An Application of Formal Methods to Cognitive Radios[*]

Konstantine Arkoudas and Ritu Chadha and Jason Chiang
Telcordia Research
One Telcordia Drive
Piscataway, NJ 08854
{konstantine, chadha, chiang}@research.telcordia.com

ABSTRACT

We discuss the design and implementation of a formal policy system regulating dynamic spectrum access (DSA) for cognitive radios. DSA policies are represented and manipulated in a proof framework based on first-order logic with arithmetic and algebraic data types. Various algebraic operations combining such policies can be easily implemented in such a framework. Reasoning about transmission requests is formulated as a satisfiability-modulo-theories (SMT) problem. Efficient SMT solvers are used to answer the corresponding queries and also to analyze the policies themselves. Further, we show how to reason about transmission requests in an optimal way by modeling the problem as an SMT instance of weighted Max-SAT, and we demonstrate that this problem too can be efficiently solved by cutting-edge SMT solvers. We also show that additional optimal operations on transmission requests can be cast as classic optimization problems, and to that end we give an algorithm for minimizing integer-valued objective functions with a small number of calls to an oracle SMT solver. We present experimental results on the performance of our system, and compare it to previous work in the field.

1. INTRODUCTION

One of the world's most prized physical resources is the electromagnetic spectrum, and particularly its radio frequency (RF) portion, stretching roughly from 10 KHz to 300 GHz. The RF spectrum is so valuable that its allocation is strictly regulated by world governments, and these days even small parts of it can be sold for billions of dollars in spectrum auctions.

As wireless devices continue to proliferate, demand for access to RF spectrum is becoming increasingly pressing, and increasingly difficult to achieve. After all, the (usable) spectrum is finite, while the demand for it continues to increase without bounds. However, it has been recognized [13, 14] that problems of spectrum scarcity and overuse arise not so much from physical shortage of frequencies, but mostly as a result of the centralized and rigid way in which spectrum allocation has been managed. Spectrum has been allocated in a *static* way: bands of the RF range are statically assigned to licenced users on a long-term basis, who then have exclusive rights to the corresponding region. Examples here include the 824–849 MHz, 1.85–1.91 GHz, and 1.930–1.99 GHz frequency bands, which are reserved for FCC-licenced cellular and personal communications services (PCS). Other parts of the RF spectrum, by contrast, are unlicensed, and anyone can transmit in those frequencies as long as their power does not exceed the regulatory maximum. Consequently, large portions of the RF range remain underutilized for long periods of time, while other parts—such as the ISM bands—are overutilized.

Observations like these provided the impetus behind the NeXt Generation (XG) Communications program, a technology development program sponsored by DARPA [9], which proposed opportunistic spectrum access as a means of coping with spectrum scarcity. The underlying approach has come to be known as dynamic spectrum access (DSA), and has evolved into a general methodology for dealing with spectrum scarcity [17]. The main idea is that a DSA network has two classes of users: primary and secondary. Primary users are licenced to use a particular spectrum band and always have full access to that band whenever they need it. Secondary users are allowed to use such spectrum portions but only as long as their use does not interfere with the operating needs of the primary users. The secondary users are typically cognitive radios [4] that can dynamically adjust their operating characteristics (such as waveform, power, etc.). Secondary users *sense* the spectrum for available transmission opportunities, determine the presence of primary users, and attempt to use the spectrum in a way that interferes as little as possible with the activities of the primary users.

Policies are used for specifying how secondary radio nodes

[*]The research reported in this paper was performed in connection with contract number W15P7T-08-C-P213 with the U. S. Army Communications Electronics Research and Development Engineering Center (CERDEC). The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the U. S. Army CERDEC, or the U. S. Government unless so designated by other authorized documents. Citation of manufacturers or trade names does not constitute an official endorsement or approval of the use thereof. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

are allowed to behave. Policies consist of declarative statements that dictate what secondary radios may or may not do, without prescribing how they might do it [18]. Different policies may be applicable in different regions of the world. Even in one and the same region there may be multiple policies in place, reflecting different constraints imposed by different regulatory bodies. Policy systems for cognitive radios should be able to load new policies on the fly.

There are two key questions in designing such a policy system, one representational and the other computational: How should we represent policies, and how can we make efficient use of them? Policies regulating DSA must be able to express rules such as the following: “Allow a transmission if its frequency is in a certain range and the power does not exceed a certain limit, *or* if the transmission is an emergency.” Thus, for representation purposes, we need at least the full power of propositional logic, as well as numbers (both integers and reals), and numeric comparison relations. It would also be helpful to have algebraic datatypes at our disposal, so that we could, for example, represent the *mode* of a transmission as one of a finite number of distinct symbolic values, such as *everyDay* or *emergency*. A certain amount of quantification would also be convenient, so that we could, e.g., quantify over all possible transmissions. We must also be able to introduce arbitrary transmission parameters, so a strong type system, preferrably with subtyping, should be available. Hence, first-order logic with arithmetic, and perhaps additional features such as algebraic datatypes and subsorting, would appear to be a natural representation choice, provided that we can make efficient use of it, a point that we will address shortly.

So much for representation. What are the main computational problems that a policy system must solve? The main problem is this: determine whether an arbitrary transmission request should be allowed or denied, given the current set of policies. In fact, if this were all there was to using policies for spectrum access, things would be relatively simple. But we usually want the policy engine to do more. For example, when a transmission request is not permissible, a simple “denied” answer is not very useful. The policy system should also tell the requesting radio why the request was denied, and more importantly, what it would take for its request to be allowed. As a simple example, the system might tell the radio: “Your request cannot be allowed in its present form, but it will be allowed if you only change your transmission power to such-and-such level.” Further, in such cases the policy system will usually have many choices as to which parts of the transmission request to modify in order to make it permissible. We then want it to make an optimal choice, i.e., a choice that satisfies the original request to the greatest possible extent.

