Declarative GUIs: Simple, Consistent, and Verified

Stephan Adelsberger
Department of Information Systems and Operations
Vienna University of Economics
A-1020 Vienna, Austria, Europe
sadelsbe@wu.ac.at

Anton Setzer
Department of Computer Science
Swansea University
Swansea SA2 8PP, UK
a.g.setzer@swansea.ac.uk

Eric Walkingshaw
School of EECS
Oregon State University
Corvallis, OR, USA
walkiner@oregonstate.edu

Abstract
Graphical user interfaces (GUIs) are a major source of bugs in real-world software systems that are notoriously difficult to prevent through traditional type systems or automated testing. Although there have been major achievements in verified software, work on verifying 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 means 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; they may involve both infinitely many states and the execution of arbitrary interactive programs.

To demonstrate the practical application of our library to develop verified GUI applications in a safety-critical domain, we present a case study developed in cooperation with the Medical University of Vienna. The case study implements a healthcare process for prescribing anticoagulants, which is highly error-prone when followed manually. Our implementation generates GUIs from an abstract description of a data-aware business process, making our approach easy to reuse and adapt to other safety-critical processes. We prove medically relevant safety properties about the executable GUI application, such as that given certain inputs, certain states must or must not be reached. The specification of such properties is defined in terms of a GUI application simulator, which conceptually simulates all possible sequences of interactions performed by the user.

ACM Reference Format:

1 Introduction
Graphical user interfaces (GUIs) are ubiquitous in modern software and form an integral part of many safety-critical systems. Due to the widespread use of interface builders and user-interface markup languages to define GUI layouts and transitions, GUIs are also a major practical application of code generation from declarative specifications. The role of declarative GUI specifications in GUI application development has a clear benefit: it provides an abstraction boundary between the user interface and the business logic. This allows user experience designers to work independently from software engineers, to develop GUIs at a higher level of abstraction, and to quickly mock-up and experiment with different GUIs independently of the rest of the application [70].

1.1 Challenges of Declarative GUIs
A supposed benefit of declarative programming is that it leads to more correct and more maintainable software. Unfortunately, this promise is not realized by the GUI specification languages currently in use. In practice, GUIs are a major source of bugs. For example, the GUI for the Mozilla project is specified using the declarative XML User-Interface Language (XUL), but a study of the Mozilla project found that the GUI is still the source of 50.1% of reported bugs and responsible for 45.6% of crashes [69]. Additionally, other researchers have noted the difficulty in maintaining consistency between GUI specifications, the underlying business logic [70], and the corresponding test suites [25].

Besides typical maintenance challenges associated with mixing generated and handwritten code [77], GUI applications pose special challenges to ensuring software quality. The most significant is that traditional software testing of GUIs is notoriously difficult [48, 51], and that traditional measures of test quality, such as code coverage, are not useful in the context of GUIs [50]. The core challenge to testing GUIs is that automated tests must simulate user interactions, but

1https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL
the range of interactions is huge and the simulated actions tend to be brittle with respect to minor changes in the GUI specification such as swapping the placement of two buttons.

Since many GUI applications are safety critical, there is a need for stronger correctness guarantees related to GUIs than software testing can provide. Others have studied the formal verification of GUI applications using model checking [49]. However, such approaches verify only an abstracted model of the GUI rather than the software itself.

1.2 A Library for Directly Verified GUIs

To address the need for strong correctness guarantees in safety-critical situations, we have developed a library for building directly verified GUI applications in Agda [6]. Agda is a dependently typed programming language and interactive theorem prover. Our library is unique in that the same declarative GUI specification that is used to generate the GUI and bindings to the business logic, is also a first-class value that can be used in Agda types and proofs. This enables us to statically guarantee basic consistency properties between the GUI and the business logic for all GUI applications built in our library. It also enables us to define and prove sophisticated properties about the behavior of GUI applications in terms of their specifications.

A major contribution of this paper is a demonstration that our declarative, directly verified approach to GUI application development can be applied to a realistic problem in a safety-critical domain. Specifically, after briefly introducing our library in Section 3, we present an extended case study developed in cooperation with the Medical University of Vienna. The case study implements a healthcare process concerning the prescription of blood thinners used in the emergency room of the Vienna General Hospital.

1.3 Domain: GUI Applications in Healthcare

Medication errors are a serious problem in healthcare, perhaps even the third leading cause of death in the US [47]. The newest blood-thinning drugs called NOACs (novel oral anticoagulants) are especially problematic. They are widely prescribed but studies have found that as many as 11–16% of NOAC prescriptions are medication errors [32, 79]. Prescribing NOACs is difficult because they have complex exclusion and dosage criteria, leading to human errors.

