Specifying Semantic Email Processes

Luke McDowell, Oren Etzioni, and Alon Halevy
University of Washington
E-mail: {lucasm,etzioni,alon}@cs.washington.edu

Abstract

Prior work has shown that semantic email processes *(*SEPs*) can be an effective tool for automating email-mediated tasks that are currently performed manually in a tedious, time-consuming, and error-prone manner. However, specifying a* SEP *can be difficult to accomplish, even for users familiar with RDF and semantic email. In response, this paper considers an approach for specifying* SEP *templates that can be* authored *once but then* instantiated *many times by untrained users. We describe the template language and provide a complete example, highlighting the key features needed to enable general* SEPs*. We then examine a number of challenges related to* SEP *authoring. In particular, we discuss the problem of verifying that a given template will always produce a valid instantiation and give the computational complexity of this problem. In addition, we discuss how to simplify the task of* SEP *authoring (and improve execution quality) by automatically generating explanations for the actions performed in pursuit of a* SEP*'s goals. Finally, we report on practical experience with our fully deployed system.*[1]

1 Introduction

In previous work [2, 7], we introduced *semantic email processes* (SEPs) and demonstrated how they can effectively manage a number of tasks that are currently performed via email in a tedious, time-consuming, and error-prone way. These processes 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 *goals*. In order to achieve these goals, the semantic email *manager* may utilize a number of *interventions* (e.g., rejecting a participant's response or suggesting changes).

McDowell et al. [7] demonstrated that SEPs can be used for a wide range of useful interactions and that important reasoning problems for SEPs are computationally tractable in many common cases. Applying this theory to real problems, however, requires the ability to create a SEP specification that corresponds to an originator's goals. Ideally, we would like an originator who is untrained in the usage of RDF and semantic email to be able to easily construct such a specification. Unfortunately, without appropriate support this task may be challenging even for trained users, because the specification must fully describe the originator's goals in terms of some RDF content to collect, express how and when to use interventions in order to achieve these goals, and support the explanation of these interventions to the participants.

The contributions of this paper are as follows. First, we describe how to specify SEPs via *templates*. This approach addresses the difficulties described above by enabling a SEP template to be *authored* once by a user familiar with semantic email but then *instantiated* many times by untrained originators. We describe the key primitives of our declarative template language, which allows authors to easily express complex goals but also raises several authoring challenges. In particular, to avoid frustrating the originator we must ensure that every possible instantiation of a template is executable, yet checking this property manually can very difficult for the SEP author. Thus, we examine how to automate this task by formally defining the problem of testing *instantiation safety*, describing its computational complexity, and providing an approximate solution. Likewise, to secure the participants' cooperation the manager's interventions must be clearly explained, yet providing these explanations in the template can be challenging or impossible for the author. Consequently, we examine how to automatically generate these explanations in terms of *why* a particular response could not be accepted and *what* responses would be more acceptable, and describe both approximate and optimal algorithms for computing these quantities.

The next section gives a brief overview of SEPs while Section 3 describes our template language and a complete example. Sections 4 and 5 examine the problems of instantiation safety and explanation generation that were discussed above. Finally, Section 6 discusses our experience with a deployed semantic email system, Section 7 examines related work, and Section 8 concludes.

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

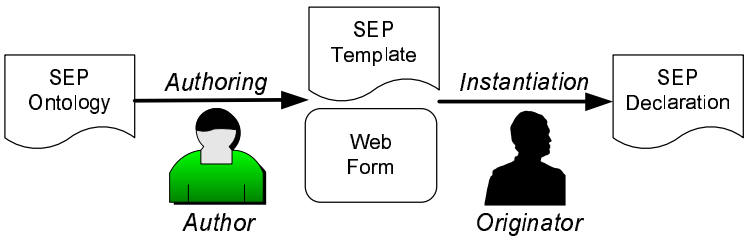

Figure 1. The creation of a semantic email process. Initially, a SEP template is *authored* by the "Author", then is later *instantiated* by the "Originator". Typically, a template is authored once and then instantiated many times.