In this paper we present an implementation of a policy system in Athena, an interactive proof system for polymorphic multi-sorted first-order logic with equality, algebraic data types, subsorting, and a higher-order functional programming language [3]. Athena’s rich logic, sort system, and programming language significantly facilitated the representation and manipulation of policies. Another innovation of our paper is the use of SMT solvers [15] for reasoning with and about policies. The problem of determining whether a transmission request is permissible by a given policy can be couched as a satisfiability problem, namely, the problem of determining whether the request is consistent with the policy. So SAT solving would seem to be a natural fit for this domain. However, policy rules in this domain also make heavy use of equality and numeric relations, not to mention symbolic values such as transmission modes. Thus, viewed as an abstract syntax tree, the body of a policy rule has an arbitrarily complicated boolean structure internally, with relational atoms at the leaves, whose semantics come from standard theories such as those of arithmetic, equality, and so on. That is what makes SMT solving an ideal paradigm for this problem. We will show that the problem of making optimal adjustments to transmission parameters can also be formulated quite naturally in this setting as a weighted Max-SAT problem, many instances of which can be efficiently solved despite the problem’s high theoretical complexity. Indeed, competitive SMT solvers such as CVC3 [7] and Yices [6] are highly optimized programs that can readily decide extremely large formulas. As a quick comparison, another recently built policy system for spectrum access, Bresap, which uses BDDs as its underlying technology, cannot handle policies with more than 20 atomic expressions [1, p. 83]. By contrast, our system can easily handle policies with many *thousands* of atomic expressions.

The remainder of this paper is structured as follows. The next section describes the representation of spectrum-access policies in Athena.[1] Section 3 discusses the integration of SMT solvers with Athena, and shows how to reduce the problem of reasoning with spectrum-access policies to SMT solving. In section 4 we model the problem of making optimal changes to transmission requests as a Max-SAT problem, and we describe how our system models and solves such problems. We also present an optimizing algorithm that uses a version of binary search in order to minimize integer-valued objective functions with very few calls to an oracle SMT solver (at most $\log(n)$ such calls, where n is the maximum value that can be attained by the objective function). This algorithm can be used to further optimize transmissions, e.g., by minimizing the distance between the values desired by the cognitive radio and the values allowed by the policy. Section 5 presents results evaluating the performance of our policy engine. Finally, section 6 discusses related work and concludes.

2. POLICY REPRESENTATION

[1]While some Athena code is presented, knowledge of the language is not necessary. Any reader familiar with formal methods and functional programming languages should be able to follow the code.

The set of radio transmissions is treated as an abstract data type with fields such as frequency, power, mode, etc. The abstract domain of transmissions is introduced in Athena as follows:

```
domain Transmission
```

A transmission field (or "parameter") can then be introduced as a function from transmissions to values of some appropriate sort. For illustration purposes, we demonstrate the declaration of the following parameters: frequency, power, mode, and time. Additional parameters can be introduced as needed.

```
declare frequency: [Transmission] -> Int
declare power: [Transmission] -> Real
declare mode: [Transmission] -> Mode
declare hours, minutes: [Transmission] -> Int
```

Here `Int` and `Real` are built-in Athena sorts representing the sets of integers and real numbers, respectively, with `Int` a subsort of `Real`. The sort `Mode` is a finite algebraic datatype:

```
datatype Mode := emergency | everyDay | ...
```

There are two distinguished unary predicates on the set of all possible transmissions: `allow` and `deny`. These predicates are used by policies to specify which transmissions are considered permissible and which are not. More concretely, a policy is represented as a pair of sentences: an *allow condition* and a *deny condition*. An allow condition (or AC for short) is of the form: [2]

$$\texttt{(forall ?x:Transmission. allow ?x <==> } \cdots\texttt{)} \quad (1)$$

while a deny condition (DC) is of the form:

$$\texttt{(forall ?x:Transmission. deny ?x <==> } \cdots\texttt{)} \quad (2)$$

Either condition may be absent, i.e., a policy may specify only an allow condition but no deny condition, or only a deny condition but no allow condition. An absent condition is represented by the unit value, `()`. The constructor for policies is therefore a procedure that takes two conditions and simply puts them together in a two-element list:

```
define make-policy :=
  lambda (AC DC) [AC DC]
```

There are two destructor or selector procedures, one for retrieving each component of the policy:

```
define [get-AC get-DC] := [first second]
```

An important operation is the *application* of a given policy condition `C` of the form (1) or (2) to a term `t` of sort `Transmission`. Let `body` be whatever sentence would normally be where the ellipses now appear in (1) (or in (2)). Then the result of the said application is the sentence obtained from `body` by replacing every free occurrence of the quantified variable `?x:Transmission` by the term `t`. In Athena code, this procedure is defined as follows:

```
define apply :=
  lambda (C t)
    match C {
      (forall x ((_ x) <==> body)) =>
        (replace-var x t body) }
```

The built-in ternary Athena procedure `replace-var` is such that `(replace-var x t p)` produces the result obtained from the sentence `p` by replacing every free occurrence of variable `x` by the term `p`. As a concrete example, here is a sample AC:

```
define AC1 :=
 (forall ?x .
   allow ?x <==>
    frequency ?x in [5000 5700] &
    power ?x < 30.0 &
    mode ?x = everyDay & hours ?x <= 11)
```

It says that a transmission is (provisionally) permissible iff its frequency is in the 5000 . . . 5700 MHz range; its power is less than 30.0 dBm; its mode is `everyDay`; and the transmission's time is no later than 11 in the morning. Here `in` is a binary procedure defined as follows:

```
define in :=
  lambda (x range)
     match range {
       [a b] => (a <= x & x <= b)}
```

