Developing GUI Applications in a Verified Setting

Stephan Adelsberger[1], Anton Setzer[2], and Eric Walkingshaw[3]

[1] Dept. of Information Systems, Vienna University of Economics, Austria
[2] Dept. of Computer Science, Swansea University, Swansea, UK
[3] School of EECS, Oregon State University, Corvallis, USA

Abstract. Although there have been major achievements in verified software, work on verifying graphical user interfaces (GUI) applications is underdeveloped relative to their ubiquity and societal importance. In this paper, we present a library for the development of verified, state-dependent GUI applications in the dependently typed programming language Agda. The library uses Agda's expressive type system to ensure that the GUI, its controller, and the underlying model are all consistent, significantly reducing the scope for GUI-related bugs. We provide a way to specify and prove correctness properties of GUI applications in terms of user interactions and state transitions. Critically, GUI applications and correctness properties are not restricted to finite state machines and may involve the execution of arbitrary interactive programs. Additionally, the library connects to a standard, imperative GUI framework, enabling the development of native GUI applications with expected features, such as concurrency. We present applications of our library to building GUI applications to manage healthcare processes. The correctness properties we consider are the following: (1) That a state can only be reached by passing through a particular intermediate state, for example, that a particular treatment can only be reached after having conducted an X-Ray. (2) That one eventually reaches a particular state, for example, that one eventually decides on a treatment. The specification of such properties is defined in terms of a GUI application simulator, which simulates all possible sequences of interactions carried out by the user.

Keywords: Agda · interactive theorem proving · GUI verification.

1 Introduction

Graphical user interfaces (GUIs) are widely used in real-world software systems. They are also a major source of bugs. For example, a study of the Mozilla project found that the web browser's GUI is the source of 50.1% of reported bugs and responsible for 45.6% of crashes [33]. Unfortunately, testing of GUIs is notoriously difficult [21,23]. The fundamental problem is that "tests must be automated, but GUIs are designed for humans to use" [30]. Automated tests must simulate user interactions, but the range of user interactions is huge, and simulated actions tend to be brittle with respect to minor changes in the GUI such as swapping the placement of two buttons.

A challenge related to GUIs is the widespread use of code generation. For example, a GUI builder enables a GUI to be graphically assembled, then generates code for the GUI which can be integrated into the rest of the application [34]. Code generation can have a negative impact on software evolution and maintenance if the GUI specification and the program logic cannot be evolved together [36]. In the worst case, handwritten customizations must be manually integrated each time the GUI code is re-generated [34]. Moreover, the dependence of handwritten code on the upstream specification is implicit.

Because of their importance and the challenges related to testing them, researchers have studied the formal verification of GUI applications using model checking [22]. However, such approaches verify only an abstracted model of the GUI rather than the software itself.

In this paper, we present a library for developing directly verified GUI applications in Agda [6]. Agda is a dependently typed programming language and interactive theorem prover. Our library supports verifying properties about GUI applications in terms of user interactions. We address the challenge of code generation by observing that such systems are *implicitly* working with *dependent types* (the hand-written parts depend on the types generated by the declarative specification), and so make this relationship *explicit*. This combines the benefits of a declarative specification with the flexibility of post-hoc customization.

Our library builds on our previous work on state-dependent object-based programming in Agda [1], which is briefly summarized in Section 2. The library itself is introduced in Section 3, beginning with an example of a GUI application written using the library, followed by a description of key aspects of the implementation. In our library, GUIs are declaratively specified as a value of an inductive data type. This *value* is used in the *types* of the controller functions that link the GUI to the underlying business logic, guaranteeing via the type system that the controller code is consistent with the GUI specification. Since the GUI specification is an inductive value, we can write arbitrary functions to modify it, improving the flexibility of GUI specifications.

In Section 5.1, we present a case study based on a healthcare process adapted from the literature [24]. Processes in the healthcare domain are mostly data-driven (e.g. patient data), include interactions with doctors (e.g. diagnosis), and are not easily modeled as finite state machines [26,24]. We demonstrate that by using dependent types, we can model such state-dependent systems with an infinite number of states.

On this note, in Sections 4 and 5, we develop a framework for specifying and proving the correctness of GUI applications. More precisely, in Section 4, we define a simulator that simulates sequences of user inputs to a GUI. Using this, we define *reachability* as whether there exists a sequence of user inputs to get from one state of the GUI to another. We then conduct a small case study to prove for a simple GUI application which states can be reached from a given state. In Section 5, we develop an example from the healthcare domain with interactive handlers (i.e. the observer pattern where an object handles GUI events) and a data-dependent GUI. We specify an *intermediate-state property* that requires passing through a specific state before reaching some other state, and we prove that the example satisfies a given property. Finally, we specify a *final-state property* that requires the application to eventually reach a given state from a start state. We show that the advanced example fulfills such a property. It turns out that the complexity lies more in the specification of the properties, while proving the properties is relatively straightforward. We discuss related work in Section 6 and conclusions in Section 7.

To summarize, the contributions of this paper are:

1. A dependently typed library for programming state-dependent GUI applications, which allows an infinite number of states and arbitrary interactive handlers.
2. A technique for specifying properties of GUIs using a simulator which simulates all possible sequences of interactions carried out by the user.
3. A framework for the verification of GUI applications involving: (1) reachability between states, (2) that one state can only be reached by passing through another state, and (3) that one eventually reaches a specific state.

