Property Inference from Program Executions [*]

Richard M. Chang
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
rchang@cs.umass.edu

George S. Avrunin
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
avrunin@cs.umass.edu

Lori A. Clarke
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
clarke@cs.umass.edu

ABSTRACT

Software verification techniques require properties that define the intended behavior of a system be specified. Generating such properties is often very difficult and serves as an impediment to the adoption of verification techniques. Techniques that leverage program executions to infer these properties are a promising avenue for automatically generating these properties. In this paper, we propose a property inference approach that leverages event traces derived from program executions to efficiently infer properties that are subtle variations of commonly occurring properties. We define *inference templates* that represent sets of these properties and describe our inference algorithm that refines these templates based on event traces.

1. INTRODUCTION

Traditionally, verification has been done by first specifying the properties to be proven and then using formal theorem proving, finite-state verification, or other such approaches to determine if the system is consistent with the specified properties. Although all parts of the verification effort can be quite demanding, it is often difficult to formulate the properties correctly or to determine all the properties that should be stated or proven about a system.

In this paper, we propose an approach that uses system executions to automatically infer properties that can then be used for verification, testing, or documentation. Our approach builds upon techniques for helping developers specify properties [2, 6]. It relies on the assumption that, although developers may not be adept at accurately specifying all the details of a system, they have knowledge about its intended behaviors that could be leveraged. Our goal is to develop a property inference approach that is able to infer properties representing these intended behaviors from program executions using limited initial guidance from developers. This approach should be automatic, efficient, and able to infer a wide variety of properties.

Previous approaches to infer properties about software systems from their program executions have typically either focused on learning arbitrary finite-state automata [1] or on inferring properties that are derived from a relatively small set of candidate property patterns [7]. While both of these approaches have been shown to be useful for inferring certain kinds of properties, they have some serious limitations. Learning arbitrary automata can result in properties containing a large number of states, making it difficult to determine if the learned properties represent the intended behavior of the system or not. On the other hand, focusing on a small set of candidate property patterns can greatly restrict the types of properties that can be inferred. One of the goals of this work is to develop an inference technique that focuses on commonly occurring patterns, but allows many variations of these patterns to be automatically inferred efficiently.

We propose an approach that builds upon the property patterns work [2], which identified a set of commonly occurring property patterns, and the Propel property pattern templates [6], which explicitly describe the alternatives that can be associated with each of these patterns. Our work further extends the Propel property templates to define *inference templates* that are then used by our inference algorithm. This algorithm, called *template refinement*, works by processing event traces gathered from program executions. As these event traces are processed, the algorithm refines these inference templates to restrict the set of properties represented by these templates to only properties that are satisfied by the set of currently processed event traces.

There are several expected benefits to this method of property inference. First, the inference templates we define are based on commonly occurring patterns, which means our inference focuses on the most likely candidate properties. Second, by defining these templates such that each one initially represents a large number of structurally similar candidate properties, we are able to infer many subtle variations of these patterns. Third, the refinements performed by the inference algorithm are event trace driven, which ensures that we return the strictest, or most refined,

[*]This material is based upon work supported by the National Science Foundation under Award No. CCF-0427071, the U. S. Army Research Office under Award No. DAAD19-01-1-0564, and the U. S. Department of Defense/Army Research Office under Award No. DAAD19-03-1-0133. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the U. S. Army Research Office or the U. S. Department of Defense/Army Research Office.

templates that are consistent with the set of processed event traces. Finally, the inference template representation allows us to compactly represent a variety of similar properties. This representation should allow our inference approach to be efficient. We also expect that this representation to be amenable to manual and automated analysis. In particular, we plan to investigate how these inference templates can be used to support an extension to invariant-based test data selection [5] and verification.

The rest of this paper is organized as follows. The next section gives a brief overview of the property inference approach. Section 3 presents the inference templates that are used by our inference algorithm. Section 4 describes the refinement algorithm that is used to infer properties from event traces. Section 5 compares our inference approach with related work. Finally, section 6 discusses the status of this work and some interesting future directions.

2. OVERVIEW OF OUR APPROACH

Each inference template used in our approach represents a set of finite-state automata. The basic idea behind our approach is to iteratively refine each inference template such that each refinement removes automata from the set represented by that template. These refinements are constructed to ensure that the set of automata removed by each refinement consists of those automata that cannot accept the event trace currently being processed. In subsequent sections, we describe these refinements and how we can generate the set of automata, called *concrete properties*, represented by a template.

