A Semantic Web Reasoner for Rules, Equations and Constraints

Daniel Elenius, Grit Denker, Mark-Oliver Stehr

SRI International, Menlo Park, California, USA
`firstname.lastname@sri.com`

Abstract. We describe a reasoner for OWL ontologies and SWRL policies used on cognitive radios to control dynamic spectrum access. In addition to rules and ontologies, the reasoner needs to handle user-defined operations (e.g., temporal and geospatial). Furthermore, the reasoner must perform sophisticated constraint simplification because any unresolved constraints can be used by a cognitive radio to plan and reason about its spectrum usage. No existing reasoner supported all these features. However, the term rewriting engine Maude, augmented with narrowing, provides a promising reasoning mechanism. This allows for a behavior similar to that of a logic programming system, while constraint simplification rules as well as operations can easily be defined and processed. Our system and general approach will be useful for other problems that need sophisticated constraint processing in addition to rule-based reasoning, or where new operations need to be added. The implementation is efficient enough to run on resource-constrained embedded systems such as software-defined radios.

1 INTRODUCTION

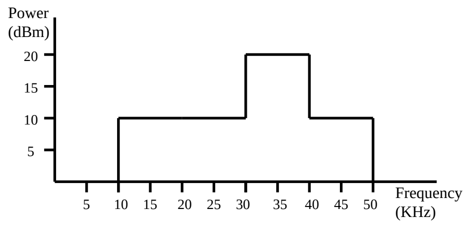
The radio frequency spectrum is a finite resource, and demand for it is increasing. Large, robust, and agile radio networks are very difficult to achieve with traditional methods. *Cognitive radios* and *dynamic spectrum access* offers a solution, where radios can use sensors to avoid interference and *reason* about spectrum usage. This area offers an interesting application domain for Semantic Web technologies.

In DARPA's neXt Generation (XG) program, declarative *policies* are used to control access to the spectrum resource. Policies define circumstances under which radios are allowed to transmit, in terms of frequencies used, power levels, geographic location, time, and so on. These concepts are defined in *ontologies*. A *policy reasoner* is used to decide whether a transmission is allowed. More background can be found in [1].

The ontologies and policies are encoded in OWL and SWRL. The semantics is the usual first-order model theory. However, the reasoning problem is not a straightforward Description Logic subsumption problem. This paper describes our policy reasoner, which is a general-purpose reasoner for OWL and SWRL, but with several additional features. Our intent here is not to give a formal characterization of our reasoner, but to motivate and describe the system, and to contrast it with other reasoning technologies.

Our approach is to translate to a target language that has reasoning mechanisms appropriate to our domain. There are several desired features of such a language and its reasoning system. Policies can be permissive or restrictive. It is intuitive to write policies as *rules*, e.g., of the form `Allow` $\Leftarrow$ *Constraints* (for permissive policies), where we try to prove `Allow` by solving the constraints. (Replace `Allow` by `Disallow` for restrictive policies). It is also useful to define auxiliary predicates using rules, for modularity, reusability, and convenience of specification. This speaks in favor of a logic programming type of language.

We also need to define certain operations, such as arithmetic on time primitives, and calculation of the distance between two geographic points. Pure logic programming does not provide an appropriate computation framework for this, and because we want purely declarative policies, procedural attachments are not an option. However, these types of operations can be defined in a natural, declarative way that allows for efficient computation, using *functions* and *equational specifications*. Another use of functions is to define so-called power masks, i.e., functions from frequency to power levels (see Figure 1).



$$\texttt{power(f)} = 0.0 \Leftarrow \texttt{f} < 10.0$$
$$\texttt{power(f)} = 10.0 \Leftarrow 10.0 \leq \texttt{f} \wedge \texttt{f} < 30.0$$
$$\texttt{power(f)} = 20.0 \Leftarrow 30.0 \leq \texttt{f} \wedge \texttt{f} < 40.0$$
$$\texttt{power(f)} = 10.0 \Leftarrow 40.0 \leq \texttt{f} \wedge \texttt{f} < 50.0$$
$$\texttt{power(f)} = 0.0 \Leftarrow 50.0 \leq \texttt{f}$$

Fig. 1. A power mask (left) is a function from frequency to power. Such functions can be intuitively defined using conditional equations (right).

Another ubiquitous feature in spectrum policies are *numerical constraints*, for frequency ranges, power levels, minimum distance to other transmitters, and so on. Interesting problems arise from the combination of numerical and other logical constraints.

