

Towards Integrated Specification and Analysis of Machine-Readable Policies Using Maude¹

Rukman Senanayake, Grit Denker, and *Jon Pearce

SRI International, Menlo Park, CA 94025, *San Jose State University, CA 95112
rukman@cs.sri.com, Grit.Denker@sri.com, *pearce@cs.sjsu.edu

1. Introduction and Objectives

A policy is defined as “a plan or course of action, as of a government, political party, or business, intended to influence and determine decisions, actions, and other matters.” The behavior of many computerized systems and applications is determined by policies. For such systems it is useful not only to specify policies in a machine-readable way, as provided by semantic markup languages such as OWL² and SWRL³, but also to analyze whether a policy meets the intention of the policy designer. The latter can be achieved by translating policies into an existing specification and analysis framework and providing check lists, such as reachability of certain system states while complying with the policy or consistency checks among sets of policies, to increase the trust of the policy designer that the formalized policy captures her intention.

We propose to investigate the use of the rule-based framework Maude⁴ for the analysis of policies. The advantage of existing rule-based formal frameworks is that they usually already support a variety of analysis tasks that can be adapted to the task at hand. We believe that a rule-based system is adequate since many policies satisfy the following two characteristics: 1. Often natural language policy uses “*If-Then-Else*” statements and, thus, can be naturally formalized as rules. For example, a return policy for an online purchase service may state “If an item is returned within 30 days, then full refund is given.” 2. Often policies define constraints on the process or behavior of a system that implements the policy.

For example, the policy “First it is determined whether an item is returnable, next the exact refund amount is calculated” defines constraints in the sense that a certain sequence of state transitions or actions must be followed. Thus many policies can be formalized as a collection of rules and often policies abstractly define the behavior of a system in terms of a state transition system where rules govern how a state changes. This is also the case in one of our projects, the DARPA XG project⁵, in which policies define the behavior of wireless radios. In the XG project policies are defined using OWL extended by a rule mechanism similar to SWRL.

¹ Supported by the Defense Advanced Research Projects Agency through Air Force Research Laboratory under Contracts FA8750-05-C-0230 and F30602-00-C-0168.

² Web Ontology Language: <http://www.w3.org/2001/sw/WebOnt/>

³ A Semantic Web Rule Language Combining OWL and RuleML: <http://www.daml.org/2003/11/swrl/>

⁴ The Maude System: <http://maude.csl.sri.com>

⁵ DARPA Next Generation (XG) project: <http://www.darpa.mil/ato/programs/xg/>

Thus, in this paper we investigate by means of example the adequacy of Maude to express OWL/SWRL policies and use the built-in analysis techniques for Maude. As shown in this paper, our approach seems feasible, and future work must investigate the existence of a formal translation from OWL/SWRL into Maude to show that inference in OWL and SWRL can be reduced to inference in Maude.

In the remainder of this paper we will introduce a small example that we will use for illustration purposes (Section 2), briefly overview the Maude framework (Section 3), apply the framework to the example (Section 4), illustrate the translation of SWRL policies to Maude (Section 5), explain the various analysis tests that can be executed in Maude on the policy (Section 6), and give a brief overview of related work and close with a brief summary in Section 7.

2. Policy Specification and Semantics

2.1. A Policy Example

We selected a real-world policy as our primary example, namely, the return policy of Amazon. Amazon uses a ‘wizard-like’ approach to guide a user through the return process for an item sold to a customer. The policy as a whole can be described by a collection of rules and a set of states through which the process transitions. The rules define transitions between states, and the user needs to provide certain information, such as “whether the item was opened or not” to determine the initial state of the system. Table 1 summarizes some of the rules applicable to Amazon’s return policy.

Rule	Description
1	Partial refund if item is returned after 30 days
2	Item is nonreturnable if stated in Product Description Page (PDP)
3	Partial refund if item is a book and has signs of usage
4	If there was a shipping error, Amazon pays return cost

Table 1. Example Rules of Amazon’s Return Policy.