Our approach defines a single inference template for each type of property to be inferred. These property types are derived from the property patterns work [2]. For example, the property patterns identified a commonly occurring property pattern called Response, which is parameterized by two events called **action** and **response**. A Response property states that whenever an instance of the **action** event occurs, it is subsequently followed by an instance of the **response** event. There are many variations of this property type. In addition to allowing the **action** and **response** events to be bound to many different event pairs, there are several allowable variations regarding the sequences of **action** and **response** events, as described by [6]. For instance, the initial state may or may not be accepting and multiple instances of the action event may or may not be allowed to happen before the response event occurs. The Response inference template we define represents a set of concrete properties that are variations of the Response property pattern.

3. INFERENCE TEMPLATES

Inference templates used in our approach build upon the property templates defined in [6]. Inference templates are similar to finite-state automata but, to allow the representation of sets of automata, have the following extensions: abstract events, concrete events, optionally accepting states, optional transitions, abstract labels, and multi-labels.

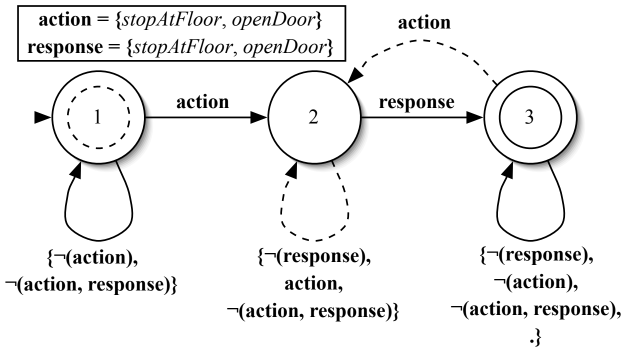
Rather than having a single event alphabet, inference templates have two event alphabets, a concrete alphabet, which consists of concrete events, and an abstract alphabet, which consists of abstract events. Concrete events correspond to events from program executions, such as method calls or variable accesses. An inference template's concrete alphabet determines the alphabet of the concrete properties the inference template represents. Abstract events parameterize an inference template and each is associated with a set of elements in the template's concrete alphabet. We refer to this set of concrete events as an abstract event's *potential bindings*. An abstract event's potential bindings set is mutable and can be modified by removing concrete events from the set. These two alphabets and potential bindings enable a single inference template to represent many concrete properties. As noted above, a Response inference template has two abstract events called **action** and **response**.

States of an inference template may be accepting, non-accepting, or optionally accepting. Optionally accepting states are states that may be treated as either accepting or non-accepting. Similarly, inference templates are also defined to have optional transitions, which are transitions that may be treated as regular transitions or completely removed from the template.

Transitions in inference templates may be labeled with either *abstract labels* or *multi-labels*. Each multi-label consists of a set of alternative abstract labels. There are three types of abstract labels: *simple labels* denoted by a single abstract event, *complement labels* which are denoted by a complement operator, ¬, followed by a comma-separated list of abstract events enclosed in parentheses, and *universal labels* which are denoted by **.**. We say that a simple abstract label is a *matching label* for a given concrete event if and only if, the potential bindings of the label's abstract event is a singleton set containing only the given concrete event. We say that a complement label is a matching label for a given concrete event if and only if the concrete event is not contained in the potential bindings of any abstract events in the list following the complement operator. We also say that an universal label is a matching label for any concrete event. A similar definition exists for multi-labels. We say that a multi-label is a matching label for a given concrete event if and only if all abstract labels the multi-label's set are matching labels for the concrete event.

3.1 Example Response Inference Template

Figure 1 displays an example of a Response inference template. We use this inference template as a running example to illustrate different aspects of our property inference approach.



action = {stopAtFloor, openDoor}
response = {stopAtFloor, openDoor}
action
action
response
1
2
3
{¬(action), ¬(action, response)}
{¬(response), action, ¬(action, response)}
{¬(response), ¬(action), ¬(action, response), .}

Figure 1: Response Inference Template

In this example figure, optionally accepting states are denoted by states with dashed inner concentric circles. The start state of this inference template is an example of an optionally accepting state. Optional transitions, such as the self-loop transition on state 2, are denoted by dashed lines. Multi-labels are denoted by braces enclosing a comma-separated list of abstract labels. Abstract labels are denoted as described in the previous section. The abstract alphabet of this inference template consists of **action** and **response**. The rectangular box above the inference template displays the potential bindings for the abstract event **action** and the abstract event **response**. The concrete alphabet of this inference template is not explicitly given here, but we know that it must contain the concrete events *stopAtFloor* and *openDoor* because these are the potential bindings for **action** and **response**.

