A World Wider than the Web: End User Programming Across Multiple Domains

Will Haines
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025 USA
haines@ai.sri.com

Melinda Gervasio
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025 USA
melinda.gervasio@sri.com

Aaron Spaulding
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025 USA
spaulding@ai.sri.com

Jim Blythe
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292 USA
blythe@isi.edu

ABSTRACT

As Web services become more diverse and powerful, end user programming (EUP) systems for the Web become increasingly compelling. However, many user workflows do not exist exclusively online. To support these workflows completely, EUP systems must allow the user to program across multiple domains. To this end, we introduce the notion of pluggable domain models—independently generated action models for different application domains that can combine to support the learning of cross-domain procedures—and we present guidelines for the development of such domain models. In the context of our work on an *Integrated Task Learning* (ITL) system, we discuss how to use pluggable domain models to facilitate cross-domain instrumentation and automation. We also explore what impact such a model has on the systems that reason over, learn, and visualize procedures. Along the way, we provide prescriptive suggestions for engineering real-world cross-domain EUP systems as well as suggestions for what sorts of user activities such as system should support. Finally, we briefly discuss some open questions that cross-domain EUP systems will need to address in the future.

A WORLD WIDER THAN THE WEB

Today's rapid proliferation of Web services, particularly with the emergence of Web 2.0, has prompted an increasingly varied use of the Web to support users' everyday tasks [8]. In the office, Web services now support many business processes: travel authorization and reimbursement, equipment purchase and requisition, and conference facilities management are just some processes that often rely on dedicated Web-based applications. At home, we visit a variety of websites to purchase books, make travel arrangements, and manage our finances. However, many user workflows, particularly in business environments, still involve non-Web applications [10]. Even as some applications begin to transition to the Web—for example, email and calendar tools—the workflows will continue to involve multiple, disparate application domains. Thus, any end-user programming (EUP) tool, particularly those designed for the business environment, must accommodate procedures learned over a variety of applications, on the Web and beyond.

Consider the job of Alice, who is responsible for maintaining a website listing all the publications by the members of a university laboratory.[1] Whenever someone in the lab produces a report, they notify Alice by email. The email message contains the citation for the paper as well as an attached electronic version of the work. The attachment may be in a single-file format, such as a PDF or Microsoft Word document, or in a multi-file format such as LaTeX. In the case of multiple files, it may come as several files or as a single, compressed folder. Alice saves the file(s), and if the paper is not already a PDF, she must convert it before renaming the file to conform to a standard *YYYY-FirstAuthorLastName-Venue.pdf* format. She then uploads the PDF file using the site administrator's Web interface. This includes filling out a form with the citation information for the paper, uploading the paper, verifying that the uploaded paper is downloadable. Finally, Alice replies to the email message, copying the direct URL link to the paper into the message for the author's benefit.

This is a task Alice repeats several dozen times a year, and she would clearly benefit by automating it. However, since it touches several different applications, including an email client, the file system, word-processing software, PDF converters, and a Web browser, any EUP tool designed for a single application can only automate part of Alice's workflow. For example, Alice could use a Web EUP system to automate the segment involving uploading the paper and citation information to the website. However, she must still manually process the email, perform the file operations, provide values for the Web form, and reply to the email. Additional single-application EUP systems could potentially automate more segments, but they would require Alice not only to learn several different interfaces but also to manually link the data from one system to another. In contrast, an EUP tool that can be used across domains could potentially automate the entire workflow, providing a significantly greater benefit for Alice.

[1] This use case was adapted from a contextual inquiry user study [2] we conducted in 2007 to observe office workers performing potentially automatable tasks on their computers.

While cross-domain EUP would clearly be valuable, it also presents many design and implementation challenges. There is a clear reason why most EUP systems tackle a single application domain: it is much easier to engineer instrumentation and automation for a single platform, the relations between different domain actions are obvious, and the procedures that can be learned are bounded. Nevertheless, we argue that the benefits provided by cross-domain EUP make it well worth attempting to meet its unique challenges.

Here, we present our approach for achieving cross-domain EUP. We introduce the notion of pluggable domain models—independently generated action models for different application domains that can combine to support the learning of cross-domain procedures—and we present guidelines for the development of such domain models. We then discuss the often-underappreciated task of instrumentation and automation, noting the additional challenges that occur when learning procedures across domains. Given these pluggable domain models, we describe the various issues and opportunities raised for reasoning, learning, and visualization, grounding the discussion within our work on an *Integrated Task Learning* (ITL) system [25]. Finally, we present avenues for future work and conclusions.

CREATING PLUGGABLE DOMAIN MODELS

To get the most mileage out of EUP systems, domain knowledge must be encoded in such a way as to support reasoning across different applications. One possible approach, realized in the CALO cognitive desktop assistant, is to develop a master shared ontology for representing not just all the objects in the world and relations between them, but also the actions or tasks involving them [7]. The different applications are required to publish instrumentation events that adhere to this ontology, and the various modules can use the centralized knowledge base. Such an approach is very powerful, supporting deep reasoning over actions and objects spanning different applications [15]. However, this power comes at a very high engineering and maintenance cost. The knowledge engineers must develop an all-encompassing ontology and component developers must commit to the shared representation to model their domains. Any changes to the ontology must thus be carefully vetted to avoid unintended consequences and to avoid significant re-engineering. In a large, distributed EUP system comprising applications that are only loosely, if at all, connected, these concerns likely present an unacceptable cost. Instead, we recommend an extensible architecture that models each domain as a separate, pluggable module. In this section, we lay out the issues that arise when specifying such domain models and we present prescriptive guidelines for the development of these models.

