeInstant)
    req_evidence(?te) and
    timeStamp(?te) = ?t and
    timeBefore(?t,ti(2006,12,30,23,59,59,0)) = true;
end
```

C.7 DISTRIBUTED2

```
/* Distributed Control policy 2
   "A policy allows XG radios to transmit if they are authorized by authority X."*/
policy distributed2 is
  use ssc_params;
  allow if
    req_authorized(X);
end
```

C.8 DISTRIBUTED3

```
/* Distributed Control policy 3
   "A policy 1 is more important than policy 2."
   This cannot currently be expressed. Policy precedence can be handled by
   the SSR by deactivating lower-precedence policies.*/
```

C.9 IDENTITY1

```
/* Node Identity policy 1
   "A policy allows XG radios to transmit in band 250 MHz to 260 MHz if they belong to Red Cross."*/
policy identity1 is
  use ssc_params;
  allow if
    centerFrequency(req_transmission) in {250.0 .. 260.0} and
    req_org(redCross);
end
```

C.10 IDENTITY2

```
/* Node Identity policy 2 (war mode)
   "A policy allows all XG radios to transmit anywhere, anytime
    if they are authorized by DoD authority as war-mode radios."
   Note: The SSR must also deactivate all disallowing policies.*/
policy identity2 is
  allow;
end
```

C.11 LBT1

```
/* Listen-Before-Talk policy 1
   "A policy restricts XG radios from transmitting if its
```

```
   peak received power sensing is more than -80 dBm."*/
policy lbt1 is
  use request_params;
  disallow if
  (exists ?se:SignalEvidence)
    req_evidence(?se) and
    peakRxPower(?se) > -80.0;
end
```

C.12 LBT2

```
/* Listen-Before-Talk policy 2
   "A policy allows XG radios to transmit if its peak received
    power sensing is at most -80 dBm and its EIRP for transmission is at most 10 mW."*/
policy lbt2 is
  use request_params;
  allow if
    (exists ?se:SignalEvidence)
      req_evidence(?se) and
      peakRxPower(?se) =< -80.0E-2 and
      meanEIRP(req_transmission) =< 10.0E-3;
end
```

C.13 LBT3

```
/* Listen-Before-Talk policy 3
   "A policy restricts XG radios from transmitting if it detected the X waveform."*/
policy lbt3 is
  use ssc_params;
  disallow if
    (exists ?se:SignalEvidence)
      req_evidence(?se) and
      detectedSignal(?se,signal_x);
end
```

C.14 LBT4

```
/* Listen-Before-Talk policy 4 (partial DFS)
   " A policy allows XG radios with transmission power control turned on to transmit in the band 5470 MHz
to 5725 MHz only if the mean EIRP is at most 30 dBm, the unwanted emission mask is below a mask
represented by a point-based function M, the in-band leakage is below a power mask represented by a
point-based function M2, the transmission bandwidth is at most 20 MHz, the nominal frequency carrier is
one the frequency values in set S, and the radios did not detect any radar signal while continuously
sensing for at least 60 seconds in the last 24 hours or if the last radar detection occurred at least 30
minutes before the latest clear 60-second long scan with power sensing threshold at most -64 dBm."*/
policy lbt4 is
  use ssc_params;
  defconst maxIBL : Powermask = pm(linear,[(1.0,2.0),(2.0,4.0)]);
  defconst maxOBL : Powermask = pm(linear,[(1.0,2.0),(2.0,4.0)]);
  defconst channels : [Frequency] = [5500.0, 5550.0, 5600.0];
  allow if
    tpc(req_transmission) = true and
    centerFrequency(req_transmission) in {5470.0 .. 5725.0} and
    meanEIRP(req_transmission) =< 30.0 and
    powermaskLessThan(inBandLeakage(transmittedBy(req_transmission)),
maxIBL) = true and
    powermaskLessThan(outOfBandLeakage(transmittedBy(req_transmission)),
      maxOBL) = true and
```