The self-loop transition on the start state is labeled with an example of a multi-label. This multi-label consists of two abstract labels, $\neg$**(action)** and $\neg$**(action, response)**. If we assume that the inference template's concrete alphabet contains a concrete event called *closeDoor*, then the multi-label on the self-loop transition is a matching label for *closeDoor* because that concrete event is not contained in the potential bindings of **action** or **response**.

The transition from the start state to state 2 is labeled with an abstract label denoted by **action**. This abstract label is not a matching label for any of the elements of the inference template's concrete alphabet because its set of potential bindings is not a singleton. If we were to modify the potential bindings for **action** by removing *openDoor*, then the abstract label **action** would be a matching label for *stopAtFloor* because **action**'s potential bindings would now be a singleton containing only *stopAtFloor*.

3.2 Concrete Property Instantiation

As stated eariler, each inference template represents a set of concrete property finite-state automata. These concrete properties can be generated from an inference template by systematically making choices with respect the extensions to finite-state machines defined by inference templates. Next, we use our example inference template in Figure 1 to illustrate this process.

In our example inference template, we can see that **action** and **response** each have two potential bindings. We disallow concrete properties where multiple abstract events are bound to the same concrete event. This means for our inference template that there are two valid bindings possible: one where **action** is bound to *openDoor* and **response** is bound to *stopAtFloor*, and another where **action** is bound to *stopAtFloor* and **response** is bound to *openDoor*.

For each of these bindings, there are many concrete properties that can be generated by making decisions with respect to optionally accepting states, optional transitions, and multi-labels. These decisions include whether or not the start start is accepting, whether or not the transition from state 3 to state 2 exists, and choosing an abstract label from the multi-label on the start state. Notice that if we want to generate only deterministic finite state automata, there are local dependencies among some of these decisions. An example of such a dependency is if the optional transition from state 3 to state 2 exists, then the self loop transition on state 3's multi-label cannot include the all abstract label or the complement of **response** abstract label. As shown later, our refinement algorithm handles such local dependencies using a post-processing step for each refinement. If we apply this concrete property instantiation to the Response inference template in Figure 1 with the given potential bindings for **action** and **response**, then the total number of concrete properties instantiated is 192. This means that this single inference template compactly represents 192 concrete properties.

4. TEMPLATE REFINEMENT

Our template refinement algorithm begins with a set of event traces gathered from program executions and an initial set of inference templates. Each event trace consists of a sequence of concrete events. Our approach allows for user guidance with respect to the initial set of inference templates. Users may specify the type of properties to be inferred and may also specify the initial potential bindings for each abstract event associated with a template.

We call the current set of inference templates to be refined the *candidate inference template set*. As illustrated below, this set can grow, when multiple possible refinements for a transition and event are possible, or shrink, as new events make some inference templates invalid. An *invalid inference template* is a template that does not represent any concrete properties that are consistent with the set of traces currently processed.

Our template refinement algorithm proceeds by processing each event trace in the input set of traces once and performing refinements on each inference template in the candidate inference template set as we go. For each inference template in our candidate set, we keep track of its current state. When we begin to process a trace, each inference template's current state is set to its start state. For each concrete event e in the current event trace, we refine each inference template T in our candidate set based on e.

The process of refining an inference template based on its current state and the event e proceeds as follows. First, we examine all outgoing transitions from the current state of T. For each transition, we check if it is a *potentially matching transition* of e. A potentially matching transition is a transition whose label either currently matches e, called a *matching transition*, or it is a transition for which there exists a refinement of T such that the transition matches e after such a refinement. If there are no potentially matching outgoing transitions, we simply mark this template as invalid and remove it from the candidate inference template set. If there is a single potentially matching transition we refine the inference template based on this transition and the concrete event e. For each potentially matching transition beyond the first, we generate a new copy of the inference template, add it to our candidate set, and refine this copy based on the concrete event e and the potentially matching transition. When there are multiple possible refinements to convert a potentially matching transition into a matching one, we also generate new copies of the inference template. Each of these copies is added to our candidate set, and we perform a unique refinement from set of the possible refinements on each copy.

After each refinement has occurred, we must check if the event most recently processed is the last event in a trace. If this is true, we must examine the updated current state of the inference template. If this state is optionally accepting, we convert it into an accepting one. If this state is accepting then we do nothing because this indicates that all the concrete properties represented by the template accept the event trace we have just finished processing. If this state is non-accepting, this means that all concrete properties represented by this inference template reject the event trace we have just finished processing. This implies that the inference template is invalid and must be removed from the candidate set.

