

# User Interface Model Discovery: Towards a Generic Approach

**Andy Gimblett**

Future Interaction Technology Lab  
Swansea University  
a.m.gimblett@swansea.ac.uk

**Harold Thimbleby**

Future Interaction Technology Lab  
Swansea University  
h.thimbleby@swansea.ac.uk

## ABSTRACT

*UI model discovery* is a lightweight formal method in which a model of an interactive system is automatically discovered by exploring the system’s state space, simulating the actions of a user; such models are then amenable to automatic analysis targetting structural usability concerns. This paper specifies UI model discovery in some detail, providing a formal, generic and language-neutral API and discovery algorithm. The technique has been implemented in prototype systems on several programming platforms, yielding valuable usability insights. The API described here supports further development of these ideas in a systematic manner.

## Author Keywords

Discovery tools; interaction programming; structural usability; reverse engineering.

## ACM Classification Keywords

H.5.2 (D.2.2, H.1.2, I.36) User Interfaces: Theory and methods

## General Terms

Algorithms

## 1. INTRODUCTION

Interactive systems are widespread and in many cases critical, but often contain design defects that may cause users problems. Traditional usability techniques cannot be expected to find all such defects: it is generally impractical for a user to exhaustively explore all of a system’s possible states and interactions, yet serious flaws may exist in the unexplored parts, only to manifest when the system is put into use—potentially with life-threatening or mission-critical consequences [22]. Conversely, processes based on conventional formal methods can guarantee exhaustive analysis but are rarely used except in certain key domains—largely because of the high level of expertise required. Furthermore, because such techniques invest a lot of effort prior to implementation, they tend either to discourage iterative

development, or to be bypassed where it takes place. Sometimes the formal methods are only applied to a model prior to implementation, and the implementation itself may have unknown bugs independent of the model.

An alternative approach is to use processes that can reliably and automatically analyse actual systems, even as developed using mainstream methods. Such processes (sometimes called *lightweight* formal methods) aim to provide the benefits of using formally-based tools without the high cost and inflexibility of conventional formal methods—and to integrate well with existing tools and techniques. These efforts should be seen in the context of Reason’s “swiss cheese model” [14]: adding new layers of defence against error to those already routinely used by developers.

Thimbleby [23] introduces one such technique: *user interface model discovery*, in which a model of a UI is automatically discovered by a tool simulating the actions of a user. In this approach, the model produced is a finite directed graph—a discrete state space whose nodes represent states of the device UI that is being discovered, and whose edges represent user actions changing that state. Various analyses of such a model may then be performed using standard graph-theoretical techniques or model checking tools [21, 23, 24] (see also section 6). While such models can become very large, experience has shown that by choosing the right level of abstraction, tractable models yielding useful results are obtainable; Jackson’s *small model hypothesis* [6] suggests that bugs can be found in small models, and if a large model is required, a user would plausibly be unable to understand the operation of that system in any case. Discovery complements purely analytical techniques such as abstract interpretation; in the presence of running code it is much simpler to apply, and does not rely on detailed knowledge of the code’s running environment—e.g., consider a virtual machine running in a web browser: its event-handling/buffering behaviour, and their interaction with the code in question, will tend to be non-trivial to interpret abstractly.

### 1.1 Contributions

This paper describes UI model discovery in detail, concentrating on the technical requirements for implementing the technique in a given setting. Its contributions (going beyond [23]) are a formal and generic description of an API for UI model discovery, and a discovery algorithm, clearly stated in terms of that API. We also describe variants and extensions to the API, with which the algorithm can perform

a variety of sophisticated UI exploration tasks. Throughout, we draw attention to certain areas where a user of the technique—nominally, an investigating programmer—must make informed choices of how to apply it, though a full exploration of these issues is beyond the scope of this paper. Our description is language-neutral, and suitable for retrospective integration with existing UI applications and development tools; as such, it is our hope that in the future, models of many UIs, implemented using various languages and toolkits, will be produced using these techniques, and hence contribute to a lifting of user interface implementation standards. In particular, integration into IDEs and development workflows seems a promising area, and one where the requirements of the API (discussed below) are easily met; reverse engineering existing systems would also be valuable, but harder to achieve in general because of the deep level of integration required.

## 2. USER INTERFACE MODEL DISCOVERY

In UI model discovery we systematically explore the state space of an interactive system by simulating the actions of a user, using standard search techniques [25] augmented with domain-specific aspects such as discovery/actuation of UI widgets. Discovery produces a directed graph whose nodes represent (sets of) system states, and whose edges represent user actions.

Our current scope is reactive devices with discrete interfaces and finite state spaces, subject to a number of assumptions. In particular: that the system responds (almost) immediately to user actions, that the responses are manifested in the system's state (possibly including the state of its UI), and that any silent or external (non-user) actions which modify the state may be modelled by (perhaps explicitly) adding them to the set of explored actions—see section 2.2.2; e.g., *tick* actions are used in the infusion pump model in [24] to represent passing time in certain modes. There are clearly many non-trivial devices that satisfy these assumptions.

We take the view that such an interactive system consists of an interface with which the user interacts, and (usually) an underlying system implementing domain-specific logic. This separation of concerns may be designed into the architecture (e.g., using the MVC pattern [15]), or may be less well-defined. We call the state of the interactive part the *GUI state*, in principle visible to the user, and we call the state of the underlying system the *inner state*.