```
   bandwidth(req_transmission) =< 20.0 and
   centerFrequency(req_transmission) in channels and
   (exists ?se:SignalEvidence,?te:TimeEvidence)
     req_evidence(?se) and
     ((not ((exists ?r:RadarSignal) detectedSignal(?se,?r)) and
     req_evidence(?te) and
     timeAfter(lastCompleteEmptyScanTime(?se), subtractTime(timeStamp(?te),
           td(1,0,0,0,0))) = true and
     timeDurationLongerThan(lastCompleteEmptyScanDuration(?se),
              td(0,0,0,60,0)) = true) or
   (((exists ?r:RadarSignal) detectedSignal(?se,?r)) and
     timeBefore(addTime(lastDetected(?se),td(0,0,30,0,0)),
              lastCompleteEmptyScanTime(?se)) = true and
     timeDurationLongerThan(lastCompleteEmptyScanDuration(?se),
              td(0,0,0,60,0)) = true and
     sensingThreshold(?se) >= -64));
end
```

C.15 LBT5

```
/* Listen-Before-Talk policy 5 (partial DFS)
   "A policy allows XG radios with transmission power control turned off or no transmission power control
to transmit in the band 5470 MHz to 5725 MHz only if the mean EIRP is at most 23 dBm, the unwanted
emission mask is below a mask represented by a point-based function M, the in-band leakage is below a
power mask represented by a point-based function M2, the transmission bandwidth is at most 20 MHz,
the nominal frequency carrier is one the frequency values in set S, and the radios did not detect any radar
signal while continuously sensing for at least 60 seconds in the last 24 hours or if the last radar detection
occurred at least 30 minutes before the latest clear 60-second long scan with power sensing threshold at
most -62 dBm."
   (same as lbt3 except no tpc, and -62 dBm)*/
policy lbt5 is
  use ssc_params;
  defconst maxIBL : Powermask = pm(linear,[(1.0,2.0),(2.0,4.0)]);
  defconst maxOBL : Powermask = pm(linear,[(1.0,2.0),(2.0,4.0)]);
  defconst channels : [Frequency] = [5500.0, 5550.0, 5600.0];
  allow if
    tpc(req_transmission) = false and
    centerFrequency(req_transmission) in {5470.0 .. 5725.0} and
    meanEIRP(req_transmission) =< 30.0 and
    powermaskLessThan(inBandLeakage(transmittedBy(req_transmission)),
      maxIBL) = true and
    powermaskLessThan(outOfBandLeakage(transmittedBy(req_transmission)),
      maxOBL) = true and
    bandwidth(req_transmission) =< 20.0 and
    centerFrequency(req_transmission) in channels and
    (exists ?se:SignalEvidence,?te:TimeEvidence)
      req_evidence(?se) and
      ((not ((exists ?r:RadarSignal) detectedSignal(?se,?r)) and
      req_evidence(?te) and
      timeAfter(lastCompleteEmptyScanTime(?se),
              subtractTime(timeStamp(?te), td(1,0,0,0,0))) = true) and
      timeDurationLongerThan(lastCompleteEmptyScanDuration(?se),
              td(0,0,0,60,0)) = true) or
    (((exists ?r:RadarSignal) detectedSignal(?se,?r)) and
      timeBefore(addTime(lastDetected(?se),td(0,0,30,0,0)),
               lastCompleteEmptyScanTime(?se)) = true and
```

```
    timeDurationLongerThan(lastCompleteEmptyScanDuration(?se),
                    td(0,0,0,60,0)) = true and
    sensingThreshold(?se) >= -62.0);
end
```

C.16 LOCATION1