Action-Oriented Domain Model

EUP is concerned primarily with automation, so the domain actions must be the primary focus of modeling. We prescribe a *dataflow* model of actions, where the effects of executing an action are characterized by the action's inputs and outputs. Specifically, each action is a named operation with a set of typed input and output parameters such that, in a procedure, outputs of actions serve as inputs to succeeding actions.

The dataflow model is particularly well suited to modeling Web services and service-oriented architectures in general, since services can be modeled straightforwardly as operations taking particular inputs and producing certain outputs. Moreover, many actions in the desktop world operate on artifacts such as email, files, and calendar entries and can thus also be easily cast into this modeling framework. For the remainder of this chapter, we will represent actions in the form *name [parameters]* where parameters are of the form *+|–paramName:paramType* with + indicating an input and – indicating an output. Figure 1 shows some representative actions for a Web browser and an email client.

Browser:

```
openURL +url:string
submitForm +formInputs:List<string>
–url:webAddress
```

Email:

```
openComposeEmailWindow
+sender:List<emailAddress>
+subject:string +body:string
–frameID:frameID
sendEmail +email:email
```

Figure 1: Some Possible Actions.

In Web service domains, it is often easier and more intuitive to implement instrumentation to generate events in terms of these actions rather than in terms of the changes they have on the world state. For example, uploading a paper to a Web server through a Web interface might be captured as an action that takes as input the publication information and generates as output the URL for the uploaded paper. Alternatively, it could be modeled in terms of the state of the browser window (and maybe the paper database) before the *Submit* button is pressed, and the state of the window (and maybe the paper database) after. If one were using citation information copied from an email message, then state-based instrumentation must also capture the state of the email client window. In general, state-based instrumentation must capture not just the conditions that may be affected by the current action, but also the conditions affected by previous and succeeding actions. Action-oriented instrumentation can be more narrowly focused and, at the same time, also more readily extensible.

However, it imposes the constraint that instrumentation and automation be modeled as direct inverses of each other—i.e., any observed action must also be directly executable.

Given a dataflow model of actions, a procedure learner can reason about the support relationships between the prerequisites and results of discrete end-user actions. This reasoning lets it perform procedure validation, provide editing support, and perform parameter and structure generalization [25]. Later, we discuss how we can extend our reasoning capabilities by attaching additional metadata to actions.

Modeling Human-Level Actions

When constructing a dataflow model, one of the keys to successful procedure representation and learning is to capture actions at the right level of granularity. Ideally, actions should be modeled at the level at which humans would typically describe their own actions and should expose the objects that humans would find relevant to the actions as arguments. For example, in an email application, it is preferable to model the actions `openComposeWindow`, `sendEmailAttachment`, and `sendEmail`, rather than low-level actions like *`moveMouse`* or *`leftClickOnMouse`*, or high-level actions like `sendReceiptsToAdmin`, or `sendQuarterlyReport`.

Capturing actions at low levels will generally result in much more compact action models, simplifying instrumentation and automation. However, it is likely to yield incomprehensible learned procedures—for example a procedure composed entirely of mouse drags and clicks. Meanwhile, capturing actions at too high a level will generally impose an impractical reasoning burden on the instrumentation to map what can actually be observed to the user's intent—imagine having to determine the purpose of sending an email message. Further, such high-level actions compose poorly because the user cannot break them down into smaller units should they want to realign them.

When modeling actions that match how users think of themselves interacting with applications, one is more likely to strike the right balance between the cost of instrumentation and user comprehension of the learned procedures. Such comprehension is essential if we ever expect to create systems that allow users to later modify and debug their procedures [25]. A user with a learned procedure that operates as a black box may not be any better off than a user who does not have the technical skills to read a scripting language. Modeling domain actions at human level is a service both to the learning algorithms and end users.

Beyond Actions: Modeling Objects and Relations

As discussed above, an action-oriented domain model presents a number of advantages for a cross-domain EUP system. However, judicious modeling of the objects in a domain and the relations between them can often simplify action modeling while also improving our ability to learn and reason over procedures. For example, suppose that the correct recipient of an email containing a travel expense report is the administrative assistant attached to the project that funded the trip. Without a representation for these relations it would not be possible to notice this requirement in an action trace, or represent it in a procedure. We can also use relations and properties as tests in conditional branches in procedures.

In ITL, we store relation models for each application along with the action models, preserving modularity. Technically, a relation such as "the project funding the trip" can be represented either as a relation or an operation, in this case: "look up the project funding the trip". However, there may not always be an observable user action to query for the relation. The choice of whether to use a relation or information-producing action in each particular case should be largely governed by what is more natural for users of the application who generate and edit procedures.