### 2.1 Motivating Example

Figure 1 shows an example of UI model discovery in action. The UI being discovered, and the discovery tool, were written in Haskell [13] using the wxWidgets [18] toolkit—see section 4.1 for further discussion. The tool, its source code, and a screencast showing its operation, may be found online.<sup>1</sup> The tool displays a number of features typical of our discovery approach.

The interface being discovered, or “System Under Discovery” (SUD), is at the top-left. By way of example, it is de-

liberately trivial: a simulation of an air conditioning control unit with controls: on/off; heat/cool; fan speed (three settings); target temperature (5–30°C). The *Control* panel is used to control and monitor UI discovery. First, the SUD's controls are listed: `sl_On/Off`, etc. The available controls and associated actions have been discovered automatically by the tool—and may vary between states of the SUD. A textual representation of the SUD's inner state is shown as “State:”; here this system has a 1-1 correspondence between GUI state and inner state, but this need not always be the case—see section 2.2.1. Beneath this panel are a number of counters indicating the discovery progress so far, and a “health check” of the discovered model (in this case, it is weakly connected, which is reasonable as discovery is still in progress). Finally, there are buttons to start discovery (which becomes “Pause Discovery” if it is running), to execute just a single cycle of the discovery algorithm, and to reset to an initial state. The *State space* panel to the right previews the SUD's state space graph as discovered so far. Below this are buttons to redraw the preview (if the window is resized), to view the state space in full using an external application (rendering from PDF or GML, a graph language), or to save it in various formats suitable for further processing.

In the illustration, discovery is only partially complete; 1012 discovery steps have occurred; 184 states have been discovered, of which 170 have been fully explored, while the rest are in a pool of 14 known but as yet unexplored states with 80 associated actions performable in those states. The SUD is simple enough for us to see that upon completion, a total of  $2 \times 2 \times 3 \times 26 = 312$  states will have been explored.

### 2.2 Requirements for UI Model Discovery

The UI model discovery algorithm simulates the actions of a user systematically performing all possible actions in all possible states. The API is the formal interface between the UI application and the discovery tool—a bridge which must be implemented before discovery may take place on a given platform. The API provides four key capabilities, which we shall now consider in turn:

- ability to compute, store and compare inner states;
- ability to identify, for a given inner state, the actions which may be performed in that state;
- ability to perform such actions;
- ability to backtrack, so all of a state's actions can be explored.

#### 2.2.1 Identifying and modelling inner state

A node in the discovered model represents an instance of the SUD's inner state; such a state consists of a collection of state variables projected from the SUD. This notion is deliberately left abstract and generic in order to support a wide range of possible implementations: it is the task of the interaction programmer modelling the SUD to choose and implement an appropriate projection. However, this task is critically important (e.g., see [4] for an in-depth examination of questions surrounding such projections) and so we consider some key factors here.