Source Code. All displayed Agda code has been extracted automatically from type checked Agda code. For readability, we have hidden some bureaucratic details and show only the crucial parts of the code. The code is available at [4].

2 Background

Agda and Sized Types We recommend the short introduction to Agda that we provided in a previous paper [1]; here, we will just repeat the basics. Agda [6] is a theorem prover and dependently typed programming language based on Martin-Löf type theory. Propositions are represented as types, and a value of a type corresponds to a proof of the preposition. Agda features a type checker, a termination checker, and a coverage checker. The termination and coverage checker guarantee that every program in Agda is total. Partiality would make Agda inconsistent as a proof language.

Agda has a hierarchy of types for describing sets of types themselves, for example, the set of all types in the usual sense is called Set in Agda. The type Set_1 includes all of Set plus types that refer to Set itself. For example, in our library, we use Set_1 to define structures that have Set as one of its components (e.g. IOInterface below). Agda has function types, inductive data types, record types, and dependent function types. A dependent function type $(x : A) \to B$ represents the type of functions mapping an element $x : A$ to an element of B where B may depend on x. The variant form $\{x : A\} \to B$ of the dependent function type allows us to omit argument x when applying the function; Agda will try to infer it from typing information, but we may still apply it explicitly as $\{x = a\}$ or $\{a\}$ if Agda cannot deduce it automatically.

To represent infinite structures we use Agda's coinductive record types, equipped with size annotations [16]. The size annotations are used to show productivity of corecursive programs [2], which we define using copattern matching [3].

State-Dependent IO In previous work [1], we gave a detailed introduction to interactive programs, objects, and state-dependent versions of interactive programs and objects in dependent type theory. The theory of objects in dependent type theory is based on the IO monad in dependent type theory, developed by Peter Hancock and the second author of this article [14]. The theoretical basis for the IO monad was developed by Moggi [25] as a paradigm for representing IO in functional programming. The idea of the IO monad is that an interactive program has a set of commands to be executed in the real world. It iteratively issues a command and chooses its continuation depending on the response from the real world. Formally, our interactive programs are coinductive, i.e. infinitely deep, Peterson-Synek trees [28], except that they also have the option to terminate and return a value. This allows for monadic composition of programs, namely sequencing one program with another program, where the latter program depends on the return value of the first program. In the state-dependent version [1], both the set of available commands and the form of responses can depend on a state, and commands may modify the state.

We introduce now an IO language for GUIs. It will include also console commands and calls to external services, such as database queries. An IO language, which we call IO interface, is a record consisting of commands a program can issue, and responses the real world returns in response to these commands.

```
record IOInterface : Set₁ where
  Command  : Set
  Response  : (m : Command) → Set
```

The fields of this record type Command and Response are its projections. They can be applied also postfix using the dot notation: if p : IOInterface, then p .Command : Set. To improve readability we omit in records bureaucratic statements field, coinductive, and open.

3 State-Dependent GUI Applications

Our library separates the structure and appearance of an application's GUIs (the view) from the *handlers* that process the events produced by user interactions (the controller). This separation of concerns is similar to current practice with model-view-controller frameworks [20] and graphical GUI-builder tools [34]. A distinguishing feature of our approach is that handlers are dependently typed with respect to the GUIs they interface with. This means that GUI specifications can be programmatically generated and dynamically modified (e.g. a button may be dynamically added at runtime) without sacrificing the static guarantee of consistency with the handler objects. As a GUI dynamically changes, the interfaces of the corresponding handler objects (which methods exist and their types) dynamically change in response. Such dynamically changing GUIs are not well supported by the GUI-builder model, and the consistency guarantees are not provided by programmatic MVC frameworks.

In the rest of this section, we introduce our library. In Section 3.1, we provide an introductory example that illustrates how to build a simple state-dependent

GUI application. In Section 3.2, we turn to the implementation of the library, introducing the basic command interface provided by the library for creating and interacting with GUIs. In previous work, we have developed a representation of state-dependent objects, which we briefly describe in Section 3.3. This representation is the basis for GUI event handlers in our library. We say that our library supports *state-dependent* GUI applications since the GUI can dynamically change based on the state of the model and since the GUI is itself a dynamically changing state of the handler objects. We use state-dependent objects to define generic handlers in Section 3.4 whose interfaces are generated from the GUIs they depend on. We also introduce a data type that collects all of the components of a GUI application together. Finally, in Section 3.5, we introduce an example of a GUI having infinitely many states, where each state differs in the GUI elements used.

3.1 Introductory Example

Consider a GUI application with two states. In one state, the GUI has a single button labelled “OK”, in the other it also has a button labelled “Cancel”. Pressing the OK button in either state switches to the other state.

Creating a GUI application in our library consists of specifying the GUI, including defining the GUI elements and their properties (e.g. the color of a button), then defining the handler objects for events those elements can produce. For our example, we first specify the GUI for the first state, which consists of a frame and a single button labelled “OK”. Then, we specify the GUI for the second state by adding a button labelled “Cancel” to oneBtnGUI. Both oneBtnGUI and twoBtnGUI are of type Frame.

```
oneBtnGUI = addButton "OK" create-frame
twoBtnGUI = addButton "Cancel" oneBtnGUI
```

Finally, we specify the initial properties of GUIs (e.g., properties for twoBtnGUI), specifically the color of the button labels (black) and the frame layout.

```
propOneBtn : properties oneBtnGUI
propOneBtn = black , oneColumnLayout

propTwoBtn : properties twoBtnGUI
propTwoBtn = black , black , oneColumnLayout
```

Observe that the types of the property specifications are dependent on the corresponding GUI values, ensuring consistency between the two.

