Cognitive Policy Radio Language (CoRaL)
A Language for Spectrum Policies
XG Policy Language

Version 0.1

ICS-16763-TR-07-001
SRI Project No. 16763
Contract No. FA8750-05-C-0230

April 1, 2007

Prepared by
Grit Denker, Daniel Elenius, Rukman Senanayake, Mark-Oliver Stehr, Carolyn Talcott, David Wilkins
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.

Contents

1. Executive Summary

Today, wireless communication is confronting two significant problems: spectrum scarcity and deployment delays. These problems derive from current procedures for the assignment of frequencies, which are centralized and static in nature [SP04]. The current scheme cannot adapt to the rapidly changing spectrum needs of users from the government, military, and commercial worlds. New technologies often cannot be used effectively because of this inflexibility, but they also provide the basis for solutions.

Spectrum is no longer sufficiently available, because it has been assigned to primary users who own the privileges to their assigned spectrum. However, studies [BM03] have shown that most of the spectrum is, in practice, unused most of the time. This observation was the starting point for DARPA's NeXt Generation (XG) Communications program, which proposes opportunistic spectrum use to increase spectrum availability. To achieve opportunistic spectrum use, radios must have the following capabilities

- Sensing over a wide frequency band and identifying primaries
- Characterizing available opportunities
- Communicating among devices to coordinate the use of identified opportunities
- Expressing and applying interference-limiting policies (among others)
- Enforcing behaviors consistent with applicable policies while using identified opportunities
-

Because of the large number of operating dimensions to be considered (frequencies, waveforms, power levels, and so forth) and the ever-changing nature of regulatory environments and application requirements, it is not feasible to design and implement optimal algorithms that allow radios to flexibly make use of available spectrum over time. Instead, a flexible mechanism must be provided that supports spectrum sharing while ensuring that radios will adhere to regulatory policies. The solution must be able to adapt to changes in policies, applications, and radio technology. The XG Program has embraced a solution based on policies.

We have implemented a device-independent Policy Reasoner (PR) that provides a software solution to opportunistic spectrum access. Our approach allows encoding of spectrum-sharing policies, ensures radio behavior that is compliant with policies, and allows policies to be dynamically changed. The PR either approves or disallows every transmission candidate proposed by a radio, based on compliance with currently active policies. Flexibility and spectrum sharing are achieved by expressing policies in a declarative language based on formal logic, and allowing devices to load and change policies at runtime. This document describes the Cognitive (Policy) Radio Language CoRaL. The reasoning architecture and techniques are described in [XGArch].

2. Introduction

2.1. Purpose and Scope

This document describes the policy language proposed by SRI for DARPA's neXt Generation (XG) communications program. It lays out the motivation for such a language within the context of the XG program, presents the key concepts and features of the proposed language, and discusses techniques for expressing such policies, underlying reasoning techniques, and issues regarding authoring and management of policies.

This document is a Request for Comments (RFC). It is important to note that ideas proposed within this document will evolve with feedback from the community at large, the goal of this document being to evolve this draft to a point where it becomes a single reference point for all aspects of the XG policy language developed by SRI. Since our language was designed to support the definition of spectrum-access policies and to be extensible so that unanticipated policy types can be encoded that might appear in the context of other application domains of cognitive radios, we call this language Cognitive (Policy) Radio Language (CoRaL).

BBN Technologies developed the XG Policy Language Framework RFC with the latest version published in April 2004 [XGPLF]. CoRaL is a new language for policy specification that is very different from what has been proposed by BBN. However, we will reuse some of the ontological concepts proposed by BBN for domain knowledge representation.

This document provides a detailed discussion of all issues pertaining to CoRaL, but does not provide details regarding the XG architecture, the reasoning technology or its components. A thorough discussion of the architecture can be found in the XG Architecture RFC [XGArch].

2.2. Motivation

The following is an excerpt from the XG Vision Document [XGV] outlining the main factors motivating the DARPA XG communications program.

"There are two significant problems confronting wireless communications with respect to spectrum use:

♦ *Scarcity:* The current method of allotting spectrum provides each new service with its own fixed block of spectrum. Since the amount of usable spectrum is finite, as more services are added, there will come a point at which spectrum is no longer available for allotment. We are nearing such a time, especially due to a recent dramatic increase in spectrum-based services and devices.

♦ *Deployment difficulty:* Currently, extensive, frequency , system by system coordination is required for each country in which these systems will be operated. As the number, size, and complexity of operations increase, the time for deployment is becoming unacceptably long.

Both problems are a consequence of the centralized, static nature of current spectrum allotment policy. This approach lacks the flexibility to aggressively exploit the possibilities for dynamic reuse of allocated spectrum over space and time, resulting in very poor utilization and *apparent* scarcity. […]

In order to address the scarcity and deployment difficulty problems, XG is pursuing an approach wherein static allotment of spectrum is complemented by the opportunistic use of unused spectrum on an instant-by-instant basis, in a manner that limits interference to primary users. In other words, the basic idea is this: a device first "senses" the spectrum it wishes to use and characterizes the presence, if any, of primary users. Based on that information, and regulatory policies applicable to that spectrum, the device identifies spectrum opportunities (in frequency, time, or even code), and transmits in a manner that limits (according to policy) the level of interference perceived by primary users. We term this approach *opportunistic spectrum access*."

Opportunistic spectrum access can be realized if the following capabilities can be achieved (a more in-depth discussion can be found in [XGV]):

a. Sensing over a wide frequency band and identifying primaries
b. Characterizing available opportunities
c. Communicating among devices to coordinate the use of identified opportunities
d. Expressing and applying interference-limiting policies (among others)
e. Enforcing behaviors consistent with applicable policies while using identified opportunities

Because of the large number of operating dimensions to be considered (frequencies, waveforms, power levels, and so forth) and the ever-changing nature of regulatory environments and application requirements, it is not feasible to design and implement optimal algorithms that allow radios to flexibly make use of available spectrum over time. Instead, a flexible mechanism must be provided that supports spectrum sharing while ensuring that radios will adhere to regulatory policies. The solution must be able to adapt to changes in policies, applications, and radio technology. The XG Program has embraced a solution based on policies. The next section gives more detailed arguments for the policy-based approach.

SRI's project focuses on the last two capabilities listed above, and this document focuses on the language for expressing policies. Spectrum agility, the ability to sense and transmit on unused spectrum, is central to the XG effort and achieving this depends on the ability to express interference-limiting policies and being able to reason about devices with behaviors based on such policies.

SRI will develop technology to achieve dynamic behaviors for devices based on policies. We will provide techniques that ensure radio behavior that is compliant with policies and that allow policies to be dynamically changed. The former is achieved by a device-independent policy reasoner that decides whether a transmission candidate as proposed by a radio is compliant with the policies. We achieve the latter by allowing devices to load and change policies on the fly instead of embedding policies into hardware, firmware, or device-specific software. We refer to devices that obey policies and avoid harmful interference[1]as "policy-based devices." These devices are flexible and adapt their behavior by changing declarative policies instead of software or hardware.

2.3. Benefits of Policy-Based Spectrum Sharing

In current radios, policies are programmed or hardwired into the radio and form an inseparable part of the radio's firmware. Typically, radio engineers use imperative (procedural) languages such as C for radio software. One could envision implementing spectrum-sharing algorithms and behaviors on radios in a similar manner.

However, this approach has obvious drawbacks. Any change in policies requires reimplementation (and reaccredidation) in the firmware of every radio that might operate in bands affected by the changes. Clearly, this approach does not scale well as technological advances lead to an increasing number of radio designs. Further, it is not scalable or flexible enough to deal with policies that are written by the many authorities in more than 200 countries or that may initially change frequently. Further compounding the problem is the fact that spectrum-sharing policies will likely have a larger number of operating dimensions (such as the state of sensed spectrum), and may initially change frequently as best practice is discovered or additional opportunities exploited.

The key difference in our approach is that declarative policies are expressed in terms of "what" should be protected or made available rather than "how" spectrum is protected or made available. Such policies are higher level than typical radio code and free from implementation details. This often makes it easier to intuitively grasp the meaning of a policy. Several considerations argue for this policy-based approach over encoding spectrum-sharing algorithms directly in radios:

- Radio behavior can quickly adapt to a changing situation. While policies themselves can be written to behave differently in different situations, the main advantage is that policies can be dynamically loaded without the need of recompiling any software on the radio. For example, a policy might be loaded to more aggressively exploit spectrum-sharing opportunities in emergencies.
- Policy changes can be limited to certain regions, frequencies, time of day, or any other relevant parameter. Since policies are platform independent, they can be loaded on

[1] (Harmful) Interference is defined in the Spectrum Policy Task Force Report as "interference which endangers the functioning of a radio navigation service or other safety services or seriously degrades, obstructs, or repeatedly interrupts a radio communication service operating in accordance with these [international] Radio Regulations."

different types of radios. Thus, new policies must only be uploaded into a radio to take effect, because each radio runs the policy reasoner on the currently loaded policies.

- Our approach decouples policy definition, loading, and enforcement from device-specific implementations and optimizations. One advantage is a reduced certification effort. When policies, policy reasoners, and devices can be accredited separately, accreditation becomes a simpler task for each component. Changes to a component can be certified without accrediting the entire system. We can certify the PR and each policy once, independent of the radio, and then test device configurations to see whether they correctly interpret PR outputs. Thus, currently certifying n device configurations for m policies would require to consider all *n*m* combinations. But in the case of the policy-based approach to XG devices, n such device configurations can be certified by just doing *n+m+1* tests, namely certifying the policy reasoner and each policy once, independent of the radio, and then test the n device configurations to see whether they correctly interpret the results from the policy reasoner. As a consequence, a change in the device implementation results in only one new check. In effect, the cost of accrediting the policies and policy reasoner is shared across all radio platforms. Radios can dynamically load accredited policies without additional certification.
- Another advantage of decoupling policies from radio implementation is that devices and policies can evolve independently over time. If a radio does not “understand” a policy, and cannot fulfill its requirements, it will not have transmissions approved by this policy, thus missing opportunities but avoiding potentially creating interference. On the other hand, if a radio has more capabilities than required by a certain policy, it can use just what is required. Thus, new policies do not require changes in radio software or hardware, and existing policies will work on new radio hardware. Today a cyclic dependency exists where regulatory bodies must wait for technology and technology must wait to see what the policies look like. This dependency is resolved by the policy-based approach that allows radio technology to develop in advance of policies and vice versa.
- A policy-based approach is extensible with respect to the kinds of policies that can be expressed. While we already know many relevant parameters and the interrelationships between various categories of policies, including structural relations such as hierarchies, we cannot foresee the degrees of freedom in policy definition that may be desirable in the future. Our approach provides the means to define new policy parameters that are relevant to spectrum-access decisions. Example parameters include functional allocations of spectrum (e.g., emergency response or aeronautical radio navigation), geographic restrictions (e.g., U.S. vs. foreign policies), temporal restrictions (e.g., time of day), host nations or authorities (e.g., U.S. vs. Europe, commercial vs. governmental), service restrictions (e.g., civil services, electronic warfare, Joint forces), and international vs. national policies on the same bands (e.g., maritime distress).
- An unprecedented amount of freedom and control of spectrum is possible as stakeholders can shape spectrum policies (as allowed by regulations) to best fit their objectives.

.

The basis for policy-defined radios is a policy language that serves as an interface between at least two different viewpoints, namely that of the regulators and that of the radio engineers. Thus, we must provide a language for expressing policies, and methods for interpreting policies and supporting efficient policy enforcement with automated reasoning.

2.4. CoRaL, BBN's XGPLF, Ontologies, OWL, Rules

2.4.1. CoRaL vs. BBN's XGPLF

We investigated BBN's XG Policy Language Framework (XGPLF) [XGPLF] and identified some shortcomings with respect to expressing some regulatory policies. One of the major drawbacks in BBN'S XGPLF stems from the fact that language constructs are defined without giving a denotational or operational semantics. For example, XGPLF allows the definition of a variety of expressions, including some that would serve a purpose similar to constraints in CoRaL, but the semantics of these expressions is not clear and the logical framework in which they should be interpreted is likewise not clear. In this version of this document we focus on the language constructs and their use in policies examples. However, the language has been designed with a denotational semantics in mind, which will be added to future versions of this document.

Moreover, BBN's XGPLF is based on OWL [OWL], and, thus, suffers the shortcomings that come along with the limited expressiveness of OWL (see Section 2.4.5).

Other shortcomings in BBN's XGPLF include its inability to adequately handle the complexity of real-world policy design. Many international regulatory bodies each independently specify policies, yet the resulting policies must be considered together for XG radio operation. Several well-known techniques can help to reduce the complexity in policy specification and make policy reasoning efficient.

One such technique is inheritance, which allows structuring policies to avoid redundant specification. For example, there will be rules that apply to all broadcast services, such as TV or HDTV. These rules can be defined on a "Broadcast" class; other, more specific, constraints for operating in TV bands will be defined in rules that operate on the "TV" subclass of "Broadcast." A rule of the form "if C then A" (read "if Condition C then action A"), where C is of class "Broadcast" would also match for all subclasses of "Broadcast."

Meta-policies and policy overriding allow the application of policies to be customized, for example, during crises situations. Meta-rules could define how to resolve a conflict between rules or in which order to apply rules. Another application is the definition of exceptions to a given policy set. A user may want to reuse the policy base of Europe, but exclude certain rules from the set. This could be represented in a meta-policy.