<sup>1</sup><http://www.cs.swan.ac.uk/~csandy/research/discovery/aircon/>

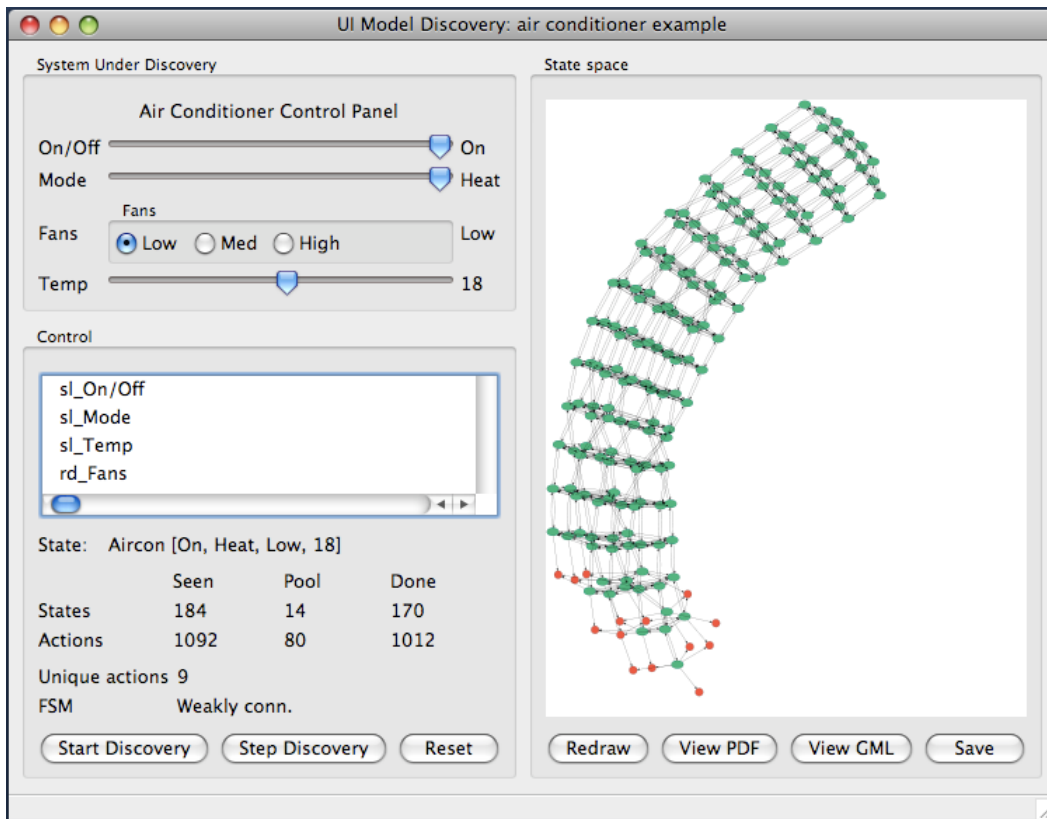


Figure 1. UI model discovery for a simple air conditioning control unit

First, which variables are to be projected? For simple systems (such as the one discussed above), we might project everything, but the more we project, the bigger the state space, and if we are investigating a particular aspect of interaction, filtering out irrelevant variables is an important abstraction technique; e.g., if we are investigating number entry in a syringe pump, then projecting the alarm volume is unnecessary. In general, tracking such ‘uninteresting’ state will introduce many ‘garbage’ states to the model, and should be avoided if possible—but note that that may *not* be possible, at least at the discovery stage: see section 3.5. Where the SUD has complex (perhaps object-oriented) state structure, it might be sufficient to simply project those objects and their relationships, provided equality and backtracking are implemented—but such thoroughness will not be necessary in general: a flat (or slightly hierarchical) namespace projecting key components will often suffice. Even ephemeral parts of the state structure may be projected: two states are clearly non-equivalent if they contain differently-named elements—though care might be required for backtracking, to ensure that an ephemeral piece of state has a home to be projected back onto. (Again, the techniques described in section 3.5 can help here.)

Second, which values are projected for each variable? Projection of all possible values may not be tenable; e.g., if we project a numerical variable directly, the state space can rapidly explode. It may be sufficient in such cases to use a small subset of possible values, or a set of named equiva-

lence classes (e.g., the classic “zero, one or many”). Conversely, too much restriction here can result in an insufficiently detailed model—though in our experience a surprising amount can be learnt from very restricted models [23].

These two factors interact to define an equivalence relation on inner states: it is the investigating interaction programmer’s task to find the appropriate equivalence relation for the analysis at hand. Again, see [4] for further discussion.

Third, how is projection implemented? Our approach assumes a certain level of intimacy between the discovery tool and SUD: the tool must be able to inspect SUD states, and interact with its GUI. Typically this means they need to be compiled together, or run in the same process space, though other approaches may be possible, such as interaction with a separate process via O/S mediation as in [11]. In any case, projection of inner state may not be possible without some modification of SUD code—though in our experience this is not always necessary, and where it is, it is minimally invasive. We primarily envisage discovery augmenting development in much the same way as a debugger or unit testing framework, in which case these requirements are easily met.

Finally, how is an inner state represented? Like an object’s state in a traditional OO language, a state can be viewed as a mapping from names to values of various types; because we need to compare states for equality, we must be able to compare their values. As a general approach, we have found

data types with trivial semantics similar to that of JSON [3] (trees of key/value pairs with basic data types such as numbers, booleans, strings, lists) to be adequate. As noted above, richer models might be used, but our experience so far has been that the state structuring and abstraction mechanisms which aid implementation of UIs are not relevant or required in the setting of discovery.

### 2.2.2 Identifying actions

Edges in the model represent discrete user actions which, assuming an event-driven GUI framework is used, generally correspond to events in that framework, such as button clicks, slider moves, text entry, menu selections, etc. Clearly we require a way to identify the actions which may be performed in a given state; the most general solution is to automatically inspect the SUD's GUI and discover the possible actions automatically—with integration implications as above. How this is implemented will depend on the implementation platform and the capabilities it offers. Restricted cases, such as learning the actions only once at the start of discovery, or hand-coding them into the discovery implementation if automatic learning is not possible, are then special cases of this approach—see section 3.6.4.

### 2.2.3 Performing actions

Having discovered possible actions, it is necessary to perform them and see where they lead; thus, we require an automatic way to trigger discovered actions which, again, in detail will vary between implementation platforms. Fortunately, every framework for which we have implemented discovery has supported this quite directly; in HTML/JavaScript, for example, a button click may be enacted simply by calling the button object's `click()` method. Obviously, to do so generally requires maintaining a reference to the widget being interacted with.

After the action has been triggered, an inner state is projected and inspected. If it is unchanged, the action had no effect—it is a self-loop (self-loops need not be explicitly represented, though they may be semantically important during subsequent analysis, particularly in cases involving nondeterminism; see [23]); otherwise, we have discovered a new edge in the state space, and possibly a new state. If we have discovered a new state, we must discover its possible actions as described above, and explore them later.

### 2.2.4 Backtracking

To explore all actions performable in a given inner state, we need backtracking. Conceivable approaches include:

1. Reset to arbitrary saved state.
2. Reset to initial state; follow saved path to desired state.
3. Undoable actions.
4. Using only actual user actions to restore an arbitrary state (sometimes called the robot exploration problem).

After backtracking, the restored inner state must be equivalent to the one seen earlier. This does not imply that the

actual underlying state must be identical—only that the projected state is *and* that the effects of possible actions are identical; if they're not, then our model will be unsound—discovering it again in a different order could lead to a different model—see section 3.5. In our experiments so far we have exclusively used strategy 1, and the algorithm (and our discussion) assumes this to be the case.

## 3. UI MODEL DISCOVERY: API & ALGORITHM

We now describe, using Haskell data types and type signatures, the API elements which must be implemented in order to use the model discovery algorithm on a given UI development platform; familiarity with Haskell is not necessary to understand this section: we shall explain the few required concepts.

We specify our API using Haskell because it is high-level, rich but compact; it is independent of a particular SUD or GUI framework; it is easily translated to other formalisms and programming languages; finally, our specification is derived from actual running/working UI model discovery code. We would like to stress however that our goal is a general description of model discovery, not just a particular implementation; we believe the formal description here enables its implementation in any adequate language, and in particular have done so in several other settings including Java, JavaScript and ActionScript [23, 24].