```
/* Location policy 1
  "A policy allows XG radios to transmit only if they are at most 30 miles away from the geographic
coordinates (39 10' 30" N, 75 01' 42")."    Note: We used this converter
    http://www.geology.enr.state.nc.us/gis/latlon.html
  to convert between deg/min/sec and decimal formats.*/
policy location1 is
 use request_params;
 defconst loc1 : Location = loc(39.175, 75.0283, 0.0);
 allow if
  (exists ?le:LocationEvidence)
    req_evidence(?le) and
    distance(location(?le),loc1) =< 48280.32; // exactly 30 miles in meters
end
```

C.17 LOCATION2

```
/* Location policy 2
  "A policy allows XG radios to transmit only if they are at least 10 miles away from every of the provided
geographic coordinate list L, where L consists of a large number of geographic coordinates ."*/
policy location2 is
 use request_params;
 /* Could be a much larger list */
 defconst loclist : [Location] = [loc(1.2,3.4,0.0), loc(1.2,3.4,10000.0)];
 allow if
  (exists ?le:LocationEvidence)
    req_evidence(?le) and
    (forall ?l2:Location in loclist)
      distance(location(?le),?l2) >= 16093.44;  // 10 miles in meters
end
```

C.18 TIME1

```
/* Time policy 1
  "A policy allows XG radios to transmit only between 06:00 and 13:00 local time."
   I made 06:00 inclusive and 13:00 exclusive.*/
policy time1 is
 use request_params;
 allow if
  (exists ?te:TimeEvidence)
    req_evidence(?te) and
    hour(timeStamp(?te)) in {6 .. 12};
end
```

C.19 TIME2

```
/* Time policy 2
  "A policy allows XG radios to transmit for no more than 1 second at a time
   and requires that the minimum off-time between transmissions is at least 100 milliseconds."*/
policy time2 is
 use request_params;
 defconst minAllowedTime : TimeDuration = td(0,0,0,0,100);
```

```
  allow if
    timeDurationLessThan(minAllowedTime,
                    minOffTime(req_transmission)) = true;
end
```

C.20 TV1

```
/* Time-Location-LBT policy (tv policy)
   "A policy allows XG radios located within USA boundaries to transmit in band 450 MHz to 600 MHz if
they transmit continuously for at most 1 second at bandwidth of at most 6 MHz, have off-time of at least
150 milliseconds, have duty cycle of at least 50% across a 2-second period and if they sample spectrum
at least once every 5 seconds for TV signals using power sensing but no sub-noise detection of DTV
signals and they received spectral power in the channel they are transmitting equivalent or less than 100
dBm and the peak power spectral density of their emission does not exceed -53 dBm/Hz."
    Notes:
    Duty cycle should probably be *at most* 50%. We don't have a way of saying "across a 2-second
period". This can be added.
    "No sub-noise detection of DTV signals"?  -- use detectedSignal(DTV)
    power spectral density = meanEIRP?   received spectral power = peakRxPower?
    Should continuous/periodic be properties of evidence instead of detector?*/

policy tv1 is
  use ssc_params;
  /* This will save some typing below */
  defconst F : Frequency = centerFrequency(req_transmission);
  allow if
    F in {450.0 .. 600.0} and
    timeDurationLessThanOrEqual(maxOnTime(req_transmission),
      td(0,0,0,1,0)) = true and
    bandwidth(req_transmission) =< 6.0 and
    timeDurationLongerThanOrEqual(minOffTime(req_transmission),
      td(0,0,0,0,150)) = true and
    meanEIRP(req_transmission) =< -53.0 and
    (exists ?se:SignalEvidence)
      req_evidence(?se) and
      peakRxPower(?se) =< 100.0 and
      {F-3.0 .. F+3.0} in scannedFrequencies(?se) and
      (exists ?d:PeriodicSignalDetector)
        detectedBy(?se) = ?d and
        sampleRate(?d) >= 0.2 and
        dutyCycle(?d) >= 0.5;
end
```

C.21 TV2