(Note that variables do not need to be explicitly annotated with their sorts. Athena has polymorphic sort inference, so a Hindley-Milner algorithm will recover the most general possible sorts of all variable occurrences.) We can now construct a policy with the above AC and with no DC as follows:

```
define policy-1 := (make-policy AC1 ())
```

Our second policy example has both an AC and a DC:

```
define policy-2 :=
  (make-policy
     (forall ?x .
       allow ?x <==> mode ?x = emergency)
     (forall ?x .
        deny ?x  <==> power ?x > 25.0))
```

A key operation on policies is that of *merging*. New policies may be downloaded into the policy database at any given time. A newly acquired policy must be integrated with the existing policy that is in place in order to produce a single merged policy with a new AC and a new DC. Merging is therefore a binary operation. The current definition of merging two policies p_1 and p_2 is simple: it is disjunction on both the AC and DC. Specifically, the AC of the merged policy allows a transmission t iff p_1 allows t *or* p_2 allows it; and it denies t iff p_1 denies it *or* p_2 denies it. Merging is

[2]A more accurate term instead of `allow` would be "allow provisionally," because, as we will see, for a transmission request to be granted, the allow condition must hold *and* the deny condition must not hold. With this caveat in mind, we will continue to use the term `allow` in the interest of simplicity.

implemented as an Athena procedure. Other alternative definitions of merging are possible, and these could be straightforwardly implemented in the same way. As an example, here is the result of merging `policy-1` and `policy-2`, as these were defined above:

```
define merged-policy :=
  (merge-policies policy-1 policy-2)

>(get-AC merged-policy)

Sentence:
  (forall ?v4:Transmission
   (iff (allow ?v4)
        (or (and (and (<= 5000
                          (frequency ?v4))
                      (<= (frequency ?v4)
                          5700))
                 (and (< (power ?v4)
                         30.0)
                      (and (= (mode ?v4)
                              everyDay)
                           (<= (hours ?v4)
                               11))))
            (= (mode ?v4)
               emergency))))

>(get-DC merged-policy)

Sentence: (forall ?v5:Transmission
            (iff (deny ?v5)
                 (> (power ?v5)
                    25.0)))
```

(Note that while Athena accepts input in either infix or prefix form, sentences are output in prefix for indentation purposes.)

We define a *transmission request* as a set of constraints over some object of sort `Transmission`, typically just a free variable ranging over `Transmission`. These constraints, which are usually desired values for some—or all—of the transmission parameters, can be expressed simply as a sentence. A sample transmission request:

```
define request-1 := (frequency ?t = 150 &
                     mode ?t = everyDay &
                     hours ?t = 16)
```

This is a conjunction specifying that the `frequency` of the desired transmission (represented by the free variable `?t`) should be 150, its mode should be `everyDay`, and its time should be between 4:00 and 4:59 (inclusive) in the evening. A transmission request may be incomplete, i.e., it may not specify desired values for every transmission parameter. The preceding request was incomplete, since it did not specify values for the `power` and `minutes` parameters.

3. POLICY REASONING

The main task of the policy system is this: determine whether the policies currently in place allow or deny a given transmission request. This is in fact the simplest formulation of the core reasoning problem, in that it only requires a yes/no answer from the policy system. But in general the policy system needs to solve more interesting versions of this problem, namely: If the transmission request is granted, find appropriate values for any missing parameters, in case the request was incomplete; if the request is not granted, find appropriate values for the transmission parameters that *would* render the transmission permissible. Moreover, in the second case, we often want to compute values in a way that is optimal w.r.t. to the given request, e.g., so that the original transmission request is disrupted as little as possible. We discuss such optimality issues in section 4.

The above problems are naturally formulated as satisfiability problems. However, the most appropriate satisfiability framework here is not that of straight propositional logic, but rather that of *satisfiability modulo theories*, or SMT for short [15]. In the SMT paradigm, the solver is given an arbitrary formula of first-order logic with equality and its task is to determine whether or not the formula is satisfiable with respect to certain background theories. Typical background theories of interest are integer and/or real arithmetic (typically linear, but not necessarily), inductive data types (free algebras), the theory of uninterpreted functions, as well as theories of lists, arrays, bit vectors, etc. Hence, most of the function symbols and predicates that appear in the input formula have fixed interpretations, given by these background theories. An SMT solver will not only determine whether a sentence p is satisfiable; if it is, it will also provide appropriate satisfying values for the free variables and/or constants that occur in p.

Athena is integrated with a number of SMT solvers, such as CVC3 and Yices; the one used for this project is Yices [6], mostly because it can solve Max-SAT problems. The main interface is given by a unary Athena procedure `smt-solve`, which takes an arbitrary first-order sentence p and attempts to determine its satisfiability with respect to the appropriate theories. If successful, `smt-solve` will return a list of values for all free variables and/or constants that occur in p. Some examples:

```
datatype Day :=
  Mon | Tue | Wed | Thu | Fri | Sat | Sun

>(smt-solve ?x = 2 & ?y = 3.4 |
            ?x = 5 & ?d = Mon)

List: [(= ?y 3.4) (= ?x 2) (= ?d Mon)]
```

The input to `smt-solve` here was a disjunction of two conjunctions. The free variable `?d` ranges over the datatype `Day`; while `?x` and `?y` range over `Int` and `Real`, respectively. The output is a list of identities providing values for each free variable that render the input satisfiable. Because the argument to `smt-solve` is a sentence, i.e., a native Athena data value, this provides a very flexible and high-bandwidth programmable interface to SMT solvers. This interface proved extremely useful for our system.

