The Specification of Agent Behavior by Ordinary People: A Case Study

Luke McDowell, Oren Etzioni, and Alon Halevy

University of Washington, Department of Computer Science and Engineering
Seattle, WA 98195 USA
{lucasm,etzioni,alon}@cs.washington.edu,
http://www.cs.washington.edu/research/semweb/email

Abstract. The development of intelligent agents is a key part of the Semantic Web vision, but how does an ordinary person tell an agent what to do? One approach to this problem is to use RDF *templates* that are authored once but then instantiated many times by ordinary users. This approach, however, raises a number of challenges. For instance, how can templates concisely represent a broad range of potential uses, yet ensure that each possible instantiation will function properly? And how does the agent explain its actions to the humans involved? This paper addresses these challenges in the context of a case study carried out on our fully-deployed system[1] for *semantic email agents*. We describe how high-level features of our template language enable the concise specification of flexible goals. In response to the first question, we show that it is possible to verify, in polynomial time, that a given template will always produce a valid instantiation. Second, we show how to automatically generate explanations for the agent's actions, and identify cases where explanations can be computed in polynomial time. These results both improve the usefulness of semantic email and suggest general issues and techniques that may be applicable in other Semantic Web systems.

1 Introduction

The vision of the Semantic Web has always encompassed not only the declarative representation of data but also the development of intelligent agents that can consume this data and act upon their owner's behalf. For instance, agents have been proposed to perform tasks like appointment scheduling [2], meeting coordination [26], and travel planning [22]. A significant difficulty with this vision, however, is the need to translate the real-world goals of untrained users into a formal specification suitable for agent execution [31, 30]. In short, how can an ordinary person tell an agent what to do?

One approach to this problem is to encapsulate classes of common behaviors into reusable *templates* (cf., *program schemas* [7, 10] and *generic procedures* [22]). Templates address the specification problem by allowing a domain-specific template to be *authored* once but then *instantiated* many times by untrained users. In addition, specifying such templates declaratively opens the door to automated reasoning with and

[1] See `http://www.cs.washington.edu/research/semweb/email` for a publicly accessible server (no installation required); source code is also available from the authors.

composition of templates. Furthermore, the resulting declarative specifications can be much more concise than with a procedural approach (see Table 1).

However, specifying agent behavior via templates presents a number of challenges:

- **Generality:** How can a template concisely represent a broad range of potential uses?
- **Safety:** Templates are written with a certain set of assumptions — how can we ensure that any (perhaps unexpected) instantiation of that template by a naive user will function properly (e.g., do no harm [32], generate no errors)?
- **Understandability:** When executing a template, how can an agent explain its actions to the humans (or other agents) that are involved?

This paper investigates these challenges via a case study that leverages our deployed system for *semantic email agents* (E-Agents).[2] E-Agents provide a good testbed for examining the agent specification problem because they offer the potential for managing complex goals and yet are intended to be used by a wide range of untrained people. For instance, consider the process of scheduling a meeting with numerous people subject to certain timing and participation constraints. E-Agents support the common task where an *originator* wants to ask a set of *participants* some questions, collect their responses, and ensure that the results satisfy some set of *constraints*. In order to satisfy these constraints, an E-Agent may utilize a number of *interventions* such as rejecting a participant's response or suggesting an alternative response.

Our contributions are as follows. First, we identify the three key challenges for template-based agents described above. We then examine specific solutions to each challenge in the context of our fully-deployed system for semantic email. For *generality*, we describe the essential features of our template language that enable authors to easily express complex constraints without compromising important computational properties. The sufficiency of these features is demonstrated by our implementation of a small but diverse set of E-Agents. For *safety*, we show how to verify, in polynomial time, that a given template will always produce a valid instantiation. Finally, for *understandability*, we examine how to automatically generate explanations of *why* a particular response could not be accepted and *what* responses would be more acceptable. We also identify suitable restrictions where such constraint-based explanations can be generated in polynomial time. These results both greatly increase the usefulness of semantic email as well as highlight important issues for a broad range of Semantic Web systems.

[2] Our prior work [19] referred to *semantic email processes*; in this work we call them E-Agents to be more consistent with standard agent terminology.

Table 1. Comparison of the size of an E-Agent specification in our original procedural prototype [9] (using Java/HTML) vs. in the declarative format described in this paper (using RDF). Overall, the declarative approach is about 80-90% more concise.

E-Agent name	**Procedural approach (number of lines)**	**Declarative approach (number of lines)**	**Size Reduction for Declarative**
Balanced Potluck	1680	170	90%
First-come, First-served	536	99	82%
Meeting Coordination	743	82	89%
Request Approval	1058	109	90%
Auction	503	98	81%

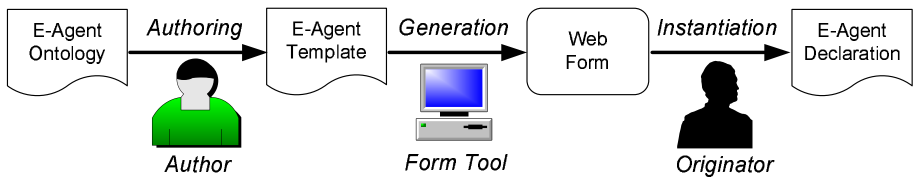

Fig. 1. The creation of a Semantic Email Agent (E-Agent). Initially, an "Author" *authors* an E-Agent template and this template is used to *generate* an associated web form. Later, this web form is used by the "Originator" to *instantiate* the template. Typically, a template is authored once and then instantiated many times.