Referring explicitly to properties and relations of objects does require additional mechanisms to be defined to support querying for object properties or relations when a procedure is executed. Compared with an alternative approach that represents each object as a tuple of its properties, however, this approach provides two distinct advantages. First, the properties themselves may be more natural for users to view and edit in their native interface. Second, the object properties may be *mutable*—that is, they may change concurrently within the domain application. In determining whether it is the same object referred to in different actions, care must thus be taken to distinguish mutable from immutable properties and to compare only the immutable properties. Third, forcing relations between objects into properties of the object tuples is often awkward and unwieldy. In situations where these factors are significant, the advantages of representing objects as references may be well worth the additional overhead.

Extensible Type System

Recall that we define an action as taking a set of *typed* inputs and outputs. These types are used to allow the learners to make reasonable comparisons and substitutions between actions operating on compatible types of data. Figure 1 shows a variety of types, ranging from simple primitives such as `string` to more complex types such as `email`.

Much like actions, it is preferable to allow application domain models to specify arbitrary types rather than restricting model authors to a finite set of possible types. Since the type system is very important in procedures that tie together steps from several different applications, it is important that compatible information provided by one application and used in another can be identified as such. For example, both an email client and a Web browser can understand email addresses, and it is important that they both settle on a common representation. We can achieve

this agreement either by having the two domains use the same name for these object types, or by providing a central module that asserts the equivalence of the types and that perhaps contains a set of operations providing object conversion as needed. Providing shared type names or conversions does not in itself solve the problem of bridging information across multiple domains by matching types. This problem is similar to the ontology alignment or database integration problems, where a lot of work has been done [12,23]. Our general approach here has been to keep a lightweight central type system that is relatively easy to align to. ITL includes a module to automatically align types into a central system based on observed values [17].

In addition we suggest a more powerful hierarchical approach that allows application domain models to build up *complex data types* from *primitive data types* and *collection data types*. To illustrate this approach, consider the hierarchical type system supported by the ITL system. ITL allows domain models to build from complex types from string, integer, float, and Boolean primitives and list, object, and named types. More complex types are built by creating lists, which consist of typed parameters; objects, which consist of typed fields; or named types, which are types that structurally represented by another primitive or complex type but are not considered equivalent to that type. Figure 2 shows how to build an email type in such a scheme:

```
Named emailAddress string

Object email:
  List<emailAddress> recipients
  emailAddress sender
  string subject
  string body
```

Figure 2: Building the emailAddress and email types

As we can see, one can build arbitrarily complex data types out of the components, while still allowing the learning systems to reason about the internals of complex types. Another useful addition to such a type system are relations between types such as “is a” and “has a.” Such additional metadata is not strictly necessary, but may extend the reasoning ability of the learners [7].

The key to the above approach is that it parallels the specification of application domain models in that it allows domain modelers to create expressive models without forcing all model engineering to happen upfront. One can declare types for an application alongside the actions and register to allow learning in a just-in-time fashion. Thus, we can define any given *application domain model* as simply the set of actions, types, and relations that describe all user-level operations over which our learners can reason and build procedures. The *full domain model* is then just the conjunction of all actions, types, and relations from all the application domain models that we wish to consider. Hence, we can build a large full domain model without the upfront design and continuing maintenance costs required by a monolithic master ontology.

ENGINEERING INSTRUMENTATION AND AUTOMATION

Despite a great deal of clever reasoning, in the end, an EUP system is only as good as the *instrumentation* that it can reason over. Likewise, EUP execution engines are worthless without robust *automation* hooks. Unfortunately, to work in non-trivial, non-custom environments, instrumentation and automation must touch a number of applications and websites, the vast majority of which were not originally designed to support such intrusions. Given such a high cost of entry and high benefit for cross-domain EUP, we suggest budgeting a large percentage of time to handle such concerns.

The following sections provide an overview of the common engineering challenges faced by application developers who wish to attach their applications to a cross-domain EUP system. Then, using ITL as a motivating example, we provide some prescriptive guidance that may reduce the programming burden associated with instrumentation/automation.

Plugging in to an End User Programming System

Let us first consider instrumentation. In a dataflow action, collecting instrumentation consists of recording the values of the input parameters in the target application, waiting for the action to be performed, then recording the values of the output parameters and sending the whole package to the learners. One possibility is to provide the target application with a way to notify the learners when an action has been executed. In this case, we can simplify the process somewhat by using the domain model to create a registry of action notifications with accompanying containers for storing inputs and outputs. Here, we can leverage the organization of a good domain model to take much of the burden off the application programmer. However, be aware that the mapping from application functions and state to domain actions is almost never one-to-one. As such, it is key to provide the programmer with flexibility, as it can be particularly onerous to restructure application logic to fit in with a given action model.