Translating the Amazon return policy into a state transition system, we get seven states and transitions as depicted in Figure 1.

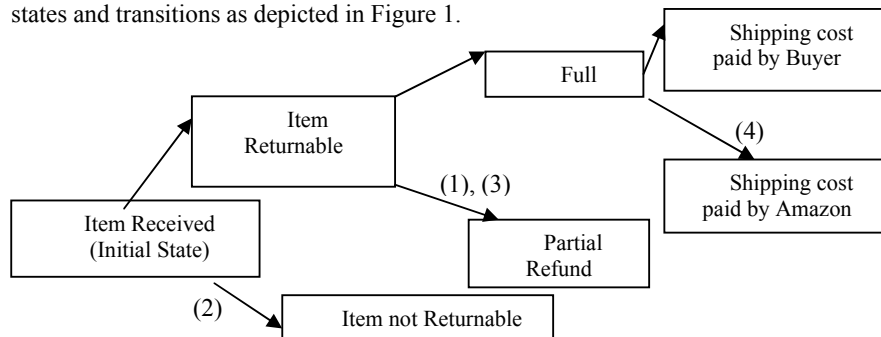


Fig. 1. State Transition System for Amazon’s Return Policy. Labels on arrows refer to rules that enable state transition. Unlabeled arrows are default transitions.

2.2. Policy Specification

The return policy was specified using the OWL and SWRL semantic markup languages. OWL was used to define the attributes, classes, and instances that collectively define a state space necessary for the return policy, and SWRL was used to define the rules of the policy.

We used the Protégé Ontology editor to encode the policies, as defined in Table 2. The class taxonomy contains classes such as ‘Manual’ which is a subclass of ‘Book’ which in turn is a subclass of ‘Item’. The class ‘Book’ has datatype properties ‘isUsed’ and so on. The rule base for the policy was specified using SWRL.

Rule	SWRL Encoding of the Rule
2	$\text{Item}(?x) \wedge \text{hasPDP}(?x, \text{"Can not be returned"}) \rightarrow \text{isReturnable}(?x, \text{false})$
3	$\text{Book}(?x) \wedge \text{isUsed}(?x, \text{true}) \rightarrow \text{hasRefundRatio}(?x, 2)$
4	$\text{Item}(?x) \wedge \text{hasCondition}(?x, \text{"Shipping Error"}) \wedge \text{hasSeller}(?x, ?y) \rightarrow \text{hasShippingCostPayee}(?x, ?y)$

Table 2. Policy Rules Defined Using SWRL.

3. The Maude System

Maude [CEL+96] is a multiparadigm executable specification language based on rewriting logic [Mes92, Mes00]. Maude sources, executables for several platforms, the manual, a primer, cases studies, and papers are available from the Maude Web site at <http://maude.csl.sri.com>.

We briefly summarize the syntax of Maude that is used in our case study. Maude specifies systems as collections of modules. System modules are rewrite theories specifying concurrent systems; they are declared with the syntax *mod...endm*. Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported modules can be added. One can also declare *sorts*, *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 (underscores) marking the argument positions.

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 *cr1* (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 or rules (and commands) may be declared globally using 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.

We model the ontology described by OWL and SWRL specifications in Maude by using Maude's notation and conventions for concurrent objects. A snapshot of a system's state can be thought of as a collection of objects (a configuration). The multiset union operator for configurations is denoted with empty syntax (juxtaposition) and (by definition of multiset) is associative and commutative.

An object has the form $\langle O : C \mid att-1, \dots, att-n \rangle$ where O is an object identifier (sort Oid), C is a class identifier (sort Cid), and $att-1 \dots att-n$ are attributes (sort $Attribute$). This notation is part of a module called `CONFIGURATION`, which is part of the standard Maude library.