2 Overview of SEP Creation and Execution

We illustrate SEPs with the running example of a "balanced potluck." The originator of the process initially sends out a message announcing the potluck and asking each participant whether they are bringing an `Appetizer`, `Entree`, or `Dessert` (or `Not-Coming`). The originator also expresses a set of goals for the potluck, which may be specified in two different ways. For a logical SEP, the originator specifies a set of *constraints* that should be satisfied by any process outcome. For example, the originator may specify that the difference in the number of appetizers, entrees, or desserts should be at most two. Alternatively, for a decision-theoretic SEP, the originator provides a function representing the *utility* of possible process outcomes (e.g., a function that becomes more negative as the potluck becomes more unbalanced).

Conceptually, the semantic email system acts as a *manager* that uses interventions to direct the process towards an outcome consistent with the originator's goals. For instance, if the manager detects that the potluck is becoming too unbalanced, the manager can *reject* a participant's response or *suggest* that a participant modify his response.

Figure 1 demonstrates how a template is used to create a new SEP. Initially, a user, who is assumed to have some basic knowledge of RDF and semantic email, authors a new template using an editor (most likely by modifying an existing template). We call this user the SEP *author*. The template is written in OWL based on a SEP ontology that describes the possible queries, goals, and messages for a process. For instance, the "balanced potluck" template defines the general balance constraints for the process, but has placeholders for *parameters* such as the participants' addresses, the specific choices to offer, and how much imbalance to permit. To enable an originator to provide these values later, the author also constructs a simple *web form* that prompts the originator for each parameter. Section 4 describes how to automatically generate such forms.

An untrained originator finds a SEP template from a public library and instantiates the template by filling out the corresponding web form, yielding a SEP *declaration* (also in OWL). The originator then invokes the process by submitting this declaration to the manager. The manager executes the declaration, using appropriate algorithms to decide how to direct the process via message rejections and suggestions.

In our implementation, the manager is a central server. When invoked, this server sends initial messages to the participants announcing the process. These messages also contain a plain text form that can be handled by any mail client. Participants reply via mail directly to the server, rather than to the originator, and the originator receives status messages from the server when appropriate. The originator can query or alter the process via additional emails or a web interface.

3 SEP Templates

This section defines the parts of a SEP template and then examines a detailed example for the balanced potluck.

3.1 Components of a SEP template

SEP templates (as well as SEP declarations) are composed of four primary parts:

Preamble: general information describing the process originator, the set of participants, and explanatory text to be sent with the initial request (e.g., "You have been invited to the following potluck...").

Queries: the set of queries to ask each participant. For instance, a potluck SEP might query each participant for the food item and the number of guests that they are bringing. Each query defines a variable name for later use and the type of valid responses to that query (e.g., integer, boolean, etc.). Queries may also specify further restrictions on what responses are considered valid (e.g., `NumGuests` must be non-negative). Finally, each query item provides an RDQL query that specifies the semantic meaning of the requested information and is used to map the participant's textual response to RDF. This RDQL portion also enables a suitably equipped participant to have an agent respond to some requests automatically (e.g., "Decline all invitations for Friday evening").

Goals: the originator's goals for the process. These goals may be expressed either as constraints that must be satisfied at every point in time (a `MustConstraint`) or con-

straints that should, if possible, be *ultimately satisfied* by the final process outcome (a `PossiblyConstraint`). Alternatively, the goals can be expressed via a utility function over the eventual process outcome, along with descriptions of the costs (social or otherwise) of asking participants to switch and the probabilities of expected participant behavior (see [7] regarding the estimation of these values). This type of goal is referred to as a `TradeoffGoal` because it strives to balance the utility of the expected process outcome against the costs of actions taken to achieve that outcome.

The manager uses the process's goals to decide when to make a rejection or suggestion. Goals may also specify some text to explain these interventions to the participants. This text may be static or dynamically generated based on the current state (e.g., "Sorry, we already have 5 more Appetizers than Desserts"). Providing enough detail in the messages so that they are understandable to the participants (and hence yield the desired cooperation) can be a challenge for the SEP author. Section 5 discusses techniques for automatically constructing these explanations.

The constraints or utility functions are written as expressions involving arbitrary arithmetic functions over constants and variables. There are three classes of variables:

- **Parameters:** a value provided by the originator when instantiating the template (e.g., `Choices`, the options to offer the participants).
- **Author-defined:** any variable explicitly defined by the SEP author. These variables may represent common subexpressions or may be used to iterate over some set (e.g., to loop through each value of `Choices`, verifying that none violate the constraints). In addition, these variables may be queries over a supporting data set that contains the responses of each participant to the originator's request. For convenience, these RDF responses are mapped to a virtual relational table that may be queried via SQL.
- **Pre-defined:** variables automatically computed by the manager (e.g., `NumResponses`, the total number of responses received so far). The system also provides a few common queries over the supporting data set (e.g., `Bringing.Entree.count()` is the number of "Entree" responses received).