When we start to process a new trace, every inference template in our candidate set has its current state updated to its start state and this refinement algorithm begins processing events in the new trace. This iterative process continues until our candidate set become empty, which means that all inference templates in our initial candidate set have become invalid, or until we have processed all traces in our set. If we have finished processing all traces and our candidate set is non-empty, then our candidate set can be returned to a user. These inference templates can be either shown to a user or the set of concrete properties represented by these templates can be explicitly generated.

4.1 A Refinement Example

In this section, we present an example that illustrates the refinements that our algorithm performs when potentially matching transitions are identified. There are two basic kinds of refinements, *potential binding refinements* and *multi-label refinements*. Potential binding refinements reduce the number of concrete properties represented by an inference template by removing concrete events from an abstract event's potential bindings set. Multi-label refinements reduce the number of possible different transitions that may be generated from a multi-label by removing abstract labels from a multi-label. Both types of refinements update an inference template's current state to the destination state of the refinement's transition.

If the potentially matching transition that our algorithm has identified is already a matching transition, we do not perform any refinements on the inference template. We simply update the current state of this inference template to be the destination of the matching transition, and the inference template remains in our candidate set to be refined when the next trace event is processed. If the multi-label or abstract label on the transition is not a matching label, then we must perform a refinement to convert the transition's label into a matching label. Our refinement strategy is dependent on the type of label.

We now consider refining our example Response inference template in Figure 1. We assume that the inference template's current state is its start state. Furthermore, we assume that the current concrete event being processed is *stopAtFloor*. There are two potentially matching transitions for this state and concrete event. The transition from the start state to state 2 and the self-loop transition on the start state are potentially matching transitions for *stopAtFloor*. Neither of these transitions is a matching transition for *stopAtFloor*, therefore, we must refine this inference template based on each of these transitions and the concrete event *stopAtFloor*. Because there are two potentially matching transitions, we create two copies of our inference template, each corresponding to a different potentially matching transition. In the figures that illustrate performing refinements, each current state and potentially matching transition that is focus of the refinement are denoted by bold lines.

First, we consider performing a refinement on this example inference template based on the transition from the start start to state 2 and the concrete event *stopAtFloor*. This transition is labeled with a simple abstract event **action**. Because the transition's label is an abstract label, we cannot attempt to perform a multi-label refinement, and we must perform a potential binding refinement.

Potential binding refinements are defined in the following way. If the abstract label is a simple label, then we remove every event other than given concrete event from the potential bindings of the label's abstract event. If the abstract label is a complement label, then we remove the given concrete event from the potential bindings of each abstract event associated with the complement label. This refinement is not necessary for universal labels because by definition they are matching labels for any concrete event. It is clear that this refinement converts potentially matching abstract labels into matching ones. If an abstract event's potential bindings is a singleton set and a potential binding refinement removes the label from this set, then the inference template becomes invalid, and it is removed from our candidate inference template set.