A typical system configuration will consist of several objects. The dynamic behavior of a concurrent object system is then axiomatized by specifying rewrite rules for each class that determine, for example, how objects can evolve from one state to another through rule application.

4. Example Policy in Maude

An abbreviated version of the generated Maude module is given in Table 3.

(1)	<pre> mod ATTRIBUTES is protecting STRING . protecting BOOL . op hasOriginalCondition:_ : String - > Attribute . op isReturnable:_ : Bool -> Attribute . op hasCondition:_ : String -> Attribute . endm </pre>	The module <code>ATTRIBUTES</code> defines the collection of datatype properties in the OWL ontology
(2)	<pre> mod OWLONTOLOGY is inc CONFIGURATION . inc ATTRIBUTES . </pre>	The module <code>OWLONTOLOGY</code> defines classes and rules. The 'inc' statements are somewhat analogous to importing of URIs that define namespaces.
(3)	<pre> sort Book . op Book : -> Book . sort Item . op Item : -> Item . sort swrlAtom . op swrlAtom : -> swrlAtom . </pre>	Each class in the OWL ontology is defined as a sort in the Maude module. This includes the defining of a constructor with the same name as the class name using the 'op' keyword.
(4)	<pre> subsort String < Oid . subsort String < Cid . subsort swrlAtom < Cid . subsort Item < Cid . subsort Book < Item . subsort Manual < Book . </pre>	The subsort definitions reflect the correct class taxonomy within Maude. All class names and object names are strings. All other classes are subclasses of the top-level class

	subsort Novell < Book .	Cid or one of its subclasses.
(5)	var atts : AttributeSet . var oid : Oid . var I1 : Int . var book : Book . var item : Item .	All variables used in the rewrite rules (see (6)) are defined.
(6)	rl [rule2] : < oid : item hasPDP: "Can not be returned", isReturnable: B1, atts > => < oid : item hasPDP: "Can not be returned", isReturnable: false, atts > .	Rewrite rule definitions consist of keyword <i>'rl'</i> (or <i>ctrl</i>), an optional name ([rule1]), the start state, and the terminating state. The example rule implements Rule 1.
(7)	op startState : -> Configuration . eq startState = < "Les Miserables" : Book isReturnable: true, hasCondition: "Used", hasRefundRatio: 0, hasPDP: "All time favorite" > .	The initial state of the ontology is defined by the operator <i>'startState'</i> . It contains instances of classes (one or more) together with their attribute values.

Table 3. Partial Maude Module for Amazon's Return Policy.

5. Translating Semantic Markup Specifications to Maude

5.1. Translating OWL Specifications to Maude

Translating the Class Taxonomy

Because of the object-oriented nature of the OWL ontology, we chose the object-oriented specification style of Maude to stay as close as possible in our translation to the original OWL/SWRL policy. The OWL-to-Maude translation process preserves the class hierarchy and this is done using the 'subsort' definitions (see section (4) of Table 3). Using the *'subsort'* keyword of Maude is semantically comparable to the OWL class taxonomy.

Some additional definitions are incorporated because of requirements within Maude, namely, the subsort definitions *'subsort String < Cid'* and *'subsort String < Oid'*. This is to support using strings as class and object names. Another important definition is *'subsort Item < Cid'*, which is analogous to the fact that every OWL named class is a subclass of the OWL:thing class (since the sort 'Cid' corresponds to the top-level class 'thing' in OWL).

Translating OWL Individuals

The collection of OWL individuals defined in the ontology is equivalent to the start state of the Maude system module. The collection of OWL individuals is retrieved and converted into a list of object definitions based on the attributes and sorts defined in the Maude model (see section (7) of Table 3). This collection of objects is the start state for the Maude rewrite logic, and is an instance of the Configuration sort. Since the Configuration sort is a multiset of objects and messages this preserves the object-oriented semantics of the OWL ontology.