Notifications: a set of email messages to send when some condition is satisfied. The target of a notification may be an arbitrary list of email recipients (possibly from a query over the underlying data set). Alternatively, the target may be the `Originator`, `Responders`, `NonResponders` (particularly useful for sending a reminder to respond after a few days), or `AllParticipants`. Finally, sending a notification to the virtual target `ProcessSummary` adds the notification text to a process-specific web page. This web page contains a table with the response of each participant; adding `ProcessSummary` notifications is useful for displaying further summary information over this data (e.g., to show the most popular response to a vote).

A notification may be triggered by a number of conditions including `OnResponseReceived`, `OnAllResponsesReceived`, `OnDateTime`, or the most general `OnConditionSatisfied`. In addition to the notifications specified in the template, our implementation allows the originator to easily create an arbitrary number of `OnDateTime` "reminders" when instantiating the process.

3.2 Template Example

Figure 2 shows a complete SEP template for our example balanced potluck. Parameters that must be instantiated by the originator are shown in bold; other variables such as `NumGuests` will be evaluated as the SEP is executed. The declaration follows the four main parts described above. First, the template specifies the participants and a suitable prompt for the initial message. Second, the template defines two queries. The `Bringing` query indicates that a valid response to this query must be in the the (originator-provided) set `Choices`. The `NumGuests` query is "guarded" so that applies only if the parameter `AskForNumGuests` is true; if so, this query will accept only non-negative integers. In both cases the `query` property provides the aforementioned RDQL query.

Third, the template specifies one `MustConstraint` goal. The constraint is evaluated over every possible `(x,y)` where `x` and `y` are in the set `(Choices - OptOut)`; `OptOut` is for choices such as "Not Coming" that should be excluded from the constraints. The constraint requires that the number of responses `x` (e.g., Appetizer) must differ from the number of responses `y` (e.g., Dessert) by no more than `MaxImbalance`. The `message` property provides an explanation to send to a participant if their response is rejected because of this constraint. This message utilizes the predefined variable `Bringing.acceptable()`, which is explained in Section 5.

Finally, the template specifies two notifications. The first notifies the originator as soon as the total number of expected guests (computed via a SQL query over the supporting data set) reaches `GuestThreshold`. The other notification updates the process summary to include counts of each type of response received. Notice the use of the `forAll` property to iterate over the possible responses, similar to its use in the `MustConstraint`.

The above example demonstrated the use of a `MustConstraint` goal; the same properties may be used to define a `PossiblyConstraint` instead. A `TradeoffGoal` follows the same general form but instead of an `enforce` property it provides a utility expression via an `optimize` property, along with additional properties to describe the associated costs and probabilities.

```
:participants   "$ParticipantsList$";
:prompt
 "You have been invited to the following potluck. Please use the form below to indicate what you are
  bringing. To ensure that our meal selection is balanced, you may be asked to modify your choice.
  Description:   $PromptDescription$
  Location:      $PromptLocation$
  Date and Time: $PromptDateTime.toUserFriendly()$";

:queries (
  [a              :StringQuery;
  :name           "Bringing";
  :query          "WHERE (?process, <rdfcal:attendee>,  ?x1),
                     (?x1,  <rdfcal:calAddress>,        ?EMAIL),
                     (?x1,  <uw:bringing>,              ?Bringing)
                    USING rdfcal FOR <http://www.w3.org/2002/12/cal/ical#>,
                    uw FOR <http://www.cs.washington.edu/research/semweb/vocab#v1_0>";
  :enumeration    "$Choices$" ]

  [a              :IntegerQuery;
  :guard          "$AskForNumGuests$";
  :name           "NumGuests";
  :query          "WHERE (?process ....";
  :minInclusive   "0"; ]  );

:goals (
  # Reject the message if it results in too much inbalance between any two pairs
  [a              :MustConstraint;
  :forAll         ([:name   "x";   :range "$Choices$-$OptOut$"]
                   [:name   "y";   :range "$Choices$-$OptOut$"]);
  :suchThat       "$x$ != $y$";
  :enforce        "abs($Bringing.{$x$}.count()$ - $Bringing.{$y$}.count()$) <= $MaxImbalance$";
  :message        "Your request to bring a $Bringing.last()$ could not be accepted.
                   Choices that could be accepted right now are $Bringing.acceptable()$.";]  );

:notifications (
  # Notify the owner if the number of guests crosses a threshold (ignore if $GuestThreshold$ is zero)
  [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.";]

  # Update the process summary
  [a              :OnMessageReceived;
  :notify         :ProcessSummary;
  :message        (  "Here's how many of each choice confirmed so far:"
                     [ :forAll   ([:name   "x";  :range "$Choices$"]);
                       :evaluate "$x$: $Bringing.{$x$}.count()$"; ] ) ] )
```