The next section gives a brief overview of E-Agents while Section 3 discusses the salient features of our template language with an extended example. Sections 4 and 5 examine the problems of instantiation safety and explanation generation that were discussed above. Finally, Section 6 considers related work and Section 7 concludes with implications of our results for both email-based and non-email-based agents.

2 Overview of Semantic Email Agents

We illustrate E-Agents with the running example of a "balanced potluck, " which operates as follows. The E-Agent originator initially creates a message announcing a "potluck-style" event — in this case, the message asks each participant whether they are bringing an `Appetizer`, `Entree`, or `Dessert` (or `Not-Coming`). The originator also expresses a set of constraints for the potluck, e.g., that the difference in the number of appetizers, entrees, or desserts should be at most two (to ensure "balance").

Figure 1 demonstrates how a template is used to create a new E-Agent. Initially, someone who is assumed to have some knowledge of RDF and semantic email authors a new template using an editor (most likely by modifying an existing template). We call this person the E-Agent *author*. The template is written in RDF based on an ontology that describes the possible questions, constraints, and notifications for an E-Agent. For instance, the potluck template defines some general balance constraints, but has placeholders for *parameters* such as the participants' addresses, the specific choices to offer, and how much imbalance to permit. Associated with each template is a simple *web form* that describes each needed parameter; Section 4 describes a tool to automatically generate such forms. An untrained originator finds an appropriate web form from a public library and fills it out with values for each parameter, causing the corresponding template to be instantiated into an E-Agent *declaration*. The semantic email server executes the declaration directly, using appropriate algorithms to direct the E-Agent outcome via message rejections and suggestions.

In our implementation, E-Agents are executed by a central server. When invoked, this server sends initial messages to the participants that contain a human-readable announcement along with a corresponding RDF component. These messages also contain a simple text form that can be handled by a human with any mail client, or by another agent based on an associated RDQL query. Participants fill out the form and reply via mail directly to the server, rather than to the originator, and the originator receives status messages from the server when appropriate.

3 Concise and Tractable Representation of Templates

Our first challenge is to ensure that a template can concisely represent many possible uses while still ensuring the tractability of E-Agent reasoning (e.g., to check the acceptability of a participant's response). This section first describes our template language via an extended example and then discusses how the language meets this challenge.

3.1 Template Example

An E-Agent *template* is a (parameterized) RDF document that specifies

- a set of **participants**,
- **questions** to ask the participants,
- **constraints** to enforce on the participants' responses, and
- **notifications** to send to the originator and/or participants at appropriate times.

We illustrate the latter three parts below with a balanced potluck template. In these declarations (shown in N3 format), variables in bold such as **$Choices$** are parameters provided by the originator; other variables such as `$x$` are computed during execution.

Questions: the set of questions to ask each participant. For instance, a potluck E-Agent might ask everyone for the food items and number of guests that they are bringing:

```
[a               :StringQuestion;
 :name           "Bring";
 :enumeration    "$Choices$"; ]

[a               :IntegerQuestion;
 :guard          "$AskForNumGuests$";
 :name           "NumGuests";
 :minInclusive   "0"; ]
```

The `enumeration` and `minInclusive` properties constrain the legal responses to these questions. In addition, the latter question is "guarded" so that it applies only if the parameter `AskForNumGuests` is true. Because a question defines data that may be accessed in multiple other locations in the template, it is important to be able to reason about whether its guard might evaluate to false (see Section 4). Finally, each question item also specifies a RDQL query (not shown) that defines the semantic meaning of the requested information and is used to map the participant's textual response to RDF [19].

Constraints: the originator's goals for the E-Agent's outcome. Each goal may be expressed either as a constraint that must be satisfied at every point in time (a `MustConstraint`) or as a constraint that should, if possible, be *ultimately satisfied* by the final outcome (a `PossiblyConstraint`). Section 5 defines the behavior of these constraints more precisely. Our simple example uses a quantified `MustConstraint` to require balance among the counts of the different choices:

```
[a            :MustConstraint;
 :forAll      ([:name   "x";    :range "$Choices$-$OptOut$"]
               [:name   "y";    :range "$Choices$-$OptOut$"]);
 :suchThat    "$x$ != $y$";
 :enforce     "abs($Bring.{$x$}.count()$ - $Bring.{$y$}.count()$)
                 <= $MaxImbalance$";
 :message     "Sorry, we can't accept a $Bring.last()$ now."; ]
```

The constraint applies to every possible combination (`x,y`) from the set `(Choices - OptOut)`; `OptOut` is for choices such as "Not Coming" that should be excluded from the constraints. The `message` property is an optional message to send to a participant in case this constraint causes their message to be rejected. This particular message is not very helpful, but specifying messages with enough detail to entice the desired co-operation from the participants can be a challenge for the E-Agent author. Section 5 discusses techniques for automatically constructing more informative messages.