Future versions of CoRaL will investigate techniques to handle policy complexity through inheritance, meta-policies, or policy overriding.

2.4.2. The Old Web and the Semantic Web

On today's Web, XML [XML] is the common representation formalism (along with auxiliary specifications like XML Namespaces [XMLNamespace] and XML Schema [XMLSchema]). XML provides a generic syntax that can be used for many representational needs. Thus, many existing XML parsers and other tools are freely available to read, manipulate, and generate this syntax.

XML claims to be self-describing—tags have names describing their meaning. Meaning is the key notion here. It is clear that XML tags have meaning only to humans. Thus, Alice generates an XML document with some named tags; Bob picks it up and figures out what the intended meaning of the tags might be.

With XML Schema, this process provides more of a common base for tags. An XML Schema describes all the types of tags that might be used, and assigns namespaces to them. Alice generates an XML document using an XML Schema document with which both Alice and Bob are familiar; Bob picks up the XML document and understands the meaning of it, since he understands what the elements in it mean according to the schema. However, the "meaning" of elements is still not accessible to a computer in the sense that no automatic inferences using the meaning of elements is possible.

So far, we have described how knowledge representation works on the "old" Web. But in the vision of the Semantic Web, the XML technologies are just low-level building blocks, on which something more sophisticated can be built. As we have seen, the use of XML simplifies syntactic concerns, and XML Schema give some structure to knowledge representation. The Semantic Web takes things a step further, replacing (or augmenting) XML Schemas with ontologies. An ontology is a "formalization of a specification of a domain" [Gruber]. A domain is what the ontology describes and talks about. This could be flights, hotels, Web services, books, and so on. Formalization means that a logic is used to give a formal semantics to an ontology. Formal semantics gives the ability to infer implicit information from explicit information, and to check the consistency of ontologies.

It is important, however, to understand that there is still, and will always be, an informal specification of the domain. An ontology can formally relate different ontological elements to each other (e.g., it can say that "all Dalmatians are dogs" and that "all dogs have four legs"), but it cannot say what the elements represent in the real world (e.g., it cannot tell us what a dog or a leg is). Machines can thus "understand" ontologies only in a very limited sense.

We note that ontologies predate the Semantic Web by a wide margin. If one counts generously, by about 2500 years (Aristotle did some of the first work in logic and ontologies). On a more conservative estimate, by several decades (computer scientists have been working on knowledge representation since the conception of Artificial Intelligence). The Semantic Web applies existing ideas to new ones, namely applying logic and ontologies to the Web.

2.4.3. Ontologies, OWL and CoRaL

XG needs a common ontology for policies so that various policy-making bodies can consistently and unambiguously refer to radios, radio capabilities and parameters, and the relevant properties of the current radio environment. In the XGPL Architecture RFC [XGArch], we describe a set of basic parameters that are common to many policies. These parameters are used to form transmission requests and replies, and policies express constraints on these parameters. Policies can also introduce new concepts themselves. For example, a TV-band policy might introduce a concept of TV Stations.

So let us take a closer look at OWL [OWL] to see whether it could be of use. CoRaL is not primarily about the Web, though XG policies might be published online. Even though OWL was introduced as a Web ontology language, it can be, and has been, used for other knowledge representation purposes.

We are not very concerned with the syntax of OWL. OWL uses XML for that purpose and we can easily provide an XML representation of CORAL for machine readability. However, we are concerned with the logic of OWL. OWL is intended to describe the structure of ontologies. The main concepts are classes, properties, and individuals. Individuals belong to classes (e.g., Alice is a Person), and properties link individuals to other individuals (e.g., Alice's mother is Julie) or to concrete values (e.g., Alice is 35 years old). OWL allows us to describe how these concepts relate to each other using various axioms and statements. OWL is not the first language of its kind, but part of a family of Description Logics [DLHB], which have been studied for a two decades. OWL, like most Description Logics, is a decidable fragment of first-order logic[2].

The main inference in OWL is subsumption. Given a class description A and a class description B, the question "Does A subsume B?" is equivalent to asking "Are all B's A's?" This kind of reasoning is useful to analyze ontologies that define many classes, to find implicit relationships between classes, and to make sure that the intended meaning of an ontology conforms with the inferred meaning. It is also possible to encode other kinds of problems as class subsumption problems. However, there are clearly many reasoning problems that cannot be encoded and solved using OWL.

2.4.4. Expressiveness vs. Tractability

The most important trade-off in the design of an XG policy language is the trade-off between expressiveness and tractability. A more expressive logic is able to represent more complicated problems and solve them using logical inference. However, it will generally be less efficient in terms of space and time consumption. In the worst case, the logic will be undecidable—we will not have any guarantees of ever getting back the answer for an inference problem, even given infinite processing power and memory[3]. For these reasons, it is important that the

[2] Decidable, but not, in the general case, tractable—OWL has worst-case ExpTime complexity.
[3] This is the case for full first-order logic, for example.

expressive features of the XG policy language are finely tuned to the problems it is intended to solve.

2.4.5. Why not OWL for CoRaL?

The expressiveness of OWL is finely tuned to the problems it is intended to solve. It does ontological reasoning, that is, reasoning about class structures, very well. So the question to ask is “Is this the kind of reasoning we need for Policy Reasoning?”

To determine what kind of expressiveness and reasoning we need in CORAL, we worked with several example policies (such as Dynamic Frequency Selection [DFS]). We first tried to encode the policies in OWL. As we have mentioned, many different kinds of problems can be encoded as class-subsumption problems, and to some extent we were able to do so with XG policies. However, we ran into expressive limitations, and our encoding was not natural, which is never a good sign.

The next step in our requirements analysis was to encode the policies in a straightforward, “natural” way in something close to first-order logic, a very expressive language. These experiments provided some insight into what kind of expressiveness we need. The following is a list of required features derived from our analysis.

- *Constraint satisfaction.* Constraints using numerical (in)equalities are central to XG policies. This is not supported by OWL.
- *Rules*. It was natural to express the “rules” of XG policies as Prolog-style rules. This is not supported by OWL.
- *Data structures*. XG policies make heavy use of tables, lists, graphs, and so forth. These kinds of data structures can be encoded in OWL, but only inelegantly.
- *Functions*. Many concepts in our XG policies are most naturally expressed as functions. OWL does not support this.

What about ontologies? As we have mentioned, there is an ontological component in CoRaL. With a relatively small set of domain concepts, we were able to formalize various example policies. Moreover, these examples did not call for ontological axioms of the kind provided by OWL, not even simple subclassing. However, in the future we will further investigate whether ontologies and subclassing can be useful to support XG policy specification.

Thus, our current conclusion is that OWL does expressive ontological reasoning, which we have not found necessary, and that we need several kinds of reasoning that OWL does not provide. One solution is to extend OWL with the kinds of reasoning that it cannot currently do. Some extensions to OWL have been proposed. For example, the Semantic Web Rule Language [SWRL] is a proposal for adding rules to OWL. But this approach still does not handle requirements such as reasoning about numerical constraints and it also leads to a very expressive language that is undecidable.

To achieve an optimal trade-off between expressiveness and tractability, we chose to design a language tightly around the reasoning required by the XG policy problem.

2.4.6. Why not OWL, Rules and Procedural Attachments for CoRaL?

Since our investigations made it clear that OWL itself was not sufficient for a policy language, the next idea was to combine OWL with some rule markup language and procedural attachments to define a suitable policy language. A rule markup language would allow capturing the main part of a policy, and procedural attachments are useful to express functions such as powermasks or other XG-domain specific concepts.

We are currently investigating the specific trade-offs between such an approach and the CoRaL language. This trade study must take into account not only expressiveness and appropriateness of the language for XG policies, but also existing and required reasoning technology.

We have reason to believe that even though a combination of OWL plus rules and procedural attachment will be powerful enough to express the policies we have encountered so far, there is no existing tool or reasoning technology that combines the various kinds of proof rules necessary to provide powerful reasoning technology for XG radios that not only return yes or no answers, but also information as to what further constraints the radio would have to satisfy in order to be granted transmission.

The latter feature, that is, returning constraints to the radio that, if satisfied, would lead to valid transmissions, is very important to achieve truly cognitive radios. Reasoners that reply only no (or yes), but do not give more information as to why the loaded policies do not allow the attempted transmission are not helpful to a cognitive radio. But cognitive radios that can gain an understanding of the policy space by making various transmission requests are potentially very powerful. Though each request is not a valid transmission and results in a "no" from the reasoner, the radio is told what further requirements it needs to satisfy in order to turn the requests into transmissions that would be compliant with the policies. The radio can exploit this information in strategies that lead to successful transmission requests.

Our current investigation seems to indicate that the reasoning technology required for such powerful reasoners is not available in existing reasoners that combine OWL with some rule language. We believe that the expressiveness of CoRaL, and the proof system we have developed for CoRaL, is necessary for an XG policy framework that supports reasoning and returning requirements to the radio. We will publish the outcome of the trade space study in the future.

2.5. SRI's XG Policy Control Project—Objectives

SRI's **XG Policy Control (XGPC)** project proposes to develop a language and algorithms that will enable compliant XG operation for a broad range of spectrum usage regulations. In addition, authoring tools for creating, analyzing, and visualizing policies will be developed.

These tools will enhance usability and contribute to acceptance of the technology. The main innovations of the proposed XGPC approach include

- An expressive and extensible policy language with denotational and operational semantics, for describing policies that meet the needs of a wide variety of spectrum regulation bodies, and for supporting policy reuse and efficient reasoning via a disciplined use of inheritance and meta-policies.

- A policy-conformance algorithm that checks the compliance of candidate transmissions using efficient, state-of-the-art reasoning technology. The design of the algorithm is independent of, yet easy to integrate into, any radio design, to encourage competitive, best-of-breed XG radio development.

- A capability to search for transmission candidates if the radio makes an underspecified, unbound transmission request, which will yield candidate parameter bindings that will satisfy policies. This capability can considerably increase the use of existing spectrum opportunities of which a radio is not aware.

- The policy language and the reasoning algorithms will be designed to be robust in the presence of conflicts as well as the absence of complete information.

- Policy authoring tools for creating and visualizing policies to support domain specialists.

The adequacy of the language will be illustrated through a library of commonly used example policies that stem from various regulatory bodies and span a wide range of spectrum services and usages. The algorithms will prove the feasibility of the policy-based approach to XG operation. The project will provide reference implementations for the reasoning algorithms in a stand-alone manner.

In summary, the objectives of SRI's XG Policy Control project are threefold:

- Expressing policies
- Reasoning about policies
- Policy authoring and management

2.5.1. Expressing Policies

The language designed by SRI is referred to as XG Policy Language 2 or CoRaL. CoRaL is an extensible language with denotational and operational semantics. The denotational semantics explains language concepts at a high abstraction level, independent of a specific implementation. Operational semantics refer to specific reference implementations of the language.

CoRaL has all the constructs needed to express XG policies in an intuitive, straightforward, and concise manner. It also supports machine readability and allows for efficient reasoning that is flexible enough to support future devices (with new feature sets). CoRaL's linguistic constructs are discussed in more detail in Section 3.

We point out that this report focuses on the specification of single policies. Future work will investigate techniques for combining multiple policies.

2.5.2. Reasoning about Policies

With respect to the reasoning techniques to be used in XG, we need to support at least three kinds of reasoning, which are further explained below:

1. Policy-conformance checking of candidate transmissions
2. Capability to search for transmission candidates
3. Policy analysis

Policy-conformance checking concerns radio requests to the policy reasoner that correspond to fully instantiated requests or candidates. A candidate consists of a set of values that fully characterize the current situation of the radio (sensed data, radio characteristics, and so on). A fully instantiated candidate is such that all variables in the candidate have concrete values (a ground term) and the set of variables is sufficient for the policy reasoner to decide whether the request is in conformance with the policies. The policy reasoner can thus return to the radio either "Yes" (meaning "transmission with parameter values as specified in candidate is allowed") or "No" (meaning "transmission with parameter values as specified in candidate is not allowed") as a reply.

There might be cases where a radio cannot or does not wish to form a fully specified candidate. A basic XG device that requests authorization to transmit may not identify all the requirements it needs to fulfill prior to making a request. This may occur because the device is not aware of all the applicable policies or to a strategy employed by the radio to not initiate costly sensing operations unless required. For example, a radio requests transmission at 100 mW, which the policies permit only if the transmission time is less than 2 ms. As the radio is not aware of this restriction, the candidate submitted to the policy reasoner will not match the policy rule. This is an example of a candidate transmission that is underspecified or unbound. It is useful if the policy reasoner accepts such unbound requests and is able to check satisfiability of policies by determining further constraints that must be satisfied. In the above example, the reasoner determines that the request would be valid if the radio limits the transmission to 2 ms. We refer to this capability as *search for transmission candidates*.

Policy analysis is useful in helping regulators and policy designers to understand the consequences of a set of policies. Analysis will become more important as policy sets grow. It will be difficult to intuitively grasp the meaning of a complex policy database that has been derived by combining several policy sets. We envision an analysis tool that permits defining scenarios of XG operation and determining permissible or prevented usages under a given set of policies using model-checking approaches. For example, we can assert a statement such as "If the measured sensor data is less than threshold x, then transmission on frequency y with power z is possible." The model-checking analysis tool would then find a counterexample to the negated assertion. If it is successful, it proves that the original assertion was true. Another approach to policy analysis is the use of theorem proving or satisfiability techniques. Given two policies P1 and P2, one can employ a theorem prover to determine whether P1 logically implies P2 or vice versa. If so, the policy set is redundant and the same reasoning results could be achieved with a smaller policy set. Satisfiability techniques can be used to determine whether the conditions of two policies can be satisfied together. If so, the two policies overlap and could be presented to a designer for further investigation—perhaps the implications in the rules are inconsistent.