Automation is somewhat easier for the application programmer, but it can be difficult for an EUP system engineer to provide a general-purpose framework for cross-domain automation that can collect the necessary *application context* to execute an arbitrary domain action. For example, when executing an `openURL` action, we have access to the URL to open as an action input, but in most browsers we also require other information such as a reference to the active tab. Not all context is appropriate to expose to the learners via the domain model, so we must have another way to access the context on demand. One solution to the problem is to create a callback framework that attaches the ability to execute arbitrary application code to each action in the domain model. As with

instrumentation, this approach must be flexible enough both to execute application operations and to gather all the program context necessary to allow such objects to operate correctly.

In summary, while instrumentation/automation engineering may not present grand AI challenges, it is a critical, under-appreciated issue for EUP, especially across domains. In the next section, we will provide some more detail as to how we have engineered the ITL application programmer's API to facilitate instrumentation and automation for third-party applications.

Crafting the Programming Model

In our deployment of ITL, we learned that instrumentation/automation is a consistent bottleneck in crafting a useful EUP system. Over several iterations, we have developed a few approaches that improve the programming model for associating application code with the actions to which it relates. While these approaches may not be necessary for all cross-domain applications, we stress that flexibility is the key programming concern in instrumenting/automating EUP client applications.

For instrumentation, we sought to address three key engineering concerns, *interoperability*, *immutable state* and *crosscutting*. Interoperability concerns the fact that equivalent types may be represented in heterogeneous ways across different applications. We earlier discussed the importance of lightweight type systems, and it turns out that we can leverage this concept for inter-application communication. In ITL, we settled on a canonical wire format for all data, recursively built out of primitives, lists, and maps based on the data type descriptions contained in the domain model. In this way, application programmers must only provide data conversion functions for their primitive application types and the framework can handle the rest of the conversion automatically.

Immutable State

The immutable state issue crops up because action inputs and outputs must be immutable values that reflect the state of the application before and after the user-level operation. If the operation or some application side effect mutates the item, the invariants of the action are violated, negatively impacting the learners. It is up to application developers to make sure that such erroneous mutation does not occur. In practice, defensive copying of the parameters solves this issue [3]. In ITL, we provide API support to deep-copy the parameters at the points which the application programmer takes the before and after operation snapshots of the application's state.

Crosscutting

Crosscutting refers to the tendency of some self-contained aspects (concerns) of a program to cut across a number of modules. Such code is hard to read and maintain [16]. Consider instrumentation, which requires the application developer to place a call to the EUP system wherever she wants to log program state. If something about these calls were to change, the developer would have to hunt down every occurrence that is entangled in code that is otherwise unrelated to the EUP system. With standard object-oriented approaches, it is impossible to fully encapsulate a crosscutting concern like instrumentation.

In ITL we decided to follow an aspect-oriented approach to solve this problem [16]. Aspect-oriented programming uses code execution intercept to factor out cross-cutting concerns; while it is not possible in all programming languages, it provides a clean solution to this problem for the increasing number of languages that support it. In our scheme, for each action, the application programmer need only specify methods to gather input and output states. Then, she simply provides locations at which the instrumentation will be triggered. The result keeps all instrumentation in one module and makes it easy to ship instrumented and uninstrumented versions of the client application. Even without aspect orientation, we suggest keeping the gathering methods in a single module and limiting the penetration of calls to this module from other modules to a minimum.

Context Gathering

As described earlier, the major problem of automation is *context gathering*—i.e., ensuring that the necessary program context is available for executing the action. We suggest *abstract factory pattern* as an elegant solution to this issue [13]. In ITL, each action uses an abstract factory to create a context object that knows how to gather context and execute the necessary code to make execution successful. In this way, one only needs to provide the template for gathering context rather than attempt to pass the context to each callback explicitly.

The ITL approach certainly is not the only method to instrument and automate a number of heterogeneous applications, but we feel that it demonstrates a number of engineering best practices for instrumentation/automation. We hope that by sharing some patterns for making this difficult process easier, we can allow EUP systems to gather more data and focus on better serving the user. Now that we have facilitated appropriate instrumentation and automation, we can focus on some interesting learning issues.

LEARNING CROSS-DOMAIN PROCEDURES

The action-oriented dataflow paradigm both presents new opportunities for learning and affords useful information that can help in the learning process. In this section, we describe various issues that arise in learning dataflow procedures across domains and present the solutions we have explored thus far within ITL.

Integrating Web Services and Other Data Sources

By representing actions in terms of their inputs and outputs, we can naturally represent procedures learned over them as higher-level actions with inputs and outputs. A beneficial

consequence of this is that we can integrate Web services and other action-oriented data sources in whole or in part into ITL. For example, if instrumentation and automation are provided at the level of the browser operations within a Web service for providing driving directions, ITL could be used to learn a procedure to drive the browser interaction. Alternatively, instead instrumentation and automation could be at the level of the Web service itself, modeling the procedure as a *single* action taking in the origin and destination addresses and providing the driving directions URL as output. This could use the Web service API directly or some intermediary such as the execution component of some other learning system tailored specifically to that Web service. This flexibility allows ITL to learn at the primitive action level but also at the level of procedures learned by other components. As long as the other learner creates an action with inputs and outputs, ITL can incorporate it into larger dataflow procedures.