Next we define handlers for our application’s two GUI states. The handler for oneBtnGUI defines a method that handles the event generated by clicking the OK button. The handler body is an interactive IO program that prints a string indicating the button click, then uses the function changeGUI to set the currently active GUI to twoBtnGUI, updating the properties and handler accordingly.

```
obj1Btn : ∀ {i} → HandlerObject i oneBtnGUI
obj1Btn .method bt = putStrLn "OK! Redefining GUI." »
                     changeGUI twoBtnGUI propTwoBtn obj2Btn
```

This is an example of copattern matching [3]: The type of obj1Btn is a record which has one field method. We define obj1Btn by determining the value for this field, which is in postfix notation written as .method. The result is a function having further argument *bt*, so we apply it to this argument and define the result. Although unused in obj1Btn, *bt* provides access to enclosing GUI elements, such as the GUI's frame.

Note that the type of the handler object HandlerObject is parameterized by the corresponding GUI value, in this case, oneBtnGUI. The type of the handler object ensures that it has methods available to handle each kind of event that the given GUI can generate.

The hidden argument $\{i\}$ is a size parameter required since handler objects are coinductive data types. It is used to check that the definition of the handler is productive, which is the copattern matching dual to termination checking.

The handler for twoBtnGUI defines a method that uses pattern matching on the argument *bt* to determine which button was clicked; this is an example of combined pattern/copattern matching.

```
obj2Btn : ∀ {i} → HandlerObject i twoBtnGUI
obj2Btn .method (firstBtn bt) = putStrLn "OK! Redefining GUI." »
                                 changeGUI oneBtnGUI propOneBtn obj1Btn
obj2Btn .method (secondBtn bt) = putStrLn "Cancel!" » keepGUI obj2Btn
```

If the click originated from the first button (OK), the active GUI is changed back to oneBtnGUI. Otherwise, if it originated from the second button (Cancel), then twoBtnGUI is retained using the library function keepGUI.

Finally, we can compile our GUI application into a NativeIO program in Agda. NativeIO is a type that represents IO programs in Haskell. This is done by calling compileProg with the arguments twoBtnGUI, propTwoBtn, and obj2Btn. The compiler of Agda translates the resulting NativeIO program via special Haskell FFI commands present in Agda into a GUI Haskell program which makes use of the wxHaskell [37] GUI toolkit.

3.2 GUI Interface

The type GuiInterface defines commands for interacting with the console, the GUI (e.g., changing a label), and communicating with a database, and the responses to those commands. Here, we only list the commands used later in this paper (console commands), see the repository [4] for the full list:

```
data GuiCommand : Set where
  putStrLn : String → GuiCommand
  getLine  : GuiCommand

GuiResponse : GuiCommand → Set
GuiResponse getLine = String
GuiResponse _       = Unit

GuiInterface : IOInterface
GuiInterface .Command  = GuiCommand
GuiInterface .Response = GuiResponse
```

3.3 State-dependent Interfaces

The handling of GUI commands will be implemented with state-dependent objects where the interface may change according to the state of the object. An object is similar to an interactive program except that it is a server-side program. An object can receive commands, and depending on the responses to those commands it returns a result and changes its internal state. With Agda being a purely functional language, we model state changes by returning an updated object together with the regular return value as a pair. In dependent type theory we also have state-dependent objects, where the methods available depend on the state of the object. Depending on the response for the method call, the object switches to a new state. So an interface for a state-dependent interface $\mathsf{Interface}^{\mathsf{s}}$ ($^{\mathsf{s}}$ indicating that it is state-dependent) consists of a set of states, a set of methods depending on the state, a set of responses depending on states and methods, and a next state function, which depends on states, methods and responses:

```
record Interfaceˢ : Set₁ where
  State  : Set
  Method : State → Set
  Result : (s : State) → (m : Method s) → Set
  next   : (s : State) → (m : Method s) → (Result s m) → State
```

We note here that a method in this framework is what corresponds to the union of all the methods together with all their arguments in normal object-oriented languages. The reason why we can bundle them together is because the result type can depend on the method, therefore there is no need for separate methods with separate result types. An Object for this interface is a program that accepts calls to objects methods (method). In response to an object method, it returns a result and an updated Object. Since this interaction might go on forever, the definition of an Object is coinductive.

3.4 Implementation of Generic GUIs

In the previous subsection, we saw that the type of a handler object depends on the value of the GUI it supports. To help understand how handler objects work, we start by taking a closer look at the type of generic handler objects.

```
HandlerObject : ∀ i → Frame → Set
HandlerObject i g = IOObjectˢ GuiInterface handlerInterface i g
```

This library function defines the type of a handler object, given a size index i and a frame that represents the GUI the handler processes events for. As described in Section 3.1, i is used to ensure that the handler is productive, a well-formedness property of co-inductive definitions.