2.5.3. Authoring Policies

In this report, we focus on policy-language concepts and their mathematical semantics. We illustrate the language features with examples and intuitively describe how the reasoning algorithms will work in the context of the examples. In future reports, we will provide particular operational semantics of the language and reasoning algorithms as well as authoring concepts for XG policies. A more comprehensive treatment of issues pertaining to authoring of policies will be presented in the XG Policy Management RFC [XGPM].

2.6. XG Policy Language Requirements

Several requirements exist for the design of a policy language in the XG domain. Here we describe some of the main requirements that informed our design.

2.6.1. Permissive and restrictive policies

CoRaL as a language should provide linguistic constructs for supporting two types of policies: permissive and restrictive. Permissive policies describe conditions under which transmission is allowed and restrictive policies describe conditions under which transmission is not allowed. The conditions of permissive policies impose constraints on a possible transmission, whereas the conditions in a restrictive policy define situations in which transmission requests will be denied. CoRaL provides built-in predicates "allow" and "disallow" and the ability to specify top-level policy rules such that a policy can be either permissive or restrictive (see Section 3.6). The XG approach is based on the broad assumption that XG devices will be designed to operate on a 'do not interfere' basis. That is, the device will not transmit if not explicitly authorized to do so by a policy governing its behavior. Therefore for most devices the default action permissible would be 'do not transmit'.

2.6.2. Ontology of domain concepts

In general, before a request for a transmission can be authorized, three types of information need to be available: the capabilities of the radio, the current environment of the radio, and the characteristics of the requested transmission. Such information is heavily dependent on domain concepts, and, thus, we define a common ontology for all such concepts. At present, CoRaL provides definition of concepts to be shared among policies and also mechanisms for extending such definitions (see Section 3.4 for further details).

2.6.3. Stateless

CoRaL was designed for both the linguistic requirements of desired policies and the reasoning capabilities that need to be provided for effective utilization of policies. Maximizing the benefits of opportunistic spectrum access by generically reasoning about machine-understandable policies will require complex inferences. Thus, it is imperative that the first generation of reasoners used in this framework be easy to verify and implement. CoRaL was therefore designed to be a stateless system, in which maintaining state and tracking state transitions are not supported. A stateless policy reasoner will be much easier to verify, and will also be simpler to design and implement. A stateful reasoner might be nearly impossible for governing authorities to verify because of the potentially unbounded number of states and the unpredictability of the quality and timing of state updates.

2.6.4. Reasoning about numerical constraints

Policies often define some conditions about transmission opportunities in terms of numerical constraints. For example, a policy may restrict transmissions to carrier frequencies within a certain range or to situations in which the peak received power is lower than a given threshold. These conditions are numerical constraints that the policy designer will define as desired. CoRaL must be able to represent these numerical constraints and the policy reasoner has to adequately reason about them. For example, if a policy defines the allowable range of carrier frequencies to be between 3100 and 3300 Hz, then the policy reasoner must be able to recognize whether or not a given carrier frequency is within this range.

2.6.5. Usable for policy designers

CoRaL is specifically designed to meet the requirements of the XG communication program and therefore provides a collection of features custom designed for policy specification and authoring. These features include,

- XG-specific data types support policy authors in constructing XG policies. For example the commonly used concept of ‘power mask’ can be encoded in CoRaL using special “syntactic sugar” (see Section 3.2.2). Special keywords such as ‘symmetric’, ‘linear’, and ‘step’ allow the author to easily complete the task of defining a specific power mask (or, for that matter, other functions).
- CoRaL provides flexible, yet not overly complex, language constructs for defining policy rules in a concise and logically correct way.
- The language supports an extensible type system. Policy designers can introduce user-defined types to meet their requirements.

3. Cognitive Radio Policy Language (CoRaL)

3.1. CoRaL Overview

As described in this higher-level overview of CoRaL, the language design is based on four main goals:

a. Be expressive enough to adequately express XG use cases
b. Support inference and reasoning capabilities
c. Be flexible and extensible enough to be long lived
d. Support machine readability

We have tested CoRaL on various types of policies. But XG use cases must also include typical radio requests and expected replies from the reasoner. This will give insight into the needed reasoning capabilities. The XG policies were developed in collaboration with Shared Spectrum Company and reviewed by the government team. The policies are summarized in 0. However, the reasoning technology will be explained in [RFC-Arch].

Extensibility is a key component of the design of CoRaL and is provided, in part, by linguistic abstraction. As the language evolves and matures, it becomes necessary to add new features to the language, that is, to add linguistic constructs. In most cases, a new construct requires a new grammatical construct to be added to the language syntax. However, in those cases in which the semantics of the new language construct is expressible in the underlying semantic domain, no additional complexity is introduced. CoRaL has a rather expressive underlying semantic domain (a subset of First Order Logic with types) and we anticipate that many syntactical extensions will be semantically explainable in this domain. Therefore, CoRaL will be extensible.

Machine readability allows policies to be loaded onto the radios. For now, we implemented a parser for CoRaL. An XML-based syntax for CoRaL could be given, though we decided to follow a different strategy. As discussed in Section 2.4.6 we are investigating the use of OWL, rules, and procedural attachments. We are considering providing an OWL/Rule syntax for a subset of CoRaL. We will discuss this in future releases of this document.

The main interest of regulators is the specification of admissible transmission behavior. They are usually not interested in how policy conformance is checked, as long as the check is correctly implemented. This is referred to as the *soundness* of the check. Regulators are not interested in the strategy used to discover opportunities, assuming that policy-conformance is ultimately enforced. Furthermore, they are not interested in verifying if a radio's strategy reasoner can exploit all transmission opportunities. Various trade offs (e.g., cost of sensing vs. need for spectrum), radio capabilities (e.g., ability to sense the spectrum), and the quality (degree of completeness) of the strategy reasoner itself will all affect which opportunities are exploited.

The main interest of radio engineers, on the other hand, is to exploit as many policy-conforming transmission opportunities as possible. This naturally leads to an incentive to enhance capabilities of both the strategy reasoner and the policy-conformance reasoner.

To best support both of these viewpoints, a policy language with a simple and unambiguous semantics is needed. Since the foremost objective is to specify—as opposed to implement—policy policy-conforming behavior, a declarative language is a considerably better fit than an imperative language like C. Cognitive (Policy) Radio Language (CoRal) is a domain-specific logic-based specification language designed to meet these needs.

CoRaL is a declarative language based on a typed version of classical first-order logic enriched by built-in and user-defined concepts. Examples of domain concepts that are shared among most policies are frequency, power, location, powermask, and signal.

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. Here, we describe the language in a bottom-up fashion, starting with the smaller building blocks, and ending with entire policies. This section gives only a quick overview of CoRaL for the novice user. A comprehensive introduction into CoRaL is given in Sections 3.2 to 3.6.

3.1.1. 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 Appendix B. 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 3.1.4), 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[4]. 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.

3.1.2. 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)**[5]
- Constants
- Lists, such as **[1,2,3]**
- Tuples, such as **(1,2)**
- Sets, such as **{1..5}**. The dots represent 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 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 expressions 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 3.1.4).

---

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

[5] 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)))**

3.1.3. 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 symbols 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.

3.1.4. 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 right 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 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 (;).

3.1.5. Policies and Ontologies

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 They are defined to support permissive as well as restrictive requirements. Policy rules are logical axioms that express under which *conditions* these predicates hold. Restrictive (disallow) rules take precedence over permissive (allow) rules when a transmission request is evaluated.

The axioms can involve any declared parameters, which represent capabilities of the radio and the results of sensing actions (among other things). These concepts are often defined in an ontology (see below).

Conditions can also use predicates, which express modes of operation, locations, and so forth. Thus, conditions allow for dynamic adjustment of policies to the current situation. For example, a rule could allow military radios to use the GSM band when a conflict starts, but not earlier. Clearly, such context-sensitive policies can respond to the situation in various ways, invoking either restrictive or permissive rules.

Numerical constraints are often used in policies specifications and can be directly expressed using built-in predicates in CoRaL (see Section 3.1.3). For example, a policy might require that for frequencies between 5000 and 5500 MHz, the transmission power should be at most 2dBm. A special syntax is available to specify powermasks, where the power is not constant but varies with frequency.

A policy can also be extended by rules in another policy without causing logical inconsistencies. For example, one policy may have a rule allowing the use of frequencies 5000 to 5500 MHz, whereas another policy might disallow the use of frequency 5250 MHz, as well as allow frequencies between 5200 and 6000 MHz. Thus, the combination of policies will allow the use of frequencies between 5000 and 6000 MHz, with the exception of frequency 5250 MHz.

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

ontology o1 is
  statements
end

Concepts that are common across several policies can be factored out into ontologies, which can represent hierarchies of types and related functions or predicates.
Formally, the only difference between ontologies and policies is that ontologies define only concepts and have no rules, whereas policies must have at least one rule and may also define concepts.

In summary, a CoRaL policy may (optionally) contain any of four types of *declarations*: type, constant, predicate, and rule. The main component is the rules, which define the behavior enforced by the policy. Rule declarations use notations that are introduced through type, constant, and predicate declarations. In the remainder of this section, we will introduce concepts in a bottom-up order, so that every concept is defined before it is used in a larger concept.

3.2. CoRaL Reserved Keywords

The next sections discuss the core constructs in the CoRaL language. For clarity, reserved keywords in CoRaL will be denoted by **Arial Narrow (Bold)** font in this discussion. User-defined literals that serve as names of data structures or user-defined types will be indicated using UPPERCASE. An CoRaL policy is a collection of valid statements, where each statement is terminated by a semicolon. Optional arguments or keywords in a specification will be enclosed in *[ ]*. Where a statement may contain more than a single optional component, the '|' symbol will be used to separate such components. The formal grammar of CoRaL can be found in Appendix A. The typing rules can be found in Appendix B. A CoRaL parser should first parse policies according to the grammar, and then type-check the policy according to the typing rules.

An alphabetical listing of all the reserved keywords used by CoRaL is given in Table 1 along with a brief description.

No	Keyword	Purpose
1	allow	Built-in constant that determines if a transmission request is approved]
2	and	Logical conjunction
3	bool	Built-in simple data type
4	const	Define a new constant
5	defconst	Define a new constant and assign a value
6	deftype	Define a new type based on an existing type
7	disallow	Built-in constant that determines if a transmission request is disapproved
8	end	Declare the end of a policy
9	exists	Existential quantifier
10	false	Built-in constant of type bool
11	False	Built-in constant of type 'Pred'
12	float	Built-in simple data type
13	forall	Universal quantifier
14	if	Declaration of a condition
15	implies	Logical implication
16	in	Membership axiom
17	include	Import shared constants into a policy
18	inf	Built-in constant indicating positive infinity
19	-inf	Built-in constant indicating negative infinity
20	int	Built-in simple data type
21	is	Declaration of a name of a policy, reply or request
22	linear	Structural axiom declaring a linear function
23	not	Logical negation
24	or	Logical disjuntion
25	policy	Define the start of a policy
26	Pred	Built-in type for declaring predicates
27	record	Group a collection of related variables
28	step	Structural axiom declaring a step-wise function (or type)
29	symmetric	Structural axiom declaring a symmetric function (or type)
30	True	Built-in constant of type 'Pred'
31	true	Built-in constant of type bool
32	type	Define a new type

Table 1 CoRaL Reserved Keywords

3.3. CoRaL Type System

CoRaL is a statically typed language based on a type system with three top-level types: function, simple, and predicate. The reserved keywords type and deftype are used in declaring and defining new types. A new type can be defined in terms of other types (using equality). The reserved keywords const and defconst are used to declare and define constants. Defining a constant involves assigning a value to the constant at the time of definition.

Figure 1 presents an overview of the type system available in CoRaL. The following discussion of the type system is based on the classification presented in this figure. User-defined types can be any of the three top-level types.

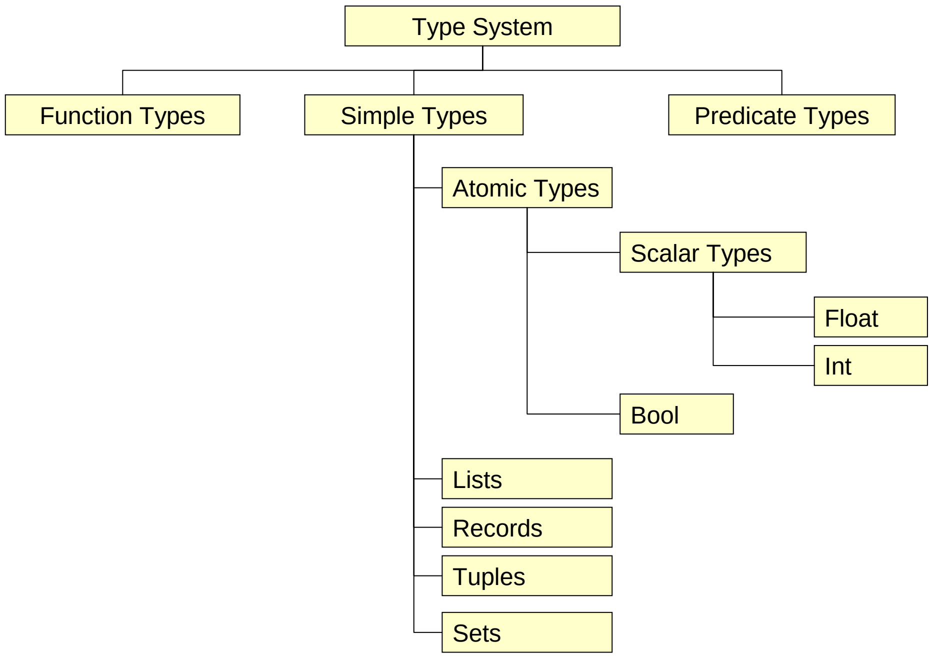


Figure 1 Overview of the type system supported by CoRaL

3.3.1. Simple Types

Simple Types are made up of atomic types and four constructors on simple types, namely lists, records, tuples, and sets. Combinations of these constructors should provide sufficient flexibility to succinctly express data aspects of XG policies.