Haskell is a pure functional language with a number of features, including the strongest and most thoroughgoing type system of any reasonably mainstream programming language. Collections of Haskell type signatures are comparable in appearance and expressiveness to signatures as found in the universal algebra/algebraic specification tradition [2]. Haskell is both *strongly* and *statically* typed: every value in a Haskell program has a particular type, and that type is fixed at compile-time. Furthermore, functions in Haskell are *first class* (they may be passed to and from functions), and subject to the same type system as atomic values. Type names start with an upper-case letter, as in *Int* or *Set x*. Here *Set* is polymorphic: it is a *parametric type*, representing a set of something; it may be instantiated by providing a concrete parameter type, as in *Set Int*. Function types are written using arrows, where multiple arguments are written using multiple arrows: for example, *Int → Int → Int* is a function taking two *Ints* and returning one. Names of values and functions start with a lower-case letter, and are given types using “`::`”, as in *double :: Int → Int*, which declares a function from *Int* to *Int*.

### 3.1 Discovery and Manipulation of GUI Controls

We start with data types representing GUI controls (widgets), their values, and the actions we can perform on them. The details will vary from framework to framework, so we leave these types loosely specified here.

```
data GuiControl = ...
data GuiValue  = ...
data GuiAction = ...
```

Note that in our conception, a *GuiAction* encapsulates not just the action to perform (say, “move slider up”) but also a reference to the control on which the action is performed. Separating them out would simply add extra parameters to some of the functions below.

We need to be able to set a control to a particular value (when resetting to some state) to determine the actions we can perform in the current state, and to perform an action:

```
setControlValue :: GuiControl → GuiValue → IO ()
getGuiActions  :: Parent w ⇒ w → IO [GuiAction]
doGuiAction    :: GuiAction → IO ()
```

Note the presence of *IO* in each of these type signatures. This indicates that the function may have side-effects, in this case interacting with the SUD’s GUI. If *IO* is absent we know the function is *referentially transparent*: it can perform no side-effects—see, e.g., *addToPool*, below, which does not modify the pool, but rather returns a *new* pool. Of course, a Java *addToPool* implementation would modify a pool rather than constructing a new one: the point is that the type signatures given here distinguish clearly between functions which interact with the SUD, and those which do not.

Consider *getGuiActions*, which discovers a GUI’s controls, and currently available actions on those controls, returning a list of *GuiActions* (see section 2.2.2). Here *w* is another parametric type, and *Parent w ⇒* is a *context* requiring that whatever *w* is, it is an *instance* of the *typeclass Parent*. Haskell typeclasses are analogous to Java interfaces: here, *Parent w ⇒* indicates that whatever *w* is (panel, window, set of windows) it must have children—and in particular, a *children* function for listing them; then, *getGuiActions* uses that function to recursively inspect a widget and its children for manipulable controls and their actions.

### 3.2 The Pool: Discovered But Unexplored States

Some mechanism is required to track those parts of the UI state space that have been discovered but not yet fully explored: we call this the “pool” of states and actions yet to explore. Note that there are many possible ways to explore the state space, with depth-first and breadth-first as extreme choices. We aim to be as generic as possible in our description; in particular, we wish to accommodate strategies—such as depth-first—which do not fully explore a given state’s actions before moving on to another state, as well as those—like breadth-first—that do.

A pool or priority queue of (state, action) pairs is sufficient; it contains one element for every unexplored action from every discovered state. Discovery implementations may then pick pairs from that pool using whatever strategy they see fit. There are many ways to implement such a pool, so again we leave it specified loosely here—though we note that it depends on the type of the inner state, which we write as *st*.

```
data Pool st = ...
```

So a value of type *Pool st* is a pool of (*st*, *GuiAction*) pairs.

However it is implemented, we need to add/remove items:

```
addToPool :: Pool st → st → [GuiAction] → Pool st
pickFromPool :: Pool st →
  (Maybe (st, GuiAction), Pool st)
```

*addToPool* takes a pool, an inner state, and a list of actions (performable in that state) and returns a new pool with the state’s actions added—one pair per action. *pickFromPool* chooses one element from the pool, returning that element (and a new pool, with that element removed). *Maybe* is another parametric type, used for computations which may in some sense “fail”—in this case, “failure” occurs if the pool is empty: then, instead of a (*st*, *GuiAction*) pair, it returns the value *Nothing*. See section 3.6 for more discussion of exploration strategies and the critical role of the pool.

### 3.3 Model Discovery

The model discovery algorithm repeatedly picks (state, action) pairs from a pool and explores them, building a state space on the fly (and sometimes extending the pool). It stops when the pool is empty. Further, it needs to be able to query and reset the SUD, and to compare inner states. All this can be packaged in a data type, a value of which encapsulates current discovery status, and some related functions:

```
data Discovery st = Discovery {
  stateSpace  :: Gr st GuiAction,
  pool        :: Pool st,
  project     :: IO st,
  reset       :: st → IO (),
  eq          :: st → st → Bool,
  getGuiActions :: Parent w ⇒ w → IO [GuiAction]
}
```

This type is polymorphic over the inner state type *st*. Here, *stateSpace* is a graph whose nodes are labelled with inner states and whose edges are labelled with GUI actions; *pool* is as described above. *stateSpace* and *pool* are initialised empty and are modified as discovery progresses.