The policy reasoning problems described earlier have a natural formulation in SMT. Specifically, consider an arbitrary transmission request *tr*, i.e., a first-order sentence with a free variable x ranging over `Transmission` (there may be other free variables as well). To determine whether *tr* is allowed by the current policy: (1) We construct a longer request, *tr′*, which is the conjunction of (a) *tr*; (b) the application of the current policy's AC to x; and (c) the application of the negation of the current policy's DC to x:

```
tr & (apply (get-total-AC) x) &
     ~ (apply (get-total-DC) x)
```

The procedure `get-total-AC` returns the AC of the current ("total") policy; likewise for `get-total-DC`. So *tr′* represents *all* the constraints imposed on the requested transmission, both by the original transmission request, *tr*, and by the policy itself, whose AC must hold for the transmission variable x while its DC must not. (2) We then run `smt-solve` on *tr'*. If the result is a satisfying assignment, then the request is granted, and all we need to do is report values for any missing parameters (parameters that did not figure in the given request). If, by contrast, `smt-solve` determines that *tr'* is unsatisfiable, then *tr* in its given form must be rejected. In that case we make a blank request consisting of the application of the policy's AC to x conjoined with the application of the negation of the policy's DC to x, and run `smt-solve` on it. This blank request therefore imposes no constraints at all on the transmission apart from those imposed by the policy. If the result is a satisfying assignment, we provide it to the user, otherwise we report that the current policy is unsatisfiable.

The Athena code for this algorithm is expressed in a procedure `evaluate-request`, which accepts an arbitrary request and processes it as described above. Here is the output for the example of the previous section:

```
>(evaluate-request request-1)

Transmission allowed. Appropriate values
for missing parameters:
[(= (power ?t1) 20.0)]
```

4. COMPUTING OPTIMAL ADJUSTMENTS TO TRANSMISSION PARAMETERS

When a transmission request is denied, there are usually many different ways of modifying it so as to make it permissible. For instance, the solver could change the desired transmission's time; or it could change its frequency; or it could change its power and mode; or it could change all of the above. Some of these actions may be preferable to others. For instance, the radio might be less willing to change the time of the transmission, or its power level, rather than the frequency. In such cases we want the policy system to return a satisfying assignment that is optimal in the sense of respecting as many such preferences of the radio as possible.

Our system achieves this in a flexible way by formulating the problem as an (SMT) instance of Max-SAT. In its transmission request, the radio can provide a weight w_i along with the desired value v_i of each transmission parameter p_i. The weight w_i reflects the importance that the radio attaches to p_i taking the value v_i. The SMT solver will then try to find a satisfying assignment for the request that has maximal total weight. In that case, the request is not just a list of constraints `[`$c_1 \cdots c_n$`]` but a list of pairs of constraints and weights `[[`c_1 w_1`]` $\cdots$ `[`c_n w_n`]]`. A sample transmission request might then be:

```
define weighted-request :=
  [[(frequency ?t = 8000) 10]
   [(power ?t = 35.0)     15]
   [(hours ?t = 13)       30]]
```

indicating that the relative importance of the frequency being 8000 is 10, that of the power being 35.0 is 15, and that of `hours` being `13` is 30. Suppose further that the policy in place has no DC and a disjunctive AC:

```
define AC1 :=
  (forall ?t . allow ?t <==>
     (frequency ?t in [5000 7000] &
      power ?t <= 30.0 & hours ?t > 12) |
     (frequency ?t in [6000 9000] &
      power ?t <= 40.0 & hours ?t <= 8))

define policy := (make-policy AC1 ())
```

Now consider evaluating `weighted-request` with respect to this policy. Clearly, the request cannot be allowed as is. We can make it permissible in more than one way: (a) we could demand a change of frequency and power in accordance with the values prescribed by the first branch of `AC1`, while keeping the `hours` parameter to the requested value of `13`; or (b) we could demand a change of the `hours` parameter only, in accordance with the second disjunctive branch of `AC`, while keeping the desired frequency and power values; or (c) we could demand that all three parameter values change. From these possibilities, only (a) is optimal, in that it honors the original request to the greatest extent possible (as determined by the given weights). Running this example in our system results in the following output:

```
>(evaluate-request weighted-request)

The following parts of the request
could not be satisfied:

(= (power ?t) 35.0) (= (frequency ?t) 8000)

Here is a maximally satisfying assignment:

[(= (mode ?t) everyDay)
 (= (frequency ?t) 5999)
 (= (power ?t) 30) (= (hours ?t) 13)]
```

By contrast, if we had changed the weight of the `hours` parameter from 30 to 20, the result would then change the `hours` parameter instead of the frequency and power, since retaining the values of the two latter parameters would result in a maximum weight of 25.

Note that weights can be arbitrary integers or a special "infinite" token, indicating a hard constraint that *must* be satisfied. In fact this infinite weight is the one that Athena attaches to the constraints obtained from the AC (and the negation of the DC) of the current policy. But radios can also use infinite weights in their requests.