Lists, records, tuples, and sets are all data types that depend on structural equality. That is, members of these data types are equivalent if and only if they are structurally identical.

3.3.1.1. Atomic Types

Atomic types are either the predefined atomic types, that is, the two scalar types 'Float', 'Int' and the type 'Bool', or a user-defined type. Using the deftype keyword, a user-defined type can be defined to be equivalent to another type. One of the more frequently used examples is the 'Frequency' type, which is an alias for the 'Float' type. The standard syntax for *defining* and *using* such types are similar, and are presented together in this section.

At present, CoRaL does not support used-defined *sub*types. We intend to extend the language to support this in the future.

The reserved keywords true and false are provided as built-in constants of the built-in type Bool.

3.3.1.1.1. Syntax

Type Declaration	type TYPEn ; deftype TYPEn = TYPE1 ; where TYPE1 is Float, Bool, Int, or another previously declared atomic type.
Constant Declaration	const CONST1 : TYPE1 ; defconst CONST1 : TYPE1 = V1 ; where V1 is a constant of type TYPE1 and TYPE1 is either Float, Bool, Int, or another previously declared atomic type.

3.3.1.1.2. Example

Type Declaration	deftype `Frequency` = Float ; deftype `Age` = Int ; type `TVType` ;
Constant Declaration	const `analog : TVType ;` const `digital : TVType ;` const `curFrequency` : `Frequency` ; defconst `minAge` : `Age = 300` ; defconst `channelAvailCheck` : Bool = true ;

3.3.1.2. Tuples

Tuples (as well as records) are used for grouping terms composed of (possibly) different types. The main difference between tuples and records is that tuples do not have field identifiers and therefore depend on the order of the terms for identification.

3.3.1.2.1. Syntax

Type Declaration	deftype TYPEn = (TYPE1, TYPE2, TYPE3, …) ; where TYPE1, TYPE2, TYPE3, etc. are previously declared simple data types.
Constant Declaration	const CONST1 : (TYPE1, TYPE2, TYPE3, …) ; defconst CONST1 : TYPEn = (V1, V2, V3,…) ; where V1, V2, V3 are constants of type TYPE1, TYPE2, TYPE3, etc., respectively.

3.3.1.2.2. Example

Type Declaration	deftype `Location` = (Float, Float) ; deftype `SignalToPower` = (`Signal`, `Power`) ;
Constant Declaration	defconst `loc1` : `Location` = (`43.56`, `78.32`) ; defconst `fullpowAnalog` : `SignalToPower` = (`fullPowerAnalogTV`, `10`) ; where `fullPowerAnalogTV` is a constant of type `Signal`.

3.3.1.3. Records

The *record* data type allows the user to define a structured compound entity. A record consists of a finite set of components. Each component consists of a field identifier and a type for each field. The field identifier is a label and the field may hold a constant of the corresponding type. The order of record elements does not matter, since access to an element of a record is achieved with the commonly used 'dot' notation. For example, the 'location' field of the 'station1' record declared in the above table could be accessed by specifying 'station1.location'. A field in a record may have a record type, in which case the 'dot' notation would be used multiple times for access to a field in the second-layer record.

3.3.1.3.1. Syntax

Type Declaration	deftype TYPEn = {FIELD1 : TYPE1, FIELD2 : TYPE2, FIELD3 : TYPE3, …} ; where TYPE1, TYPE2, TYPE3, etc. are previously declared simple data types. FIELD1, FIELD2, FIELD3, etc. are user-defined labels.
Constant Declaration	const CONST1 : {FIELD1 : TYPE1, FIELD2 : TYPE2, FIELD3 : TYPE3, …} ; defconst CONST1 : TYPEn = {FIELD1 = V1, FIELD2 = V2, FIELD3 = V3, ….} ; where V1, V2 etc are constants of type TYPE1, TYPE2, TYPE3, etc., respectively.

3.3.1.3.2. Example

Type Declaration	deftype `TVStation` = {`location` : `Location`, `gradeB` : `Polygon`, `type` : `TVType`} where `Location, Polygon` and `TVType` are previously defined simple data types. In particular, `Location` is a pair (tuple) of Floats.
Constant Declaration	defconst `station1` : `TVStation` = {`location` = (`54.5`, `23.65`), `gradeB` = `P1`, `type` = `analog`}; where `P1` is a constant of type `Polygon`, and `analog` is a constant of type `TVType`.

3.3.1.4. Sets

The *set* data type allows the user to define types based on a range of scalar values, that is, Int or Float or a type defined to be equivalent to a scalar type (such as Frequency). The range type declaration assumes the user will specify the minimum value of the range as the first argument and the maximum value as the second argument. A check to enforce this is not done by the type system. For example, a definition such as "defconst `r1` : {`Frequency`} = {`5350..5200`} ;" would result in an empty range.

3.3.1.4.1. Syntax

Type Declaration	deftype TYPEn = {TYPE1} ; where TYPE1 is a scalar type.
Constant Declaration	const CONST1 : {TYPE1} ; defconst CONST1 : {TYPE1} = {V1..V2} ; where V1, V2, are constants of type TYPE1, which is a scalar type.

3.3.1.4.2. Example

Type Declaration	deftype `FreqRange` = {`Frequency`} ;
Constant Declaration	defconst `allowedRange1 : FreqRange` = {`5150..5350`} ;

3.3.1.5. Lists

The *list* data type allows a user to define a simple type composed of a homogeneous listing of comma (‘,’) separated values. Since the ‘list’ type allows declaration of types based on other ‘simple types’, it is possible to create lists based on records, tuples, sets, and atomic types. It is not possible within CoRaL to define recursive lists.

3.3.1.5.1. Syntax

Type Declaration	deftype TYPEn = [TYPE1] ; where TYPE1 is a previously declared simple type
Constant Declaration	const CONST1 : [TYPE1] ; defconst CONST1 : [TYPE1] = [V1, V2, ….] ; where V1, V2, etc. are constants of type TYPE1.

3.3.1.5.2. Example

Type Declaration	deftype `DFSCarFreq` = [`Frequency`] ;

	deftype `BeaconSignals` = [{`beacon : Beacon`, `signalpower : Power`, `signalage : Duration`}] ; deftype `FreqRanges` = [{`Frequency`}] ; where `Beacon, Power`, and `Duration` are previously defined simple data types.
Constant Declaration	defconst `dfsFreqList` : `DFSCarFreq` = [`5180`, `5200`, `5220`, `5240`, `5260`, `5280`, `5300`, `5320`, `5500`, `5520`, `5540`, `5560`, `5580`, `5600`, `5620`, `5640`, `5660`, `5680`, `5700`] ; defconst `receivedBeacons` : `BeaconSignals` = [{`beacon` = `B1`, `signalPower = 200`, `signalage = 30s`}, {`beacon` = `B2`, `signalPower = 42`, `signalage = 110s`}, {`beacon` = `B3`, `signalPower = 90`, `signalage = 56s`] ; defconst `eirpRanges` : `FreqRanges` = [{`5150..5350`}, {`5470..5735`}] ; where `30s, 110s, 56s` are constants of type `Duration`, and `B1, B2,` and `B3` are constants of type `Beacon.`

3.3.2. Function Types

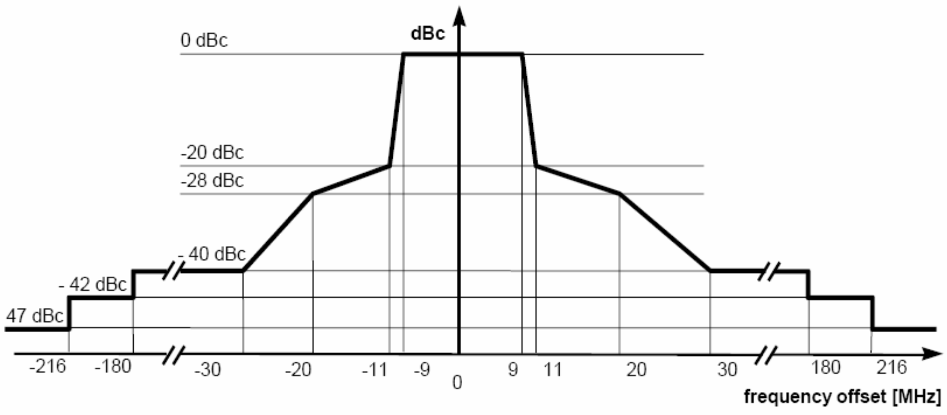
Here, we discuss function definitions.

3.3.2.1. Syntax

Type Declaration	deftype TYPEn = TYPE1 -> TYPE2 ; deftype TYPEn = (TYPE1,TYPE2,…) -> TYPEm ; where TYPE1, TYPE2, etc. are identifiers of previously declared types
Constant Declaration	const CONST1 : TYPE1 -> TYPE2; defconst CONST1 : TYPE1 -> TYPE2 = *[*symmetric*]* linear | step [(U1, V1), (U2, V2), …] ; where V1, V2, etc. are constants of type TYPE2 and U1, U2, etc. are constants of type TYPE1.

	const CONST1 : (TYPE1, TYPE2, ...) -> TYPEm;
	
	defconst CONST1 : (TYPE1, TYPE2, …) -> TYPEm =
	    [symmetric] linear | step [((U1, V1, …), W1), ((U2, V2, …), W2), …] ;
	
	where Ui, Vi (i=1…n) are constants of types TYPE1, TYPE2, etc. and W1, W2, etc. are constants of type TYPEm.

3.3.2.2. Example

Type Declaration	deftype `PowerMask` = `Frequency` -> `Power` ; deftype `Distance` = (`Location`, `Location`) -> Float ; where `Location` is defined as a tuple composed of a pair of floats. Thus, the above type definition is equivalent to deftype `Distance` = ((Float, Float), (Float, Float)) -> Float ;
Constant Declaration	const `maxInBandLeakage : PowerMask ;` To demonstrate the definition of a function in CoRaL, let us consider an example from the ETSI-DFS policy [DFS]. The following spectral power mask depicts the levels for unwanted emissions within the 5 GHz RLAN bands.  Such a power mask can be defined in CoRaL using the function type as demonstrated below: defconst `maxInBandLeakage` : `Powermask` = symmetric linear [(`0`, `0`), (`9`, `0`), (`11`, `-20`), (`20`, `-28`), (`30`, `-40`), (`180`, `-40`), (`180`, `-42`),

(216, -42), (216, -47), (inf, -47)] ;

OR

defconst `maxInBandLeakage` : `Powermask` = symmetric
linear [(0, 0), (9, 0), (11, -20), (20, -28), (30, -40)]
step [(30, -40), (180, -42), (216, -42), (inf,-47)] ;

The levels for unwanted emissions outside the 5 GHz RLAN band is given by the following table in [DFS]. For this table it is possible to use a step function to represent the specification, as demonstrated below:

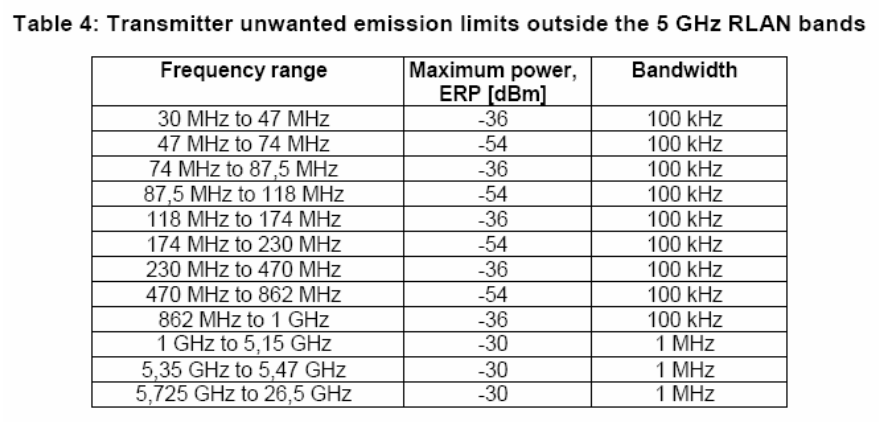
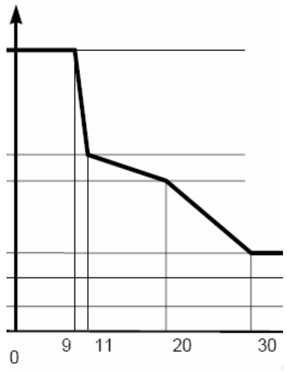


Table 4: Transmitter unwanted emission limits outside the 5 GHz RLAN bands

Frequency range	Maximum power, ERP [dBm]	Bandwidth
30 MHz to 47 MHz	-36	100 kHz
47 MHz to 74 MHz	-54	100 kHz
74 MHz to 87,5 MHz	-36	100 kHz
87,5 MHz to 118 MHz	-54	100 kHz
118 MHz to 174 MHz	-36	100 kHz
174 MHz to 230 MHz	-54	100 kHz
230 MHz to 470 MHz	-36	100 kHz
470 MHz to 862 MHz	-54	100 kHz
862 MHz to 1 GHz	-36	100 kHz
1 GHz to 5,15 GHz	-30	1 MHz
5,35 GHz to 5,47 GHz	-30	1 MHz
5,725 GHz to 26,5 GHz	-30	1 MHz

defconst `maxOutsideBandLeakage` : `PowerMask` =
step [(30, -36), (47, -54), (74, -36), (87.5, -54), …] ;

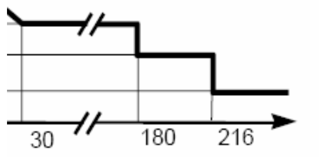
The ‘symmetric’ keyword (optional) is used to define a map that is symmetric around the Y-axis. This is helpful in making definitions more compact. The optional ‘linear’ and ‘step’ keywords are used to specify what type of interpolation is to be used in the intervening range of the given Cartesian coordinates.