The type of a handler is a state-dependent object that supports GuiInterface commands (see Section 3.2). The state of the object is the GUI specification; it is parameterized by a Frame, which determines the interface of the object as defined below.

```
handlerInterface : Interfaceˢ
handlerInterface .State        = Frame
handlerInterface .Method f     = methodsG f
handlerInterface .Result f m   = returnType f
handlerInterface .next f m r   = nextStateFrame f r
```

We need a type for the methods, which depend on the frame. As mentioned before all individual methods of an object are bundled together into one single one. For each individual GUI component such as a button, and each event corresponding to this component, we require one method for handling it. The method for a component is the sum of all the methods for its events, and the methodsG function creates the sum of all the methods of each component of the frame. When an individual event is called, it obtains as arguments the parameters of this call. Since the parameters are part of the method, an event method is the product of the parameters of this method call together with an element representing all the components of the frame. It is the task of the user to implement these methods, when he creates a handler object for a frame definition.

Since a handler object's interface (i.e. what methods it provides to handle GUI events) is determined by its state, the interface dynamically updates with corresponding changes to the GUI specification.

An event handler method is an interactive program that has three possible return options:

```
data returnType (f : Frame) : Set where
  noChange          : returnType f
  changedAttributes : properties f → returnType f
  changedGUI        : (fNew : Frame) → (properties fNew) → returnType f
```

In the first case, the GUI remains unchanged. In the second case, we simply return the changed properties. In the third case we transform the given Frame into a newly created GUI.

The function nextStateFrame carries out the calculation of the new successor state after a method is finished. The state is updated only in case of the return option changedGUI.

```
nextStateFrame : (f : Frame)(r : returnType f) → Frame
nextStateFrame f noChange                = f
nextStateFrame f (changedAttributes x)   = f
nextStateFrame f (changedGUI fNew x)     = fNew
```

3.5 A GUI with an Unbounded Number of States

In this subsection, we present an introductory example that both illustrates the use of the GUI data type and demonstrates that we can develop GUIs with infinitely many states where each state differs in the GUI elements used. The example is a GUI application with n buttons, where clicking any button expands the GUI into one with $n+1$ buttons.[4] The helper function nFrame constructs a

[4] The library contains a more interesting example [4] where clicking button b_i extends the GUI with i additional buttons.

GUI with n buttons, while nProp defines its corresponding properties (a black label for each button organized in a one-column layout). The nGUI function constructs a GUI application with n buttons combining the GUI, its properties, and a handler that for any button press replaces the GUI application with a new one containing $n + 1$ buttons.

```
nFrame : (n : ℕ) → Frame
nFrame 0       = create-frame
nFrame (suc n) = addButton (show n) (nFrame n)

nProp : (n : ℕ) → properties (nFrame n)
nProp 0       = oneColumnLayout
nProp (suc n) = (black , nProp n)

nGUI : ∀{i} → (n : ℕ) → GUI {i}
nGUI n .defFrame         = nFrame n
nGUI n .property         = nProp n
nGUI n .obj .method m = changeGUI (nGUI (suc n))
```

The type GUI represents a GUI application. It is a record with three fields: .defFrame defines the GUI, .property specifies its properties, and .obj contains the GUI's handler object, whose field .method is invoked when a button is clicked. The argument m to the handler method indicates which button was clicked, but in this case, we ignore it since clicking any button updates the GUI to add one more button. Note that the GUI application defined above is dynamically expanding, which is difficult to design using standard GUI builders since they allow constructing only finitely many GUIs for a particular application.

4 Proof of Correctness Properties of GUIs

We reason about GUI applications by reasoning about the GUI's states. The state of a GUI is given by a frame, its properties, and its handler object. When an event is triggered, an IO program is executed, and the return value determines which state to transition to. Thus, we can reason about the transition graph for a GUI application by reasoning about the exit points of the handler. However, a complication is that IO programs are coinductive, meaning they may have an unbounded number of interactions and never terminate. Ideally, IO programs for event handlers would be inductive since we typically want event handlers to always terminate so that the GUI is responsive. However, this is much more difficult to integrate within a general GUI framework since GUI applications are naturally coinductive.

4.1 A Simulator for GUI Applications

To cope with coinductive IO programs in event handlers, we do not reason about GUI states directly, but instead introduce an intermediate model of a GUI application, where the IO programs in handlers are unrolled into potentially infinitely many states. This model is itself coinductive and we cannot reason about it directly since an infinite sequence of IO commands will induce an infinite number of states. Therefore, we instead reason about *finite simulations* of the GUI model.

To define the model, we first introduce a data type to indicate whether an event handler has been invoked or not. The notStarted constructor indicates that the handler has not yet been invoked, while started indicates the handler has been invoked. The constructor started has as an additional argument *pr* corresponding to the IO program still to be executed.

```
data MethodStarted (f : Frame) (prop : properties f)
                   (obj : HandlerObject ∞ f) : Set where
  notStarted : MethodStarted f prop obj
  started    : (m : methodsG f) (pr : IO' GuiInterface ∞ StateAndGuiObj)
             → MethodStarted f prop obj
```

The handler is parameterized by the size value ∞. Sizes are ordinals that limit the number of times a coinductive definition can be unfolded. There is an additional size ∞ for coinductive definitions that can be unfolded arbitrarily many times. A more detailed explanation of sizes and ∞ can be found in Section 3 of [17].