Occasionally there may be additional requirements on the assignments returned by the policy system, beyond honoring the original request to the greatest extent possible. For example, if a parameter value must change, we may want the new value to be as close as possible to the original requested value. If we are not allowed to transmit at the exact desired time, for instance, we may want to transmit as close to it as possible (as can be allowed by the currently installed policies). To meet such requirements we generally need the ability to perform optimization by minimizing some objective function, in this case the absolute difference between desired and permissible parameter values. Most SMT solvers do not perform optimization (apart from Max-SAT, in the case of Yices, though see below), but we can efficiently implement an integer optimizer in terms of SMT solving. The idea is to use binary search in order to discover the optimal cost with as few calls to the SMT solver as possible: at most $O(\log n)$ calls, where n is the maximum value that the objective function can attain. Specifically, let c be an arbitrary constraint that we wish to satisfy in such a way that the value of some "cost" term t is minimized, where *max* is the maximum value that can be attained by the cost function (represented by t). (Tf this value is not known *a priori*, it can be taken to be the greatest positive integer that can be represented on the computer.) The algorithm is the following: We first try to satisfy c conjoined with the constraint that the cost term t is between 0 and half of the maximum possible value: $0 \leq t \leq (max\ \mathbf{div}\ 2)$. If we fail, we recursively call the algorithm and try to satisfy c augmented with the constraint $(max\ \mathbf{div}\ 2) + 1 \leq t \leq max$. Whereas, if we succeed, we recursively call the algorithm and try to satisfy c augmented with the constraint $0 \leq t \leq (max\ \mathbf{div}\ 4)$. Some care must be taken to get the bounds right on each call, but this algorithm is guaranteed to converge to the minimum value of t for which c is satisfied, provided of course that the original constraint c is satisfiable for *some* value of t. This algorithm is implemented by a ternary Athena procedure `smt-solve-and-minimize`, such that

```
(smt-solve-and-minimize c t max)
```

returns a satisfying assignment for `c` that minimizes `t` (whose maximum value is `max`).

For example, suppose that `x`, `y`, and `z` are integer variables to be solved for:

```
define [x y z] := [?x:Int ?y:Int ?z:Int]
```

subject to the following disjunctive constraint:

```
define c :=
  (x in [10 20] & y in [1 20] &
   z in [720 800]) |
  (x in [500 600] & y in [30 40] &
   z in [920 925])
```

Suppose also that the desired values are `x = 13`, `y = 15`, `z = 922`. The task is to find values for these variables that satisfy c while diverging from the desired values as little as possible. We can readily model this problem in a form that is amenable to `smt-solve-and-minimize` as follows. First we define the objective-function term t, as the sum of the three differences:

```
define t := (?d-x:Int + ?d-y:Int + ?d-z:Int)
```

with the difference terms defined as follows:

```
define d-x-def :=
  (ite (x > 13) (?d-x = x - 13)
                (?d-x = 13 - x))
define d-y-def :=
  (ite (y > 15) (?d-y = y - 15)
                (?d-y = 15 - y))
define d-z-def :=
  (ite (z > 922) (?d-z = z - 922)
                 (?d-z = 922 - z))
```

Here `ite` is a simple if-then-else procedure:

```
define ite :=
  lambda (c x y) ((c ==> x) &  (~ c ==> y))
```

Assume that we do not know the exact maximum value that `t` can attain, but we know that it cannot be more than 10^6. We can then solve the problem with the following call:

```
define diff-clauses :=
  (d-x-def & d-y-def & d-z-def)
define query := (c & diff-clauses)

>(smt-solve-and-minimize query t 1000000)

List: [(= ?x 13) (= ?y 15) (= ?z 800)
       (= ?d-x 0) (= ?d-y 0) (= ?d-z 122)]
```

This solution was found by a total of 8 calls to the SMT solver (for a total time of about 100 milliseconds). Why were only 8 calls required when we started the binary search with a maximum of 10^6? One would expect about $\log 10^6$ calls to the smt solver, i.e., roughly 20 such calls. However, our implementation uses the information returned by each call to the SMT solver to speed up the search. That often results in drastic shortcuts, cutting down the total number of iterations by more than a factor of 2.

This procedure enables our system to optimize any quantity that can be given an integer metric. Moreover, unlike independent branch-and-bound algorithms for integer programming, it has the advantage that it allows not just for numeric constraints, but for arbitrary boolean structure as well, along with constraints from other theories such as reals, lists, arrays, bit vectors, inductive datatypes, etc. While more sophisticated approaches to optimization in the SMT paradigm are beginning to emerge (e.g., [2]), the implementation described here has been quite adequate for our purposes.

5. PERFORMANCE

In order to test our system we wrote an Athena program that generates test instances, with two independently controllable variables: parameter number and policy number. In particular, we defined a procedure `make-policy-set` with two parameters, `param-num` and `policy-num`. The output is a list of `policy-num` policies, where each policy involves `param-num` transmission parameters. The latter came from a pre-declared pool of 100 transmission parameters, half of them integer-valued and the other half real-valued. We distinguished the following types of policies:

- *permissive* policies, which have only an AC;
- *prohibitive* policies, which have only a DC and no AC;
- *mixed* policies, which have both.
- *Ordering*$/N$ policies. These policies are parameterized over $N > 0$. Specifically, an ordering/N policy is one whose AC and/or DC is of the following form:

  $$\texttt{(forall ?t:Transmission . } D \texttt{ ?t <==>} \\ \texttt{(}f_{i_1}\texttt{ ?t } R_1\ c_{i_1}\texttt{) \& ... \& (}f_{i_k}\texttt{ ?t } R_k\ c_{i_k}\texttt{))}$$

  where D is either `allow` or `deny`, and $\forall j \in \{1, \ldots, k\}$: f_{i_j} is a transmission parameter; $R_j \in \{<, >, <=, >=\}$; and c_{i_j} is a constant number of the appropriate sort. We refer to the ordering constraints $\texttt{(}f_{i_j}\texttt{ ?t } R_j\ c_{i_j}\texttt{)}$ as the *atoms* of the AC (or DC). An ordering/N policy may be permissive, prohibitive, or mixed, but its total number of atoms (i.e., the number of the AC's atoms plus the number of the DC's atoms) must equal N.
- *Equational*$/N$ policies. The AC and/or DC of such a policy is of the same form as shown above, except that each R_i is the identity relation. These identities are the *atoms* of the condition. The total number of atoms (of the AC and DC together) must be N.
- *Inequational*$/N$ policies. Same as above, except that the relation in question here is inequality.
- *Range*$/N$ policies. The AC and/or DC of such a policy is of the same form as that of an ordering/N policy, except that each R_i is the relation `in`, and each constant c_i is a range $\texttt{[}a\ \ b\texttt{]}$.
- *Disjunctive-Conjunctive*$/N$ policies. The AC and/or DC of such a policy is a disjunction of conjunctions, where each atom is of the form $\texttt{(}f_x\texttt{ ?t in [}\cdots\texttt{])}$. The number of atoms are required to add up to N.