```
/* Time-Location-LBT policy 2 (tv policy 2)
   "A policy allows XG radios located within USA boundaries to transmit in band 450 MHz to 600 MHz if
they transmit continuously for at most 1 second at bandwidth of at most 6 MHz, have off-time of at least
150 milliseconds, have duty cycle of at least 50% across a 2-second period and if they sample spectrum
at least once every 5 seconds for TV signals using sub-noise detection of DTV signals and they received
spectral power in the channel they are transmitting equivalent or less than 107 dBm and the peak power
spectral density of their emission does not exceed -53 dBm/Hz plus 1 dB for every 1 Db the received
signal is below -107 dBm up to a maximum of -40  dBm/Hz."*/
policy tv2 is
  use ssc_params,math;
  /* This will save some typing below */
  defconst F : Frequency = centerFrequency(req_transmission);
```

```
  /* The max tx power depends on the max rx power.
     Typical situation to use a function. */
  const maxPower : Power->Power;
  (forall ?r:Power)
  maxPower(?r) = min(-40.0,-53.0+(-107.0-?r));

  allow if
    F in {450.0 .. 600.0} and
    timeDurationLessThanOrEqual(maxOnTime(req_transmission),
      td(0,0,0,1,0)) = true and
    bandwidth(req_transmission) =< 6.0 and
    timeDurationLongerThanOrEqual(minOffTime(req_transmission), td(0,0,0,0,150)) = true and
    (exists ?se:SignalEvidence, ?rx:Power)
      req_evidence(?se) and
      peakRxPower(?se) = ?rx and
      ?rx =< 107.0 and
      meanEIRP(req_transmission) =< maxPower(?rx) and
      {F-3.0 .. F+3.0} in scannedFrequencies(?se) and
     (exists ?d:PeriodicSignalDetector)
       detectedBy(?se) = ?d and
       sampleRate(?d) >= 0.2 and
       dutyCycle(?d) >= 0.5;
end
```

C.22 BUGGY

```
/* Buggy policy to test parser and type checker. */

policy buggy is
  use ssc_params;
  defconst maxIBL : Powermask = pm(linear,[(1.0,2.0),(2.0,4.0)]);
  defconst maxOBL : Powermask = pm(linear,[(1.0,2.0),(2.0,4.0)]);
  defconst channels : [Frequency] = [5500.0, 5550.0, 5600.0];

  allow
    tpc(req_transmission) = true and
    centerFrequency(req_transmission) in {5470.0 .. 5725.0} and
    meanEIRP(req_transmission) <= 30.0 and
    powermaskLessThan(inBandLeakage(transmittedBy(req_transmission)),maxIBL) = true and
    powermaskLessThan(outOfBandLeakage(transmittedBy(req_transmission)),maxOBL) = true and
    bandwidth(req_transmission) =< 20.0 and
    centerFrequency(req_transmission) in channels and
    (exists ?se:SignalEvidence,?te:TimeEvidence)
      req_evidence(?se) and
        ((not ((exists ?r:RadarSignal) detectedSignal(?se,?r)) and
        req_evidence(?te) and
        timeAfter(lastCompleteEmptyScanTime(?se),
                    subtractTime(timeStamp(?te), td(1,0,0,0))) = true and
        timeDurationLongerThan(lastCompleteEmptyScanDuration(?se),td(0,0,0,60,0)) = true) or
        (((exists ?r:RadarSignal) detectedSignal(?se,?r)) and
        timeBefore(addTime(lastDetected(?se),td(0,0,30,0,0)),lastCompleteEmptyScanTime(?se)) = true
and
        timeDurationLongerThan(lastCompleteEmptyScanDuration(?se),td(0,0,0,60,0)) = true and
        sensingThreshold(se) >= -64));
end
```

APPENDIX D: REQUESTS

D.1 REQUEST1