Notifications: a set of email messages to send when some condition is satisfied, e.g., to notify the originator when the total number of guests reaches `GuestThreshold`:

```
[a          :OnConditionSatisfied;
 :guard      "$GuestThreshold$ > 0";
 :define     [ :name  "TotalGuests";
               :value "[SELECT SUM(NumGuests) FROM CURR_STATE]"];
 :condition "$TotalGuests$ >= $GuestThreshold$";
 :notify    :Originator;
 :message   "Currently, $TotalGuests$ guests are expected.";]
```

3.2 Discussion

The example above illustrates two different ways for accessing the data collected by the E-Agent: via a pre-defined variable (e.g., `Bring.last(),Bring.$x$.count()`) or, less commonly, by utilizing an explicit SQL query over a virtual table constructed from the RDF (e.g., as with `TotalGuests`). The former method is more convenient and allows the author to easily specify decisions based on a variety of views of the underlying data. More importantly, if the constraints refer to response data only through such pre-defined variables, then they are guaranteed to be *bounded*, because they enable the agent to summarize all of the responses that have been received with a set of counters, where the number of counters is independent of the number of participants. For this type of constraints (which still enables many useful E-Agents), the optimal decision of whether to accept a message can always be computed in polynomial time [19]. Thus, the language enables more complex data access mechanisms as necessary but helps authors to write E-Agents that are computationally tractable.

This example also highlights additional key features of our language, including:

- Guards (e.g., with `$AskForNumGuests$`)
- Sets, membership testing, and set manipulation (e.g., `$Choices$-$OptOut$`)
- Universal quantification (e.g.. `forAll $x$`... enforce this constraint)
- Question types/restrictions (e.g., `IntegerQuestion, minInclusive`)
- Multiple constraint types (e.g., `MustConstraint` vs. `PossiblyConstraint`)
- Math. functions and comparisons (e.g., `abs(x-y) <= $MaxImbalance$`)
- Pre-defined queries over the supporting data set (e.g., `$Bring.last()$`)

Among other advantages, guards, sets, and universal quantification enable a single, concise E-Agent template to be instantiated with many different choices and configurations. Likewise, question types and restrictions reduce template complexity by ensuring that responses are well-formed. Finally, multiple constraint/notification types, mathematical functions, and pre-defined queries simplify the process of making decisions based on the responses that are received. Overall, these features make it substantially easier to author useful agents with potentially complex functionality.

Using this template language, we have authored and deployed a number of E-Agents for simple tasks such as collecting RSVPs, giving tickets away (first-come, first-served), scheduling meetings, and balancing a potluck. Our experience has demonstrated that this language is sufficient for specifying a wide range of useful E-Agents (see Table 1).

4 Template Instantiation and Verification

The second major challenge for template-based specifications is to ensure that originators can easily and safely instantiate a template into an E-Agent declaration that will accomplish their goals. This section first briefly describes how to acquire and validate instantiation parameters from the originator. We then examine in more detail the problem of ensuring that a template cannot be instantiated into an invalid declaration.

Each E-Agent template must be accompanied by a web form that enables originators to provide the parameters needed to instantiate the template into a declaration. To automate this process, our implementation provides a tool that generates such a web form from a simple RDF *parameter description*:

Definition 1. *(parameter description) A parameter description ϕ for a template τ is a set $\{R_1, ..., R_M\}$ where each R_i provides, for each parameter P_i in τ, a name, prompt, type, and any restrictions on the legal values of P_i. Types may be simple (Boolean, Integer, Double, String, Email address) or complex (i.e., a* set *of simple types). Possible restrictions are: (for simple types) enumeration, minimal or maximal value, and (for sets) non-empty, or a subset relationship to another set parameter.*

For instance, the following (partial) parameter description relates to asking participants about the number of guests that they will bring to the potluck:

```
[a         :TypeBoolean;
 :name     "AskForNumGuests";
 :enumeration (
  [:value :True;  :prompt "Yes, ask for the number of guests";]
  [:value :False; :prompt "No, don't ask about guests";] ) ]
[a         :TypeInteger;
 :name     "GuestThreshold";
 :prompt   "Notify me when # of guests reaches (ignored if 0):";
 :minInclusive "0"; ]
```

The form generator takes a parameter description and template as input and outputs a form for the originator to fill out and submit. If the submitted variables comply with all parameter restrictions, the template is instantiated with the corresponding values and the resulting declaration is forwarded to the server for execution. Otherwise, the tool redisplays the form with errors indicated and asks the originator to try again.