To summarize, we need a reasoner that allows rules in the logic programming style, functions defined using equations, and flexible handling of numerical constraints.

This remainder of this paper is organized as follows. In Section 2 we examine existing reasoning technologies with regard to the desired features. Section 3 describes the system in which we implemented our reasoner, and its underlying logic. Section 4 describes our reasoner, and shows how it can be used as a Web reasoning tool. We end with some conclusions in Section 5.

2 RULES, EQUATIONS, AND CONSTRAINTS

Here, we examine the features needed by the policy language in more detail, and discuss existing technologies supporting them. There are two somewhat (but not completely) separate issues. The first is the combination of logic programming with functions and equations. The second is the addition of a flexible notion of constraints.

2.1 Combining Rules and Equations

Combining a rule system with functions and equations could be seen as combining the logic programming and functional programming paradigms. There is a large body of work on this topic (see [2, 3] for extensive surveys). Following [4], we take the relational paradigm to be about *multidirectional*, *nondeterministic*, and *not necessarily convergent* computation, and the functional paradigm to be about *directional*, *deterministic*, and *convergent* computation (see Table 1).

Functions, Equations	Relations, Rules
Deterministic *No failure or backtracking*	Nondeterministic *Failure/backtracking*
Directional *Takes inputs and produces outputs*	Multidirectional *No specific inputs/outputs*
Terminating *Usually expected to terminate on legal input*	Not necessarily terminating *Can enumerate infinitely many instances of arguments*
Evaluation *Reduction*	Deduction *Search/resolution/ unification*

Table 1. Comparison of two paradigms of computation.

Having a language that supports only one of the two paradigms usually forces users to make unnatural encodings in order to support the missing functionality. One one hand, functions encoded as relations in logic programming cannot take advantage of the efficient evaluation of deterministic functions that a functional programming language can perform. On the other hand, functional programming cannot take advantage of the built-in search and partially instantiated data structures that logical programming supports. Thus, a combination of both paradigms is called for.

There are two fundamental approaches to the problem of combining relations and functions [3]. One is to start with the relational approach and add functions. The other is to start with the functional approach and add relations.

In some sense, relations are more general than functions and can emulate functional behavior. For example, functional notations can be translated into Prolog through *flattening*, and determinism of function evaluation can be achieved by using Prolog features like cuts. This approach is used in [5] and [6], which both take logic programming as the starting point, and add functional notation through a translation approach. These approaches make a syntactical distinction between rules and equations, and between relations and functions. Normal Prolog mechanisms are used for rules and relations, and the translated versions with a special encoding to behave like functions are used for the equational/functional side.

Systems that take functional programming as the starting point instead treat relations and logical connectives (*and*, *or*, etc.) as boolean functions, and generalize their expressiveness by allowing new variables on the right-hand sides of equations. Thus, these languages typically do not make a distinction between relations and functions.

Regardless of the starting point, the choices of operational principles are similar, the main choices being *residuation* (e.g., Le Fun [4]) or *narrowing* (e.g., Curry [7]). The idea of residuation is to delay evaluation of terms containing logical variables until the variables have been instantiated due to the evaluation of other terms. Narrowing works by using unification instead of matching when a redex is matched with the left-hand side of an equation, thus allowing logical variables in the redex. For example, given the equations

$$brother(Phil) = Tom$$

$$brother(Phil) = Jack$$

and the redex $brother(x) = Tom$, normal reduction does not work since we have a variable in the redex. However, with narrowing, the redex unifies with the first equation and the binding $x = Phil$. In general, narrowing can result in several different solutions, which means that backtracking or some equivalent mechanism must be provided. For example, the redex $brother(Phil) = y$ unifies with both equations, with the bindings $y = Tom$ and $y = Jack$. Narrowing is the most general approach, encompassing both unification and reduction, and thus supporting both the functional and the relational paradigms. Our solution is based on a functional language, Maude [8], and uses narrowing.

2.2 Constraints

Constraints are fundamental in the policies we considered in the XG project – in fact we take the view that policies *are* constraints. There are many different notions of "constraints". We will make clear what we mean by constraints, and how this compares to the constraints of Constraint Logic Programming (CLP) [9].