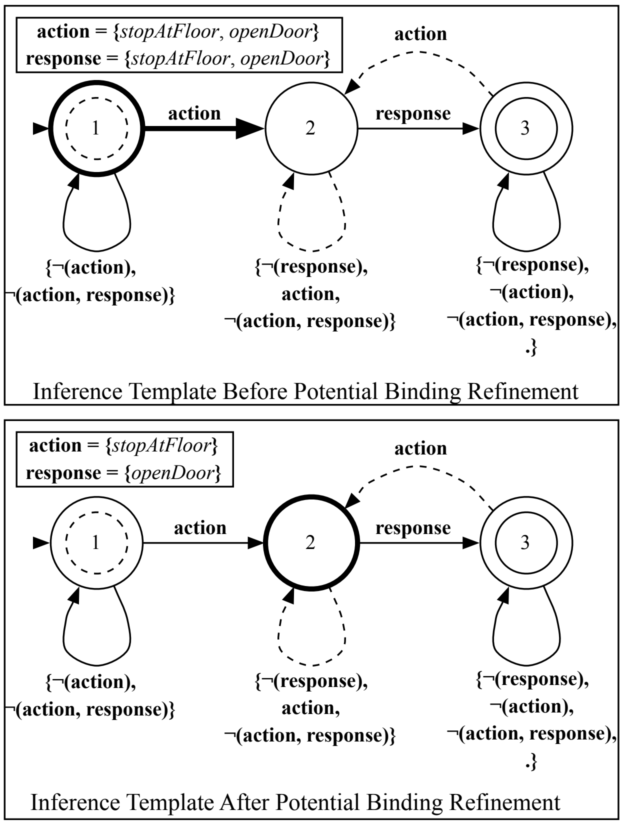
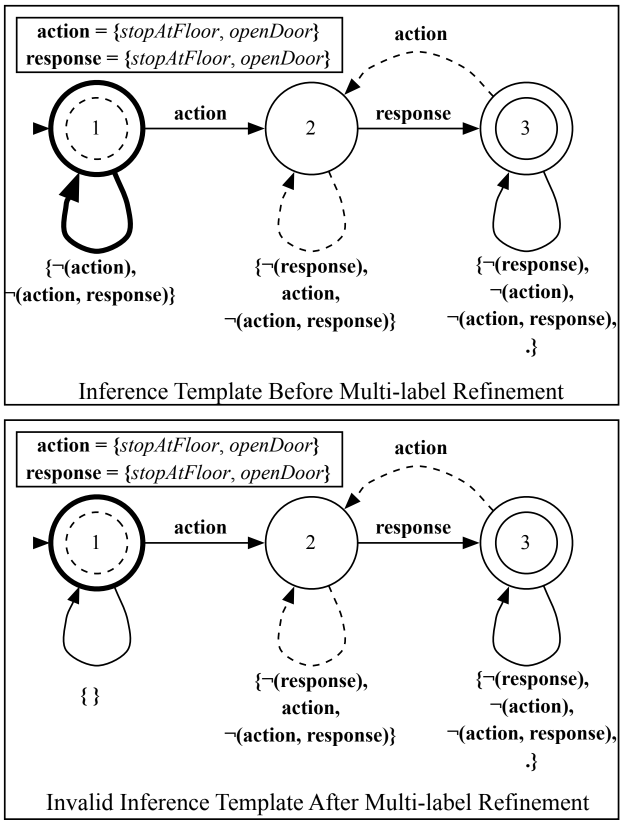

action = {stopAtFloor, openDoor}
response = {stopAtFloor, openDoor}
action
response
1
2
3
{¬(action), ¬(action, response)}
{¬(response), action, ¬(action, response)}
{¬(response), ¬(action), ¬(action, response), .}
Inference Template Before Potential Binding Refinement
action = {stopAtFloor}
response = {openDoor}
action
response
1
2
3
{¬(action), ¬(action, response)}
{¬(response), action, ¬(action, response)}
{¬(response), ¬(action), ¬(action, response), .}
Inference Template After Potential Binding Refinement

Figure 2: Example Potential Binding Refinement

Figure 2 shows the result of performing a potential binding refinement based on the given transition and the concrete event *stopAtFloor*. The result of this refinement is a new inference template whose abstract event **action** has had the concrete event *openDoor* removed. Note that the abstract label denoted by **action** is now a matching label for the event *stopAtFloor*. The other result of this refinement is that the inference template's current state has been updated to state 2. We have also removed *stopAtFloor* from the potential bindings of **response**. This is basically an optimization we can perform because we are disallowing concrete property instantiations generated by having more than one abstract event bound to the same concrete event. If a refinement ever returns an inference template containing an abstract event with a singleton set for its potential bindings, then we can safely remove the concrete event in that set from the potential bindings of all other abstract events.

We now consider performing refinements on our example inference template based on the second potentially matching transition previously identified. This transition is labeled with a multi-label consisting of two abstract labels, ¬**(action)** and ¬**(action, response)**. When we are refining a template based on a multi-label, we may have to perform combinations of both potential binding and multi-label refinements. First, we create a copy of the inference template and attempt to perform a multi-label refinement. A *multi-label refinement* removes from a multi-label each abstract label that is not a matching label for the given concrete event. If all abstract labels are removed, then the inference template become invalid. After performing a multi-label refinement on our example inference template, we will consider performing a combinations of potential binding and multi-label refinements.