4.1 Instantiation Safety

Unfortunately, not every instantiated template is guaranteed to be executable. For instance, consider instantiating the template of Section 3 with the following parameters:

```
AskForNumGuests = False
GuestThreshold  = 50
```

In this case the notification given in Section 3 is invalid, since it refers to a question symbol `NumGuests` that does not exist because the parameter `AskForNumGuests` is false. Thus, the declaration is not executable and must be refused by the server. This particular problem could be addressed either in the template (by adding an additional `guard` on the notification) or in the parameter description (by adding a restriction on `GuestThreshold`). However, this leaves open the general problem of ensuring that *every* instantiation results in a *valid declaration*:

Definition 2. *(valid declaration) An instantiated template δ is a valid declaration if:*

1. **Basic checks:** *δ must validate without errors against the* E-*Agent ontology, and every expression $e \in \delta$ must evaluate to a valid numerical or set result.*
2. **Enabled symbols:** *For every expression $e \in \delta$ that is* enabled *(i.e., does not have an unsatisfied guard), every symbol in e is defined once by some enabled node.*
3. **Non-empty enumerations:** *For every enabled* `enumeration` *property $p \in \delta$, the object of p must evaluate to a non-empty set.*

Definition 3. *(instantiation safety) Let τ be a template and ϕ a parameter description for τ. τ is instantiation safe w.r.t. ϕ if, for all parameter sets ξ that satisfy the restrictions in ϕ, instantiating τ with ξ yields a valid declaration δ.*

Instantiation safety is of significant practical interest for two reasons. First, if errors are detected in the declaration, any error message is likely to be very confusing to the originator (who knows only of the web form, not the declaration). Thus, an automated tool is desirable to ensure that a deployed template is instantiation safe. Second, constructing instantiation-safe templates can be very onerous for authors, since it may require considering a large number of possibilities. Even when this is not too difficult, having an automated tool to ensure that a template remains instantiation safe after a modification would be very useful.

Some parts of verifying instantiation safety are easy to perform. For instance, checking that every *declaration* will validate against the E-Agent ontology can be performed by checking the *template* against the ontology, and other checks (e.g., for valid numerical results) are similar to static compiler analyses. However, other parts (e.g., ensuring that a symbol will always be enabled when it is used) are substantially more complex because of the need to consider all possible instantiations permitted by the parameter description ϕ. Consequently, in general verifying instantiation safety is difficult:

Theorem 1. *Given τ, an arbitrary* E-*Agent template, and ϕ, a parameter description for τ, then determining* instantiation safety *is co-NP-complete in the size of ϕ.*

This theorem is proved by a reduction from $\overline{SAT}$. Intuitively, given a specific counter-example it is easy to demonstrate that a template is *not* instantiation-safe, but proving that a template is safe potentially requires considering an exponential number of parameter combinations. In practice, ϕ may be small enough that the problem is feasible. Furthermore, in certain cases this problem is computationally tractable:

Theorem 2. *Let τ be an* E-*Agent template and ϕ a parameter description for τ. Determining* instantiation safety *is polynomial time in the size of τ and ϕ if:*

- *each* `forAll` *and* `enumeration` *statement in τ consists of at most some constant J set parameters combined with any set operator, and*
- *each* `guard` *consists of conjunctions and disjunctions of up to J terms (which are boolean parameters, or compare a non-set parameter with a constant/parameter).*

These restrictions are quite reasonable and still enable us to specify all of the E-Agents described in this paper (using $J \leq 4$). Note that they do not restrict the total number of parameters, but rather the number that may appear in any one of the identified statements. The restrictions ensure that only a polynomial number of cases need to be considered for each constraint/notification item, and the proof relies on a careful analysis to show that each such item can be checked independently while considering at most one question at a time.[3]

4.2 Discussion

In our implementation, we provide a tool that approximates instantiation safety testing via limited model checking. The tool operates by instantiating τ with all possible parameters in ϕ that are boolean or enumerated (these most often correspond to general configuration parameters). For each possibility, the tool chooses random values that satisfy ϕ for the remaining parameters. If any instantiation is found to be invalid, then τ is known to be not instantiation safe. Extending this approximate, non-guaranteed algorithm to perform the exact, polynomial-time (but more complex) testing of Theorem 2 is future work.

Clearly nothing in our analysis relied upon the fact that our agents are email-based. Instead, similar issues will arise whenever 1.) an author is creating a template that is designed to be used by other people (especially untrained people), and 2.) for flexibility, this template may contain a variety of configuration options. A large number of agents, such as the RCal meeting scheduler [26], Berners-Lee et al.'s appointment coordinator [2], and McIlraith et al.'s travel planner [22], have the need for such flexibility and could be profitably implemented with templates. This flexibility, however, can lead to unexpected or invalid agents, and thus produces the need to verify various safety properties such as "doing no harm" [32] or the instantiation safety discussed above. Our results highlight the need to carefully design the template language and appropriate restrictions so that such safety properties can be verified in polynomial time.

5 Automatic Explanation Generation

While executing, an E-Agent utilizes rejections or suggestions to influence the eventual outcome. However, the success of these interventions depends on the extent to which they are understood by the participants. For instance, the rejection "Sorry, the only dates left are May 7 and May 14" is much more likely to elicit cooperation from a participant in a seminar scheduling E-Agent than the simpler rejection "Sorry, try again." The E-Agent author can manually encode such explanations into the template, but this task can be difficult or even impossible when constraints interact or depend on considering possible future responses. Thus, below we consider techniques for simplifying the task of the E-Agent author by automatically generating explanations based on *what* responses are acceptable now and *why* the participant's original response was not acceptable.