To support such composability, it is critical that the inputs and outputs of the different services or learned procedures be semantically aligned. One approach is to omit semantic typing and annotate parameters with only their basic data—for example, type the parameter to a browser navigation command as a string rather than as a URL. While this approach will work, it leads to an explosion in the search space for matching parameters and to inefficient learning. Thus, as discussed earlier, instead we advocate the use of a lightweight type system into which the inputs and outputs of the different actions can then be mapped. In the past, we have used the semantic mapping component of PrimTL [17] to integrate new data sources. However, conceptually, other techniques for ontology alignment can be used to relate inputs and outputs of different services.

Programming by Demonstration

Programming by demonstration (PBD) or programming by example has been a popular EUP paradigm since its introduction a few decades ago [9,19]. PBD is a particularly attractive methodology for nontechnical end users because it relies on a very natural form of interaction—demonstration—that requires minimal input from the user. Recent years have seen resurgence in enhanced PBD approaches as adaptive AI systems have begun to tackle the acquisition of complex workflows (e.g., [1,6]).

Within the dataflow paradigm, we can characterize the basic learning task as one of generalizing a demonstration comprising a sequence of executed actions into a procedure that can be used to achieve the same task in future similar situations. There are two basic aspects to generalization: 1) *parameter generalization* to essentially convert observed constants into variables and 2) *structure generalization* to induce procedural structure over the observed straightline sequence.

A dataflow-oriented action model introduces a number of more specific issues for learning procedures from demonstration. First is *dataflow validation*—ensuring that every input is supported by a previous output. Second is the related issue of parameter generalization through expression formulation—essentially, determining how to replace constants not simply with variables but with functional expressions over previous variables. The third issue is the induction of loops over collections of objects, in contrast to counting loops or while loops.

In ITL, the PBD capability is provided by the LAPDOG procedure learning component [14,15]. LAPDOG was designed specifically to learn dataflow procedures, addressing each of the issues above while taking advantage of the inherent structure provided by the action-oriented data model discussed previously. We now present each of the issues in dataflow procedure learning in turn, discussing our approach to handling them in LAPDOG and remaining open problems.

Dataflow Validation

To be executable, the inputs of every action in a dataflow procedure must be supported by previous outputs. In the simplest case, an input is directly supported by an output—i.e., they are the same value. A slightly more complex case involves inputs that can be supported by expressions over previous outputs; this is discussed further in the next section.

The most interesting case arises when no such directly or easily derivable supports can be found based on the observed demonstration. Discounting the situation where this occurs due to insufficient instrumentation, missing supports may occur due to unobservable mental actions that the user performs in the process of accomplishing a task. For example, a user might search for all Italian restaurants in a city and then proceed to email the names and links for only the five-star-rated ones. However, all that a PBD system will observe is that the user sent out some information from some subset of the list of restaurants. The fact that it was the five-star-rated subset is something that needs to be inferred.

In LAPDOG, we address this problem through two main techniques for dataflow completion. The first involves a heuristic search in the space of possible relations within a knowledge base [15]. The second might be characterized as planning in the space of information-producing actions, such as string manipulation operations, named entity extractors, and classification operations [14]. While the first involves search over a relatively static knowledge base, the second involves search over dynamically generated data. Both techniques leverage aspects of the domain model prescribed earlier: the first involving relations between objects accessible through some query mechanism and the second involving non-observable but executable information-producing actions.

Extended Parameter Generalization

In a specialized case of inferred relations between known outputs and required inputs, we can consider accessor and

construction operations over lists and tuples. LAPDOG utilizes unlimited tuple field access and limited list element access. Specifically, individual values may be supported by any field value of a tuple but only by the first or last element of a list. The rationale is that while the individual fields of a tuple as well as the first and last elements of a list are meaningful, the other elements of a list are rarely so. LAPDOG also utilizes list and tuple construction, allowing list and tuple values to be supported by constructor operations over values matching their constituent parts.

Loop Induction

Within the dataflow paradigm, one of the most common loops involves a loop over the elements of a collection (i.e. a set or a list). In the case where the loop is over a collection that is explicitly observed in the demonstration (e.g., the output of a previous action), we can leverage this information to detect loops. Intuitively, if we can find a similar sequence of actions operating over each element of the collection, we can induce a loop. In LAPDOG, we leverage this information to find loops over collections, where the loop body is identical over all iterations [11].

A more interesting situation arises when loops occur over collections that are not explicitly observed but can be inferred from previous outputs. A simple case of this involves a sorting operation on a list to generate another, potentially differently ordered, list. For example, a set of person names may be sorted alphabetically while a list of employee IDs may be sorted in increasing numeric order. A straightforward extension involves the application of predefined actions that generate lists. For example, a travel system might have an operation that takes a set of travel authorization requests and outputs the list awaiting approval or another that takes an employee ID and a date range and outputs the list of all travel within that specified dates. Inserting these kinds of actions into the learned procedure is a natural extension of LAPDOG's dataflow completion capabilities [14,15].

Another interesting situation involves a loop that uses the accumulated outputs of a previous loop. For example, imagine an administrative assistant making online hotel reservations registrations for a number of people and then emailing each person with their reservation confirmation ID. This presents an interesting learning challenge because it requires the preceding loop to be learned in order to determine that it will generate the list that the succeeding loop needs [11].