Now, a state in the GUI model can be represented by the GUI, its properties, the handler, and the invocation state.

```
data ModelGuiState : Set where
  state : (f : Frame) (prop : properties f) (obj : HandlerObject ∞ f)
          (m : MethodStarted f prop obj) → ModelGuiState
```

Using this model, we can simulate the execution of GUI applications. To do this, we define a simulator for state-dependent IO programs. Depending on the state of the GUI model, the simulator must trigger GUI events or provide responses to IO commands, then move to the next state in the model. The following function defines the available actions at each state in the model.

```
modelGuiCommand : (s : ModelGuiState) → Set
modelGuiCommand (state g prop obj notStarted)              = methodsG g
modelGuiCommand (state g prop obj (started m (exec' c f))) = GuiResponse c
modelGuiCommand (state g prop obj (started m (return' a))) = ⊤
```

If the model is in a notStarted state, the event simulator can trigger an event drawn from the methods supported by the GUI interface. If the model is in a started state, then there are two sub-cases: If the IO program has not finished, the program has the form (exec' c f). This means that the next real-world command to be executed is c and once the world has provided an answer r to it, the interactive program continues as (f r). (Previously we used do instead of exec, but do has now become a keyword in Agda.) In this case the GUI is waiting on a response to the IO command c, which the event simulator must provide. The second subcase is, if the remaining IO program has already returned. Then the simulator can take the trivial action (⊤) to return to the notStarted state.

Using this definition, we can define a transition function for the simulator with the following type:

```
modelGuiNext : (s : ModelGuiState) (c : modelGuiCommand s) → ModelGuiState
```

That is, given a model state and an action of the appropriate type, we can transition to the next model state.

To simplify proofs over the model, the transition function makes a few optimizations. First, in the case where the new state corresponds to a completed IO program, we can skip to the next notStarted state directly rather than requiring this unit step be made explicitly. Second, we reduce sequences (shorter than a given finite length) of consecutive trivial IO actions, such as print commands, into single transition steps.

We can define a state-dependent IO interface (an element of IOInterface^s) for the simulator (see Section 2)), which incorporates the previous definitions in a straightforward way.

```
modelGuiInterface : IOInterfaceˢ
modelGuiInterface .State              = ModelGuiState
modelGuiInterface .Command            = modelGuiCommand
modelGuiInterface .Response s m       = ⊤
modelGuiInterface .next     s m r     = modelGuiNext s m
```

Using this, we define a relation between states s and s' of the GUI, which states that s' is reachable from s if running the simulator from s can produce s'.

```
_-gui->_ : (s s' : ModelGuiState ) → Set
s -gui-> s' = IOˢind_p0 modelGuiInterface s s'
```

Here, s -gui-> s' expresses that we can get from s to s' in a finite number of steps. It is therefore defined inductively which allows to reason about it inductively. Finally, we introduce a one-step relation that states that s is reachable from s' by executing one step of the simulator.

```
data _-gui->¹_ (s : ModelGuiState ) : (s' : ModelGuiState)→ Set where
  step : (c : modelGuiCommand s) → s -gui->¹ modelGuiNext s c
```

The correctness proofs using the simulator are included in the code repository [4].

5 State Transition Properties

In this section, we demonstrate the definition and proof of properties relating to state transitions using an example from the healthcare domain. In Section 5.1, we consider the property that any path from one state to another in a GUI application must pass through a given intermediate state. In Section 5.2, we consider the property that all paths through a GUI application end up in the same final state. In both cases, the main challenge is to express the property to be proved, while the proof is relatively straightforward.

Such properties are well covered by existing approaches based on model checking [22]. However, the advantage of our approach is that we prove such properties for the implementation of the GUI application directly, rather than for an abstracted model of it.

5.1 Intermediate-State Properties

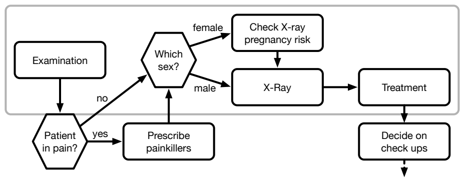

Fig. 1: Process model of a fracture treatment process.

Consider the healthcare process illustrated in Figure 1, which is adapted from [24]. The relevant part of the process is highlighted. Specifically, it consists of four states corresponding to steps in the process: an initial examination, performing an X-ray, assigning treatment, and a risk check for pregnancy in which the patient is asked about a potential pregnancy. The last state is only performed for female patients. In this subsection, we will build a GUI application abstracting this part of the process and prove that all paths from the initial examination to treatment pass through the intermediate X-ray state. We define a generic mechanism for expressing such intermediate-state properties, which illustrates how other properties on GUI applications can be defined.

Below is the straightforward specification of the GUI for the initial examination state. The GUIs for other states are defined similarly.

```
frmExam : Frame
frmExam = addBtn "Examination" create-frame
```

The controller for the initial examination state is shown below. After pressing the button, the system interactively asks whether the patient is female or male. If the answer is female, the controller invokes changeGUI to change the application to the pregnancy-test state. (A version which includes a check of the correctness of the user input is discussed in [5].) Otherwise, the controller changes to the X-ray state. This is an example of a data-dependent GUI with interaction.

```
hdlExam : ∀ i → HandlerObject i frmExam
hdlExam i .method {j} (btn , frm) =
  exec (putStrLn "Female or Male?") λ _ →
  exec getLine λ s →
  hdlExamProgEnd i (string2Sex s)

hdlExamProgEnd : (i : Size)(g : Sex) → HandlerIOType i frmExam
hdlExamProgEnd i female = changeGUI frmPreg propOneBtn (hdlPreg i)
hdlExamProgEnd i male   = changeGUI frmXRay propOneBtn (hdlXRay i)
```

The GUI and controller for the X-ray state is not shown, but it provides a single button that when pressed transitions to the treatment state.