In the XG architecture, the policy reasoner must prove a special atom `Permit` based on the policies and facts available:

$$\mathit{Facts}, \mathit{Policies} \vdash \texttt{Permit}$$

Policies will have axioms about the `Permit` predicate, so that the proof obligation becomes

$$\mathit{Facts} \vdash \mathit{Constraint}$$

where *Constraint* is a formula resulting from combining all those `Permit` axioms. The *facts* come from a transmission request, where the radio states what its current configuration is, what its intended transmission looks like, and what the state of the environment is, as far as the radio can determine by using its sensors. As a simplistic example, consider the following policies:

$$\texttt{Permit} \Leftrightarrow \texttt{Allow} \wedge \neg\texttt{Disallow}$$

$$\texttt{Allow} \Leftarrow 500 < \texttt{freq} \wedge \texttt{freq} < 1000$$

$$\texttt{Allow} \Leftarrow 1200 < \texttt{freq} \wedge \texttt{freq} < 1400$$

$$\texttt{Disallow} \Leftarrow 900 < \texttt{freq} \wedge \texttt{freq} < 1300$$

The first, "top-level", policy relates permissive and restrictive policies. The given top-level policy just says that we need to find *some* policy that allows, and *no* policy can disallow. Other top-level rules can be used to account for priorities and other relationships between policies. With these policies, after expanding the definition of `Permit`, we get the proof obligation

$$\begin{aligned} \mathit{Facts} \vdash (500 < \texttt{freq} \wedge \texttt{freq} < 1000 \vee 1200 < \texttt{freq} \wedge \texttt{freq} < 1400) \\ \wedge\neg(900 < \texttt{freq} \wedge \texttt{freq} < 1300) \end{aligned}$$

which can be further simplified to

$$\mathit{Facts} \vdash 500 < \texttt{freq} \wedge \texttt{freq} \leq 900 \vee 1300 \leq \texttt{freq} \wedge \texttt{freq} < 1400$$

If *Facts* contains for instance $\texttt{freq} = \texttt{800}$, the whole constraint reduces to `True`, which means that the proof is completed and the radio can transmit. However, we are also interested in the case where such facts are *not* provided, because radios can make underspecified requests as a way of querying for available transmission opportunities. Thus, whatever remains of the constraint after simplification has been performed should be returned as a result to the radio.

The description above could be subsumed under a very general view of CLP. However, the variant of CLP that has been implemented in current Prolog systems is less general, in at least three respects. First, CLP does not handle negation in a clean, logical way. For example, we would like to be able to get simplifications like

$$\neg\texttt{freq} < 500 \rightarrow \texttt{freq} \geq 500$$

In Prolog, negation is handled by the negation-as-failure method, which precludes such inferences. Second, CLP does not handle disjunction. For example, it cannot perform the simplification

$$\texttt{freq} < 500 \vee \texttt{freq} > 400 \rightarrow \texttt{True}$$

Third, in Prolog/CLP, only special constraint formulas are returned when they are not completely satisfied, whereas we view *all* formulas as constraints. While there are proposals to handle the negation [10] and disjunction [11] limitations, the third limitation is of a more fundamental nature.

The view of constraints and constraint simplification that we have suggested above lends itself very well to a formalization and implementation as a term rewriting system [12], where the derivations above are instances of some rewrite rules of the form $A \to B$. Indeed, at first glance it may look as if most functional programming languages (e.g., Haskell or ML) or functional logic programming languages (e.g., Curry or Escher) can be used for this purpose, since they allow us to write equations, which are usually interpreted as left-to-right rewrite rules. However, two limitations of these languages prohibit this: their *constructor discipline* and their inability to handle *associative* or *commutative* functions.

Constructors are a subset of the function symbols that cannot be further reduced. Most functional programming languages restrict the equations one can write such that the left-hand sides must be of the form $f(t_1, \ldots, t_n)$, where f is a nonconstructor function symbol and t_i are terms containing only constructor symbols or variables. This constructor discipline allows one to define computable functions and to execute them in an efficient way, but it limits us when we want to define a more general rewrite relation, such as our constraint simplification relation. For example, we will need rewrite rules/equations such as

$$A \wedge \texttt{true} = A$$

$$A \vee \texttt{true} = \texttt{true}$$

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

where A, B, and C are boolean variables. From the first two equations, it is clear that neither $\wedge$ nor $\vee$ are constructors, as they can be eliminated (in fact, `true` and `false` are the only constructors of the boolean algebra). Thus, the third equation does not adhere to the constructor discipline, since it contains nested nonconstructor function symbols.