A call of the form `(make-policy-set N P)` returns a list L containing `P` policies, each of which involves `N` transmission parameters. This list is roughly equally partitioned among all the preceding types. Specifically, 20% of the policies in L are ordering/`N` policies. About 1/3 of these ordering/`N` policies are permissive, 1/3 are prohibitive, and 1/3 mixed. The next 20% of L consists of equational/`N` policies, and these are again evenly split between permissive-, prohibitive-, and mixed-policy subsets. The third 20% contains inequational/`N` policies, and so forth. Once a list L of policies is thereby obtained, we combine them all into a single policy by folding the `merge-policies` operator over L, with the `empty-policy` as the identity element. It is with respect to this merged policy that we tested transmission requests. The requests were generated randomly.

Figure 1 shows the SMT-solving times for processing plain transmission requests (i.e., without weights attached to the various transmission parameters), for various combinations of policy-set sizes and transmission parameter numbers, where the policy sets are evenly mixed as described above. Figure 2 shows the corresponding times for optimal processing of transmission requests, i.e., requests that attach weights to each of the transmission parameters and are solved as Max-SAT problems. In order to stress-test the implementation further, we repeated the experiments with sets of policies that were more structurally complex, doing away with simple ordering, equational, and inequational policies, and using instead only range and disjunctive-conjunctive policies. The corresponding results are shown in Figure 3 and in Fig-

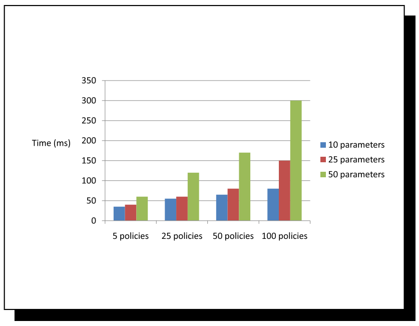

Figure 1: SMT solving times for processing regular transmission requests.

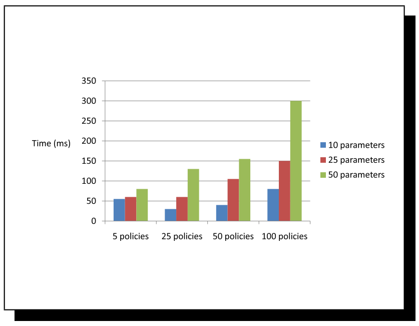

Figure 2: Max-SAT SMT solving times for optimized processing of transmission requests.

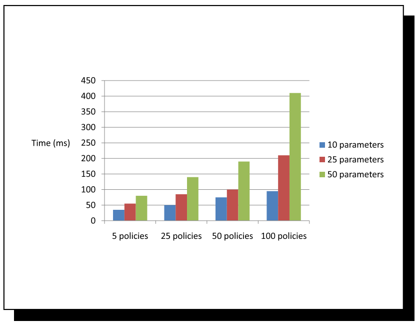

Figure 3: Solving times for processing of transmission requests with structurally complex policies.

ure 4. With these sort of structurally complex policies in place, the highest increase occurs in optimal processing of transmission requests with more than 50 policies and parameters.

The graphs show only the time spent on SMT solving, which is the bulk of the work that needs to be done when processing transmission requests. We do not include the time spent on translating from Athena to SMT notation and back. This translation is a straightforward linear-time algorithm (linear in the size of the formula to be translated, on average, since it uses hash tables for the necessary mappings between Athena and SMT namespaces), and the time spent on it in most cases is neglibible. For huge policies containing hundreds of thousands of nodes, the translation does take longer, though still typically a fraction of a second. Virtually all of the translation cost is incurred when translating the policy, since the transmission requests are tiny by comparison. Observe, however, that there is no reason to be translating the entire policy anew each time the engine needs to process a new request. Instead, the (merged) policy can be translated off-line, once, and subsequently cached in SMT notation in some file. Thus, the translation time is an one-time, off-line cost.

All experiments were performed on a 2.4 GHz Intel Core i3-370M dual-core processor with 4GB RAM, running Windows 7. The `time` command of the Cygwin shell was used

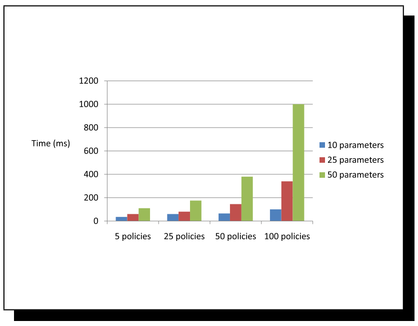

Figure 4: Max-SAT SMT solving times for transmission requests with structurally complex policies.

to get the timing data. In most cases, the reported system time (the `sys` entry of `time`'s output) was 0.000, i.e., too small to be reliably measured, and hence the time reported here is `real` (wall clock) time, which includes time segments spent by other processes and times during which the SMT solver was blocked (e.g., waiting for I/O to complete). Therefore, the times reported here are overly conservative; the actual times spent on SMT solving are smaller.

6. RELATED WORK & CONCLUSIONS

We have presented an implementation of a policy system for dynamic spectrum access. Policies are represented and manipulated in Athena, a formal proof system for first-order logic with polymorphism, subsorting, inductive datatypes, arbitrary computation, and definitional extension. Most of these features have proven useful in the engine's implementation; others would become useful if the engine were to be extended so that it used, e.g., structured ontologies instead of flat data types.