To reason about our GUI application, we define the corresponding coinductive model as described in Section 4.1. We first model the initial examination state as stateExam along with two intermediate states corresponding to interactions with the user: $\mathsf{stateExam}_1$ is the state reached after querying the sex of the patient, while $\mathsf{stateExam}_2$ is the state reached after the user provides a response.

```
stateExam : ModelGuiState
stateExam = state frmExam propOneBtn (hdlExam ∞) notStarted

stateExam₁ : (c : methodsG frmExam) → ModelGuiState
stateExam₁ = modelGuiNext stateExam

stateExam₂ : (c : methodsG frmExam) (str : String) → ModelGuiState
stateExam₂ c str = modelGuiNext (stateExam₁ c) str
```

Similarly, we model the perform-X-ray state as stateXRay and the assign-treatment state as stateTreatm.

We expect that our GUI application implements the healthcare process in Figure 1. As mentioned, we want to ensure that we never assign a treatment without first performing an X-ray. To support stating such a property, we define a predicate (*path* goesThru s'), which expresses that a *path* from states s to s' passes through a state t. The path from s to s' is a member of the type -gui->. The predicate is defined inductively, where the constructors exec_i and return_i are the inductive forms of the usual exec and return constructors for coinductive programs.

```
_goesThru_ : {s s' : ModelGuiState}(q : s -gui-> s')(t : ModelGuiState) → Set
_goesThru_ {s} (execᵢ c f) t = s ≡ t ⊎ f _ goesThru t
_goesThru_ {s} (returnᵢ a) t = s ≡ t
```

In the exec_i case, the current state is either equal to t or all subsequent states must pass through the t. In the return_i case, the path is the trivial path, and therefore the current state must be equal to t.

Now we can show that any path through the GUI application from the initial-examination state to the treatment state passes through the X-ray state. We need to prove this property not only for the initial state but also for all intermediate states. The proof for the initial state is shown below by matching on exec_i and showing that all subsequent states have this property. We match the case return_i with the empty case distinction (indicated by ()), which is always false.

```
examGoesThruXRay : (p : stateExam -gui-> stateTreatm) → p goesThru stateXRay
examGoesThruXRay (execᵢ c f) = inj₂ (exam₁GoesThruXRay c (f _))
examGoesThruXRay (returnᵢ () )
```

Cases for most of the other states are similarly straightforward. However, we show in detail the interesting case of $\mathsf{stateExam}_2$ in which we need to make a case distinction on the value of (string2Sex *str*). For this, we use the magic with construct, which allows extending a pattern by matching on an additional expression. The symbol ...| expresses that the previous pattern is repeated, extended by a pattern matching on the with-expression.

```
exam₂GoesThruXRay : (c : methodsG frmExam) (str : String)
  (path : stateExam₂ c str -gui-> stateTreatm) → path goesThru stateXRay
```

```
exam₂GoesThruXRay c str path with (string2Sex str)
... | male   = XRayGoesThruXRay path
... | female = checkPregGoesThruXRay path
```

The main difficulty in this proof is to recognize the need for the intermediate states stateExam$_1$ and stateExam$_2$. Once these intermediate states are made explicit and the property is defined, the proof itself is straightforward.

5.2 Final-State Properties

Another property we might want to prove about our healthcare process application is that all paths eventually lead to a treatment. To support the definition and proof of such properties, we define the following inductive data type, which states that all paths from an initial state *start* eventually reach a state *final*.

```
data _-eventually->_ : (start final : ModelGuiState) → Set where
  hasReached : {s : ModelGuiState} → s -eventually-> s
  later : {start final : ModelGuiState} (fornext : (m : modelGuiCommand start)
            → modelGuiNext start m -eventually-> final) → start -eventually-> final
```

The constructors for this type state that the property holds if either the current state is the state to be reached (hasReached) or all subsequent states have this property (later). Since the data type is inductive, this expresses that eventually the final state is reached.

We now show that in the GUI application defined in Section 5.1, the treatment state is always reached. Again we prove this property for all states between the initial state and the treatment state. Once again, the interesting case is stateExam$_2$, where we need to make a case distinction on Sex as before.

```
exam₂EventuallyTreatm : (c : methodsG frmExam) (str : String)
   → (stateExam₂ c str) -eventually-> stateTreatm
exam₂EventuallyTreatm c str with (string2Sex str)
... |   female = checkPregEventuallyTreatm
... |   male   = xRayEventuallyTreatm
```

Here we see the benefits of defining GUIs in a generic way—proving properties about them is straightforward since one can follow the states of the GUI as given by our data type.

6 Related Work

In our previous article [1], we introduced an Agda library for object-based programs. We demonstrated the development of basic, static GUIs. In this paper, we have extended this work by adding a declarative specification of GUIs and the GUI-creation process is now automatic. We also demonstrate the verification of GUI applications in Sections 4–5.