To show the deficiency of functional programming languages with regard to *associative* and *commutative* functions, consider what would happen if we try to reduce a term $\texttt{true} \vee A$ using the rules above. We want to use the second rule, but it does not match, since the terms are in the wrong order. However, $\vee$ is commutative, and we could encode this using another equation,

$$A \wedge B = B \wedge A$$

but adding this equation will make the term rewriting system nonterminating, since any term rewritten by this equation still matches the equation, and can thus be rewritten again indefinitely. A similar argument applies to the associativity of conjunction and other operators. One solution is *AC matching*, where we can declare that an operator is associative and/or commutative, instead of using an equation. AC matching means matching modulo these properties, and can be done in a built-in, terminating way. Thus, our solution is to use a system which does not demand a constructor discipline and which supports AC matching, namely, Maude [8].

3 EQUATIONAL LOGIC, REWRITING LOGIC, AND MAUDE

Equational logic (EL) [13] is the subset of first-order logic with $=$ as the only predicate symbol, and equations as the only formulas (i.e., there are no logical connectives). The rules of deduction of EL are *reflexivity*, *congruence*, *transitivity*, and *symmetry*. Despite being a very small subset of first-order logic, equational logic can be used to define any computable function. Furthermore, EL can be used as a programming language, by treating equations as left-to-right rewrite rules (i.e., ignoring the symmetry rule), and using *reduction* as the operational semantics. Viewed as rewrite systems, theories in EL are expected to be terminating and confluent. This means that the order in which redexes and rewrite rules are chosen does not matter; we will reach the same result regardless, and in a finite number of steps. Sometimes *conditional* equations are allowed, which means that Horn clauses can be used in addition to plain equations, as in the power mask definition in Figure 1.

Rewriting logic (RL) [14] is similar to EL on the surface, in that it allows for (possibly conditional) rewrite rules. However, the rules are not semantically equations, and the rule systems are not expected to be confluent. Therefore, reduction cannot be used as the operational semantics, since different choices of redexes and rules can lead to different results. Instead, the rewrite rules are interpreted as nondeterministic state transitions, and the operational mechanism is *search* in the state space. Analogously to EL, RL can also support conditional rewrite rules.

Maude [8] is a multiparadigm executable specification language encompassing both EL and RL. The Maude interpreter is very efficient, allowing prototyping of quite complex test cases. Maude also provides efficient built-in search and model checking capabilities. Maude is reflective [15], providing a meta-level module that reflects both the syntax and semantics of Maude. Using reflection, the user can program special-purpose execution and search strategies, module transformations, analyses, and user interfaces. Maude sources, executables for several platforms, the manual, a primer, cases studies, and papers are available from the Maude website `http://maude.cs.uiuc.edu`.

We briefly summarize the syntax of Maude that is used in this paper. Maude has a module system, with

– *functional* modules, specifying equational theories, which are declared with the syntax `fmod`... `endfm`
– *system* modules, which are rewrite theories specifying systems of state transitions; they are declared with the syntax `mod`... `endm`

These modules have an *initial model semantics* [16]. Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* and *operators*. Operators are introduced with the `op` keyword followed by the operator name, the argument and result sorts. An operator may have mixfix syntax, with the name containing '`_`'s marking the argument positions. A binary operator may be declared with equational *attributes*, such as `assoc`, `comm`, and `id: <identity element>` stating, for example, that the operator is associative, commutative, and specifying an identity element for the operation. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. Equational axioms are introduced with the keyword `eq` (or `ceq` for conditional equations) followed by the two terms being declared equal separated by the equality sign `=`. Rewrite rules are introduced with the keyword `rl` (or `crl` for conditional rules) followed by an optional rule label, and terms corresponding to the premises and conclusion of the rule separated by the rewrite sign `=>`. Variables appearing in axioms and rules (and commands), may be declared globally using the keyword `var` or `vars`, or "inline" using the variable name and its sort separated by a colon; for example, `n:Nat` is a variable named `n` of sort `Nat`. Rewrite rules are not allowed in functional modules.

Maude has a `reduce` command for equational reduction in functional modules, and a `search` command for breadth-first search in the state space of system modules. The search mechanism allows searching for the first answer, all answers, or only answers matching some goal term. The search mechanism encompasses the reduction mechanism, as equational reduction is performed before each application of rewrite rules.

3.1 Logic Programming in Maude

Maude also has a `narrow` command. Narrowing in Maude is similar in many ways to the `search` mechanism mentioned above. Like search, narrowing nondeterministically selects rewrite rules, generates choice points, and can return answers in the same ways. There are two differences: 1) new variables are allowed on the right-hand side of rewrite rules, and 2) when there are uninstantiated variables in a redex, unification is used instead of matching.