Figure 2. SEP template for a "Balanced Potluck" process, shown in N3 format. Variables in bold (e.g., **$Choices$**) are parameters provided by the originator when instantiating the template. Other variables are defined inside the declaration (e.g., `$x$`, `$TotalGuests$`) or are automatically computed by the system (e.g., `$Bringing.acceptable()$`).

4 Template Instantiation and Verification

This section describes how *parameter descriptions* can be used to generate and validate a web form for instantiating templates, then examines the problem of templates that, when instantiated, may yield invalid declarations.

4.1 Parameter Descriptions

Each SEP 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 converts a simple OWL *parameter description* into such a web form. Figure 3 shows a partial example for our example balanced potluck. The description provides a name, type, and prompt for each parameter, along with any restrictions on the legal values of that parameter. For instance, the first parameter block specifies that `Choices` is a set of strings, while the second parameter indicates that `OptOut` is a set of strings that must be a subset of `Choices`. The last two parameters relate to asking participants about the number of guests that they will bring to the potluck.

The form generator tool 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 manager for execution. Otherwise, the tool redisplays the form with errors indicated and asks the originator to try again.

4.2 Instantiation Safety

This section considers the problem of templates which, when instantiated, may yield invalid declarations, explores the significance of this problem, and describes the solutions adopted in our system. There are a number of possible errors that might render a declaration invalid, including:

```
:parameters (
  [a          :TypeList;
   :name      "Choices";
   :prompt    "Choices for the recipients to choose from (enter a comma-separated list)" ]

  [a          :TypeList;
   :name      "OptOut";
   :prompt    "Choices to exclude from these restrictions (enter a comma-separated list)";
   :subset    "$Choices$" ]

  [a          :TypeSelectOne;
   :name      "AskForNumGuests";
   :choices   ([:value :True;   :prompt "Yes, ask how many guests each person is bringing"     ]
               [:value :False;  :prompt "No, don't ask about guests"] ) ]

  [a              :TypeInteger;
   :name          "GuestThreshold";
   :prompt        "Notify the originator when the number of guests reaches (enter 0 to ignore):";
   :minInclusive  "0" ]
)
```

Figure 3. Part of a *parameter description* for the potluck template of Figure 2. Additional elements for variables such as `MaxImbalance` are not shown.

1. **Missing/multiple properties:** e.g., each `MustConstraint` must have exactly one `enforce` property.
2. **Wrong object type:** e.g., the `name` property must point to a literal string, not a resource.
3. **Ambiguous names:** e.g., the same variable or query name must not be defined twice.
4. **Expression errors:** e.g., `"x ++ y"` is not permitted.
5. **Undefined symbols:** e.g., a `condition` property must refer only to variables that have been defined.
6. **Empty set:** e.g., properties such as `enumeration` require that their argument is not the empty set.

Errors 1 and 2 above can be detected automatically by validating the template against the corresponding OWL ontology, while errors 3, 4, and sometimes 5 can be automatically detected by a static analysis of the template. However, some occurrences of errors 5 and 6 will only occur for particular instantiations of the template and thus cannot be detected by examining the template alone. For instance, considering instantiating the potluck template in Figure 2 with the following (partial list of) parameters:

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

In this case the first notification in Figure 2 is invalid, since it refers to a query `NumGuests` that does not exist because `AskForNumGuests` is false. Thus, the declaration is not executable and must be refused by the manager. This particular template problem could be addressed either by adding an additional `guard` on the first notification or by adding a parameter restriction on `GuestThreshold`. However, this leaves open the general problem of ensuring that *no* instantiation results in an invalid declaration:

Definition 4.1 (**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 ϕ, instantiating τ with ξ yields a valid SEP declaration. □

This problem is of significant practical interest for two reasons. First, because 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 to ensure that a deployed template is instantiation safe is desirable. 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.