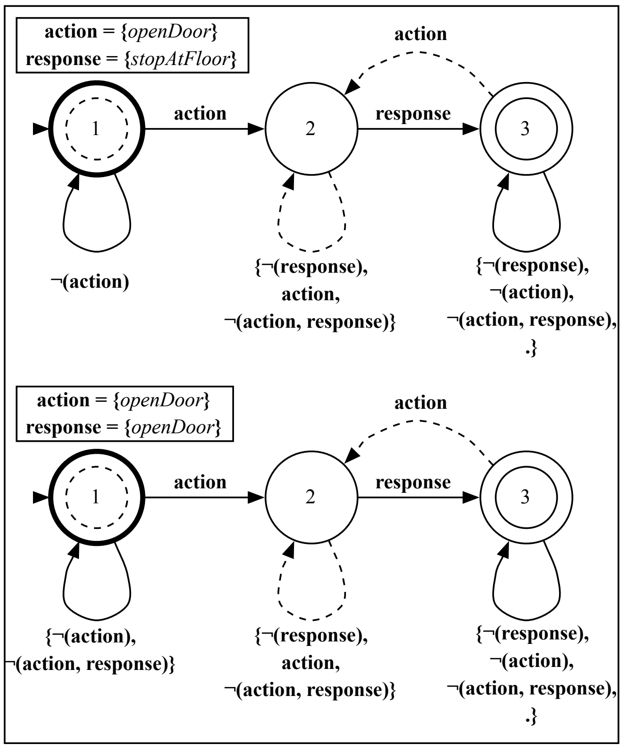


action = {stopAtFloor, openDoor}
response = {stopAtFloor, openDoor}
action
action
response
1
2
3
{¬(action), ¬(action, response)}
{¬(response), action, ¬(action, response)}
{¬(response), ¬(action), ¬(action, response), .}
Inference Template Before Multi-label Refinement
action = {stopAtFloor, openDoor}
response = {stopAtFloor, openDoor}
action
action
response
1
2
3
{ }
{¬(response), action, ¬(action, response)}
{¬(response), ¬(action), ¬(action, response), .}
Invalid Inference Template After Multi-label Refinement

Figure 3: Example Multi-label Refinement

Figure 3 shows the result of performing a simple multi-label refinement on a copy of our example inference template based on the self-loop transition on its start state and the concrete event *stopAtFloor*. Because *stopAtFloor* is contained in the potential bindings for both abstract labels associated with the complement labels that comprise the multi-label on this transition, neither of these abstract labels are matching labels for the concrete event, and both are removed from the multi-label by this multi-label refinement. This results in an invalid inference template.

Next, we consider performing refinements on copies of our example inference template that perform a potential binding refinement followed by a multi-label refinement. We call refinements of this type *multi-stage refinements*. Figure 4 shows the result of performing multi-stage refinements on our example Response inference template. Because there are two possible multi-stage refinements based on this transition and concrete event, our algorithm generates two copies of the original inference template. The first inference template in this figure corresponds to a multi-stage refinement that first performed a potential binding refinement that removed *stopAtFloor* from **action**'s potential bindings followed by a multi-label refinement that removed the ¬**(action, response)** abstract label from the multi-label. The second inference template in this figure is corresponds to a multi-stage refinement that first performed a potential binding refinement that removed *stopAtFloor* from both **action**'s and **response**'s potential bindings. The refinement stops at this point because this inference template is invalid. As stated earlier, we are disallowing concrete properties derived from binding multiple abstract events to the same concrete event. Because of this restriction, this inference template does not represent any valid concrete properties is invalid.



action = {openDoor}
response = {stopAtFloor}
action
action
response
1
2
3
¬(action)
{¬(response), action, ¬(action, response)}
{¬(response), ¬(action), ¬(action, response), .}
action = {openDoor}
response = {openDoor}
action
action
response
1
2
3
{¬(action), ¬(action, response)}
{¬(response), action, ¬(action, response)}
{¬(response), ¬(action), ¬(action, response), .}

Figure 4: Results of Multi-stage Refinements

There is a final processing step that must be performed before a refinement completes. If the refinement's transition is optional, we make the transition a normal transition. We then modify the multi-labels on all other outgoing transitions from the transition's source state to remove all abstract labels that are matching labels for our concrete event. If an optional outgoing transition's multi-label becomes empty, we remove it. If a normal transition's multi-label becomes empty, the inference template becomes invalid. Finally, we remove any outgoing optional transition that whose label is a matching label for the concrete event the refinement was based on. This ensures that our inference template represents only deterministic finite-state properties.

The example refinements performed on our example inference template based on the concrete event *stopAtFloor* have reduced the number of concrete properties represented by our candidate inference template set from the 192 the example inference template in Figure 1 represented. The candidate set consists of the first inference template in Figure 4 and the inference template after the refinement from Figure 2. These refinements have eliminated 48 possible concrete properties.

If we continue this process with subsequent events *"stopAtFloor, openDoor, closeDoor, move, stopAtFloor, openDoor"* from a sample trace, we get the candidate inference template set shown in Figure 5. The candidate inference template set, which now includes 3 templates, represents 56 concrete properties, reduced from 192 when we began processing the trace.