As mentioned in Section 2.1, the addition of narrowing to a functional language gives us the ability to subsume logic programming. We encode relations and logical connectives as boolean functions. The logical connectives are defined in a `BOOL` module in Maude's "prelude", which contains statements like

```
op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
op not_ : Bool -> Bool [prec 53] .
vars A B C : Bool .
eq true and A = A .
eq false and A = false .
eq A and A = A .
```

The `assoc` and `comm` attributes declare the associativity and commutativity of the operators. The `prec` attribute sets the precedence of an operator so that the mixfix notation is parsed correctly without a proliferation of parentheses. Note that `not` is simply a truth function that takes `true` to `false` and vice versa, and corresponds to classical negation. We do not have a notion of negation-as-failure since we deal with a language with classical first-order models.

User-defined predicates are encoded in a way similar to the connectives. For example, an n-ary predicate `P` is encoded as

```
op P : T1 ... Tn -> Bool .
```

where `Ti` are the sorts of the arguments of `P`.

Rules are encoded as rewrite rules. For example,

$$\forall x, y, z : uncle(x, y) \Leftarrow parent(x, z) \wedge brother(z, y)$$

is encoded in Maude as

```
vars x,y,z : Ind .
rl uncle(x,y) => father(x,z) and brother(z,y) .
```

Note the implicit existential variable `z` on the right-hand side. Normal rewriting logic does not support this, but narrowing allows us to handle this. We do not allow existential variables in a negative context (i.e., under an odd number of `nots`). This would correspond to universal quantification, which the system does not support.

Facts are rules without bodies. For example, *father*(*John*, *Bob*) is encoded as

```
rl father(John,Bob) => true .
```

Note the use of rewrite rules (`rl`) rather than equations (`eq`) for user-defined facts and rules. This is motivated by operational concerns; we want to be able to use narrowing on these facts and rules. The model-theoretic semantics of Maude rewrite rules as state transitions does not directly reflect our interpretation of the rules as implications in first-order logic. However, our use of Maude is sound with regard to the first-order model theory.

4 MAUDE AS A WEB REASONING TOOL

We now describe how Maude can be used as a Web reasoning tool. First we describe how to translate (parts of) OWL ontologies and SWRL rules into Maude, and how the system can be used in a way similar to other rule-based systems like Prolog. Then we describe the novel features where our system goes beyond other rule-based systems: user-defined or "built-in" operations, and domain-specific simplification rules. Our examples come from the spectrum policy domain as discussed above, but we believe that these features are generally useful for a wide variety of problems. Finally, we discuss some implementation details.

4.1 Encoding OWL and SWRL in Maude

To use Maude as the reasoning engine for spectrum policies, we need to translate our policies to Maude. The policies are written as SWRL rules, i.e., Horn clauses, and refer to OWL ontologies. We can also translate a significant portion of axioms from OWL ontologies into Horn logic [17]. Once we have all our statements in Horn clause form, it is straightforward to encode them in a very direct way in Maude, using the scheme described in Section 3.1. Some specifics of the encoding follow.

First, we note that OWL does not have types or sorts in the sense of Maude or other programming languages. However, Maude operators need to be declared with sorted signatures. We therefore introduce one sort of OWL individuals, `Ind`, and one sort of OWL data values, `Data`. These are the two semantic domains of OWL models. We translate OWL individuals, classes, and properties as follows (`***` denotes a comment in Maude):

```
op Radio1 : -> Ind .              *** an individual
op Radio : Ind -> Bool .          *** a class
op detector : Ind Ind -> Bool .   *** an object property
op frequency : Ind Data -> Bool . *** a datatype property
```

The signatures here are perhaps best understood as follows: An individual is an operator with no argument that returns itself (a constant). A class is an operator that takes an individual as an argument, and returns `true` or `false`, depending on whether or not the individual is a member of the class. A property is an operator that takes a subject and an object as arguments, and returns `true` or `false` depending on whether or not that subject has that object as a property value for the property in question.

We treat *functional* properties separately, translating them as

```
op role : Ind -> Ind           *** a functional object property
op power : Ind -> Data         *** a functional datatype property
```

where the operator works as a function—it takes the subject as an argument and returns the object. This encoding makes reasoning more efficient in Maude, since it is essentially a functional language.