We begin by defining more precisely a number of relevant terms. Given an E-Agent, the *supporting data set* is an RDF data store that holds responses from the participants to the questions posed by the E-Agent. The *current state* D is the state of this data set given all of the responses that have been received so far. We assume that the number of participants is known and that each will eventually respond.

[3] See McDowell [18] for details on the proofs of this paper's theorems.

A *constraint* is an arbitrary boolean expression over constants, parameters, and variables. Variables may be arbitrary expressions over constants, parameters, other variables, and queries that select or aggregate values for some question in the data set. Constraint satisfaction may be defined in two different ways. First,

Definition 4. *(MustConstraint) A* `MustConstraint` *C is a constraint that is satisfied in state D iff evaluating C over D yields* `True`.

If a response would lead to a state that does not satisfy a `MustConstraint` C, it is rejected. For example, for the potluck we would not accept a dessert response if that would lead to having 3 more desserts than entrees or appetizers. In many cases, however, such a conservative strategy will be overly restrictive. For instance, we may want to continue accepting desserts so long as it is still *possible* to achieve a balanced final outcome. Furthermore, a `MustConstraint` is usable only when the constraints are initially satisfied, even before any responses are received, and thus greatly limits the types of goals that can be expressed. Hence, we also define a second constraint type:

Definition 5. *(PossiblyConstraint) A* `PossiblyConstraint` *C is a constraint that is* ultimately satisfiable *in state D if there exists a sequence of responses from the remaining participants that leads to a state D' so that evaluating C over D' yields* `True`.

This approach permits more flexibility with the constraints and with the sequence of responses, though computing satisfaction for such constraints is more challenging.

For simplicity, we assume that the constraints C_D are either all `MustConstraints` or all `PossiblyConstraints`, though our results for `PossiblyConstraints` also hold when C_D contains both types. In addition, some results below mention *bounded* constraints (see Section 3.2), a restricted type that still supports a wide range of agents (including all those discussed in this paper). Recall that a sufficient condition for being bounded is for the constraints to access data only via "pre-defined" variables.

5.1 Acceptable Responses

Often the most practical information to provide to a participant whose response led to an intervention is the set of responses that would be "acceptable" (e.g., "An Appetizer or Dessert would be welcome" or "Sorry, I can only accept requests for 2 tickets or fewer now"). This section briefly considers how to calculate this *acceptable set*.

Definition 6. *(acceptable set) Let Λ be an* E*-Agent with current state D and constraints C_D on D. Then, the* acceptable set *A of Λ is the set of legal responses r such that D would still be satisfiable w.r.t. C_D after accepting r.*

For a `MustConstraint`, this satisfiability testing is easy to do and we can compute the acceptable set by testing a small set of representative responses. For a `PossiblyConstraint`, the situation is more complex:

Theorem 3. *Given an* E*-Agent Λ with N participants and current state D, if the constraints C_D are bounded, then the acceptable set A of Λ can be computed in time polynomial in N, $|A|$, and $|C_D|$. Otherwise, this problem is NP-hard in N.*

In this case we can again compute the acceptable set by testing satisfiability over a small set of values; this testing is polynomial if C_D is bounded [19]. In addition, if C_D is bounded then either $|A|$ is small or A can be concisely represented via ranges of acceptable values, in which case the total time is polynomial in only N and $|C_D|$.

5.2 Explaining Interventions

In some cases, the acceptable set alone may not be enough to construct a useful explanation. For instance, suppose an E-Agent invites 4 professors and 20 students to a meeting that at least 3 professors and a quorum of 10 persons (professors or students) must attend. When requesting a change from a professor, explaining *why* the change is needed (e.g., "We need you to reach the required 3 professors") is much more effective than simply informing them *what* response is desired (e.g., "Please change to Yes"). A clear explanation both motivates the request and rules out alternative reasons for the request (e.g., "We need your help reaching quorum") that may be less persuasive (e.g., because many students could also help reach quorum). This section discusses how to generate explanations for an intervention based on identifying the constraint(s) that led to the intervention. We do not discuss the additional problem of translating these constraints into a natural language suitable for sending to a participant, but note that even fairly simple explanations (e.g., "Too many Appetizers (10) vs. Desserts (3)") are much better than no explanation.

Conceptually, an E-Agent decides to reject a response based on constructing a *proof tree* that shows that some response r would prevent constraint satisfaction. However, this proof tree may be much too large and complex to serve as an explanation for a participant. This problem has been investigated before for expert systems [24, 29], constraint programming [14], description logic reasoning [20], and more recently in the context of the Semantic Web [21]. These systems assumed proof trees of arbitrary complexity and handled a wide variety of possible deduction steps. To generate useful explanations, key techniques included abstracting multiple steps into one using rewrite rules [20, 21], describing how general principles were applied in specific situations [29], and customizing explanations based on previous utterances [4].