Previous work in this area includes BBN's XGPLF [8] and SRI's Coral [10, 11]. XGPLF does not have a formal semantics and is limited in what it can express (it is based on OWL [19]). Moreover, XGPLF cannot model inheritance. To the best of our knowledge, the only implementation of a policy reasoning engine based on XGPLF is the one built by the Shared Spectrum Company [12]. They used SWI-Prolog as the underlying reasoning engine. In field tests using resource-limited radio devices, SSC replaced the Prolog reasoner with "a simplified reasoner developed in C/C++" [12, p. 504]; no details are provided regarding the design, implementation, or correctness of this simplified reasoner. SSC's engine can only return yes/no answers to transmission requests.

Coral (Cognitive Radio Policy Language) is a new language specifically designed for expressing policies governing the behavior of cognitive radios. Like Athena, Coral offers a rich and extensible first-order logical language that includes arithmetic and allows for the introduction and definition of new function symbols. It also features subtyping and can therefore express inheritance and ontologies. Some notable differences from Athena include: (a) Unlike Coral, Athena has polymorphism. Thus, e.g., it is not necessary to introduce lists for integers and lists for booleans separately; a generic parameterized definition is sufficient. (b) Athena has automatic sort inference based on the Hindley-Milner algorithm, which is convenient in practice because it allows for shorter—and usually cleaner—specifications. (c) Athena has a general-purpose formally defined programming language that can seamlessly interact with its logical layer, and which can be used to dynamically construct and manipulate logical formulas very succinctly. This programming language offers procedural abstraction, as well as side effects through mutation, including reference cells and vectors, features which are often important for efficiency. The ability to compute with terms and sentences as primitive data values was very useful. While Coral can express some computations via universally quantified identities which can then be interpreted as executable rewrite rules by a tool such as Maude [16], it does not offer procedural abstraction, i.e., it is not possible to define arbitrary procedures.

Two policy engines based on Coral have been implemented [10, 11]. The initial implementation used Prolog as the underlying reasoning engine, and was only able to return yes/no answers to transmission requests. The second (and current) implementation uses a custom-made proof system to reason about transmission requests. The system has four kinds of proof rules: forward chaining, backward chaining, partial evaluation based on conditional rewriting, and constraint propagation and simplification. The proof system is implemented in the rewriting engine Maude. One issue with this implementation is that a positive answer is given only if the transmission request is an *exact match* of the operative policy conditions. But that is not likely to be the case in practice. Most transmission requests are likely to be incomplete or to diverge from the policy, at least in some small degree. In such cases, the Coral implementation does not return values for the relevant transmission parameters. Rather, it returns an arbitrarily complicated first-order formula representing all the possible "opportunity constraints" that the radio could use to modify its request so as to make it permissible. But that is not likely to be of much use to the requesting party. It is not realistic to require that whoever made the transmission request should be able to understand and reason about arbitrarily complicated logical formulas in order to understand the policy system's output.

A second issue with this implementation is efficiency. The Coral team has not published precise benchmarks describing their engine's performance for variable numbers and types of policies (and for variable numbers of transmission parameters), but they have released a demo of their implementation that can be used to evaluate transmission requests with respect to policies that have a fixed set of transmission parameters, namely: location (latitude and longitude); time (hours, minutes, and seconds); sensed power; emissions; network id; and role (slave/master). Even with this fairly small set of transmission parameters, and with only 11 active policies in place, it has been reported [1, p. 18] that evaluating a single transmission request took the Coral engine 58 seconds, and resulted in an output formula comprising 75 constraints. In the preceding section we saw that our implementation can evaluate transmission requests with several dozens of parameters and with over 50 complex policies in place in a fraction of a second; this is so even when the requests are processed optimally. Moreover, the Coral engine has no notion of optimality. It does not allow the requesting party to specify that some parameters are more important than others, or to ask that the request should be satisfied to the greatest extent possible, or with as little divergence from the requested values as possible.

Another policy engine for spectrum access is Bresap, a system that was recently implemented at Virginia Tech [1, 5]. Unlike the other systems discussed above, Bresap is limited to policies that can be adequately represented as finite Boolean functions, which it encodes as binary decision diagrams (BDDs). Therefore, unlike Coral and our system, Bresap is not suitable as an expressive language for specifying policies. For instance, it has no facilities for introducing new policy concepts and/or rules. Indeed, Bresap is not a language. It is a system that reads certain types of policies expressed in XML and converts them to BDDs. This is done by assigning a distinct Boolean variable to each atomic expression that appears in the policy. For instance, a policy such as "allow transmission in the frequency range $a \ldots b$ if the power does not exceed m" would result in the introduction of two distinct Boolean variables x_1 and x_2, with x_1 corresponding to the atom $a \leq f \leq b$ (where f stands for the transmission's frequency); and with x_2 corresponding to the atomic expression $p \leq m$ (where p stands for the transmission's power). The accepting condition could then be represented by the boolean function $x_1 \wedge x_2$. Graph algorithms are used to merge different policy BDDs. With a single BDD in place representing the result of merging all active policies, Bresap can then accept an incoming transmission request and evaluate it. A transmission request must be an attribute-value list of the form $a_1 = v_1, \ldots, a_n = v_n$, where a_i is a transmission parameter (such as mode or frequency) and v_i is the corresponding desired value. Notably, Bresap is capable of handling underspecified (incomplete) requests. It is also capable of attaching costs to transmission parameters (where a given cost indicates the penalty paid for changing the value of that parameter), and then modifying the parameter values of a transmission request in a way that minimizes the overall cost.