5.2. Translating SWRL Specifications to Maude

The SWRL rule base is translated into a Maude rule collection. A Maude rule has a start-state definition and an end-state definition. These definitions are simply

collections of objects with certain attributes, and a rule as a whole depicts how the values of the attributes change when the rule is applied.

For example, Rule 4 of Table 2 is translated into a Maude rule by the following technique:

- i. The SWRL class membership predicate is identified for the relevant class of objects that the rule affects. In Rule 4 this is equivalent to ‘Item(?x)’ and the Maude representation of this would be ‘oid : item’ where ‘oid’ is a variable (typically of type String) used as an object identifier.
- ii. The collection of attributes in the antecedent and the consequent of the SWRL rule are retrieved and translated into equivalent attribute definitions of Maude. For example, ‘hasSeller(?x,?y)’ is written in Maude as ‘hasSeller: Y’ where ‘Y’ is a variable of a suitable type (typically String).
- iii. The Maude rule is generated by converting the antecedent of the SWRL rule to the start state and the consequent to the end state.

The resulting Maude rule follows:

Rule 4	<pre> rl [Rule-5] < oid : item hasCondition: "Shipping Error", hasSeller: S1, hasShippingCostPayee: S2, atts > => < oid : item hasCondition: "Shipping Error", hasSeller: S1, hasShippingCostPayee: S1, atts > </pre>
--------	--

The rule says that an item with condition “Shipping Error” can be rewritten to a state where the payee for the shipping cost is set to the seller (independent of the values in the hasShippingCostPayee before). In the above Maude rule, ‘atts’ is a variable representing a set of attributes a class may have. This is a simple technique used in Maude to avoid listing the entire set of attributes for every state, since ‘atts’ represents any subset of the attribute collection of a class, so that only the attributes affected by the current rule need to be listed.

6. Analyzing Policies Using Maude

The dynamic behavior of a concurrent object system is axiomatized by specifying rewrite rules for each type of state transition that can take place in the system. For example, a rewrite rule may define the transition into a state where an item is not returnable.

Maude [CEL+96] is not just a language. It also has an interpreter, making it an executable, multiparadigm specification language. The Maude interpreter is very efficient, allowing prototyping of quite complex test cases. In addition to modeling capabilities, Maude provides efficient built-in search and model checking capabilities. Maude is reflective [Cla98], providing a meta-level module that reflects both its syntax and semantics. Using reflection, the user can program special-purpose execution and search strategies, module transformations, analyses, and user interfaces. Maude has several built-in analysis capabilities. For example, the user can define an initial system state and use Maude to apply rules to find a final state. This is called “default execution”, since one possible sequence of rules is applied. Obviously, a final state is found only for rules that describe finite-state systems or, for an infinite-state

system, when the number of rules applied is restricted by the user. Besides default execution, Maude supports state search in at least two ways.

(1) A user can define an initial system state (which can be represented in a symbolic way) and use a search strategy in Maude to determine all possible combinations of rule applications and the resulting states. If this analysis results in more than one final state for a given set of rules, it means that the system specified by the rules is not deterministic and that the order in which policy rules are applied matters. This is usually not the intent of policy designers, and in particular in a situation where several policies are combined, it is important to know whether there are such “side effects” due to rule ordering. This test can be generalized to test policy consistency. We define a strategy that finds sequences of rule applications that result in contradictory statements (e.g., `refundRatio=partial` and `refundRatio=full`).

(2) Maude also supports a search function for Linear Temporal Logic (LTL) formulas. The user can define a state for which he would like to know whether the state is reachable from a given initial state with the set of rules. Maude will find a path, if it exists. This type of analysis is useful to check the adequacy of policies with regard to the user’s intuition. For example, a user might expect for a given situation a certain policy decision or outcome by applying the rules. She can check this by running the LTL model checker that is part of Maude. If the desired and expected outcome is not achieved, then the rule base does not correctly reflect the intention of the user. More generally, the user might want to test what are all possible reachable states (if a finite state system is described by the set of rules). In addition to analysis, Maude can serve as an execution engine for rules or services since any Maude specification can be efficiently executed.