The remaining four members of *Discovery* are callback functions, specified as parameters when *Discovery* is initialised (see below). *project* computes the SUD’s current inner state, as discussed in section 2.2.1. Dually, *reset* takes an inner state and imposes it onto the GUI using *setControlValue* for backtracking—see section 2.2.4. *eq* compares two inner states for equivalence, and should create equivalence classes of inner states as appropriate—see section 2.2.1. Finally, *getGuiActions* is as described in section section 3.1.

To initialise such a value, we provide the callback functions just mentioned, and construct a *Discovery st* with empty *stateSpace* and *pool*; then *initDiscovery* should immediately call *project* and *getGuiActions* to compute seeds for the state space and pool.

```
initDiscovery :: IO st → (st → IO ()) →
  (st → st → Bool) → (w → IO [GuiAction]) →
  Discovery st
```

Given a state, we need to know if it has been seen before (using *eq* to compare states):

```
isStateNew :: Discovery st → st → Bool
```

When we find new states and new edges, we add them to the state space, returning new *Discovery* values. (We do not have to check for new edges because we only add a state’s actions to the pool once, when we first meet the state; see details below.)

```
addState :: Discovery st → st → Discovery st
addEdge :: Discovery st → (st, GuiAction, st) →
  Discovery st
```

Finally, an auxiliary function checks if the pool is empty:

```
finished :: Discovery st → Bool
```

Figure 2 summarises the API, pointing out which parts may be re-used for multiple SUDs written in the same GUI toolkit, and which need to be defined once per SUD. Most of the work can be done just once per toolkit: only the data type *st* and the parameters passed to *initDiscovery* ever need to be tuned for a specific SUD (but see section 3.6).

Notes:

1. As described in section 3.6, several *pickFromPool* implementations may be desirable to implement various exploration strategies; however, in each case, the strategy need only be implemented once per toolkit.
2. As described in section 2.2.1, the data type *st* used to represent inner states may be definable just once in a generic way, which can then be re-used for many SUDs; if it is not possible or desirable to do so, then a per-SUD representation can be used.

### 3.4 An Algorithm for UI Model Discovery

Given the API, algorithm 1 shows pseudocode for the UI model discovery algorithm.

---

**Algorithm 1** The UI model discovery algorithm.

---

```
initDiscovery
while ¬ finished do
  s, a ← pickFromPool
  reset SUD to inner state s
  doGuiAction a
  s' ← project new inner state
  if ¬ [s eq s'] then
    if isStateNew s' then
      addState s' to stateSpace
      a' ← getGuiActions for state s'
      addToPool s', a'
    end if
    addEdge (s, a, s') to stateSpace
  end if
end while
```

---

In practice the algorithm need not be implemented exactly as

shown here. Thus, for the air conditioning control described in section 2.1, a single cycle of the **while** loop is enacted by clicking “Step Discovery,” while “Start Discovery” starts it cycling to completion (though it may be paused). There is no while loop directly visible in its code—but the overall strategy above is encoded faithfully.

### 3.5 Nondeterminism

The above algorithm always produces a deterministic model: in a given state, a given action is only ever explored once, so the model *cannot* contain states with multiple identically-labelled actions leading to different destinations.

However, the use of a state projection may cause the *algorithm* to operate nondeterministically: if some aspect of state is not projected (and thus not reset upon backtracking), but influences the effect of an action, then multiple runs of the algorithm with different exploration orders might produce differing models. We argue that this would be a sign of an ill-formed projection, which fails to capture some essential element of state (but see below), and propose *stochastic checking* of models to detect such cases. An obvious approach here is to perform a full stochastic exploration (see section 3.6.1) after discovery is complete, and check that the two models thus produced are isomorphic; a more practical approach is to interleave stochastic checks with model discovery by revisiting discovered state/action pairs at random, allowing early detection of ill-formed projections.

‘Uninteresting’ (but necessary, in the above sense) aspects of state may, of course, be filtered from the complete discovered model, and *this* can lead to a nondeterministic model. Such models may be meaningful and provide valuable insight [4]; e.g. in [23], the ‘device model’ of a simulation of a handheld calculator projects all of its inner state and is deterministic, but the ‘user model’ (produced by filtering out those parts of the state which are hidden from the user) is nondeterministic and provided the insight that the decimal point button never updates the display.

### 3.6 Variations and Extensions

The API and algorithm are deliberately generic. We now describe some useful extensions and variations to the basic picture. Note that with the possible exception of directed exploration (section 3.6.7), each of these extensions requires modification only of API elements: the structure of the algorithm remains identical.

#### 3.6.1 Exploration order

The order in which the state space is explored is determined by the implementation of the *Pool* data type and its associated functions. It is possible to implement almost any desired strategy without modifying the rest of the algorithm. In particular, using a stack for the pool will yield *depth-first* exploration whereas a FIFO queue yields *breadth-first*. *Stochastic* exploration can be implemented using any collection data type supporting random access, and this may be useful for checking for hidden modes or for inadequate state abstractions. If the state space is fully explored, each of these strategies eventually produces the same result, albeit in