Health information technology (HIT) systems can reduce medication errors [61]. However, the HIT system itself can introduce a new class of errors related to software bugs. Magrabi et al. [45] provide a detailed overview of HIT-related errors. One noteworthy example is that Trinity Health System (at the time, operating 46 hospitals in the US) reported an error where its GUI application would post doctors’ orders to the wrong medical charts [45]. There is evidence that HIT errors are widespread and that many of these errors are specifically related to GUIs. The Pennsylvania Patient Safety Authority [43] reported 889 medication errors attributed to HIT systems within six months. Graber et al. [24] reported 248 malpractice claims attributed to HIT systems. Finally, a review of FDA data in 2011 [56] found 120 error reports related to HIT systems; 50.2% of these errors were related to the user interface, while 20.8% were related to calculation errors or other software bugs.

Potential software errors and liability issues are a likely reason that medication prescriptions are still usually performed manually, even though this is also error prone for some medications, such as NOACs.

In our case study, we use our library to build a GUI application that implements part of a realistic healthcare process for prescribing NOACs. First, we develop a declarative specification of the process in collaboration with a doctor at Vienna General Hospital. Next we develop a verified program that implements this process as an executable GUI application. The application is statically guaranteed to be consistent with the process specification. Finally, we define and prove additional medically relevant properties about the process and the GUI application. Our case study demonstrates how formal verification can be used to rule out many of the errors that plague safety-critical HIT systems. Notably, our approach enables directly verifying the healthcare process itself, the business logic that realizes that process, and the executable GUI program all within a single framework.

1.4 Role of Dependent Types

Our case study also illustrates the advantages of declarative specifications as first-class values. In our library, GUI specifications are values of a coinductive data type. This means we can write arbitrary functions to query and modify GUI specifications. Thus, in implementing the NOAC prescription process, we do not define each stage of the GUI application directly, but instead generate them from the process specification (which is itself an inductive value). The required GUI specifications are generated and used as dependently typed inputs to the rest of the application generation code, ensuring that the GUIs are consistent with the controllers that realize the parent process specification. This demonstrates that first-class declarative specifications support both reuse and correctness. Once we have a data type for specifying healthcare processes and functions for translating them into GUI applications, we can more easily generate GUI applications for other healthcare processes and immediately get the same strong consistency guarantees demonstrated in our case study.

Dependent types are also central to proving medically relevant properties about healthcare processes and their realization as executable GUI applications. Of course, by exploiting the Curry-Howard correspondence we can define the properties that we want to prove as types and prove them by

---

2We use a coinductive rather than inductive data type because we allow potentially infinite sequences of interactions.
To summarize, this paper makes the following contributions:

- A library for programming state-dependent GUI applications, which has at its core a type for defining first-class, generic, declarative GUI specifications, and which make essential use of dependent types. This library goes beyond finite state machines in two ways: it allows for infinitely many states, and it supports arbitrary real-world interactions (i.e. arbitrary IO actions) when transitioning from one GUI state to the next.

- A demonstration using a real-world case study that our approach can be used to develop safety-critical GUI applications in the medical domain. This demonstration illustrates how, using our library we can:
  - Generate GUI specifications from a high-level, data-dependent process description in a way that is guaranteed to be type safe and consistent with both the process description and the business logic.
  - Verify data-dependent properties of the actual executable GUI application, such as the reachability and unreachability of GUI states.
  - Perform all of the above within the same framework and language, avoiding the need to define external models of the application or translate between frameworks, which are potential sources of errors.

**Source Code.** All displayed Agda code has been extracted automatically from type-checked Agda code [5]. For readability, we have hidden some details and show only the crucial parts of the code. The full source code is available online.³

³https://github.com/stephanadelsb/PPDP18

## 2 Background

**Agda and Sized Types** Agda [6] is a theorem prover and dependently typed programming language based on Martin-Löf type theory. In Agda, propositions are represented as types; a proposition is proved by providing a value of its type. 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, which is required for the consistency of the logic.

In dependently typed languages, types can contain values and functions can produce types. To assign a type, for example, to a type-producing function, Agda needs a type of which is called Set. In fact, Agda has infinitely many type levels. The next level, Set₁, extends the collection of types by Set itself and types formed from it, while Set₂ is the next type level above Set₁, and so on.

A dependent function type is written as \((x : A) \rightarrow B\), which maps an element \(x\) of type \(A\) to an element of type \(B\), where the type \(B\) may depend on \(x\). Wrapping the argument in curly brackets, such as \(\{x : A\} \rightarrow B\), allows us to omit arguments when applying the function, and to rely on Agda to infer it from typing information. We may still apply the argument explicitly using the notation \(\{x = a\}\) or \(\{a\}\), for example, when Agda cannot deduce it automatically.

Agda has inductive types (introduced by `data`) and record types. To represent infinite structures we use Agda’s coinductive record types, equipped with size annotations [30]. The size annotations are used to show the productivity of corecursive programs [2], which we define using copattern matching [3].