Facts are translated into Maude rewrite rules as in the following examples:

```
rl Radio(Radio1) => true .                  *** class-instance fact
rl detector(Radio1,Detector1) => true .     *** property-value fact
rl role(Radio1) => BaseStation .    *** functional property-value fact
```

Finally, *rules* are translated exactly as shown in Section 3.1.

4.2 Operations

Recall from Section 1 that one of our motivations for using a reasoning technology that supports functions and equations was that we needed to define certain operations on temporal and geospatial entities. These operations should be deterministic, directional, and terminating, and should be evaluated rather than "reasoned" about. In other words, they should be defined as functions. Whereas we need to use rewrite rules (`rl`) for user-defined axioms in order to support more general reasoning using narrowing, we use equations (`eq`) for these defined operations.

Note that SWRL has a number of built-in operations of this functional flavor, e.g., for math, time, comparisons, and strings.[1] We propose that additional operations should be treated in the same way as SWRL built-ins, i.e., used in "built-in atoms" in SWRL rules.

These kinds of operations cannot be defined in OWL or SWRL. The Protégé ontology development environment [18] supports adding implementations of new built-ins in Java [19]. However, using a functional framework, as we propose, provides many well-known advantages of declarative languages, such as clear semantics, intuitive and transparent specifications, and better integration with reasoning. In fact, one could give the SWRL builtins a formal semantics (which they currently lack) by defining them using Maude equations.

Ideally, we would like to have an overall semantic framework that encompasses both OWL (and SWRL) and the equational specification of these "built-ins." This is an area of future research. It is also an open question whether all the operations should be considered "built in" to the reasoner, or whether users could also define their own operations. The latter option would provide

[1] See `http://www.w3.org/Submission/SWRL/`

great flexibility, as it allows the same reasoner to be applied to new domains, where any new operations required can be defined by users themselves, rather than having to modify the reasoner with new "built-ins". There are, however, some practical details to resolve, e.g., regarding which syntax to use for these user-defined operations.

Currently, our reasoner specification includes many of the SWRL built-ins, and certain geospatial operations. Because of space restrictions, we cannot show sample implementations of operations here, but our reasoner specification can be downloaded at `http://xg.csl.sri.com/`. See, in particular, the `TIME` and `GEO` modules.

4.3 Simplification Rules

The second motivation for using a reasoning technology that supports equations is our need to write custom constraint simplification rules, as discussed in Section 2.2. In fact, this is where Maude really differentiates itself from most other systems, because it can do reduction, rewriting, and narrowing *modulo associativity and commutativity (AC)*. Handling associativity and commutativity in a built-in way is critical for encoding logical rewriting systems such as the one in question here, because lacking this capability, we would be forced to add AC axioms, which would make the system nonterminating, as discussed previously.

While Maude's prelude already includes certain constraint simplification rules, such as the trivial ones we showed in Section 3.1, we needed to go further and implement our custom rules, for two reasons: 1) the combination of ordering constraints (e.g., $<$, $\leq$) and boolean constraints (e.g., $\wedge$, $\vee$) introduces many opportunities for simplification, and 2) the desire to get answers in a certain form.

The latter point could use some additional clarification. All the simplification rules transform a constraint into another, equivalent, constraint. The purpose of all rules is to move the constraint to a "simpler" form, until we have reached the "simplest" possible form. However, it is not always obvious what the simplest form is. In the radio policy domain, we have some guidance from the domain. The policy engine should return a constraint that can be recognized by the radio as a set of transmission opportunities, where each opportunity is straightforward to interpret. We can achieve this by using *disjunctive normal form* (DNF) as the target of our constraint reduction. A DNF formula is a disjunction of conjunctions, i.e., it has the form

$$(A \wedge B \wedge C) \vee (D \wedge E) \vee (F \wedge G) \vee ...$$

We can think of each of the conjuncts $(A \wedge B \wedge C)$, $(D \wedge E)$, or $(F \wedge G)$ as an opportunity, because it is enough for the radio to satisfy *one* of the disjunctions in order to satisfy the entire constraint. For example, if the radio provides the facts D and E, then the formula above reduces to

$$(A \wedge B \wedge C) \vee \texttt{True} \vee (F \wedge G) \vee ... \rightarrow \texttt{True}$$

using the rule

$\texttt{True} \vee P \rightarrow \texttt{True}$

Furthermore, for the chosen opportunity, all the constraints have to be satisfied.