7. Related Work and Concluding Remarks

Semantic Web Service (SWS) approaches (e.g., OWL-S or WSML) describe services in semantically meaningful and machine-readable ways, and thus overcome the issues of informal descriptions or natural language policies that describe a Web service’s behavior. For example, the process specification of a service may say that one cannot use the service without successfully completing a login phase or that a service will always go through an order confirmation phase before charging a credit card. Our approach could be used to formalize such process constraints on a high abstraction level, and define a “refinement” or “implementation” relationship between high-level policies and a service model. In this way, one could define a semantics for service models using rewriting logic. Other approaches to give semantics to OWL-S processes are [NM02] and [AHS02]. [NM02] gives semantics to some parts of the process model in terms of Petri Nets, and [AHS02] gives an operational semantics. A Maude formalization of the OWL-S process model in terms of rules describing the state transitions would provide both a denotational and an executable semantics, since Maude specifications are efficiently executable [CDE+03] and possess a well-founded semantics based on rewriting logic [Mes92].

Recent work on machine-readable policy and rule languages (e.g., SWRL, RuleML, SWSL, Rei [KFJ05] and KAoS [UBJ+03]) enables the formal definition of policies and rules. Some of the languages have tools for analysis of policies, such as checking whether a given action or instance matches a policy or whether policies are

consistent. Rei and KAoS make use of existing technology to achieve their reasoning capabilities. Rei uses prolog-style reasoning and KAoS uses existing Description Logic reasoning. In a manner somewhat similar to the Rei and KAoS approaches, we are interested in investigating how the rule-based Maude technology can be used to help with policy and service specification and analysis.

We designed and implemented a mapping from example policies expressed in SWRL to executable Maude specifications. We support the analysis of machine-readable policies in an integrated fashion from the policy editor, using Maude's built-in analysis capabilities. This encourages us to further investigate how the Maude environment and its theoretical underpinning (i.e., rewriting logic) could be of use as a framework for OWL/SWRL policies.

References

[AHS02] A. Ankolekar, F. Huch, and K. Sycara. Concurrent Execution Semantics for DAML-S with Subtypes, First Int. Semantic Web Conference (ISWC), Sardinia (Italy), June, 2002.

[CDE+03] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In Nieuwenhus (ed). *Rewriting Techniques and Applications*. Springer, LNCS 2706, pp 77-87, 2003.

[CEL+96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude, In J. Meseguer (ed), *Rewriting Logic and Its Applications*, First Int. Workshop, Asilomar, CA, pp 65-89, 1996.

[Cla98] M. Clavel. *Reflection in General Logics, Rewriting Logic, and Maude*. Ph.D. Dissertation, University of Navarre, 1998.

[KFJ05] L. Kagal, T. Finin, and A. Joshi. Rei: A Policy Specification Language. See <http://rei.umbc.edu/>.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency, *Theoretical Computer Science*. 96(1):73-155, 1992.

[Mes00] J. Meseguer. Rewriting Logic and Maude: A Wide-spectrum Semantic Framework for Object-based Distributed Systems. In S. Smith and T. Talcott (eds), *Formal Methods for Open Object-based Distributed Systems, FMOODS 2000*, pp 89-117. Kluwer, 2000.

[NM02] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services, Proc. of the Eleventh International World Wide Web Conference (WWW-11), Honolulu, May 2002.

[Protégé01] N. F. Noy, M. Sintek, S. Decker, M. Crubezy, R. W. Ferguson, and M. A. Musen. Creating Semantic Web Contents with Protege-2000, *IEEE Intelligent Systems* 16(2):60-71, 2001.

[UBJ+03] A. Uszok, J. Bradshaw, J. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement, Proc. IEEE Workshop on Policy, 2003.