We have developed an alternative version of this library to use a simpler, custom-built backend [5], rather than the wxHaskell backend used in this paper. The newer backend supports a much simpler version of the GUI interface types described in Section 3.2 and the handler objects described in Section 3.4 (in fact, much of the complexity of these types is not exposed in this paper for presentation reasons; for full details, see the library [4]), which simplifies proofs and improves the scalability of our approach. However, the more complex approach described in this paper is more generic and has the advantage of being built on an existing and widely used GUI toolkit (wxWidgets). In particular, this version supports GUIs with nested frames, separates properties from the GUIs they apply to, allows modifying properties without redrawing the entire GUI, and supports adding new components from wxWidgets relatively easily. Additionally, the wxHaskell backend supports concurrency and integrates better with the host operating system than our custom backend. The downside is that wxHaskell is an inherently imperative and concurrent toolkit, which makes interfacing with Agda non-trival and leads to the increased complexity of this version (for the technical details, again see the library [4]). In addition, in [5], we add a representation of business processes in Agda and automatically compile such processes into executable GUIs. Taking advantage of the simpler design, in [5] we also implement a larger, realistic case study but proved only reachability statements, whereas in this paper we perform a simpler case study but also cover intermediate state properties.

Functional Reactive Programming (FRP) is another approach for writing GUIs in functional languages [35], where a behaviour is a function from time to values, and an event is a value with occurrence time. In connection with dependent types, FRP has been studied from the foundational perspective [31] and for verified programming [18].

Formlets [12] are abstract user interface that define static, HTML based forms that combine several inputs into one output. However, they do not support sequencing multiple intermediate requests.

Formalizations in Isabelle of end-user-facing applications have been studied for distributed applications [8] and conference management systems [19]. However, only the respective core of the server is verified.

Process Models in the Healthcare Domain were specified using Declare [24], also with extensions to model patient data [26]. In the current paper, in contrast to [24,26,11] and other approaches, we apply formal verification using a theorem prover (Agda) and provide machine-checked proofs. We found only two papers using formal specifications: Debois [13] proves in Isabelle a general result of orthogonality of events. Montali et al. [27] developed models via choreographies. But the latter is limited to finite systems and doesn't deal with interactive events.

Furthermore, healthcare processes are safety critical and testing GUI applications is a major challenge [21]. Sinnig et al. [32] argued that it is important to formalize business processes together with the design information/requirements of the user interface (e.g. GUI) in a single framework. This is directly supported by our library and a benefit of our use of dependently typed GUI programming.

Verification of user interfaces for medical devices. An approach to the verification of user interfaces based on a combination of theorem proving and model checking is presented in [15]. In particular, [15] focuses on the problem of how to demonstrate that a software design is compliant with safety requirements (e.g., certain FDA guidelines). Their solution is elegant in their combined use of model checking, theorem proving, and simulation within one framework. A difference to our work is that [15] verifies models of devices while we verify the software (e.g., handlers of GUI events) directly. Furthermore, we allow the verification of GUI software with an infinite number of states which is not the focus of [15].

Idris and Algebraic Effects. Bauer and Pretnar [7] introduced algebraic effects. Brady [9] adapted this approach to represent interactive programs in Idris [10]. In [1], Sect. 11 we gave a detailed comparison of the IO monad in Agda and algebraic effects in Idris and a translation between the two approaches. Regarding GUIs, we only found a forum post [29], which shows that GUI programming should be possible using the FFI interface of Idris but has yet to be performed.

7 Conclusion and Future Work

Verification of GUI-based programs is important because they are widely used, difficult to test, and many programs are safety critical. We demonstrate a new approach to generically representing and verifying GUI applications in Agda. Our approach makes essential use of dependent types to ensure consistency between the declarative GUI specification and the rest of the system. We demonstrated a generic mechanism for expressing intermediate-state properties. For example, we proved that between the initial examination and the treatment, the GUI application must pass through the intermediate X-ray state. We also considered the property that all paths through a GUI application arrive at a particular final state. Finally, we presented the generation of working GUI applications, including GUI programs with an unbounded number of states.

A limitation of our current approach is that, although the underlying GUI framework supports concurrency, we do not have a way to represent or reason about this explicitly in our library. Concurrency is important for defining GUI applications that remain responsive while a long-running event handler executes. We are working on an extension to our library that allows introducing and reasoning about concurrency, such as defining multiple threads and proving that they are fairly executed.

Acknowledgments The first and second author were supported by the project CA COST Action CA15123 European research network on types for programming and verification (EUTYPES). The second author was additionally supported by CORCON (Correctness by Construction, FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638), COMPUTAL (Computable Analysis, FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962), CID (Computing with Infinite Data, Marie Curie RISE project, H2020-MSCA-RISE-2016-731143). The third author was supported by AFRL Contract FA8750-16-C-0044 under the DARPA BRASS program.