Item	Purpose	Re-use
<b>data</b> <i>GuiControl</i>	Data type for GUI controls	toolkit
<b>data</b> <i>GuiValue</i>	Data type for GUI values	toolkit
<b>data</b> <i>GuiAction</i>	Data type for GUI actions	toolkit
<i>setControlValue</i>	Set a GUI control to a given value	toolkit
<i>getGuiActions</i>	Learn available actions in SUD's current state	toolkit
<i>doGuiAction</i>	Perform some GUI action on some GUI control	toolkit
<b>data</b> <i>Pool st</i>	Data type for pool of (state, action) pairs to explore	toolkit
<i>addToPool</i>	Add a discovered state and its actions to a pool	toolkit
<i>pickFromPool</i>	Pick the next item to explore from a pool	see note 1
<b>data</b> <i>st</i>	Data type for inner states	see note 2
<i>project</i>	Project inner state from SUD	SUD
<i>reset</i>	Given an inner state, reset SUD to that state	SUD
<i>eq</i>	Compare two inner states for equality	SUD
<b>data</b> <i>Discovery st</i>	Data type encapsulating state space and pool	toolkit
<i>initDiscovery</i>	Initialise model discovery algorithm	toolkit
<i>isStateNew</i>	Check if a state is new or has been seen before	toolkit
<i>addState</i>	Add a newly-discovered state to the state space	toolkit
<i>addEdge</i>	Add a newly-discovered edge to the state space	toolkit
<i>finished</i>	Check if the pool is empty or not	toolkit

Figure 2. API summary

different orders; however, some of the extensions described below can break that assumption.

### 3.6.2 Conditional exploration

In *conditional exploration* we ignore particular states or actions; we need only modify *addToPool* or *pickFromPool*. For example, introduce a predicate on  $(st, GuiAction)$  pairs to *addToPool*, so it only explores pairs for which the predicate is true (compare this with the version in section 3.2):

$$addToPool :: ((st, GuiAction) \rightarrow Bool) \rightarrow Pool\ st \rightarrow st \rightarrow [GuiAction] \rightarrow Pool\ st$$

By modifying *pickFromPool* similarly, we can achieve the same effect at a later stage. If the filtering condition is fixed for the entire exploration, there is no difference between these two approaches; if it can vary (depending on state space size, say) then there is a real difference, and one approach might be preferable to the other.

### 3.6.3 Filtering inner state items

As described in section 2.2.1, *project* dictates which aspects of inner states are projected into the model and, as such, it strongly influences the contents and size of the discovered model. Thus in the air conditioning control example, ignoring the temperature slider reduces the discovered model to just 12 states. An obvious way to increase the flexibility of model discovery, then, is to allow some *run-time* control over this. A simple way to do this is to add a parameter to *project*, listing the model state members to be retained—and to expose that list to user control via the discovery control interface itself.

How inner state members are identified for such filtering depends on the representation used: if the inner state is flat,

simple names suffice; if it is a JSON-like tree, a path language such as *XPath* ([www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)) is required. Assuming the existence of a *StateItem* type fulfilling this role, the type signature of *project* then becomes:

$$project :: [StateItem] \rightarrow IO\ st$$

### 3.6.4 Controlling action discovery frequency

The algorithm calls *getGuiActions* in *every* state; for full generality this is required, in order to deal with dynamic interfaces in which elements come and go (i.e. most non-trivial GUIs). However, in particular cases it might be unnecessary or undesirable.

If the GUI is static, or if all actions can be discovered upfront, a single call on initialisation will suffice, and this can be implemented without altering the algorithm by *memoising* *getGuiActions*, that is having it cache the results of its first call and return them immediately for all subsequent calls.

Where automatic discovery of actions is not possible, or as a performance optimisation measure, a last resort is to hard-code the control actions into the discovery tool, i.e. in *getGuiActions*—assuming there is a way to relate these hard-coded references to the actual widgets and actions.

In any case, if the SUD has a dynamic interface, it is necessary to ensure that *doGuiAction* is a no-op (so it keeps the SUD in the same state) for unavailable actions.

### 3.6.5 Context-sensitive exploration

Suppose there is some target state we are trying to work towards, because we are interested in how the user gets from *A* to *B*, say. We might wish to implement a hill-climbing

strategy or similar, in which case we need to know the current state and the global context, so we can try to pick an appropriate action leading in the right direction.

In *context-sensitive exploration*, then, exploration may be influenced by both the current state and the current picture of the state space; thus, it is necessary to add these as parameters to *pickFromPool*:

$$\text{pickFromPool} :: st \rightarrow Gr\ st\ GuiAction \rightarrow Pool\ st \rightarrow (Maybe\ (st,\ GuiAction), Pool\ st)$$

### 3.6.6 Initialisation

As described above, model discovery starts with a single state in the state space, and that state's actions in the pool—where that state is just whatever state the SUD is in when discovery is begun. A possible extension, if there is a programmatic way to specify inner states, is to seed discovery with more than one. Such an approach could be useful for the kind of search mentioned in section 3.6.5: concurrent searches starting from several points may lead to a desired result state faster. In order to be effective, some sort of scripted control over exploration would be desirable, for example allowing seed collections to be defined easily—see section 3.6.7.

### 3.6.7 Directed exploration and scripting

An interesting avenue of future work is *directed exploration*, in which, rather than proceeding entirely automatically, discovery is consciously directed, either interactively or programmatically, by the analyst. That is, a finer level of control is offered to the interaction programmer, allowing the focussed and flexible application of the ideas already presented in this section.

For example, while some of the systems described in section 4 implement conditional exploration by allowing the programmer to filter explored actions before discovery begins, the set of actions explored is fixed while it is running. Instead, we might offer the ability to interrupt discovery and modify that set, so that different parts of the model involve different actions (this could be one way to tackle *modes*).