In our context, the proof trees have a much simpler structure that we can exploit. In particular, proofs are based only on constraint satisfiability (over one state or all possible future states), and each child node adds one additional response to the parent's state in a very regular way. Consequently, we will be able to summarize the proof tree with a very simple type of explanation. These proof trees are defined as follows:

Definition 7. *(proof tree) Given an* E-*Agent A, current state D, constraints C_D, and a response r, we say that P is a* proof tree *for rejecting r on D iff:*

- *P is a tree where the root is the initial state D.*
- *The root has exactly one child D_r, representing the state of D after adding r.*
- *If C_D is all* `MustConstraints`*, then D_r is the only non-root node.*
- *If C_D is all* `PossiblyConstraints`*, then for every node n that is D_r or one of its descendants, n has all children that can be formed by adding a single additional response to the state of n. Thus, the leaf nodes are only and all those possible final states (e.g., where every participant has responded) reachable from D_r.*
- *For every leaf node l, evaluating C_D over the state of l yields* `False`.

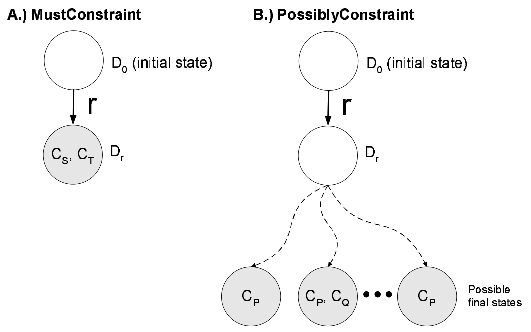

Fig. 2. Examples of proof trees for rejecting response r. Each node is a possible state of the data set, and node labels are constraints that are *not* satisfied in that state. In both cases, response r must be rejected because every leaf node (shaded above) does not satisfy some constraint.

Figure 2A illustrates a proof tree for `MustConstraints`. Because accepting r leads to a state where some constraint (e.g., c_T) is not satisfied, r must be rejected. Likewise, Figure 2B shows a proof tree for `PossiblyConstraints`, where C_P and C_Q represent the professor and quorum constraints from the example described above. Since we are trying to prove that there is no way for the constraints to be ultimately satisfied (by any outcome), this tree must be fully expanded. For this tree, every leaf (final outcome) does not satisfy some constraint, so r must be rejected.

We now define a simpler explanation based upon the proof tree:

Definition 8. *(sufficient explanation) Given an* E-*Agent* Λ*, current state* D*, constraints* C_D*, and a response* r *such that a proof tree* P *exists for rejecting* r *on* D*, then we say that* E *is a* sufficient explanation *for rejecting* r *iff,*

- *E is a conjunction of constraints that appear in C_D, and*
- *for every leaf node n in P, evaluating E over the state of n yields* `False`.

Intuitively, a sufficient explanation E justifies rejecting r because E covers every leaf node in the proof tree, and thus precludes ever satisfying C_D. Note that while the proof tree for rejecting r is unique (modulo the ordering of child nodes), an explanation is not. For instance, an explanation based on Figure 2A could be C_S, C_T, or $C_S \wedge C_T$. Likewise, a valid explanation for Figure 2B is $C_P \wedge C_Q$ (e.g., no way satisfy both the professor and quorum constraints) but a more precise explanation is just C_P (e.g., no way to satisfy the professor constraint). The smaller explanation is often more compelling, as we argued for the meeting example, and thus to be preferred [6]. In general, we wish to find the explanation of minimum size (i.e., with the fewest conjuncts):

Theorem 4. *Given an* E-*Agent* Λ *with* N *participants, current state* D*, constraints* C_D*, and a response* r*, if* C_D *consists of* `MustConstraints`*, then finding a minimum sufficient explanation* E *for rejecting* r *is polynomial time in* N *and* $|C_D|$*. If* C_D *consists of* `PossiblyConstraints`*, then this problem is NP-hard in* N *and* $|C_D|$.

Thus, computing a minimum explanation is feasible for `MustConstraints` but likely to be intractable for `PossiblyConstraints`. For the latter, the difficulty arises from two sources. First, checking if any particular E is a sufficient explanation is

NP-hard in N (based on a reduction from ultimate satisfiability [19]); this makes scaling E-Agents to large numbers of participants difficult. Second, finding a minimum such explanation is NP-hard in the number of constraints (by reduction from SET-COVER [12]). Note that this number can be significant because we treat each `forAll` quantification as a separate constraint; otherwise, the sample potluck described in Section 3 would always produce the same (complex) constraint for an explanation. Fortunately, in many common cases we can achieve a polynomial time solution:

Theorem 5. *Given an* E*-Agent Λ with N participants, current state D, constraints C_D, and a response r, if C_D is bounded and the size of a minimum explanation is no more than some constant J, then computing a minimum explanation E is polynomial time in N and $|C_D|$.*

This theorem holds because a candidate explanation E can be checked in polynomial time when the constraints are bounded, and restricting E to at most size J means that the total number of explanations that must be considered is polynomial in the number of constraints. Both of these restrictions are quite reasonable. As previously mentioned, bounded constraints arise naturally when using our template language and permit a wide range of functionality. Likewise, E-Agent explanations are most useful to the participants when they contain only a small number of constraints, and this is adequate for many E-Agents (as in the meeting example above). If no sufficient explanation of size J exists, the system could either choose the best explanation of size J (to maintain a simple explanation), approximate the minimum explanation with a greedy algorithm, or fall back on just providing the participant with the acceptable set described in the previous section.