Simplifying to DNF, however, is not enough. For example, the following constraint is in DNF

$(\texttt{freq} < 500 \wedge \texttt{power} < 10) \vee (\texttt{freq} > 400)$

but it can be simplified further to

$\texttt{power} < 10 \vee \texttt{freq} > 400$

This constraint is still in DNF and equivalent to, but simpler in some sense than, the first form. Our simplification rules take care of such cases.

4.4 Implementation

Above, we have discussed the *principles* of our engine. Here, we describe the main components of our implementation. The engine needed to be implemented in C/C++ in order to run on resource-constrained radios. Maude is implemented in C++ and highly optimized, so this posed no particular problem. The policies are written in OWL, SWRL, and SWRL FOL, using the XML presentation syntax. There was no existing C/C++ parser for OWL, and no software support for SWRL FOL. Thus, we implemented our own parser/writer for OWL+SWRL+SWRL FOL using the XML presentation syntax.

The Maude reasoner back end consists of several distinct parts, as shown in Figure 2:

– A translator from the parser's representation of OWL/SWRL to Maude, using the encoding in Section 4.1.
– The Maude reasoner specification, containing Maude equations representing the simplification rules and built-in operations.
– The Maude engine itself. Normally, Maude runs as an interactive console-based tool, but we used an experimental programming API to Maude.
– A translator from Maude results back to OWL/SWRL.

On a more detailed level, the Maude specification consists of a number of Maude modules: `TRM`, `TIME`, `GEO`, `POLICY`, `SIMP`, `DNF`, `CNF`, and `REASONER`. The `TRM` module contains basic definitions of the boolean algebra, arithmetic, and ordering constraints. The `TIME` and `GEO` modules contain built-in functions for temporal and geospatial reasoning. The `POLICY` module contains the translation of all the policies and facts that are currently loaded. When the reasoner first starts, the `POLICY` module is empty. Whenever new policies or facts are loaded, this module is updated. `SIMP`, `DNF`, and `CNF` contain different parts of the proof system. `SIMP` does a number of simplifications such as eliminating negation (as far as possible) and implication. `CNF` converts to conjunctive normal form, and does some simplifications that can be done only in this form. `DNF` converts to disjunctive normal form, and does some simplifications that can be done only in

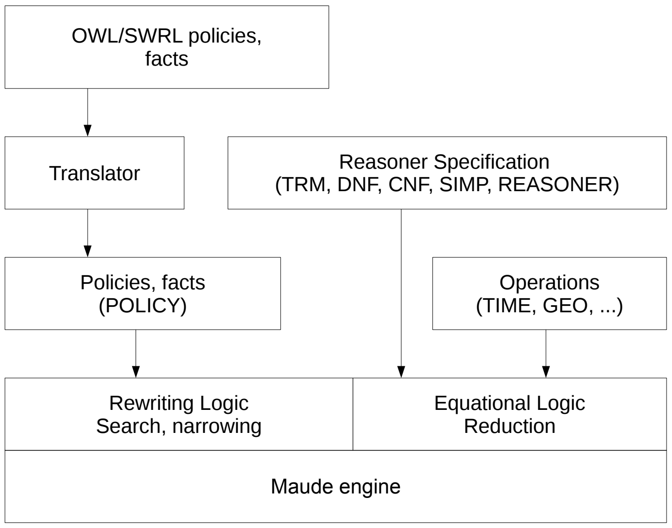

OWL/SWRL policies, facts
Translator
Reasoner Specification (TRM, DNF, CNF, SIMP, REASONER)
Policies, facts (POLICY)
Operations (TIME, GEO, ...)
Rewriting Logic Search, narrowing
Equational Logic Reduction
Maude engine

Fig. 2. Components of the Maude-based reasoner.

this form. If these three modules were combined into one, the reasoning would never terminate. For example, it could transform between `CNF` and `DNF` back and forth indefinitely. The `REASONER` module controls the ordered execution of the reasoning modules by using the Maude meta level. First, narrowing is done in the `POLICY` module, then reduction in `SIMP`, `CNF`, and `DNF`, in that order. The narrowing step looks for *all* answers. To exploit simplification opportunities between different answers, the answers are `or`'d together before simplification, again using meta level Maude code.

5 CONCLUSIONS

Dynamic spectrum access offers an interesting application area for Semantic Web technologies. OWL allows us to define concepts related to the radio domain, and spectrum access can be controlled by policies written as SWRL rules. In addition to rules and ontologies, we needed to define operations (e.g., temporal and geospatial) using equational specifications. Furthermore, we wanted the reasoner to perform sophisticated *constraint simplification*. Any unresolved constraints can be used by a cognitive radio to plan and reason about its spectrum usage.