Combining the notion of having to infer a collection of objects and accumulating a new list within a loop, we get the situation where we have a loop over a collection for which the learning system must propose a loop that will generate the required list from a previously known list. For example, imagine needing a list of employee IDs. Given a list of employee records, we could generate this list by looping over the employee records and collecting the employee ID from each.

VISUALIZING CROSS-DOMAIN PROCEDURES

End-user programming is fundamentally a programming task and, as such, it inevitably involves abstraction. For non-programmers, dealing with abstract procedures is difficult because such users tend to think of programs as the set of concrete actions that end users experience at runtime, rather than as more general abstract control structures [22]. Rode and Rosson demonstrated this difficulty in the Web domain and, from our deployment of ITL, we also conclude that in complex, cross-domain environments, the user's need to understand abstract procedures is both vital and difficult to support [24]. Fortunately, by augmenting pluggable domain-models, we can support the user without having to know about every domain in advance.

To leverage the end user's tendency to conceive of procedures in terms of runtime actions, we can combine an appropriately abstract domain model with human-readable annotations to make action specifications more concrete. First, remember that we recommend defining the domain model in terms of atomic user interactions. This level of abstraction affords a straightforward mapping from action to a human-readable *displayed step* that reflects an atomic GUI interaction with the end user. In ITL, we implemented this mapping using a *template* approach that adds metadata to domain model actions to specify a human-readable display as well as to specify pointers domain application properties that must be queried for missing display information. Figure 3 illustrates this approach.

Raw Source:

```
openComposeEmailWindow
+sender:["haines@ai.sri.com"]
+subject:"Explaining Templates"
+body:"An explanation" -frameID:ID12345
```

Action Template:

```
openComposeEmailWindow := Opened window: $frameID
```

Apply Action Template:

```
Open window: ID12345
```

Data Type Template:

```
frameID := Compose $frameID.subject
```

Apply Data Type Template:

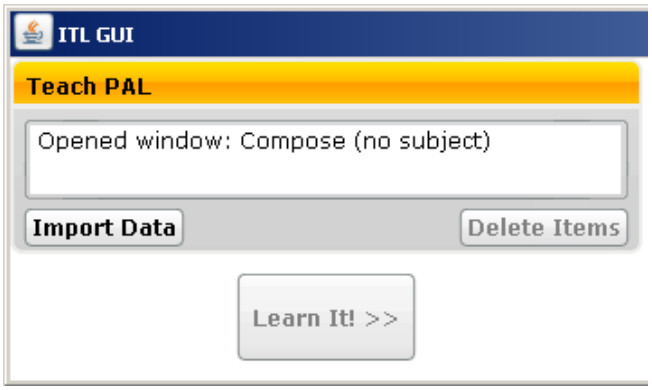

Figure 3. Action Metadata Application—Before and After

The action template indicates that the `openComposeEmailWindow` action should display as "Open window:" concatenated to the display value of the the `frameID` parameter, which in this case refers to an application property, the identifier of the window in which the email will be composed. Next, we include a template for the `frameID` data type, which queries the application to find an application-specific representation—in this case, the subject displayed in the frame. If the `frameID` instead happened to be a variable, we would instead display just the variable's name.

Also important to note is that this template does not present all arguments of an action to the user—in particular, the input arguments of the `openComposeEmailWindow` action are never shown. Our research indicates that some parameters simply complicate a user's understanding of the overall procedure flow [25]. For example, though the procedure executor might need to know screen pixel positions, such information is irrelevant to most end users. As such, adding the ability to suppress parameters and even entire actions to a "details" view is another simple way to improve user comprehension of complex procedures. Further, users may not perform a certain demonstration perfectly, making and correcting mistakes along the way. Indicative of this are certain combinations of actions that negate each other, such as a file being open then immediately closed, or an email compose window being created and edited but not saved or sent. These could also be hidden to simplify the event trace.

MODIFYING CROSS-DOMAIN PROCEDURES

A complete framework for end-user programming should support editing of procedures as well as their learning by demonstration. Given an understandable representation of their procedures, users want to make changes that cover a range of complexity, from changing constant parameters in steps to adding conditions and iterative loops. Simple edits may often be required when the task to be performed by an existing procedure changes slightly or to correct an initial hypothesis from another learning component. Support for multiple domains increases the chance that users will also need to add new steps to procedures, modify step ordering, or change the structure of the procedure. This is because domains may be more or less reliant on a graphical interface, where demonstration-based techniques are natural, leading the user to supplement demonstration by choosing available actions from a menu or describing them, and composing within an editor.

In the dataflow-oriented model, full user support for editing poses many of the same challenges faced by demonstration-based learning. For example, users may insert an action but omit auxiliary steps or queries that provide inputs for that action. In a dataflow model, those missing steps must themselves make use of inputs that are established earlier in the procedure. The use of typing in our domain specification allows us to cast the problem of inferring missing steps as compositional search over a graph of data types in which queries or steps are composed to form a path from existing inputs to those that are needed.