For example, the partial definition linear [(0,0),(9,0),(11,-20),(20,-28),(30,-40)] specifies that linear interpolation should be used in the range of $0 \leq X \leq 30$.

This type of specification essentially breaks down the power mask to individual line segments specified using the Cartesian coordinates of the points that make up the line segments as shown in the figure to the left.

The ‘step’ keyword is used in specifying value ranges that take the form of a step function, such as the range of values shown in the figure to the right.



Two predefined constants of type Float are available for representing +∞ and -∞, namely inf and –inf.

3.3.3. Predicates

The Pred keyword is used for declaring types which are predicate types. There are four built-in predicates in XGPL2, namely allow, disallow, True, and False. The predicates allow and disallow have a special meaning associated with them. They are used exclusively to represent whether or not a device should be allowed to transmit , with respect to a given transmission request within a policy. The built-in predicates True and False are not to be confused with the built-in constants true, false of type Bool.

A user-defined predicate may consist of any number of arguments of a function type. Therefore, arguments to predicate types may be of any type other than predicate types.
Predicates are predominantly used as the consequent (head) of a policy rule.

3.3.3.1. Syntax

Type Declaration	deftype TYPEn = Pred ; deftype TYPEn = Pred(TYPE1, TYPE2, TYPE3, …) ; where TYPE1, TYPE2, etc. are previously defined function types.
Constant Declaration	const CONST1 : Pred ; const CONST1 : Pred(TYPE1, TYPE2, TYPE3, …) ; defconst CONST1 : Pred = *[*True | False*]*; where TYPE1, TYPE2, etc. are previously declared function types and True and False are predefined constants of the predicate type.

3.3.3.2. Example

Type Declaration	The grammar allows usage of ‘deftype’ to declare a ‘Pred’ type, but we do expect this to be used infrequently.

Constant Declaration	`const frequency_ok : Pred ;`
	
	`const inband_leakage_ok : Pred(Frequency -> Power) ;`
	
	`const in_analog_gradeB : Pred(Location, TVStation, Int) ;`

Comparisons on functions representing powermasks are very common in XG. For example, in the ETSI-DFS policy [DFS], it is required to evaluate whether an XG device has an in-band leakage less than that of the in-band leakage defined in Section 3.3.2.2. This comparison would require a predicate as follows

```
const =< : Pred(Frequency -> Power, Frequency -> Power) ;
```

that can be used in the following way:

```
inband_leakage_ok if =<(inbandLeakage, maxInbandLeakage) ;
```

where `inbandLeakage` is the given device's in-band leakage as defined above and `maxInbandLeakage` as defined in Section 3.3.2.2. However, because comparisons on functions representing powermasks are very common in XG, we will support such a predicate as a built-in predicate.

3.4. Sharing Type and Constant Declarations Across Policies

It is important for policies to be able to share definitions of constants so that they can refer to the same concepts. Sharing provides a basic mechanism through which one can develop a shared ontology. The reserved keyword include is used for including declarations into a policy that are defined elsewhere. In CoRaL, we provide a set of shared, basic type definitions frequently required by policies, and a set of constant declarations that are required for a radio and the policy reasoner to function properly. These declarations are summarized in a policy named 'XGTypes.' At present, this collection contains only a minimal set of shared constant definitions that are used by the example policies given in Appendix B. 'XGTypes' is available in Appendix B.a. The XG Architecture RFC [XGArch] describes these definitions in detail. For a policy to make use of these predefined types and constants, it must include them using the statement:

```
include XGTypes;
```

In general, all types or constants used in a policy must be declared before use. This can be done in the policy itself prior to the first use, or in another policy that is included. Explicit declarations have the advantage of not having any hidden assumptions regarding basic type definitions.

When a policy designer might want to use different names for types that may already be defined in another policy, such as the XGTypes policy, one can create multiple aliases using the defconst construct. For example, let us assume that the DFS policy [DFS] has a declared device parameter named dfsBandwidth. Suppose a policy designer wants to define a parameter channelWidth, which is semantically the same as the dfsBandwidth, but the designer wants to use channelWidth to formulate the policy. This can be done as follows:

```
defconst channelWidth = dfsBandwidth ;
```

Creating such aliases for constants can be especially useful when constants (and variables) ranging over multiple policies might be required. In such cases, an alias allows each policy to be formulated with the wording desired by its author, yet overall semantic equivalence is established through defconst definitions. These definitions make sure that things with different names mean the same thing across policies. This will be of great use when multiple policies are merged and analyzed for determining inconsistencies or undesired consequences.

3.5. Terms

Before we can describe policy rules, we need to address how terms are formed.

3.5.1. Syntax

Prefix- and Infix Function Symbols	The unary – is a built-in prefix function symbol. The following symbols are built-in infix function symbols: +, -, *, /.
Term Declaration	Term ::= CONST | VAR | Value | Term.FIELD | Term InfixFuncSymb Term | PrefixFuncSymb Term | FUNC(Term1, Term2, …) | Term (Term1, Term2, …) where CONST is any previously defined constant; VAR is a previously defined variable (any user-defined identity/string); VALUE is an expression that appears on the right side of a defconst expression, i.e., a list, record, type, tuple, set, or function expression; FIELD is defined in a record type definition; and FUNC is a previously defined constant of type function. The 'Term Term' form is for function application where the first term is itself a function application that returns a function (see example below).

3.5.1.1. Example

Term Declaration	For example: const add-n : int->(int->int);

	could define a function that, given an integer *i*, returns a function that takes an integer and adds *i* to it. add-n(1)(2) is then a term. Another example is the definition of a function to add two powermasks, i.e., const addpmask : (Frequency->Power,Frequency->Power) ->(Frequency->Power) that can be used in a term such as addpmask(IBL1, IBL2) where IBL1 and IBL2 are constants of type Frequency->Power.

3.6. Rules

The main component of a policy is the collection of rules that define the behavior enforced by the policy. Rules can be specified in CoRaL in many ways, but there are two main types: standard and equational. There are two main differences: (1) standard rules consist of first-order formulas while equational rules consist of propositional formulas, and (2) equational rules provide the ability to assign a value (constant) to a function (whereas a standard rule evaluates a predicate to true or false).

The semantics of rule declarations will be covered in a future version of this document, which will give a complete denotational semantics for CoRaL. Some subtle issues regarding the usage of quantifiers and types of formulas are not covered in this document.

Figure 2 provides an overview of all the syntactic components that are applicable to rule declarations.

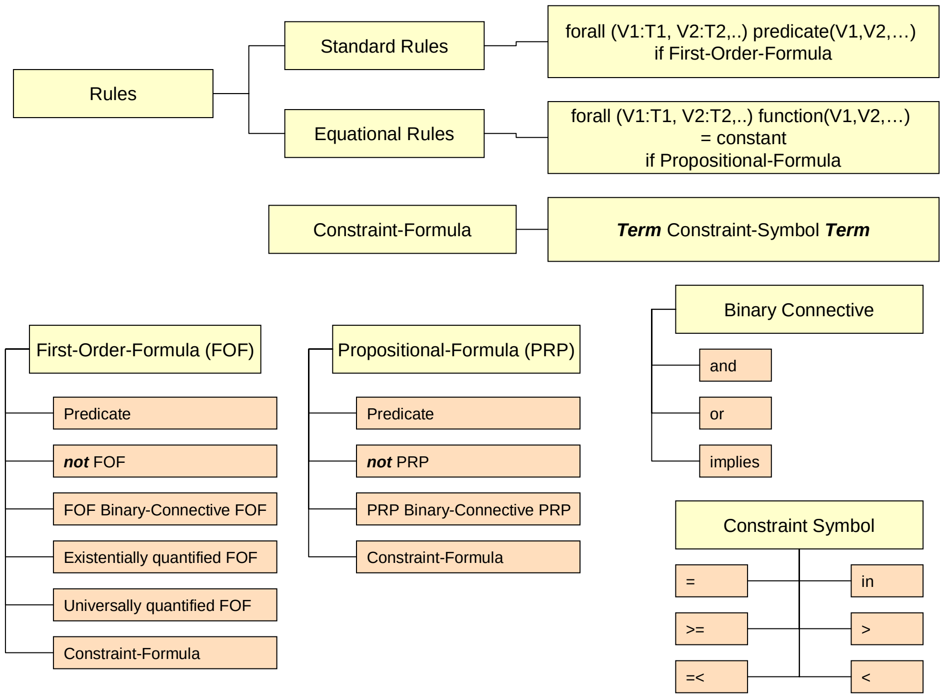

Rules
Standard Rules
forall (V1:T1, V2:T2,..) predicate(V1,V2,…) if First-Order-Formula
Equational Rules
forall (V1:T1, V2:T2,..) function(V1,V2,…) = constant if Propositional-Formula
Constraint-Formula
Term Constraint-Symbol Term
First-Order-Formula (FOF)
Predicate
not FOF
FOF Binary-Connective FOF
Existentially quantified FOF
Universally quantified FOF
Constraint-Formula
Propositional-Formula (PRP)
Predicate
not PRP
PRP Binary-Connective PRP
Constraint-Formula
Binary Connective
and
or
implies
Constraint Symbol
=
in
>=
>
=<
<

Figure 2 Overview of syntactic constructs applicable to rule declarations

Standard Rules: A standard rule consists of an (optional) universal quantification component followed by a predicate, the 'if' keyword, and a first-order formula. The intuitive meaning of this language construct is that the predicate (which might contain variables) evaluates to true for all variable assignments if the first-order formula in the condition is true.

3.6.1. Standard Rule Syntax

<u>Standard Rules</u>	*[forall* (V1:$TYPE_1$, V2:$TYPE_2$,…) *]* pred1*[*(V1,V2,…)*]* *if* | pred2*[*(TERM1, TERM2,..)*]* | *ConstraintFormula* | *FirstOrderFormula BinaryConnective FirstOrderFormula* | [*exists* | *forall* (U1:$TYPE_A$*[* in TERMm *]*, U2:$TYPE_B$ *[* in TERMm *]*,,…) *FirstOrderFormula* | *not FirstOrderFormula* Where: - V1, V2, etc. are previously declared variables - $TYPE_1$,$TYPE_2$, etc. are previously declared types - pred1, pred2 are previously declared predicates - declaration of predicate pred1 should match the arity and types of the variables in the universal quantification - TERM1, TERM2, … are terms with matching types to those of the definition of pred2. - a *ConstraintFormula* is two terms connected by one of the built-in constraint symbols (i.e., >, <, >=, =<, in). These are used for numeric constraint formulas. - a *BinaryConnective* is either and, or or implies - $TYPE_A$,$TYPE_B$, etc. are previously declared *simple* types. We note that our quantified variables are always typed, and their types can be *finite* or *infinite*. Finitely quantified formulas pose no problems. However, we allow only one very specific form of infinitely quantified formulas. Without giving details, we note that the restrictions on infinitely quantified formulas are such that we can ensure proper processing between the radio and the policy reasoner. Examples of infinitely quantified formulas are given below. More syntactical and semantic details on infinitely quantified formulas will be given in the next version of the document that will contain the semantics of CoRaL. - a *FirstOrderFormula* may take one of the following six different forms: 1. A simple predicate 2. Existentially quantified first-order formula 3. Universally quantified first-order formula 4. A first-order formula preceded by the 'not' logical unary operator 5. Two or more first-order formulas joined by binary connectives 6. A constraint formula

3.6.2. Standard Rule Examples

Rule w/ Predicate	`allow if device_ok; /* where device_ok is a predicate*/` [see DFS policy in Appendix B.b]

Rule w/ existentially quantified formula
(finite variable type)
	`channel_ok if(exists R:Role)`
	`     currentRole(R) and`
	`     (R=master and channel_ok_master) or`
	`     (R=slave and channel_ok_slave);`
	
	[see DFS policy in Appendix B.b]

Rule w/ universally quantified formula
(finite variable type)
	`emission_test_ok if`
	`     (forall (S:Signal,P:Power) in maxSignalPowers)`
	`       max_detected(S,P,`*`someAge`*`,`*`someSensingParams`*`);`
	
	[see LowerUHF policy in Appendix B.d]

Rule w/ negated formula	`(forall M:int)database_test_ok(M) if`
	`     not in_restricted_area(M);`
	
	[see LowerUHF policy in Appendix B.d]

Rule w/ universally quantified formula (infinite variable type)	`allow if frequency_ok and tests_ok;`
	
	`(forall M:int)`
	`db_and_em_test_ok(M) if`
	`     dbtest_ok(M) and emission_test_ok(M);`
	
	[see LowerUHF policy in Appendix B.d]

Rule w/ constraint formula	`frequency_ok if carrierFrequency in {470..512};`
	
	[see LowerUHF policy in Appendix B.d]
	
	`precision_ok if`
	`     precision =< 20.0E-6;`
	
	[see DFS policy in Appendix B.b]

Equational Rules: Equational Rules consist of an optional universal quantification component followed by a function and a value assignment, and then the 'if' keyword followed by a propositional formula.