The following aspects are involved:

- ability to interrupt automated discovery, either manually or using a breakpoint-like approach (defined conditionally, rather than locationally);
- ability, when paused, to modify discovery criteria such as exploration order, action/state filtering conditions, and inner state projection;
- ability to perform discovery step-by-step, possibly with fine control over *pickFromPool*'s behaviour, to allow choice of state and/or action explored—perhaps graphically, via the state space preview;
- support for all of these tasks via the discovery control GUI and/or programmatically, for instance via a domain specific language (DSL) [9].

As discussed in section 1, we see model discovery as suitable for integration into existing development workflows, and it is clear that a DSL for discovery control is an essential component of such efforts. For without such a capability, manual intervention would be required for all but the simplest of cases, and smooth integration into iterative development workflows based on automated regression testing would be (nearly) impossible. Thus, we consider directed exploration, and particularly language support for same, to be an important area of future research.

## 4. CASE STUDIES

### 4.1 Air Conditioning Control

We simulated, using Haskell and wxWidgets, an *Eberle* air conditioning system control panel, and wrote a model discovery tool embedding that simulation (see section 2.1). We found Haskell to be an expressive language for this task, and that the discipline encouraged by its strong type system led to various better understandings of the structure of the discovery process itself, and ultimately to this paper. The SUD and discovery tool were compiled together (indeed, we embedded the SUD UI in the tool UI, but this is not required), so the tool's access to the SUD is fixed at compile-time. The wx toolkit provides runtime access to the UI components, so *getGuiActions* was easily implemented using the *children* function (yielding hierarchical structure), facilities for inspection of widgets' types and bounds where relevant, etc. Similarly, *setControlValue* and *doGuiAction* were easily implemented using standard widget-property-setting machinery (e.g. allowing a slider to be set to a particular value programmatically). The SUD's inner state was represented using a simple record type with four components, and bound to its GUI state using the Observer design pattern, applied bidirectionally—in the SUD code, not the discovery tool, of course. Then the discovery tool's *project* function is a simple extraction of that state record via some exposed variable in the SUD. The SUD's interface is simple and orthogonal enough that components may be dropped from the projection freely without incurring any hazards mentioned in section 3.5, so while this example helped us clearly understand the mechanisms of discovery, it does not provide insight into the task of choosing an appropriate projection function. As noted in section 3.4, the algorithm is encoded essentially directly, except that it does not cycle freely, but under user control. The statespace is built using a standard Haskell graph library, and rendered using the *graphviz* tool. Several different pool types were implemented, including queue- and stack-based.

### 4.2 Infusion Pump

Thimbleby & Oladimeji [24] discuss model discovery and analysis of a simulation of the Alaris GP infusion pump, a device for controlling drug delivery to hospital patients. The simulation and discovery tool were written in *ActionScript* on the *Flex* (<http://www.adobe.com/products/flex>) platform.

The discovery algorithm is not encoded directly (this work predates the formulation given here) but its essential strategy is followed. There are two main differences. First, the

*Pool* is a queue of unexplored states, with all of a state's actions explored in sequence, per state—a quite natural way to implement breadth-first discovery. Second, to explore from a given state, the tool first builds a list of 'useful' actions, namely those button presses (with *doGuiAction* implemented via Flex's *Button.performClick* method) which change the state, *then* it explores those actions in turn; consequently, each 'useful' button is pressed twice per state—an inefficiency which would be avoided by following our algorithm more directly. The result of *getGuiActions*, i.e. the list of all *possible* button presses, is hard-coded (see section 3.6.4); we suggest that in many cases—particularly for research purposes—this is a reasonable approach, but that for full applicability and integration into software development workflow, the more general approach advocated in this paper is clearly beneficial.

An SUD-specific inner state representation was used: a flat bundle of some 20 strings and booleans (e.g. *volumeUnit*, *pressureLevel*, *infusionMethod*), with *eq* defined naturally and no restriction (in any of our experiments) on which variables were projected. The SUD is considerably more complex than the aircon controller, with more dependent variables and a bigger statespace (and more untracked aspects of state). Consequently, and because Flash is anyway slower than compiled Haskell, discovery takes considerably longer. The SUD involves number entry, so for some analyses, *conditional exploration* (section 3.6.2) was used, with some numerical variables (e.g. flow rate) restricted to subsets of their possible values using simple bounds—e.g., if a value is at an upper bound, the action to increase it is not explored. This helped keep the statespace (and discovery) tractable; different restrictions were used at different times, depending on what was being investigated, by modifying the discovery code directly.

## 5. RELATED WORK

*GUI Ripping* [8] dynamically constructs a model of a running GUI to aid test case creation. A GUI's state is modelled as sets of widgets, properties and values; like our state spaces, *event flow graphs* model event paths, but with events on nodes, not edges. The motivation is model-based GUI testing, and there is no consideration of the underlying state and its relation with the GUI. A depth-first search algorithm is used, with Windows/C++ and Swing implementations. In [10], a tool which crawls rich AJAX web applications is described; the end product is a state space similar to ours, and the algorithm appears to be a special case of that described in this paper, with domain-specific aspects (such as handling of the browser's "back" button) tightly integrated. It would be interesting to replicate this work using our API/algorithm.

In [11, 12] a "skeleton" formal model of a GUI is generated automatically by dynamic analysis, then completed manually to produce a test oracle. The formalism used for the models is a rich pre/post specification language, *Spec#*. The GUI is explored via the operating system's window manager, so the analysis tool and SUD need not be written in the same language or run together—as the motivation is reverse engineering existing systems, this seems appropriate. GUI mod-

elling using both static and dynamic analysis is considered in [7], using *AspectJ* and thus targetting Java only. Their focus is on clean software engineering rather than HCI/usability concerns, however, and the models extracted focus on static structure rather than system behaviour.