An editing tool for a typed dataflow model should provide at least two kinds of support. First, it should provide editing support for users, not only to add primitive steps, but also to add conditions or loops by suggesting candidates based on queries and lists that are available. Second, it should warn the user if the edited procedure misses critical inputs or otherwise has potential flaws and should use dataflow information to suggest potential fixes. A third, desirable characteristic is to allow users to copy steps between procedures to facilitate best practices, while using the dataflow model to ensure the resulting procedure is executable.

In ITL, Tailor is used to provide a procedure editing capability [4,25]. Tailor exhibits all these desired characteristics, as we describe below. Tailor allows users to add or delete steps, add conditions and iterative loops, and to copy steps between procedures. It searches over possible queries and actions arranged in the same space to find plausible missing steps, composing steps and queries if needed.

Support for Adding Conditions and Loops

Tailor uses compositional search over a graph of data types to infer missing steps or queries when users add steps. In general, however, users find the process of adding a brand new step difficult and do not perform it often, preferring to copy or move steps. The same search technique, however, can support a wide range of activities, including copying steps, generating potential fixes for flaws, and, as we describe here, adding conditions or loops.

When the user invokes Tailor in ITL, she may choose to add a condition or loop around a set of steps without providing any information about the condition or loop. This lack of specificity simplifies the interface and reduces the cognitive burden on the user, who may find it difficult to specify a conditional or loop without assistance. Tailor uses compositional search along with heuristics to generate a set of reasonable candidate specifications. Once Tailor arrives at a set of candidates for a new action, condition, loop or a change to a parameter value, the user interface can present them as options. Here it is critical that the user can understand both the current procedure and the available alternatives in order to make a reasoned choice. The alternatives are displayed within the procedure visualization described above and should use similar templates to provide a uniform view. By presenting the user with appropriate bounds, we make it easier to create complex control structures and limit the user's capacity to make errors.

Support for Editing Errors or Flaws

Nevertheless, users still often make errors when editing procedures. After the user makes a modification, Tailor checks a procedure for simple errors, for example if a step

has been deleted although it produced a value that was used later in the procedure [4]. To do this check, Tailor performs a symbolic analysis of the procedure, aiming to find important errors before the procedure is executed. This means that it does not know, for example, which of several conditional branches may be taken during execution or how long a loop will be followed. ITL's execution engine is also capable of interleaving many concurrent actions, and this means that one cannot prove that global variables will be unavailable when a step is to be run [20]. Because of this, Tailor only provides a warning for an unbound global variable at the time that a modification removes or reorders a step or query that provides a value.

For each warning, Tailor uses templates to provide a set of potential fixes that may include reordering steps, removing them, or undoing user's last edit. In some cases, modifications requiring several coordinated edits can be made by picking one edit and choosing the appropriate recovery steps. Further, Tailor can use compositional search to suggest steps that may be added to provide missing inputs.

Support for Copying Steps Between Procedures

Our user interviews revealed that users frequently desire the ability to copy steps from a previously learned procedure to a new one [25]. This request makes sense; by copying all or part of a procedure, users can reuse long demonstrations or complex constructs, such as conditions and loops. The procedures learned in ITL use no global variables, so the variables in the steps that are copied must be replaced by terms in the target procedure, either by (1) changing them to an existing variable, (2) changing them to a constant, or (3) adding auxiliary steps to establish a new variable. Tailor finds potential replacements of all kinds using the same compositional search technique [5]. This method naturally prefers to use an existing variable or constant for each copied variable, as this leads to a shorter solution. We extended this capability to enable copying sequences of steps, by composing the variable mappings of the component steps. We also added domain-specific heuristics that replace variables with constants when the intended value is known.

DISCUSSION AND FUTURE WORK

Clearly, there are both large benefits and considerable costs associated with an extensible cross-domain EUP system such as ITL. We have explored in detail some of the concerns associated with creating such a system, but there are a number of other challenges and potential benefits that we have not explored in detail to date. Here we briefly discuss a few of the issues that we hope to explore.

Consistency in a Heterogeneous Environment

A widely recognized design principle, consistency [21] is difficult enough to achieve in an unregulated environment like the Web. When attempting to integrate Web applications with desktop applications, the concept of consistency becomes even more vague. One option is to return to the native application to edit procedure parameters. While this leverages users' familiarity with that application and makes sense for certain dialogs (such as *save as* options) it is problematic for other operations like defining loops. A second option, managing editing operations entirely within the EUP tool raises new questions. Should the EUP system follow platform conventions, Web standards, or some other standard entirely? An extensible visualization system like the one in ITL should allow us to test various approaches with end users, but there are currently no clear answers.

Supporting Procedure Reuse

In addition to reusing one's own procedures, an EUP system should support users sharing procedures. Given that making procedures understandable to the author is difficult, making them understandable to others is even harder. This problem is compounded when there is a wide range in the computational literacy of the user population. Advanced users may be comfortable with complex structures, such as conditionals and iteration, which may confuse novice users who attempt to take advantage of shared procedures. A simple improvement that we have explored is to create a means for users to explicitly define arbitrary steps within a procedure and to enter descriptions summarizing the procedure and individual steps. Similar to comments in code, this metadata can help users understand and evaluate shared procedures; however, they will only be useful if users are motivated to add them to their procedures.