References

1. Abel, A., Adelsberger, S., Setzer, A.: Interactive programming in Agda – objects and graphical user interfaces. Journal of Functional Programming **27**, 38 (Jan 2017), `https://doi.org/10.1017/S0956796816000319`
2. Abel, A., Pientka, B.: Well-founded recursion with copatterns and sized types. JFP **26**, e2 (2016), iCFP 2013 special issue
3. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming infinite structures by observations. In: POPL'13. pp. 27–38. ACM, New York (2013)
4. Adelsberger, S., Setzer, A., Walkingshaw, E.: Deveoping GUI applications in a verified setting (2017), `https://github.com/stephanpaper/SETTA18`, git respository
5. Adelsberger, S., Setzer, A., Walkingshaw, E.: Declarative GUIs: Simple, consistent, and verified. In: Int. Conf. on Principles and Practice of Declarative Programming (PPDP). ACM (2018)
6. Agda Community: Agda Wiki (2017), `http://wiki.portal.chalmers.se/agda`
7. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers (2012), `http://arxiv.org/abs/1203.1539`, arXiv
8. Bauereiß, T., Gritti, A.P., Popescu, A., Raimondi, F.: CoSMeDis: A distributed social media platform with formally verified confidentiality guarantees. In: 2017 Symposium on Security and Privacy. pp. 729–748. IEEE, US (2017)
9. Brady, E.: Resource-dependent algebraic effects. In: Hage, J., McCarthy, J. (eds.) TFP'14, Netherlands, 2014. pp. 18–33. Springer, Cham (2015)
10. Brady, E.: Type-driven Development with Idris. Manning Publications, Greenwich, Connecticut, 1 edn. (2017)
11. Chiao, C.M., Künzle, V., Reichert, M.: Towards object-aware process support in healthcare information systems. In: eTELEMED'12. IARIA, Delaware, US (2012)
12. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: The essence of form abstraction. In: APLAS'08. pp. 205–220. Springer, Berlin (2008)
13. Debois, S., Hildebrandt, T., Slaats, T.: Concurrency and asynchrony in declarative workflows. In: BPM'15. pp. 72–89. Springer, Cham (2015)
14. Hancock, P., Setzer, A.: Interactive programs in dependent type theory. In: CSL'00. pp. 317–331. Springer, Berlin/Heidelberg (2000)
15. Harrison, M.D., Masci, P., Campos, J.C., Curzon, P.: Verification of user interface software: the example of use-related safety requirements and programmable medical devices. IEEE Trans. Hum.-Mach. Syst. **47**(6), 834–846 (2017)
16. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL'96. pp. 410–423. ACM, New York, NY, USA (1996)
17. Igried, B., Setzer, A.: Defining trace semantics for CSP-Agda (30 Jan 2018), `http://www.cs.swan.ac.uk/~csetzer/articles/types2016PostProceedings/igriedSetzerTypes2016Postproceedings.pdf`, accepted for publication in Postproceedings TYPES 2016, 23 pp, avilable from `http://www.cs.swan.ac.uk/~csetzer/articles/types2016PostProceedings/igriedSetzerTypes2016Postproceedings.pdf`
18. Jeffrey, A.: LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In: PLPV '12. ACM, New York (2012)
19. Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: CAV'14. pp. 167–183. Springer (2014)
20. Krasner, G.E., Pope, S.T.: A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. JOOP **1**(3), 26–49 (1988)
21. Memon, A.M.: GUI testing: Pitfalls and process. Computer **35**(8), 87–88 (2002)
22. Memon, A.M.: An event-flow model of GUI-based applications for testing. Software Testing, Verification and Reliability **17**(3), 137–157 (2007)
23. Memon, A.M., Xie, Q.: Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. IEEE T SOFTWARE ENG **31**(10), 884–896 (2005)

24. Mertens, S., Gailly, F., Poels, G.: Enhancing declarative process models with dmn decision logic. In: BPMDS'15. pp. 151–165. Springer, Cham (2015)
25. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55 – 92 (1991)
26. Montali, M., Chesani, F., Mello, P., Maggi, F.M.: Towards data-aware constraints in Declare. In: SAC'13. pp. 1391–1396. ACM (2013)
27. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographies. ACM Trans. Web **4**(1), 3:1–3:62 (Jan 2010)
28. Petersson, K., Synek, D.: A set constructor for inductive sets in Martin-Löf's type theory. In: CTCS'89. vol. 389, pp. 128–140. Springer-Verlag, London, UK (1989)
29. Pinson, K.: GUI programming in Idris? (19/12 2015), `https://groups.google.com/forum/#!topic/idris-lang/R_7oixHofUo`, google groups posting
30. Ruiz, A., Price, Y.W.: Test-driven GUI development with TestNG and Abbot. IEEE Software **24**(3), 51–57 (2007)
31. Sculthorpe, N., Nilsson, H.: Safe functional reactive programming through dependent types. In: ICFP'09. pp. 23–34. ACM (2009)
32. Sinnig, D., Khendek, F., Chalin, P.: Partial order semantics for use case and task models. Formal Aspects of Computing **23**(3), 307–332 (2011)
33. Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C.: Bug characteristics in open source software. Empirical Software Engineering **19**(6), 1665–1705 (2014)
34. Valaer, L.A., Babb, R.G.: Choosing a user interface development tool. IEEE Software **14**(4), 29–39 (1997)
35. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: PLDI'00. pp. 242–252. ACM, New York (2000)
36. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Software **31**(3), 79–85 (2014)
37. wiki: WxHaskell (Retrieved 9 Feb 2017), `https://wiki.haskell.org/WxHaskell`