No existing reasoner supported all these features, but we found that Rewriting Logic provided a promising reasoning mechanism. We built our reasoner on top of Maude, an Equational Logic and Rewriting Logic system developed at SRI. Maude was extended with *narrowing*, which allows it to achieve behavior similar to that of a logic programming system, i.e., goal-oriented reasoning with rules. At the same time, the equational part of Maude is ideal for defining a constraint simplification system, as well as for defining operations. Thus, we were able to include all our desired features in one system. The resulting system subsumes both equational logic and logic programming. It can also easily be extended or modified in two ways. First, because the constraint simplification part is specified in the Maude language, as opposed to hard coded into the reasoning engine, it can be modified to better suit different needs. Second, new operations can be added as additional equational specifications.

The policy domain motivated our work, but the reasoner is not limited to this domain, since it operates on any OWL ontologies and SWRL rules. In particular, our system will be useful for problems that need sophisticated constraint processing in addition to rule-based reasoning, or where new operations need to be added. The implementation is also efficient enough to run on resource-constrained embedded systems such as software-defined radios.

ACKNOWLEDGMENTS

We thank Steven Eker (SRI) for maintaining the Maude system and responding to our feature requests, including the entire narrowing functionality. This research was supported by DARPA's neXt Generation (XG) Communications Program under Contract Number FA8750-05-C-0230.

REFERENCES

1. Wilkins, D.E., Denker, G., Stehr, M.O., Elenius, D., Senanayake, R., Talcott, C.: Policy-based cognitive radios. IEEE Wireless Communications **14** (2007) 41–46
2. Hanus, M.: The integration of functions into logic programming: From theory to practice. J. Log. Program. **19/20** (1994) 583–628
3. Hanus, M.: Multi-paradigm declarative languages. In Dahl, V., Niemelä, I., eds.: ICLP. Volume 4670 of Lecture Notes in Computer Science., Springer (2007) 45–75
4. Ait-Kaci, H., Lincoln, P., Nasr, R.: Le fun: Logic, equations, and functions. In: IEEE Symposium on Logic Programming. (1987)
5. Casas, A., Cabeza, D., Hermenegildo, M.V.: A syntactic approach to combining functional notation, lazy evaluation, and higher-order in LP systems. In Hagiya, M., Wadler, P., eds.: FLOPS. Volume 3945 of Lecture Notes in Computer Science., Springer (2006) 146–162
6. Naish, L.: Adding equations to NU-Prolog. In: Proc. Third International Symposium on Programming Language Implementation and Logic Programming, Passau, Germany, Springer-Verlag Lecture notes in computer science number 528 (1991) 15–26
7. Hanus, M., Kuchen, H., Moreno-Navarro, J.: Curry: A truly functional logic language. In: Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming. (1995)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: All about Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Volume 4350 of Lecture Notes in Computer Science. Springer (2007)
9. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. Journal of Logic Programming **19/20** (1994) 503–581
10. Stuckey, P.J.: Negation and constraint logic programming. Information and Computation **118** (1995) 12–33

11. Backer, B.D., Beringer, H.: A CLP language handling disjunctions of linear constraints. In: ICLP. (1993) 550–563
12. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, New York, NY, USA (1998)
13. O'Donnell, M.J.: Equational logic as a programming language. Massachusetts Institute of Technology, Cambridge, MA, USA (1985)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96** (1992) 73–155
15. Clavel, M., Meseguer, J.: Reflection and strategies in rewriting logic. In: Rewriting Logic Workshop 96. Number 4 in Electronic Notes in Theoretical Computer Science, Elsevier (1996)
`http://www.elsevier.nl/locate/entcs/volume4.html`.
16. Wirsing, M.: Algebraic specification. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Volume B. North-Holland (1990) 675–788
17. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: Proc. Twelfth International World Wide Web Conference (WWW 2003), ACM (2003) 48–57
18. Knublauch, H., Fergerson, R., Noy, N., Musen, M.: The Protégé OWL plugin: An open developoment environment for Semantic Web applications. In McIlraith, S., Plexousakis, D., van Harmelen, F., eds.: Proc. 3rd Intern. Semantic Web Conference (ISWC 2004), Hiroshima, Japan, November 2004, Springer (2004) 229–243 LNCS 3298.
19. O'Connor, M., Das, A.: A mechanism to define and execute SWRL built-ins in Protégé-OWL (2006)