Another issue that arises with shared procedures is that a given procedure may contain certain types of personal data, such as names, emails, and mailing addresses. These types of information will need to be identified and personalized in order for a user to take advantage of a shared procedure. Creating a personal data store for these data types, as CoScripter (formally Koala) does [18], may help to avoid confusion.

SUMMARY AND CONCLUSIONS

In this chapter, we discussed the benefits of implementing a cross-domain EUP system as well as the unique challenges associated with such an endeavor. Using our experience building the cross-domain ITL system, we recommend building an action-oriented set of pluggable domain models. Leveraging such a model, we see that we can reduce the burden of instrumentation and automation as well as support the learning of, reasoning over, and visualization of cross-domain procedures. By modeling the world around us in a modular, extensible way, we can better allow end users to automate their workflows on the desktop, the Web, and perhaps beyond.

ACKNOWLEDGMENTS

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-07-D-0185/0004. Any opinions,

findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), or the Air Force Research Laboratory (AFRL).

REFERENCES

1. Allen, J., Chambers, N., Ferguson, G., et al. Plow: A collaborative task learning agent. *Proceedings of the National Conference on Artificial Intelligence*, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999 (2007), 1514.
2. Beyer, H. and Holtzblatt, K. *Contextual design: defining customer-centered systems*. Morgan Kaufmann, 1998.
3. Bloch, J. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, 2008.
4. Blythe, J. Task learning by instruction in Tailor. *Proceedings of the 10th international conference on Intelligent user interfaces*, ACM New York, NY, USA (2005), 191-198.
5. Blythe, J. and Russ, T. Case-based reasoning for procedure learning by instruction. *Proceedings of the 13th international conference on Intelligent user interfaces*, ACM New York, NY, USA (2008), 301-304.
6. Burstein, M., Laddaga, R., McDonald, D., et al. POIROT–Integrated Learning of Web Service Procedures. *Proc. AAAI*, (2008).
7. Chaudhri, V.K., Cheyer, A., Guili, R., Jarrold, B., Myers, K.L., and Niekrasz, J. A Case Study in Engineering a Knowledge Base for an Intelligent Personal Assistant. *the Proc. of the 2006 Semantic Desktop Workshop, Athens, GA*, (2006).
8. Cockburn, A. and McKenzie, B. What do web users do? An empirical analysis of web use. *International Journal of Human-Computer Studies 54*, 6 (2001), 903-922.
9. Cypher, A. and Halbert, D.C. *Watch What I Do: Programming by Demonstration*. MIT press, 1993.
10. Dragunov, A.N., Dietterich, T.G., Johnsrude, K., Mclaughlin, M., Li, L., and Herlocker, J.L. Tasktracer: a desktop environment to support multi-tasking knowledge workers. *Proc. IUI*, (2005), 75-82.
11. Eker, S., Lee, T., and Gervasio, M. Iteration Learning by Demonstration. *Proc. AAAI 2009 Spring Symposium on Agents that Learn from Human Teachers*, (2009).
12. Euzenat, J. and Valtchev, P. Similarity-based ontology alignment in OWL-lite. *ECAI*, (2004), 333.
13. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Reading, MA, 1995.
14. Gervasio, M., Lee, T.J., and Eker, S. Learning email procedures for the desktop. *Proc. AAAI 2008 Workshop on Enhanced Messaging*.
15. Gervasio, M. and Murdock, J. What Were You Thinking? Filling in Missing Dataflow Through Inference in Learning from Demonstration. *Proc. IUI*, (2009).
16. Kiczales, G., Lamping, J., Mendhekar, A., et al. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*. 1997, 220-242.
17. Lerman, K., Plangprasopchock, A., and Knoblock, C.A. Semantic labeling of online information sources. *International Journal on Semantic Web & Information Systems 3*, 3 (2007), 36-56.
18. Leshed, G., Haber, E.M., Matthews, T., and Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise. (2008).
19. Lieberman, H. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers San Francisco, 2001.
20. Morley, D. and Myers, K. The SPARK agent framework. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, IEEE Computer Society Washington, DC, USA (2004), 714-721.
21. Nielsen, J. and Molich, R. Heuristic evaluation of user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, ACM New York, NY, USA (1990), 249-256.
22. Pane, J.F., Ratanamahatana, C., and Myers, B.A. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Human-Computer Studies 54*, 237 (2001), 264.
23. Parent, C. and Spaccapietra, S. Issues and approaches of database integration. *Commun. ACM 41*, 5es (1998), 166-178.
24. Rode, J. and Rosson, M.B. Programming at Runtime: Requirements and Paradigms for Nonprogrammer Web Application Development. *IEEE Symposium on Human-Centric Computing Languages and Environments*, (2003).
25. Spaulding, A., Blythe, J., Haines, W., and Gervasio, M. From Geek to Sleek: Integrating Task Learning Tools to Support End Users in Real-World Applications. *Proc. IUI*, (2009).