Theorem 4.1 *Let τ be a template and ϕ a parameter description for τ. If τ is an arbitrary* SEP *template, then* instantiation safety *is co-NP-complete in the size of ϕ.*

Thus, in general determining instantiation safety is difficult, though in practice ϕ may be small enough that the problem is tractable. 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 with type boolean or enumeration (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. We conjecture that an exact, polynomial time algorithm exists for most common cases; this will be considered in future work.

5 Automatic Explanation Generation

While executing the process, the manager utilizes rejections or suggestions to influence the eventual SEP outcome. However, the usefulness of these interventions depends on the extent to which they are understood by the participants. For instance, the rejection "Sorry, only 2 tickets left" is a much more helpful response to a request for 4 tickets than a simple "No." Likewise, the suggestion "Please consider an

Appetizer instead" is much more likely to elicit the desired cooperation than a suggestion with no hint as to a more acceptable potluck dish. As previously mentioned, the SEP author may provide such messages in the template, but providing them in sufficient detail can be a challenging and time-consuming task.

Below we consider techniques for *simplifying* the task of the SEP author by automatically computing such explanations. We focus first on explaining the reasons for a particular intervention and then briefly consider computing the set of currently acceptable responses. Note that in some cases these methods can also be viewed as *improving* the quality of explanations that are possible for a SEP. For instance, because the decision about whether to suggest a different response due to a `TradeoffGoal` depends on a complex balance of possible future states vs. action costs, an automatically-generated explanation can provide much more detail than the manually-encoded explanation of even the most thorough SEP author. Finally, the discussions below relate to all SEPs, but specific complexity results apply only when the constraints/utilities are *bounded* or *K-partitionable* [7]; this includes many common cases and all SEPs discussed in this paper.

5.1 Explaining Interventions

In this subsection we consider how to generate explanations for an intervention based on identifying the constraint(s) or utility term(s) that led to the intervention. We do not discuss the additional problem of translating these terms into a natural language suitable for sending to a participant, but note that even fairly simple explanations (e.g., "Appetizer Count (10) too high vs. Dessert Count (3)") are much better than no explanation.

In our implementation the manager intervenes only with rejections for a `MustConstraint` or `PossiblyConstraint`, and only with suggestions for a `TradeoffGoal`(cf., [7]). We consider each of these in turn.

MustConstraint: In this case all constraints must be initially satisfied, so any constraint that is not satisfied after adding a response r to state D is an explanation for rejecting r. Note that, for explanations, we treat each `forAll` possibility as a separate constraint; otherwise, the sample potluck of Figure 2 would always produce the same (complex) constraint for an explanation.

PossiblyConstraint: For a `PossiblyConstraint`, there is no guarantee that the constraints are satisfied, only that it is possible for them to be *ultimately satisfied* when more responses are received. In this case an explanation should satisfy the following definition:

Definition 5.1 (**sufficient explanation**) Given a data set D and constraints C_D such that D is ultimately satisfiable but would not be after adding response r, we say that $E \subseteq C_D$ is a *sufficient explanation* for rejecting r iff *no* sequence of responses from the participants, beginning with r, will put D in a state that satisfies E. □

Intuitively, a sufficient explanation E justifies rejecting r, because accepting r precludes ever satisfying E. Using a modified form of our ultimate satisfiability theorem [7], testing if any particular E is a sufficient explanation can be done in polynomial time. We could use this procedure repeatedly to compute the *minimum-size* sufficient explanation, but this might require time exponential in the number of constraints. Alternatively, we could use an approximate, greedy algorithm that first ranks each constraint by the number of possible future states for which it is *not* satisfied, then incrementally adds to E the constraint with the largest rank until every possible state is covered. Note that, in the particularly useful case where $|E_{minimum}| = 1$, this algorithm is guaranteed to find the optimal solution.

TradeoffGoal: In this case, it is more difficult to single out specific terms that are responsible for a manager's suggestion, because every term contributes to the process utility to some extent, either positively or negatively. However, if the manager decides to make a suggestion, then the expected improvement must outweigh the certain cost of this action. Thus, for non-zero costs, there must be a significant difference in the utility of the state where the manager requested a switch (S_{sw}) vs. where the manager did not (S_0).