3.6.3. Equational Rule Syntax

Equational Rules	*[forall* (V1:TYPE$_1$, V2:TYPE$_2$,…) *]* func*[*(V1,V2,…)*]* = TERM *[if* | pred*[*(TERM1,TERM2,..)*]* | *ConstraintFormula* | *PropFormula BinaryConnective PropFormula* | *not PropFormula]* Where: - V1, V2, etc. are variables - TYPE$_1$,TYPE$_2$, … are previously declared types - func is a previously declared function with matching arity and types of the variables in the universal quantification - TERM is a term - pred is a previously declared predicate - TERM1, TERM2, etc. are terms that match the types and arity of the previously declared predicate pred. - a *ConstraintFormula* is two terms connected by one of the built-in constraint symbols (i.e., >, <, >=, =<, in). These are used for numeric constraint formulas. - *BinaryConnective* is either and, or or implies - *PropFormula* is a propositional formula that may take one of the four following forms: 1. A simple predicate 2. A constraint formula 3. A propositional formula preceded by the 'not' logical unary operator 4. Two or more propositional formulas joined by binary connectives

3.6.4. Equational Rule Examples

Unconditional Equational Rule	`(forall S:Speed,A:TimePoint)`
	`modificationFactor(S,A) = S*A+10;`

Equational Rule w/ predicate	`(forall T:TVStation)`
	`gradeB(T) = digitalGradeB(T) if digital(T);`

Equational Rule w/ constraint formula	`(forall S:Speed,A:TimePoint)`
	`modificationFactor(S,A) = 0 if A<`*`smallAge`*`;`
	
	where *`smallAge`* is a previously declared constant of type `TimePoint`.
	
	[see Lower UHF policy in Appendix B.d]

3.7. Expressing Policies

An XG policy is a collection of valid statements enclosed between the policy start and end declarations. The start of a policy is defined by the statement `"policy POLICYNAME is".` POLICYNAME is the name of the policy, and XG policy files are expected to have a filename consisting of the policy name followed by the '.xg' extension. The end of a policy is declared by writing the `'end'` keyword as a single statement.

A CoRaL policy may (optionally) contain any of four types of declarations: type, constant, predicate, and rule (as well as the declaration of the start/end of a policy and the include statements). None of the four types is mandatory. Therefore, a policy may consist of any combination of these declarations. The generic structure of a policy is given in **Figure 3**.

Policy Start	`policy` POLICYNAME `is`
Policy Imports	`include` …
Type Declarations	`type` … `deftype` …
Constant Declarations	`const` … `defconst` …

Predicate Declarations	`const` …: `pred;`
	`defconst` …: `pred;`

Rule Declarations	`allow if` … `disallow if` …
Policy End	`end`

Figure 3 Generic structure of an XG policy

The term *XG statement* (or expression) refers to a valid type, constant, predicate, or rule declaration, and not the start or end declaration of a policy (which are special expressions).

Every valid XG statement must end in a semicolon (';'), which is the statement delimiter for XG policies. White space may be present on either side of the statement delimiter.

CoRaL allows multiline comments by enclosing text in the tags '/*' and '*/' as in a number of other languages. The following is an example of a single comment, which will be ignored by the policy reasoner:

```
/* This is an example of a
     multi-line comment in
               CoRaL          */
```

4. Discussion

4.1. Naming Convention

It is recommended that policy authors use the Lower CamelCase notation as the naming convention for non-predicate constants. That is, constants should start with a lower-case letter and should not contain any white space or special characters such as '-, % ^, @, $.' The constant name could be a collection of related words joined together as a single name where each word's first letter is uppercase, except for the first word's first letter. For example, a name of a constant for an analog, low-powered TV signal could be 'analogLowPoweredTV'.

It is recommended that policy authors use the Upper CamelCase notation as the naming convention for types. The only difference between Upper CamelCase and Lower CamelCase is that in Upper CamelCase the first letter of the name must be uppercase. For example, a name of a function type representing power masks could be 'PowerMask'.

The recommended naming convention for predicates is for the predicate name to consist of a collection of lowercase, related words separated by the underscore ('_') character. For example, a predicate indicating that a given device's sensing threshold is acceptable may be named 'sensing_threshold_ok'.

4.2. Guidelines for type declarations

CoRaL provides a rich, statically typed system that can be used to create numerous user-defined types to suit a multitude of uses. Most of these types were developed with certain usage scenarios in mind and some of them may be more suited to a given task than to others. Here, we present some broad guidelines on effective use of the type system.

CoRaL allows users to define abstract types using the `'type'` keyword. For example, a type that represents the different kinds of detection parameters of a device can be declared using the statement:

```
type DetectionParameters;
```

Such a type declaration makes no other claims regarding the new user-defined type 'DetectionParameter' other than the fact that it is an abstract type. Such an abstract type allows the user to extend this type later by defining functions on this type. For example, the user can define constants of function types in the following way:

```
const minDuration : DetectionParameters -> TimeDuration ;
const dwellTime : DetectionParameters -> TimeDuration ;
const minRate :DetectionParameters -> SensingRate ;
const allFrequencies:DetectionParameters -> [Frequency] ;
     /* list of frequencies */
```

A closer analysis of the above reveals that these declarations are done in a functional way and provide an extensible solution that can be extended by further function definitions. In contrast, defining `DetectionParameters` as a record construct, as given below, is not extensible.

```
deftype DetectionParameters = {minDuration: TimeDuration,
                               dwellTime: TimeDuration,
                               minRate: SensingRate,
                               allFrequencies: [Frequency]};
```

4.3. Guidelines for rule declarations

The two built-in predicates 'allow' and 'disallow' play a major role in policy specification. A policy can be either permissive or restrictive, based on how it defines these predicates. An instantiated transmission request fully characterizes the given XG devices capabilities, its environment, and the transmission parameters for which it is seeking permission to transmit. Whether or not a policy allows this transmission to take place depends on whether the 'allow' predicate or the 'disallow' predicate evaluates to true after applying the policy to the given request. For example, the following rule is permissive for the DFS policy [DFS] given in Appendix B.b:

```
allow if transmission_ok and radio_ok;
```

In the case of the Radar-band: Sensing-based access policy (see Appendix B.c), one of the requirements is "If the peak received power is greater than -80dBm, the device must not transmit". Such a requirement can be encoded in CoRaL a:

```
disallow if peak_received_power >= -80;
```

4.4. Future Work

Policies can originate from different sources and can address different aspects of policy. For example, a policy could be a regulatory policy or a policy governing the characteristics of a device or a network. Therefore, it is necessary to be able to inherit, extend, delegate, or combine different types of policies or those originating from different sources. Such an ability is not provided in the current version of CoRaL, but will be one of the major future extensions.

If multiple policies are to be combined for reasoning, then the language must support variables and constants ranging over policies. Such support is not in this version of the language, but it may be added in the future. Combining multiple policies into one policy may prove to be a useful feature in the case of testing for inconsistencies within policy bases. We plan to support this in future language versions.

Many units are used for measuring various properties in radio communication, such as Hz, MHz, mW, and dBm. CoRaL at present does not support units within its type system or conversion between units. Introduction of such support is one of the more important tasks planned for future versions.

A stateless policy reasoner will be much easier to verify, and will also be simpler to design and implement. A stateful reasoner might be nearly impossible for governing authorities to verify because of the potentially unbounded number of states and the unpredictability of the quality and timing of state updates. Thus, the current version of CoRaL does not support states.

Policy management requires support for version control and author identification. We plan to extend CoRaL by introducing language constructs specifically designed for these purposes. This would reduce the complexities associated with policy management. We will discuss this topic in the forthcoming XG Policy Management RFC.

Appendix A. CoRaL Syntax

We present the syntax of XGPL-Full in a modified version of EBNF.

| means or, ‘x’ means add literal ‘x’ to output, [**x**] means **x** is optional, {**x**} means one or more **x**, <**x**> means a comma-separated list of one or more **x**’s.

Note that there are additional syntax rules that cannot be captured in a context-free grammar such as this one. For example, a **TypeID** must have been declared before it can be used in the second part of a **deftype** statement.

For the purpose of this document, we define two top-level units: **Policy** and ***Comment*.**

Types

TypeID ::== *e.g. FreqList, FreqTable*
FeatureID ::== *e.g. location, gradeBContour*
ScalarType ::== **TypeID |** ‘Float’ **|** ‘Int’
AtomicType ::== **TypeID | ScalarType |** ‘Bool’
SimpleType ::== **TypeID | AtomicType |**
‘[‘ **SimpleType** ‘]’ **|**
‘{‘ <**FeatureID** ‘:’ **SimpleType**> ‘}’ **|**
‘(‘ <**SimpleType**> ‘)’ **|**
‘{‘ **ScalarType** ‘}’
FunctionType ::== **TypeID | SimpleType |**
SimpleType ‘->’ **SimpleType |**
‘(‘ <**SimpleType**> ‘)’ ‘->’ **SimpleType**
Type ::== **FunctionType |** ‘Pred’ [‘(‘ **<FunctionType>** ‘)’]
TypeStatement ::== ‘type’ <**TypeID**>
DefTypeStatement ::== ‘deftype’ **TypeID** ‘=’ **Type**

Constants

ConstID ::== ‘inf’ **|** ‘-inf’ **|** ‘true’ **|** ‘false’ **|** ‘True’ **|** ‘False’**|** *e.g. frequency_ok, MyFrequency, MyTable*
Value ::== **ConstID |**
‘{‘ <**FeatureID** ‘=’ **Term**> ‘}’ **|**
‘[‘ <**Term**> ‘]’ **|**
‘(‘ <**Term**> ‘)’ **|**
‘{‘ **Term** ‘..’ **Term** ‘}’ **|**
[‘symmetric’] ‘step’ **|** ‘linear’ ‘[‘ < ‘(‘ **Term** ‘,’ **Term** ‘)’ > ‘]’
ConstStatement ::== ‘const’ <**ConstID**> ‘:’ **Type**
DefConstStatement ::== ‘defconst’ **ConstID** ‘:’ **Type** ‘=’ **Value**

Formulas

VariableID ::== *x,y,…*
FunctionID ::== *modificationFactor,…*
BuiltinPred ::== ‘allow’ **|** ‘disallow’
PredSymbol ::== **BuiltinPred | ConstID**
PrefixFuncSymbol ::== ‘-‘ **|** …
InfixFuncSymbol ::== ‘+’ **|** ‘-’ **|** ‘*’ **|** ‘/’ **|** …
ConstraintSymbol ::== ‘=’ **|** ‘<’ **|** ‘>’ **|** ‘=<’ **|** ‘>=’ **|** ‘in’ **|** …
Term ::== **ConstID | VariableID | Value | Term** ‘.‘ **FeatureID |**

Term InfixFuncSymbol Term | PrefixFuncSymbol Term | Term Term
Term ’(‘ **Term** ’)’
Atom ::== **PredSymbol** [’(‘ <**Term**> ’)’]
ConstraintFormula ::== **Term ConstraintSymbol Term**
AtomicFormula ::== **Atom | ConstraintFormula**
BinaryConnective ::== ‘and’ **|** ‘or’ **|** ‘implies’
PropFormula ::== **AtomicFormula |**
PropFormula BinaryConnective PropFormula |
‘not’ **PropFormula**
EqHead ::== **FunctionID** [‘(‘ **<VariableID>** ‘)’] ‘=’ **Term**
EqRule ::== [‘(‘ ‘forall’ <<**VariableID**> ‘:’ **Type**> ‘)’] **EqHead** [‘if’ **PropFormula**]
FOFormula ::== **AtomicFormula |**
‘(‘ ‘exists’ <<**VariableID**> ‘:’ **SimpleType** [‘in’ **Term**]> ‘)’ **FOFormula |**
‘(‘ ‘forall’ <<**VariableID**> ‘:’ **SimpleType** [‘in’ **Term**]> ‘)’ **FOFormula |**
FOFormula BinaryConnective FOFormula |
‘not’ **FOFormula**
Head ::== **PredSymbol** [‘(‘ **<VariableID>** ‘)’]
Rule ::== [‘(‘ ‘forall’ <<**VariableID**> ‘:’ **Type**> ‘)’] **Head** [‘if’ **FOFormula**]

Policies

PolicyID ::== *e.g. DFSPolicy*
ImportStatement ::== ‘include’ **PolicyID**
Policy ::== ‘policy’ **PolicyID** ‘is’ {**PStatement** ‘;’} ‘end’
PStatement ::== **ImportStatement | TypeStatement | DefTypeStatement | ConstStatement |**
DefConstStatement | Rule | EqRule

Comments

Text ::== *any text*
Comment ::== ‘/*’ **Text** ‘*/’

Appendix B.   CoRaL Typing Rules

A *typing context* Γ is a sequence of *statements*. There are six kinds of statements:

- Formulas (we use F, G, etc. to denote these)
- *type* statements
- *deftype* statements
- *const* statements
- *defconst* statements
- Type statements of the form x:T, meaning x is of type T

We use a special type *Type* as the type of types. So, x:Type means that x is a type. Γ |- means that Γ is well-typed (i.e., all the statements in Γ are well-typed). Γ |- S means that statement S is well-typed if Γ is well-typed. ∅ is the empty context. S, S', and so on will be used for statements. Γ, Γ', and so on will be used for contexts. Γ,S means the context Γ with S added to it. [A:=B]S means substitute B for all occurrences of A in statement S (using standard capture-free substitution). A rule

A_{1}, … A_{n}

B

Means that the conclusion B can be derived from the premises A_{1},…A_{n}. If there is nothing above the line, then B can always be derived.