4.2 Prototype Implementation

A prototype of our approach has been implemented using Java. A representation of inference templates and the refinement algorithm described above have been implemented. This prototype can successfully infer properties from traces gathered from simple Java programs, including an elevator control system similar to the one used in the examples throughout this paper. The current implementation supports a limited number of property types, but the framework that has been implemented is very flexible and other property types can easily be added because the core representation and algorithm remain the same for different property types.

5. RELATED WORK

As stated previously, the approach described in this paper builds upon the the property patterns [2] and Propel property pattern templates [6]. Both the Propel property pattern templates and inference templates capture many variations of the property patterns. In fact, the Response inference template used throughout the paper was derived from the Propel Response property pattern template. A key difference between our approach and the Propel approach for property specification is that the Propel approach for property generation is user driven while our approach is event trace driven. The Propel property pattern templates assist users manually specifying properties, while inference templates are automatically refined based on event traces from program executions.

Several other approaches for inferring properties using data from program executions have been proposed. Ernst et al. proposed automatically inferring likely invariants from dynamic traces containing variable value data [3]. This approach focused on learning invariants over these variables that hold at specific program points. Our approach is intended to infer more general properties involving the allowable orderings of event over entire program executions.

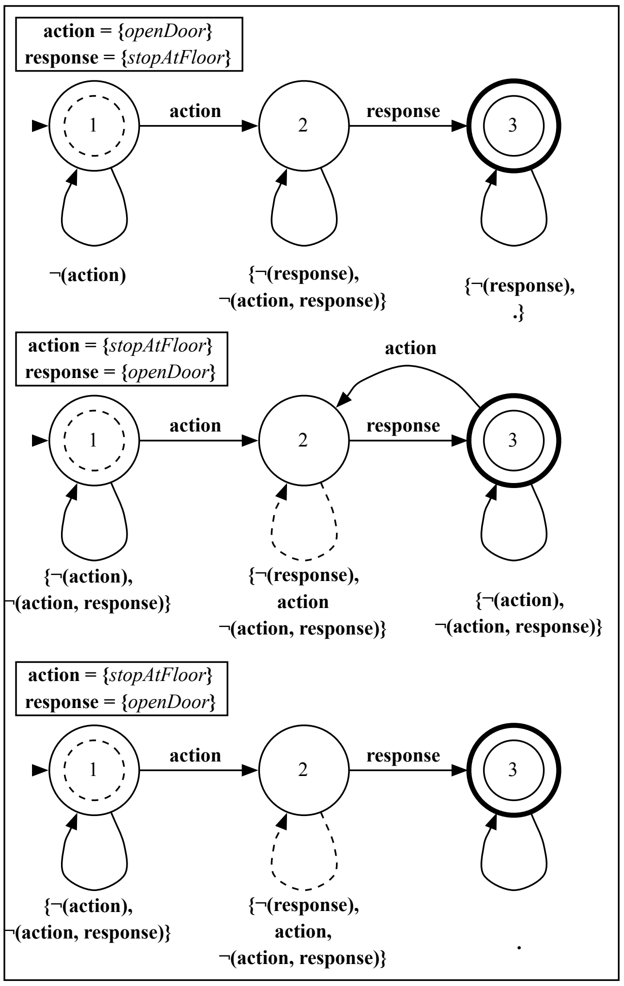

action = {openDoor}
response = {stopAtFloor}
1
action
2
response
3
¬(action)
{¬(response), ¬(action, response)}
{¬(response), ·}
action = {stopAtFloor}
response = {openDoor}
action
{¬(action), ¬(action, response)}
{¬(response), action ¬(action, response)}
{¬(action), ¬(action, response)}
action = {stopAtFloor}
response = {openDoor}
{¬(action), ¬(action, response)}
{¬(response), action, ¬(action, response)}
·

Figure 5: Resulting Candidate Inference Template Set

Two other approaches closely related to ours have been proposed, each focusing on inferring properties that should be true over entire program executions. Ammons et al. proposed an approach based on machine learning called *specification mining* [1], and Yang and Evans proposed a property inference approach that, much like our approach, focuses on inferring variations of the property patterns [7].

Specification mining differs from our approach in several ways. The properties that this approach attempts to infer can be arbitrary finite-state properties. The finite state machines returned by this approach may have many states which can make manual validation of these properties difficult. Also, the properties that are inferred may not be consist with the set of traces used as the training set. Our approach on the other hand focuses on commonly occurring patterns with very few states and is guaranteed to return properties that are consistent with all traces that have been processed.