A drawback of Bresap is that the underlying policy representation framework, that of BDDs (or propositional logic, more generally), lacks semantics for the numeric constraints that pervade spectrum access policies. To put it simply, BDDs do not know arithmetic, and thus do not have the wherewithal to "understand" that $<$ is transitive, that $1 + x$ is greater than x, that 10 and $14 - 4$ are the same object, and so on. Whereas an SMT solver immediately realizes that $f < 10$ conjoined with $f \geq 30$ is contradictory, because it uses a dedicated reasoning procedure for arithmetic, in the approach of Bresap these two atoms would result in the introduction of two Boolean variables, x_1 and x_2, the former representing $f < 10$ and the latter representing $f \geq 30$. Thus, the conjunction of x_1 and x_2 is represented by the innocuous-looking Boolean function $x_1 \wedge x_2$, a perfectly consistent condition. To represent the fact that in the present context this is actually inconsistent, one has to append to it ad hoc clauses such as $C = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$. This extra information, C, conjoined with $x_1 \wedge x_2$, would then allow us to conclude that the condition is unsatisfiable. That is, in fact, what Bresap does: It "semantically analyzes" the various atomic expressions in the policy in order to generate additional constraints—so-called "auxiliary rules" [1, section 3.5]—that prevent the engine from taking "illogical" paths in the BDD. This approach has some disadvantages.[3] First, it is an essentially ad hoc encoding of arithmetic facts, the generation of which should not be part of the policy engine's trusted computing base. Second, the additional encoded information, even if it is polynomial in size, can significantly blow up the resulting BDD. Other efficiency issues in Bresap include the conversion algorithm that produces the BDD from the atomic Boolean expressions of the policy. This algorithm has exponential time complexity in the number of expression variables. As a result, Bresap is not currently able to handle policies with more than 20 atomic Boolean expressions [1, p. 83].

In conclusion, we believe that the combination of a programmable and expressive formal framework such as Athena with an efficient SMT solver is a highly suitable implementation vehicle for spectrum access policy engines. It combines rich expressivity with efficient performance, a consideration that is likely to be crucial for cognitive radios. To the best of our knowledge, SMT solvers have not been used before for reasoning about policies, although we believe that they are ideal for this task. Indeed, we believe that there is not much that is special about spectrum access, and that the same approach we have introduced here could be used to represent and reason about policies in other domains.

[3]Note that this criticism is not peculiar to Bresap. It applies to all policy engines that represent semantically rich policies (requiring at least linear arithmetic) as Boolean functions.

7. REFERENCES

[1] A. A. Deshpande.
Policy Reasoning for Spectrum Agile Radios. Masters thesis, Electrical and Computer Engineering Department, Virginia Tech, 2010.

[2] A. Cimatti and A. Franzn and A. Griggio and R. Sebastiani and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS 2010*, pages 99–113, 2010.

[3] K. Arkoudas. Athena.
`http://www.pac.csail.mit.edu/athena.`

[4] B. A. Fette. *Cognitive Radio Technology*. Academic Press, 2nd edition, 2009.

[5] B. Bahrak and A. A. Deshpande and M. Whitaker and J. M. Park. BRESAP: A Policy Reasoner for Processing Spectrum Access Policies Represented by Binary Decision Diagrams. In *New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on*, pages 1 –12, April 2010.

[6] B. Dutertre and L. de Moura. The Yices SMT Solver. Tool paper, available online from
`yices.csl.sri.com/tool-paper.pdf.`

[7] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302.
Springer-Verlag, 2007. Berlin, Germany.

[8] BBN Technologies. XG Working Group, XG Policy Language Framework, Request for Comments, version 1.0. `www.ir.bbn.com/projects/xmac/pollang.html`, 2004.

[9] Darpa. News Release for Next Generation (XG) Communications Program Request for Comments.
`www.darpa.mil/news/2004/xg_jun_04.pdf.`

[10] G. Denker, D. Elenius, R. Senanayake, M.-O. Stehr, and D. Wilkins. A Policy Engine for Spectrum Sharing. In *New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on*, pages 55 –65, April 2007.

[11] D. Elenius, G. Denker, M. O. Stehr, R. Senanayake, C. Talcott, and D. Wilkins. CoRaL–Policy Language and Reasoning Techniques for Spectrum Policies. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, pages 261–265, 2007.

[12] F. Perich. Policy-Based Network Management for NeXt Generation Spectrum Access Control. In *New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on*, pages 496 –506, April 2007.

[13] FCC Spectrum Policy Task Force. Report of the spectrum efficiency working group. Available from
`www.fcc.gov/sptf/files/SEWGFinalReport_1.pdf.`

[14] I. F. Akyildiz and W.-Y. Lee and M. C. Vuran and S. Mohanty. NeXt Generation / Dynamic Spectrum Access / Cognitive Radio Wireless Networks: A Survey. *Computer Networks Journal (Elsevier)*, 50:2127–2159, September 2006.

[15] L. de Moura and B. Dutertre and N. Shankar. A Tutorial on Satisfiability Modulo Theories. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 20–36. Springer, 2007.

[16] M. Clavel and F. Durán and S. Eker and P. Lincoln and N. Martí-Oliet and J. Meseguer and C. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in LNCS, pages 76–87. Springer, 2003.

[17] Q. Zhao. A Survey of Dynamic Spectrum Access. *IEEE Signal Processing Magazine*, 24(3):79–89, 2007.

[18] R. Chadha and L. Kant. *Policy-Driven Mobile Ad hoc Network Management*. Wiley-IEEE Press, 2007.

[19] World Wide Web Consortium. OWL 2 Web Ontology Language Document Overview. Available from
`www.w3.org/TR/owl2-overview/`, 2009.