Many different types of agents can describe their goals in terms of a set of constraints [17, 23], and often need to explain their actions to users. Our results show that while generating such explanations can be intractable in general, the combination of simple explanations and modest restrictions on the constraint system can enable explanation generation in polynomial time.

6 Related Work

McDowell et al. [19] describe work related to the general notion of semantic email, including its relation to existing workflow and collaboration systems. Here we focus on research relevant to the agent specification problem.

Other projects have considered how to simplify the authoring of Semantic Web applications, e.g., with Haystack's Adenine programming language [27]. Adenine resembles our template language in that it can be compiled into RDF for portability and contains a number of high-level primitives, though Adenine incorporates many more imperative features and does not support the types of declarative reasoning that we describe. Languages such as DAML-S and OWL-S [5] enable the description of an application as a Semantic Web *service*. These languages, however, focus on providing details needed to *discover* and *invoke* a relevant service, and model every participant as another web service. Our work instead concisely specifies an E-Agent in enough detail so that it can be directly *executed* in contexts involving untrained end users.

More generally, E-Agent templates could be viewed as an instance of *program schemas* [7, 10] that encapsulate a general class of behavior, e.g., for automated program synthesis [10] or software reuse [7, 1]. Similarly, McIlraith et al. [22] propose the use of *generic procedures* that can be instantiated to produce different compositions of web services. Concepts similar to our definition of instantiation safety naturally arise in this setting; proposals for ensuring this safety have included manually-generated proofs [7], automatically-generated proofs [10], and language modification [1]. Our work focuses on the need for such schemas to be safely usable by ordinary people and demonstrates that the required safety properties can be verified in polynomial time.

Recent work on the *Inference Web* [21] has focused on the need to explain a Semantic Web system's *conclusions* in terms of base data and reasoning procedures. In contrast, we deal with explaining the agent's *actions* in terms of existing responses and the expected impact on the E-Agent's constraints. In this sense our work is similar to prior research that sought to explain decision-theoretic advice (cf., Horvitz et al. [11]). For instance, Klein and Shortliffe [16] describe the VIRTUS system that can present users with an explanation for why one action is provided over another. Note that this work focuses on explaining the relative impact of multiple factors on the choice of some action, whereas we seek the simplest possible reason why some action could *not* be chosen (i.e., accepted). Other relevant work includes Druzdzel [8], which addresses the problem of translating uncertain reasoning into qualitative verbal explanations.

For constraint satisfaction problems (CSPs), a *nogood* [28] is a reason that no *current* variable assignment can satisfy all constraints. In contrast, our explanation for a `PossiblyConstraint` is a reason that no *future* assignment can satisfy the constraints, given the set of possible future responses. Potentially, our problem could be reduced to nogood calculation, though a direct conversion would produce a problem that might take time that is exponential in N, the number of participants. However, for bounded constraints, we could create a CSP with variables based on the *aggregates* of the responses, rather than their specific values [19]. Using this simpler CSP, we could then exploit existing, efficient nogood-based solvers (e.g., [15, 13]) to find candidate explanations in time polynomial in N. Note though that most applications of nogoods have focused on their use for developing improved constraint solving algorithms [28, 15] or for debugging constraint programs [25], rather than on creating explanations for average users. One exception is Jussien and Ouis [14], who describe how to generate user-friendly `nogood` explanations, though they require that a designer explicitly model a user's perception of the problem as nodes in some constraint hierarchy.

7 Conclusions and Implications for Agents

This paper has examined how to specify agent behavior. We adopted a template-based approach that shifts most of the complexity of agent specification from untrained originators onto a much smaller set of trained authors. We then examined the three key challenges of generality, safety, and understandability that arise in this approach. For E-Agents, we discussed how high-level features of our template language enable the concise specification of complex agent behavior. We also demonstrated that it is possible to verify the instantiation-safety of a template in polynomial time, and showed how to generate explanations for the agent's actions in polynomial time. Together, these techniques both simplify the task of the E-Agent author and improve the overall execution quality for the originator and the participants of an E-Agent. In addition, our polynomial time results ensure that these features can scale to E-Agents with large numbers of participants, choices, and constraints.

While we focused on the context of semantic email, our results are relevant to many other agent systems. For instance, almost any agent needs some ability to explain its behavior, and many such agents react to the world based on constraints. We showed that generating explanations can be NP-hard in general, but that the combination of simple explanations and modest constraint restrictions may enable explanation generation in polynomial time. Likewise, an agent template should support a wide range of functionality, yet ensure the *safety* of each possible use. There are several different types of safety to consider, including that of doing no harm [32], minimizing unnecessary side-effects [32], and accurately reflecting the originator's preferences [3]. We motivated the need for instantiation safety, a type that has been previously examined to some extent [10, 1], but is particularly challenging when the instantiators are non-technical users. Our results also highlight the need to carefully design template languages that balance flexibility with the ability to efficiently verify such safety properties.