A "specification miner" which turns a set of program API call traces into a probabilistic regular grammar or state machine modelling the system observed is described in [1]. While GUIs are not considered specifically, the probabilistic aspect is interesting: in this paper we assume complete exploration is possible—were that not the case, an approach modelling expected or observed user behaviour probabilistically might be fruitful.

On the static analysis side, [5] describes static analysis of Java/Swing systems, producing models partitioned into separate framework/GUI/model aspects; [16] describes static analysis of Swing code for reverse engineering purposes; and [17] builds on this to present a language-independent framework for such analysis, and its integration into model-based testing; finally, [19, 20] describe GUI static analysis with a focus on generic detection of GUI elements and their relationships, exemplified in the GTK and Qt frameworks.

## 6. CONCLUSIONS AND FURTHER WORK

We have presented a formal and generic description of *UI model discovery*, a lightweight formal method which, in conjunction with related analysis techniques, provides an accessible and effective way to inform improving the usability of interactive systems. We have provided a blueprint for the technique's systematic application elsewhere, and discussed some advanced features. We demonstrated that the technique can be applied in various real contexts, based on accurate simulations of actual devices. Our key contribution, however, is the reusable and abstract API for user interface discovery.

Beyond the process of discovery itself, there is of course much research to be done on techniques analysing discovered models for usability properties; this is concurrent work in progress, which we expect to inform model discovery as new requirements are identified. Examples include a number of analyses suggested in [21], checking and suggestion of use cases and test cases, and automatic computation of Myhill-equivalent [4] action sequences over the model.

Confining our attention to *discovery*, however, we see many promising areas of future research:

- Programmatic control of discovery. As described in section 3.6.7, this is a critical area of future work; in essence, this will involve extending the API to provide finer control throughout—a DSL for discovery control is then a natural extension, providing scripting capabilities and aiding integration into existing tools (see below).
- Related: further exploration of advanced features such as conditional and directed exploration.
- Our stated aim is to develop a technique sufficiently light-

weight to be used as part of existing development workflows; as such, integration into the tools and environments used, such as Eclipse, is an obvious future step.

- Implementation of model discovery in more contexts, and application to more complex examples—both of which will surely lead to further insights into the process and how it is to be managed by the investigating programmer.

## 7. ACKNOWLEDGEMENTS

We are grateful to Patrick Oladimeji and Chitra Acharya for their work in producing example simulation/discovery systems following the techniques described here, and to Max Wilson, Patrick Oladimeji and Parisa Eslambolchilar for feedback on earlier versions of this paper. This work was supported by the EPSRC grant EP/F020031.

## 8. REFERENCES

1. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.
2. M. Bidoit and P. D. Mosses. *CASL User Manual*. LNCS Vol. 2900 (IFIP Series). Springer, 2004.
3. D. Crockford. RFC 4627 - The application/json media type for JavaScript Object Notation. Technical report.
4. A. Dix. *Formal Methods for Interactive Systems*. Academic Press, 1991.
5. M. B. Dwyer, R. Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 154–163. IEEE Computer Society, 2004.
6. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
7. P. Li and E. Wohlstadter. View-based maintenance of graphical user interfaces. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 156–167. ACM, 2008.
8. A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 260. IEEE Computer Society, 2003.
9. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
10. A. Mesbah, E. Bozdogan, and A. v. Deursen. Crawling AJAX by inferring user interface state changes. In *ICWE '08: Proceedings of the 2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE Computer Society, 2008.
11. A. C. R. Paiva, J. C. P. Faria, and P. M. C. Mendes. Reverse engineered formal models for GUI testing. In *FMICS '07: International Workshop on Formal Methods for Industrial Critical Systems*, LNCS 4916, pages 218–233. Springer, 2008.
12. A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. A. M. Vidal. A model-to-implementation mapping tool for automated model-based GUI testing. In *ICFEM '05: 7th International Conference on Formal Engineering Methods*, LNCS 3785, pages 450–464. Springer, 2005.
13. S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.
14. J. Reason. Human error: models and management. *BMJ*, 320(7237):768–770, March 2000.
15. T. M. H. Reenskaug. Models - views - controllers. Technical note, Xerox PARC, 1979.
16. J. C. Silva, J. C. Campos, and J. Saraiva. Combining formal methods and functional strategies regarding the reverse engineering of interactive applications. In *Interactive Systems: Design, Specification and Verification*, volume 4323 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 2007.
17. J. C. Silva, J. Saraiva, and J. C. Campos. A generic library for GUI reasoning and testing. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 121–128. ACM, 2009.
18. J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, 2005.
19. S. Staiger. Reverse engineering of graphical user interfaces using static analyses. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 189–198. IEEE Computer Society, 2007.
20. S. Staiger. Static analysis of programs with graphical user interface. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 252–264. IEEE Computer Society, 2007.
21. H. Thimbleby. *Press On. Principles of interaction programming*. MIT Press, Boston, USA., 2007.
22. H. Thimbleby. User-centered methods are insufficient for safety critical systems. *Proceedings of USAB'07 - Usability & HCI for Medicine and Health Care*, 4799:1–20, 2007.
23. H. Thimbleby. Contributing to safety and due diligence in safety-critical interactive systems development by generating and analyzing finite state models. In *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 221–230. ACM, 2009.
24. H. Thimbleby and P. Oladimeji. Social network analysis and interactive device design analysis. In *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 91–100. ACM, 2009.
25. P. H. Winston. *Artificial Intelligence*. Addison-Wesley, 1992.