We seek to identify the terms that explain most of this difference. In particular, given a n-term utility function

$$U(s) = u_1(s) + ... + u_n(s)$$

we define the change δ_u in each utility term as

$$\delta_u = u(S_{sw}) - u(S_0).$$

We wish to identify a set $E \subseteq \{u_1, .., ..u_n\}$ such that:

$$\sum_{u \in E} \delta_u \geq \beta[U(s_{sw}) - U(S_0)]$$

i.e., so that the terms in E explain at least β of the change.

Note that, when generating an explanation, we are primarily interested in terms indicating that a switch is beneficial, i.e., where $\delta_u > 0$. If we only consider such terms, then a greedy algorithm suffices to identify the explanation E of guaranteed minimal size: set E to $\emptyset$, then incrementally add to E the term with the largest δ_u until E explains at least β of the total change. If we wish to consider utility terms with both positive and negative changes, then this problem becomes more challenging (cf., Klein and Shortliffe [5]).

SEP name	Procedural approach			Declarative approach			Size Reduction for Declarative
	Java code	Forms (HTML)	Total	Template	Forms (Params)	Total	
Balanced Potluck	1283	397	1680	113	57	170	90%
First-come, First-served	301	235	536	66	33	99	82%
Meeting Coordination	471	272	743	60	22	82	89%
Request Approval	772	286	1058	80	29	109	90%
Auction	392	111	503	55	43	98	81%

Table 1. Comparison of the size (in number of lines) of different ways of specifying a SEP. For the procedural prototype, the first numerical section displays the size of the Java code for encoding the SEP functionality, size of the HTML for acquiring parameters from the originator, and the total of these two. For the declarative approach, the second section displays the size of the template (OWL, in N3 format), size of the parameter description, and the total. The final column shows the percentage reduction in the size of a SEP when changing from the procedural approach to the declarative approach.

5.2 Acceptable Responses

In addition to explaining the reasons why a particular response led to an intervention, we would also like to be able to inform participants about what responses would be more "acceptable." (One could also ask for an explanation of *why* a given response is acceptable, but we do not consider that problem here.) For a constraint-based SEP, if adding a response r to the current state D still permits the constraint to be satisfied (either immediately or ultimately, as appropriate) then r is acceptable. We previously proved that computing whether a specific response r is acceptable with respect to a set of `MustConstraints` or `PossiblyConstraints` could be done in polynomial time [7]. A corollary is that it is also possible to compute in polynomial time the entire set of responses which are acceptable from state D. Our system currently computes and makes available this result for `MustConstraints` (e.g., the use of `Bringing.acceptable()` in Figure 2).

For a `TradeoffGoal`, we define an "acceptable" response as one that is "good enough" so that the manager will not respond with a change suggestion. We might also be interested in computing responses that are "better" than others, e.g., those which result in an expected utility in the top 25% compared to other possible responses. This information could be used when making a suggestion ("Please consider one of these values..."), or could be displayed as part of the process summary to assist participants that have yet to respond. Given a particular state D, both such problems can be solved in polynomial time by comparing the expected utility of states before and after an additional response is received.

6 Experience and Future Work

Our semantic email system is deployed and may be freely used by anyone without any software installation; the source code for deploying other instances of the server is also available. So far we have developed simple processes for functions like collecting RSVPs, giving tickets away, scheduling meetings, and balancing a potluck. Despite very limited publicity, our semantic email server has seen growing interest. For instance, a DARPA working group has adopted semantic email for all of its meeting scheduling and RSVP needs, students have used semantic email to schedule seminars and Ph. D. exams, and semantic email has been used to organize our annual database group and departmental-wide potlucks.

Our experience has led to a number of observations regarding SEP authoring and instantiation. First, our template language is sufficient for specifying a wide range of useful SEPs. Second, these declarative specifications are much simpler and more concise than corresponding specifications written in Java. For instance, Table 1 displays the number of lines of OWL needed for a number of sample SEP templates vs. the number of lines of Java/HTML needed in our original prototype (which utilized process-specific procedures). Overall, the declarative approach requires about 80-90% fewer lines than the procedural approach. These figures are approximate – neither approach had a goal of using as few lines as possible – but give a flavor for the conciseness of the declarative approach.