```
request request1 is
  //centerFrequency(req_transmission) = 5200.0;   /* config1 */
  capability(req_radio,xg_v_5);                    /* device1 */
  capability(req_radio,p);                         /* device2 */
  capability(req_radio,pr);                        /* device3 */

  public const te : TimeEvidence;
  req_evidence(te);
  public const se : SignalEvidence;
  req_evidence(se);

  req_authorized(X);              /* distributed2 */

  /* identity1 */
  centerFrequency(req_transmission) = 255.0;
  req_org(redCross);

  /* lbt1 and lbt2 */
  peakRxPower(se) = -85.0E-2;
  meanEIRP(req_transmission) = 5.0E-3;

  /* lbt3 */
  detectedSignal(se,signal_x);

  /* time1, distributed1 */
  timeStamp(te) = ti(2006,12,30,11,50,0,0);

  /* time2 */
  minOffTime(req_transmission) = td(0,0,0,0,150);

end
```

D.2 REQUEST2

```
request request2 is

 /* This is for testing tv1.xg and tv2.xg */

 centerFrequency(req_transmission) = 500.0;
 maxOnTime(req_transmission) = td(0,0,0,0,500);
 minOffTime(req_transmission) = td(0,0,0,1,0);
 bandwidth(req_transmission) = 5.0;
 meanEIRP(req_transmission) = -55.0;

 public const se : SignalEvidence;
 req_evidence(se);
 peakRxPower(se) = -110.0;
 scannedFrequencies(se) = {450.0 .. 550.0};
```

```
  public const psd : PeriodicSignalDetector;
  detectedBy(se) = psd;
  sampleRate(psd) = 0.5;
  dutyCycle(psd) = 0.7;

end
```

D.3 REQUEST3

```
/* This is to test lbt4.xg
   Operations used: timeAfter, subtractTimes, powermaskLessThan
*/
request request3 is

    defconst IBL : Powermask = pm(linear,[(1.0,1.50),(2.0,2.50)]);
    defconst OBL : Powermask = pm(linear,[(1.0,1.50),(2.0,2.50)]);

    tpc(req_transmission) = true;
    //tpc(req_transmission) = false;

    centerFrequency(req_transmission) = 5550.0;
    meanEIRP(req_transmission) = 25.0;
    inBandLeakage(transmittedBy(req_transmission)) = IBL;
    outOfBandLeakage(transmittedBy(req_transmission)) = OBL;
    bandwidth(req_transmission) = 20.0;

    public const se : SignalEvidence;
    req_evidence(se);
    public const te : TimeEvidence;
    req_evidence(te);

    timeStamp(te) = ti(2006,5,5,12,0,0,0);
    lastCompleteEmptyScanTime(se) = ti(2006,5,5,7,0,0,0);
    lastCompleteEmptyScanDuration(se) = td(0,0,2,0,0);

end
```

D.4 REQUEST4

```
request request4 is

  /* This is for testing location1.xg and location2.xg */

  public const le : LocationEvidence;
  req_evidence(le);
  location(le) = loc(39.0, 75.0, 0.0);

end
```

D.5 REQUEST5

```
request request5 is

 /* This is for testing location1.xg and location2.xg */
```

```
 public const le : LocationEvidence;
 req_evidence(le);
 location(le) = loc(80.2, 80.4, 150000.0);

//loc(1.2, 3.4, 20000.0);

end
```

D.6 REQUEST6

```
request request6 is

  /* Tests the beacon1.xg policy */

  const se : SignalEvidence;
  req_evidence(se);
  detectedSignal(se,permitUse);
  scannedFrequencies(se) = {200.0 .. 400.0};
  lastCompleteEmptyScanDuration(se) = td(0,0,0,0,100);
  lastDetected(se) = ti(2006,5,17,11,59,58,0);
  timeStamp(se) = ti(2006,5,17,12,0,0,0);

end
```

D.7 REQUEST7

```
request request7 is

   public const le : LocationEvidence;
   req_evidence(le);
   location(le) = loc(2.0,3.0,100.0);

end
```