Thus, many agents could benefit from a high-level, declarative template language with automatic safety testing and explanation generation. Collectively, these features would simplify the creation of an agent, broaden its applicability, enhance its interaction with the originator and other participants, and increase the likelihood of satisfying the originator's goals. Future work will consider additional ways to make agent authoring and instantiation easier with the goal of bringing the Semantic Web vision closer to practical implementation.

8 Acknowledgements

This research was supported in part by NSF grant DARPA contract NBCHD030010, IIS-0312988, and ONR grant N00014-02-1-0324 for Oren Etzioni; NSF CAREER grant IIS-9985114 and ITR grant IIS-0205635 for Alon Halevy; and by NSF Graduate Research and Microsoft Endowed Fellowships for Luke McDowell. Jeff Lin assisted with the implementation. Thanks to Gerome Miklau, Natasha Noy, Stani Vlasseva, and the anonymous reviewers for their helpful comments on improving this work.

References

1. E. E. Allen. *A First-Class Approach to Genericity*. PhD thesis, Rice University, Houston, TX, 2003.
2. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.
3. C. Boutilier. A POMDP formulation of preference elicitation problems. In *AAAI-02*, pages 239–246, 2002.
4. G. Carenini and J. D. Moore. Generating explanations in context. In *Intelligent User Interfaces*, pages 175–182, 1993.
5. DAML Services Coalition. DAML-S and OWL-S. http://www.daml.org/services.
6. M. G. de la Banda, P. J. Stuckey, and J. Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 32–43. ACM Press, 2003.

7. N. Dershowitz. Program abstraction and instantiation. *ACM Trans. Programming. Language Systems*, 7(3):446–477, 1985.
8. M. Druzdzel. Qualitative verbal explanations in bayesian belief networks. *Artificial Intelligence and Simulation of Behaviour Quarterly*, 94:43–54, 1996.
9. O. Etzioni, A. Halevy, H. Levy, and L. McDowell. Semantic email: Adding lightweight data manipulation capabilities to the email habitat. In *WebDB*, 2003.
10. P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, July 2000.
11. E. J. Horvitz, J. S. Breese, and M. Henrion. Decision theory in expert systems and artificial intelligence. *International Journal of Approximate Reasoning*, 2:247–302, 1988.
12. D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
13. U. Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, 2001.
14. N. Jussien and S. Ouis. User-friendly explanations for constraint programming. In *ICLP 11th Workshop on Logic Programming Environments*, Paphos, Cyprus, Dec. 2001.
15. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in csp search. In *Principles and Practice of Constraint Programming*, October 2003.
16. D. A. Klein and E. H. Shortliffe. A framework for explaining decision-theoretic advice. *Artificial Intelligence*, 67(2):201–243, 1994.
17. A. K. Mackworth. Constraint-based agents: The ABC's of CBA's. In *Constraint Programming*, pages 1–10, 2000.
18. L. McDowell. *Meaning for the Masses: Theory and Applications for Semantic Web and Semantic Email Systems*. PhD thesis, University of Washington, Seattle, WA, 2004.
19. L. McDowell, O. Etzioni, A. Halevey, and H. Levy. Semantic email. In *Proc. of the Thirteenth Int. WWW Conference*, 2004.
20. D. L. McGuinness and A. Borgida. Explaining subsumption in description logics. In *IJCAI (1)*, pages 816–821, 1995.
21. D. L. McGuinness and P. Pinheiro da Silva. Infrastructure for web explanations. In *Second International Semantic Web Conference*, October 2003.
22. S. McIlraith, T. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2):46–53, March/April 2001.
23. A. Nareyek. *Constraint-Based Agents*, volume 2062 of *Lecture Notes in Computer Science*. Springer, 2001.
24. R. Neches, W. R. Swartout, and J. D. Moore. Explainable (and maintainable) expert systems. In *IJCAI*, pages 382–389, 1985.
25. S. Ouis, N. Jussien, and P. Boizumault. k-relevant explanations for constraint programming. In *FLAIRS'03*, St. Augustine, Florida, USA, May 2003. AAAI press.
26. T. Payne, R. Singh, and K. Sycara. Calendar agents on the semantic web. *IEEE Intelligent Systems*, 17(3):84–86, 2002.
27. D. Quan, D. Huynh, and D. R. Karger. Haystack: A platform for authoring end user semantic web applications. In *Second International Semantic Web Conference*, October 2003.
28. T. Schiex and G. Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
29. W. Swartout, C. Paris, and J. Moore. Design for explainable expert systems. *IEEE Expert*, 6(3):58–647, 1991.
30. L. G. Terveen and L. T. Murray. Helping users program their personal agents. In *CHI*, pages 355–361, 1996.
31. R. Tuchinda and C. A. Knoblock. Agent wizard: building information agents by answering questions. In *Intelligent User Interfaces*, pages 340–342, 2004.
32. D. Weld and O. Etzioni. The first law of robotics (a call to arms). In *Proc. of AAAI*, 1994.