Second, despite the usage we have seen, the group of people instantiating new SEPs seems to be much smaller than the group of people who have learned about and shown enthusiasm for the system. While some of this effect is to be expected, we also believe that a significant cause is that the system still requires too much *initial* work to launch a new SEP. In spite of SEPs' advantages, when faced with a particular data-collection task it is easier in the short-term to just send a non-semantic email message and deal with the consequences later. Essentially, originators need more *instant gratification* to motivate them to use the semantic system, as we have also described in the web context [6]. One way to address this problem would be to have authors provide basic versions of their SEPs that provide defaults for almost all parameters, akin to how modern search engines hide most of their functionality behind an "advanced" interface. An additional improvement would be to allow originators to easily modify the default parameters while a

process is executing. This both eliminates the need for upfront work and simplifies appropriate parameter selection, since the originator can delay this task until a few illustrative responses have arrived.

7 Related Work

Some hardcoded email processes, such as the meeting request feature in Outlook and invitation management via *Evite* have made it into popular use already. Our work provides a *general, declarative* infrastructure for SEPsand analyzes the inference problems it needs to solve to manage processes effectively and guarantee their outcome. Likewise, workflow and collaboration systems such as Lotus Notes/Domino and Zaplets offer scripting capabilities and some graphical tools that could be used to implement sophisticated email processes. However, these systems lack support for reasoning about data collected from a number of participants (e.g., as required to balance a potluck or ensure that a collected budget satisfies aggregate constraints). In addition, such processes are constructed from arbitrary pieces of code, and thus lack the formal properties that our declarative model provides.

Recent work on the *Inference Web* [8] 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 manager's *actions* in terms of existing responses and the expected impact on the SEP's goals. In this sense our work is more similar to prior research that sought to explain decision-theoretic advice (cf., Horvitz et al. [3]). For instance, Klein and Shortliffe [5] 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 to compute the simplest possible reason why some action could *not* be chosen (i.e., accepted). Other relevant work includes Druzdzel [1], which addresses the problem of translating uncertain reasoning into qualitative verbal explanations.

For constraint satisfaction problems, a *nogood* [9] is a reason that no *current* variable assignment can satisfy all constraints. In contrast, our notion of a sufficient 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, but this would not exploit the special structure of SEPs that ensures that a candidate explanation can be checked in polynomial time. Jussien and Ouis [4] 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.

8 Conclusions

This paper has described our template-based approach to specifying SEPs. Templates greatly increase the usability of SEPs by shifting most of the complexity of specifying a SEP from untrained originators onto a much smaller set of trained SEP authors. In addition, our template language enables authors to easily express complex goals in a way that can be automatically pursued by the manager via interventions. We also examined a number of challenges to authoring high-quality templates that arise in this context. In particular, we explored the problem of verifying that a given template will always yield a valid instantiation, described the computational complexity of this problem, and presented an approximate solution. We also discussed multiple ways in which the manager can automatically generate useful explanations for its interventions. Collectively, these techniques both simplify the task of the SEP author and improve the overall execution quality for the originator and the participants of a SEP. Future work will consider additional ways to make SEP authoring and instantiation even easier and continue to promote SEP adoption by a wide range of users.

References

[1] M. Druzdzel. Qualitative verbal explanations in bayesian belief networks. *Artificial Intelligence and Simulation of Behaviour Quarterly*, 94:43–54, 1996.

[2] O. Etzioni, A. Halevy, H. Levy, and L. McDowell. Semantic email: Adding lightweight data manipulation capabilities to the email habitat. In *Sixth Int. Workshop on the Web and Databases*, 2003.

[3] 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.

[4] N. Jussien and S. Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on on Logic Programming Environments*, Paphos, Cyprus, 1 Dec. 2001.

[5] D. A. Klein and E. H. Shortliffe. A framework for explaining decision-theoretic advice. *Artificial Intelligence*, 67(2):201–243, 1994.

[6] L. McDowell, O. Etzioni, S. D. Gribble, A. Halevy, H. Levy, W. Pentney, D. Verma, and S. Vlasseva. Mangrove: Enticing ordinary people onto the semantic web via instant gratification. In *Second International Semantic Web Conference*, October 2003.

[7] L. McDowell, O. Etzioni, A. Halevey, and H. Levy. Semantic email. In *World Wide Web*, 2004.

[8] D. L. McGuinness and P. Pinheiro da Silva. Infrastructure for web explanations. In *Second International Semantic Web Conference*, October 2003.

[9] 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.