Top-level/structural rules
The empty context is well-typed.

∅ |-

A context is well-typed if its last statement is well-typed given that all but the last statements are well-typed, and the latter is true.

Γ |-   Γ |-S

Γ,S |-

The following five rules allow us to strip off statements from the right side of the context. This is used to "look up" the type of a variable in a context.

Γ |- x:T

Γ,F |- x:T

```
Γ |- x:T          x ≠ c
------------------------
Γ, const c:T’ |- x:T

Γ |- x:T          x ≠ c
--------------------------------
Γ, defconst c:T’=M |- x:T

Γ |- x:T          x ≠ t
-------------------------
Γ, type t |- x:T


Γ |- x:T          x ≠ t
-------------------------
Γ, deftype t=M |- x:T
```

Types and constants

```
Γ |-              t is not declared/defined in Γ
---------------
Γ |- type t


Γ |- T:Type       t is not declared/defined in Γ
-------------------
Γ |- deftype t=T

Γ |- T:Type       c is not declared/defined in Γ
------------------
Γ |- const c:T

Γ |- T:Type       Γ |- M:T          c is not declared/defined in Γ
--------------------------------
Γ |- defconst v:T=M

Γ |- [X:=T]S             Γ |- T:Type             t is not declared/defined in Γ
-----------------------------------------
Γ, deftype t=T |- S

Γ, const c:T |- S                  c is not declared/defined in Γ
----------------------------
Γ, defconst c:T=M |- S
```

```
----------------------
Γ, type t |- t:Type

----------------------------
Γ, deftype t=T |- t:Type

---------------------
Γ, const c:T |- c:T
```

Basic types and type operators

```
--------------------
Γ |- Integer:Type

------------------
Γ |- Float:Type

--------------------
Γ |- Boolean:Type

Γ |- T:Type
--------------
Γ |- [T]:Type

----------------
Γ |- {Int}:Type

---------------------
Γ |- {Float}:Type

Γ |- T1:Type,…, Γ |- Tn:Type
------------------------------------
Γ |- (T1,T2,…,Tn):Type

Γ |- T1:Type, T2:Type
---------------------------
Γ |- T1->T2:Type

Γ |- T1:Type,…, Γ |- Tn:Type
-----------------------------------
Γ |- Pred(T1,…,Tn):Type

---------------
Γ |- Pred:Type
```

Terms

```
----------------------------------
Γ |- integer numeral : Integer

------------------------------
Γ |- float numeral : Float

----------------------
Γ |- true : Boolean

-----------------------
Γ |- false : Boolean

Γ |- t1:T,…, Γ |- tn:T              n>0
------------------------------
Γ |- [t1, t2, …, tn] : [T]

Γ |- T:Type
---------------
Γ |- [] : [T]

Γ |- t1:Int        Γ |- t2:Int
------------------------------
Γ |- {t1..t2} : {Int}


Γ |- t1:Float     Γ |- t2:Float
----------------------------------
Γ |- {t1..t2} : {Float}


Γ |- t1:T1,…, Γ |- tn:Tn
-----------------------------------------
Γ |- (t1, t2, …, tn) : (T1,T2,…Tn)

Γ |- t:T            Γ |- f:T->T'     (f is a prefix function symbol)
-----------------------------------
Γ |- f(t) : T'
```

Note that we use tuples to represent multiple arguments. This leaves us with two sets of parentheses in the multiple argument case, and we need the following rule to remove one of the sets of parentheses.

```
Γ |- F(t):T                (f is a prefix function symbol)
------------------
Γ |- f((t)) : T


Γ |- t1:T1       Γ |- t2:T2       Γ |- f:(T1,T2)->T3     (f is an infix function symbol)
------------------------------------------------------------
Γ |- t1 f t2 : T3


Γ |- xi:T1  (1≤i≤n)      Γ |- yi:T2  (1≤i≤n)
Γ |- T1:Int or Γ |- T1:Float
Γ |- T2:Int or Γ |- T2:Float
--------------------------------------------------
Γ |- linear[(x1,y1),…,(xn,yn)] : T1->T2


Γ |- xi:T1  (1≤i≤n)      Γ |- yi:T2  (1≤i≤n)
Γ |- T1:Int or Γ |- T1:Float
Γ |- T2:Int or Γ |- T2:Float
--------------------------------------------------
Γ |- step[(x1,y1),…,(xn,yn)] : T1->T2
```

Formulas

```
Γ |- p:pred(T1,…,Tn)          Γ |- t1:T1, …, Γ |- tn:Tn     (p is a prefix predicate symbol)
-------------------------------------------------------------------
Γ |- p(t1,…tn)


Γ |- p : pred(T1,T2)           Γ |- t1:T1       Γ |- t2:T2     (p is an infix predicate symbol)
-------------------------------------------------------------------
Γ |- t1 p t2

Γ |- F            Γ |- G
---------------------------
Γ |- F and G

Γ |- F            Γ |- G
--------------------------
Γ |- F or G
```

```
Γ |- F              Γ |- G
--------------------------
Γ |- F implies G

Γ |- F              Γ |- G
--------------------------
Γ |- F iff G

Γ |- F              Γ |- G
--------------------------
Γ |- F if G

Γ |- F
-----------
Γ |- not F

Γ, x1:T1,…, xn:Tn |- F
------------------------------------
Γ |- (forall x1:T1,…, xn:Tn) F

Γ, x1:T1,…, xn:Tn |- F
-------------------------------------
Γ |- (exists x1:T1,…, xn:Tn) F

Γ, x:T |- F       Γ |- t:[T]
--------------------------------
Γ |- (forall x:T in t) F

Γ, x:T |- F       Γ |- t:{T}
------------------------------
Γ |- (forall x:T in t) F

Γ, x:T |- F       Γ |- t:[T]
------------------------------
Γ |- (exists x:T in t) F

Γ, x:T |- F       Γ |- t:{T}
-----------------------------
Γ |- (exists x:T in t) F
```

CoRaL Example Policies

a. CoRaL Built-In Types and Constants

```
policy XGTypes is

/* Types */
deftype Power = Float;
deftype Frequency = Float;
...
type Signal;
const GPSSignal:Signal;

type DetectionParameters;

const minDuration:DetectionParameters->TimeDuration;
const dwellTime:DetectionParameters->TimeDuration;
const minRate:DetectionParameters->SensingRate;
const allFrequencies:DetectionParameters->{Frequency};

/* alternatively:
deftype DetectionParameters =
      {minDuration:TimeDuration,
       dwellTime:TimeDuration,
       minRate:SensingRate,
       allFrequencies:{Frequency}};
*/

deftype Powermask:Frequency->Power;
const < : Pred(Powermask,Powermask);


/*
(forall P1:Powermask, P2:Powermask)
< if
      (forall F:Frequency)
            P1(F) < P2(F);
*/

/* Constants (request parameters) */

const currentLocation : Position;
const currentTime : TimePoint;

const currentAuthority : Pred(Authority);
const currentRole : Pred(Role);
const currentMode : Pred(Mode);

const max_detected : Pred(Signal, Power,
                    TimeDuration, DetectionParameters);

const received : Pred(Message,Age);

const inBandLeakage : Frequency->Power;
const outsideBandLeakage : Frequency->Power;
const spuriousEmissions : Frequency->Power;

const precision : ?

const waveformDetection : Pred(Waveform);
const signalDetection : Pred(Signal);
const minSensingThreshold : Power;

const carrierFrequency : Frequency;
const waveformUsed : Waveform;
const maxPower : Power;

const timeInt : {TimePoint};
const maxContinuousOnTime : TimeDuration;
const minOffTime : TimeDuration;

const spreadingError : ?

const maxSpeed : Speed;

end
```

b. DFS Policy in CoRaL

The formalization below is based on the description of DFS given in [DFS].

```
policy DFS is

include XGTypes.xg;

/* Type declarations */
deftype Powermask = Frequency->Power;

/* Constant declarations */
defconst dfsFreqList : [Frequency] = [5180,5200,..];

defconst maxInBandLeakage : Powermask =
      linear [(0,0),(9,0),..];

defconst maxOutsideBandLeakage : Powermask =
      step [(30,-36),(47,-54),..];

defconst maxSpuriousEmissions : Powermask =
      step [(30,0),(30,-57),..];

/* Predicate declarations */
const transmission_ok : Pred;
const radio_ok : Pred;
const frequency_ok : Pred;
const power_ok : Pred;
const channel_ok : Pred;
const channel_ok_master : Pred;
const channel_ok_slave : Pred;
const channel_ok_slave_rd : Pred;
const channel_ok_slave_no_rd : Pred;
const in_band_leakage_ok : Pred;
const outside_band_leakage_ok : Pred;
const spurious_emissions_ok : Pred;

/* Policy Rules */
allow if transmission_ok and radio_ok;

transmission_ok if
      frequency_ok and power_ok and channel_ok;

radio_ok if
      in_band_leakage_ok and
      precision_ok;

frequency_ok if
      carrierFrequency in dfsFreqList;

power_ok if
      (carrierFrequency in {5150..5350} implies maxPower=<23) and
      (carrierFrequency in {5470..5725} implies maxPower=<30);

channel_ok if
      (exists R:Role)
            currentRole(R) and
            (R=master and channel_ok_master) or
            (R=slave and channel_ok_slave);

channel_ok_master if
      spreadingError =< 0.10 and
      signalDetection(radarSignal) and

      ((exists MaxRadarPower:Power, MaxThreshold:Power)
            max_detected(radarSignal, MaxRadarPower, age
                     {minDuration=60, minRate=5,
                     allFrequencies={5250..5350,5470..5725}}) and
            (MaxRadarPower >= 200 and MaxThreshold =< -64)) or
```
/*
The way “allFrequencies” is defined seems very useful. Perhaps it should be done with a union operator. Consider allowing user-defined constraint formulas as characteristic functions of sets.
*/

```
      ((exists MaxRadarPower:Power,
               MaxThreshold:Power)
            max_detected(radarSignal, MaxRadarPower, age
                     {minDuration=60, minRate=5,
                     allFrequencies={5250..5350,5470..5725}}) and
            (MaxRadarPower < 200 and MaxThreshold =< -62));

channel_ok_slave if
      (signalDetection(radarSignal) and channel_ok_slave_rd) or (not
      signalDetection(radarSignal) and
            channel_ok_slave_no_rd)

channel_ok_slave_rd if
      (exists MaxRadarPower:Power, MaxThreshold:Power)
            max_detected(radarSignal, MaxRadarPower, some-value,
                    {minDuration=60, minRate=5,
                  allFrequencies={5250..5350,5470..5725}}) and
         (MaxRadarPower >= 200 implies MaxThreshold =< -64)

channel_ok_slave_no_rd if
      received(okToTransmit,10);

precision_ok if
      precision =< 20.0E-6;

in_band_leakage_ok if
      inBandLeakage =< maxInBandLeakage;

outside_band_leakage_ok if
      outsideBandLeakage =< maxOutsideLeakage;

spurious_emissions_ok if
      spuriousEmissions =< maxSpuriousEmissions;

end
```

c. Radar Band Sensing-Based Policy in CoRaL

The following formalization is based on the description of sensing-based access in the Radar band proposed by the XG Working Group and published in [XGPolicyExamples]. Because this document is not publicly available, we reproduce the relevant text below:

i. Natural Language Description of Radar Band Sensing-Based Access

Synopsis: This example applies to XG devices that are capable of operating in the radar "S" band, have a good RF sensing ability, but do not use other sources of information such as geo-location systems or Internet access to avoid interference to primary band users. An XG device in the radar "S" band must sense within the intended frequency of operation and is authorized to transmit if the received signal meets some conditions. Transmit power is regulated as a function of the received power. From a regulatory perspective, XG is considered a secondary user operating on a not-to-interfere basis with respect to primary licensed radar operations.

Specific requirements for cognitive (XG) radios (hereinafter referred to as the "device") operating in "S" RADAR bands are

(a) Operation shall be contained within 3100 to 3300 MHz.

(b) The device must have a sensing threshold of –110 dBm or less in a 100-kHz channel.