Yang and Evan's approach defines several variations of the property patterns and a partial order over these variations so that the strictest of these variations can be inferred by generating a subset of the variations. Event traces are then processed eliminating properties that would reject these event traces.

Our approach is able to infer many more variations of property pattern types through its use of inference templates. For the Response property pattern, Yang and Evans identify 8 variations of this pattern for each ordered pair of events. Our approach can infer 96 different concrete properties that are variations of Response for each ordered pair of events.

Also, rather than explicitly generating all of these concrete properties and eliminating the ones that would reject event traces, the refinements we have defined remove invalid concrete properties from the set represented by an inference template by modifying the inference template and generating new copies only when there are multiple way to refine a given template. Yang and Evans define a partial order over their Response variations to reduce the number of them that are explicitly generated. They are able to infer which of the 8 variations of Response for a given ordered pair of events would accept a trace by explicitly generating 3 variations. If we were to apply their technique to the Response property example above with two possible ordered pairs of events, (*openDoor, stopAtFloor*) and (*stopAtFloor, openDoor*), we would have explicitly generated 6 properties to infer which of the 16 properties held. Our approach was able to infer which of the 192 candidate concrete properties were consistent with the trace above by generating 3 valid inference templates.

6. CONCLUSIONS AND FUTURE WORK

This paper describes an approach for inferring properties from event traces gathered from program executions. We described how inference templates are used to represent many variations of similar properties compactly and presented our template refinement algorithm. We also presented an example of this algorithm processing an event trace and refining a candidate inference template set consisting of Response inference templates. This example was able to quickly remove many properties from the set represented by our candidate template set while generating very few new inference templates. This approach appears promising, but more experimental validation is required. In particular, we intend to improve our prototype implementation and apply our technique to event traces gathered from the execution of realistic programs.

This approach lends itself to several applications and future directions. During our discussion of the approach we did not discuss explicitly from what types of program executions will be deriving event traces. This approach could be applied to event traces gathered from test suites. In this context, the candidate inference templates returned to a user could be manually analyzed to assess the quality of the test suite for concrete property coverage. A candidate inference template set with many templates containing optional transitions, multi-labels, etc. could be used to manually or automatically generate test cases to refine these inference templates. The goal of generating these new test cases would be to produce test cases whose event traces would eliminate concrete properties from the set represented by our candidate inference template set.

Our approach could also leverage program executions of deployed software. The event traces used by our approach could correspond to event traces gathered from program executions of deployed software. Aside from using field data to infer properties of deployed software, we may be able to use a technique similar to the one used in [4] to distribute our inference approach to instances of deployed software. Each instance would start with a candidate inference template set, which would be refined locally based on that instance's event traces. A central location could gather each deployed instance's candidate inference template set and compute the intersection of all these sets. Each deployed instance would then receive an updated candidate inference template set representing concrete properties that are consistent with all event traces processed by any deployed instance. The potential benefit of such an approach would be the reduced data transmission overhead achieved by our compact representation. Rather than sending event traces to a central location, deployed instances could send a candidate inference template set derived from those traces. This representation could potentially result in much less data being transferred when compared with sending event traces.

Another important area that we intend to explore is the use of *context information* in our inference approach. In object-oriented programming languages, such as Java, the properties of a program often involve more than just an ordering of simple events, such as method calls. Context information like the calling thread, object instance, and parameters to a method call may be an important part of a property. In recent work [7], Yang and Evans address this issue by proposing slicing an event trace into multiple traces based on context information. We intend to modify our approach to allow abstract and concrete events with varying amounts of visible context information. Our template refinement algorithm could then be modified to support refinements based on this context information.

7. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proc. Symp. Princ. Prog. Lang.*, pages 4–16, New York, NY, USA, 2002. ACM Press.

[2] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. Int. Conf. Softw. Eng.*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. Int. Conf. Softw. Eng.*, pages 213–224, 1999.

[4] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proc. Int. Symp. Softw. Testing Anal.*, pages 65–69, New York, NY, USA, 2002. ACM Press.

[5] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Object-Oriented Programming, 19th European Conf.*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

[6] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Propel: an approach supporting property elucidation. In *Proc. Int. Conf. Softw. Eng.*, pages 11–21, New York, NY, USA, 2002. ACM Press.

[7] J. Yang and D. Evans. Automatically discovering temporal properties for program verification. Technical report, Department of Computer Science, University of Virginia, 2005.