(c) Devices must incorporate a mechanism for opportunity identification within the spectrum in question. The following criteria must be met:

   1) The device must sense received RF power within the intended frequency channel. The frequency channel must be no more than 100 kHz in width. There

must be a look-through interval of minimum 3 seconds and a dwell time[6] of 500 µs.

2) The maximum authorized transmit power depends upon the sensed received power from licensed radar transmitters, as follows:
   - I. If the peak received power in sensing is less than -100 dBm, the maximum transmit power is 50 mW.
   - II. If the peak received power is greater than or equal to -100 dBm, but less than –80 dBm, the maximum transmit power is 10 mW.
   - III. If the peak received power is greater than –80 dBm, the device must not transmit.

3) Before XG transmissions may be authorized or an increase in power level may be authorized, the sensed condition must continuously be within the prescribed limit for at least 5 min.

(d) Devices must incorporate a mechanism for interference-limited use of the spectrum opportunity identified in c:

1) No more than 1% of the power must be outside of the bandwidth of the intended frequency channel.

2) The maximum continuous on time must be 1 second and the minimum off time must be 100 ms.

3) When the results of sensing indicate that XG must lower its transmitter power level or completely stop transmitting, the change must be made within 1 s.

[6] *Look-through* is the period between successive sensing operations, and the *dwell time* is the time over which energy is accumulated and integrated by the sensor.

ii. Formalization of Radar Band Sensing-Based Access in CoRaL

```
policy RadarBandSensingBased is

include XGTypes.xg;

/* Predicate declarations */

const device_ok : Pred;
const sensing_ok : Pred;
const usage_ok : Pred;
const frequency_ok : Pred;
const sensing_threshold_ok : Pred;
const channel_width_ok : Pred;
const look_through_interval_ok : Pred;
const dwell_time_ok : Pred;
const peak_received_power_ok : Pred;
const signal_age_ok : Pred;
const outside_band_leakage_ok : Pred;
const max_ontime_ok : Pred;
const min_offtime_ok : Pred;
const channel_closing_time : Pred;
const tx_ok : Pred;


/* Policy Rules */
allow if device_ok and sensing_ok and usage_ok;

device_ok if
      frequency_ok and sensing_threshold_ok and channel_width_ok;

sesing_ok if
      look_through_interval_ok and dwell_time_ok and
      peak_receieved_power_ok and signal_age ok;

usage_ok if
      outside_band_leakage_ok and max_ontime_ok and min_offtime_ok and
      channel_closing_time ok;


frequency_ok if carrier_frequency in {3100..3300};

sensing_threshold_ok if sensing_threshold =< -110;

channel_width_ok if channel_width =< 100;

look_through_interval_ok if look_through_interval =< 3;

dwell_time_ok if dwell_time =< 500;

peak_received_power_ok if
      (peak_received_power < -100 implies max_tx_power=50) and
      (peak_received_power >= -100 and peak_recieved < -80 implies
      max_tx_power=10)

disallow if peak_received_power >= -80;



outside_band_leakage_ok if outside_band_leakage < 0.1;

max_ontime_ok if max_ontime =< 1;

min_offtime_ok if min_offtime >= 100;

channel_closing_time_ok if channel_closing_time =< 1;

end
```

d. Lower UHF Policy in CoRaL

This policy is based on a draft description of an illustrative regulatory policy for the lower UHF television broadcast band 470 to 512 MHz. This draft description was provided by Shared Spectrum Company. Because it is not readily available, we include the text below.

i. Natural Language Description of Lower UHF

This is a draft set of policies for the lower UHF television broadcast band 470 to 512 MHz (i.e., Channels 14 to 20).

XG will have determined at the outset of operations the availability of position-determination capability within defined accuracy from a global positioning satellite.

Before testing a frequency for use, the system will

(1) Check whether a positive GPS signal has been received within an interval of a few seconds.
(2) If a signal has not been received within that interval but within a larger pre-set interval, it may proceed further with the algorithm but a modification of the algorithm to make it proportionately more demanding. The modification factor will be selected from a GPS modification table reflecting the time since the last observation and assumptions about the maximum speed of the XG unit.
(3) If no adequate GPS information data is available, the XG unit will check for an adequate beacon signal that can be analyzed with an alternative algorithm.
(4) If neither an adequate GPS nor beacon signal is received, the radio will be rejected transmission on the frequency.

When XG concludes that it has sufficiently reliable position determination, it must perform successively two basic types of evaluation. The first is purely a function of the position determination and a database of FCC licensed TV stations.

The database will be derived from two sources. The first consists of relevant data from the Federal Communication Commission's Media Bureau's Consolidated DataBase System ("CDBS"). It can be found at http://www.fcc.gov/mb/databases/. The second will be a database created by Shared Spectrum to capture data with respect to the public safety operations licensed to the 13 cities specified in 47 CFR 90.303. The mobile data will be configured so as to use the 13 geographic centers specified in Section 90.303 and consider circles 80 miles in radius around each mobile center as the equivalent of a Grade B television broadcast station contour.

1. **Database Test**:

The first step with the database is to cull out all broadcast stations within 200 miles of the determined XG position.

The next step, for each TV station selected, is to apply the Longley-Rice program to compute its Grade B contours. XG must first distinguish between analog and digital stations from the database. For analog stations, the calculation is governed by 47.CFR 73.683(a) and for digital stations it is governed by 47 CFR 73.622(e). Shared Spectrum has a copy of the Longley-Rice program that can be incorporated into XG.

If any computed Grade B contour includes the determined XG position, the frequency is rejected. If the frequency is not rejected, we go to the second part of the Emission test, under which frequencies will be rejected if the locational data shows them to be within 80 miles of any geographical center specified in Section 90.303.

2. **Emission Test**

Before applying an emission test, XG must first determine what kind of incoming signal is being detected. It could be any of the following:

a) full power analog TV broadcast (as high as 5,000 kW) [See 47 CFR 73.614]
b) full-power digital TV broadcast
c) analog LPTV stations (as high as 150 kW) [See 47 CFR 74.635]
d) digital LPTV stations (as high as 15 kW) [See 47 CFR 74.635]
e) television STL and TV relay stations licensed under 74.602(h)
f) public trunked mobile stations licensed in certain cities under Section 22.657 et seq.
g) offshore radiotelephone stations licensed in the Gulf of Mexico with paired assignments set forth in 47 CFR 22.1007 with erp as high as 1000 W for base stations and 100 W for mobile units
h) low-power auxiliary stations under 47 CFR 47 801 et seq., such as wireless microphones, cue and control communications, and synchronization of TV signals operating over about 100 meters
i) legacy biomedical telemetry equipment under 47 CFR 15.242military tactical or training uses permitted after coordination with FCC field personnel under the NTIA Manual Section 7.15.3.4a

Feature detection (such as SYNC signals) will be used to distinguish among the signals, and appropriate acceptable threshold levels will be applied accordingly. It may be possible to persuade the FCC to permit XG to disregard the last three categories, but thos categories should be handled if possible. Each type of use will be associated by a programmed table with an acceptable threshold level. When that level is exceeded, use of the frequency will be disallowed.

Stations in the Gulf of Mexico may need to be handled by a wide-area blackout near the Gulf. Handling of scattered public-trunked mobile systems has not yet been addressed.

ii. Formalization of Lower UHF in CoRaL

```
policy LowerUHF is

include XGTypes.xg;
```

```
/* Type declarations */

type TVType;

deftype TVStation = {location:Location, gradeB:Polygon,
                        type:TVType..};
deftype City = {location:Location,..};

/* Constant declarations */
const analog:TVType;
const digital:TVType;

const positiveGps, beacon, fullPowerAnalogTV, fullPowerDigitalTV,
       analogLPTV, digitalLPTV, tvSTLandTVRelay,
       publicTrunkedMobile, offshoreGSM, legacyBiomedTelemetry,
       militaryTacticalOrTraining of Signal;

defconst station1:TVStation = {location=.., gradeB=.., type=..};
defconst station2:TVStation = {location=.., gradeB=.., type=..};
. . .

defconst city1:City = {location=..};
defconst city2:City = {location=..};

defconst maxSignalPowers : [(Signal,Power)] =
      [(fullPowerAnalogTV,10),
       (fullPowerDigitalTV,15),
        ..];

/* Predicate declarations */
const frequency_ok : Pred;
const tests_ok : Pred;
const pos_determination_ok : Pred(Int);
const strict_pos_determination : Pred(Int);
const loose_pos_determination : Pred(Int);
const modification_factor : Pred(Speed,TimeDuration,Int);
const beacon_pos_determination : Pred;
const database_test_ok : Pred(Int);
const in_restricted_area : Pred(Int);
const in_gradeB : Pred(Location,TVStation,Int);
const in_analog_gradeB : Pred(Location,TVStation,Int);
const in_digital_gradeB : Pred(Location,TVStation,Int);
const emission_test_ok : Pred;

/* Function declarations */
const modificationFactor : (Speed,Age)->Int;

/* Policy Rules */
allow if frequency_ok and tests_ok;

frequency_ok if carrierFrequency in {470..512};

tests_ok if
      (exists Age:TimePoint)
            received(positiveGPS,Age) and
            db_and_em_test_ok(modificationFactor(maxSpeed,Age));
```

```
tests_ok if
      (exists Age:TimePoint)
            received(beaconSignal,Age) and
            db_and_em_test_ok(modificationFactor(maxSpeed,Age));

(forall S:Speed,A:TimePoint)
modificationFactor(S,A) = 0 if A<smallAge;

(forall S:Speed,A:TimePoint)
modificationFactor(S,A) = S*A+10 if A>=smallAge;

(forall M:int)
db_and_em_test_ok(M) if
      dbtest_ok(M) and emission_test_ok(M);

(forall M:int)
database_test_ok(M) if
      not in_restricted_area(M);

(forall M:int)
in_restricted_area(M) if
      (exists T:TVStation)
            distance(currentLocation,T.location,D) and
            D =< 200 and
            in_gradeB(currentLocation,T,M);


(forall L:Location,T:TVStation,M:Int)
in_gradeB(L,T,M) if
      (T.type=analog and
      in_analog_gradeB(L,T,M)) or
      (T.type=digital and
      in_digital_gradeB(L,T,M));

(forall L:Location,T:TVStation,M:Int)
in_analog_gradeB(L,T,M) if


(forall L:Location,T:TVStation,M:Int)
in_digital_gradeB(L,T,M) if
      ?

emission_test_ok if
      (forall (S:Signal,P:Power) in maxSignalPowers)
            max_detected(S,P,someAge,someSensingParams);

end
```

Appendix C. References

[BM03] “In the Matter of Establishment of an Interference Temperature Metric to Quatify and Manage Interference and to Expand Available Unlicensed Operation in Certain Fixed, Mobile and Satellite Frequency Bands. ” W. J. Byrnes and M. McHenry. FCC Comment, ET Docket No. 03-237, April, 2004.

[DFS] “ETSI Standard EN 301 893 V1.2.2 (2003-06). Broadband Radio Access Networks (BRAN); 5 GHz high performance RLAN; Harmonized EN covering essential requirements of article 3.2 of the R&TTE Directive”. Reference DEN/BRAN-002000-2, European Telecommunications Standards Institute (ETSI), 2003. http://www.etsi.org.

[DLHB] “Description Logic Handbook. Theory, Implementation and Applications.” Edited by Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, Peter Patel-Schneider. Cambridge University Press, January, 2003.

[Gruber] “What Is an Ontology?” Tom Gruber. http://www-ksl.stanford.edu/kst/what-is-an-ontology.html. See also T.R. Gruber “A translation approach to portable ontologies”. *Knowledge Acquisition,* 5(2): 199-200, 1993. http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html

[OWL] “OWL Web Ontology Language Reference.” Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein, Franklin W. Olin. Edited by Mike Dean, Guus Schreiber. W3C Recommendation, February, 2004. http://www.w3.org/TR/owl-ref/.

[SP04] “Spectrum 101. An Introduction to Spectrum Management.” J.A. Stine and D.L. Portigal. MITRE, Technical Report MTR 04W0000048, 2004.

[SWRL] “SWRL: A Semantic Web Rule Language Combining OWL and RuleML.” Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean. W3C Member Submission, May 2004. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

[XGArch] “XG Policy Architecture and APIs.”. Daniel Elenius, Grit Denker, and David Wilkins, SRI International, ICS-16763-TR-07-002, 2007.

[XGPLF] “XG Working Group. XG Policy Language Framework.” Request for Comments. Version 1.0, April 16, 2004. BBN Technologies, Cambridge, Massachusetts, U.S.A.

[XGPM] “XG Policy Management.” Request for Comments. SRI International. *To appear.*

[XGPolicyExamples] “XG Policy Examples (Draft).” Request for Comments, Version 0.1. XG Working Group. BBN Technologies, Cambridge, Massachusetts, U.S.A.

[XGV] “XG Working Group. The XG Vision. Version 2.0.” BBN Technologies, Cambridge, Massachusetts, U.S.A.

[XML] “Extensible Markup Language (XML) 1.0 (Third Edition).” Edited by Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau. W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/

[XMLNamespace] “Namespaces in XML 1.1.” Edited by Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin. W3C Recommendation, February 2004. http://www.w3.org/TR/xml-names11/.

[XMLSchema] “XML Schema Part 0: Primer Second Edition” Edited by David C. Fallside, Priscilla Walmsley. W3C Recommendation, October 2004. http://www.w3.org/TR/xmlschema-0/.