**State-Dependent IO** In previous work [1], which was based on work of the second author [67], we gave a detailed introduction to interactive programs and objects, and to state-dependent 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 Hancock and Setzer [26–28, 68]. The theoretical basis for the IO monad was developed by Moggi [53]. It was pioneered by Peyton-Jones and Wadler [59, 72–75] as a paradigm for representing IO in functional programming, especially Haskell. 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-Synak trees [58], except that they also have the option to terminate and return a value. This allows for monadic composition of programs, that is, sequencing one program with another program, where the second 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.
For this paper, we introduce a generic IO interface for describing the commands a program can issue and the responses the world can return.

```agda
record IOInterface : Set where
  Command : Set
  Response : Command → Set
```

The interface is a record with two fields: Command and Response. Record fields can be applied postfix using the dot notation, for example, if \( p : \text{IOInterface} \), then \( p\_\text{Command} : \text{Set} \). To improve readability, throughout the paper we omit some bureaucratic Agda keywords from record definitions, such as field, coinductive, and open.

### 3 A Library for Declarative, State-Dependent GUI programming

This section will introduce our library for state-dependent GUI applications. In Section 3.1, we introduce the library from the programmer’s perspective, demonstrating how to build a simple state-dependent GUI application using the library. A noteworthy aspect of the example application is that it has infinitely many states, where each state differs in the GUI elements used. In Section 3.2, we introduce a representation of state-dependent objects from previous work that underlies our implementation of GUI event handlers. We then use this to define generic handlers in Section 3.3 whose interfaces are generated from the GUIs they depend on. The handler is part of a data type that collects all of the components of a GUI application together.

In our library, the structure and appearance of an application’s GUIs (the view) is declaratively specified separately from the handler objects that process events generated by users interacting with those GUIs (the controller). This separation of concerns is similar to current practice with model-view-controller frameworks [41] and graphical GUI-builder tools [70]. However, a distinguishing feature of our approach is that handler objects 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.

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. The dynamic interfaces of handler objects, based on the type of their GUIs, are illustrated in the following example.

### 3.1 Introductory Example: A GUI with Infinitely Many States

In this subsection, we present a simple 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. Clicking button \( b_i \) extends the GUI with \( i \) additional buttons. First, we define a function \( \text{nFrame} \) that constructs a GUI with \( n \) buttons. Creating an empty frame is trivial. A button is added to a frame by providing its label as a String.

```agda
nFrame : (n : ℕ) → Frame
nFrame 0 = emptyFrame
nFrame (suc n) = addButton (show n) (nFrame n)
```

Next we define a function \( \text{nGUI} \) that constructs a GUI application with \( n \) buttons.

```agda
infiniteBtns : ∀ i → (n : ℕ) → GUI i
infiniteBtns n .gui = nFrame n
infiniteBtns 0 .obj = ()
infiniteBtns (suc n) .obj .method (m , _) =
  returnGUI (infiniteBtns (n + finToℕ (m)))
```

The argument \( ∀ \{i\} \) introduces a hidden dependency on a size \( i \), which is also a hidden argument of \( \text{GUI} \). This argument is boilerplate needed so that Agda’s termination checker accepts the corecursive definition of \( \text{infiniteBtns} \). The GUI application is defined by defining the two fields of the constructed GUI value: the \( \text{GUI\.gui} \), and its handler \( \text{.obj} \). The GUI is constructed by \( \text{GUI} \). The handler object must handle all of the events that can be triggered from the GUI, in this case button clicks from \( n \) different buttons. For the case when \( n = 0 \), the handler will be empty, indicated by the base case of the handler method. In the non-empty case, the handler object’s method accepts an argument of the form \((m, s)\), where \( m \) is a finite number (bounded by the number of buttons in the GUI) that indicates which button was clicked, and \( s \) contains all of the strings obtained by text boxes in the GUI. Since there are no text boxes in the GUI, the second component will be empty and is not used in the handler (indicated by \( . \)). The result of the handler method is an interactive program that returns a new GUI application to replace the current one—in this case, a new infinite button GUI with \( m \) additional buttons.

While this example is admittedly contrived, the ability to define dynamically expanding GUIs is an expressiveness gain over standard GUI builders. GUI builders only support constructing finitely many GUIs for a particular application.

Finally, we compile our GUI application into an element of NativeO. This is a type that can be compiled using Agda’s foreign function interface (FFI) into native Haskell code, and can then be executed in Haskell. We use Agda’s do notation, which is similar to Haskell’s, and supports creating
an interactive program from a sequence of IO actions, where outputs of preceding actions can be used by later actions.

```
main : NativeIO Unit
main = do win <- createWindowFFI
       compile win (infiniteBtns 3)
```

### 3.2 State-dependent Objects

GUI events are handled by objects, which we have defined in our previous work [1], where they are described in detail. An object can receive commands, and, depending on the responses to those commands, it returns a result and changes its internal state. Since Agda is a purely functional language, we model state changes by returning an updated object together with the return value as a pair. In dependent type theory, we also have state-dependent objects where the methods available depend on the external state of the object, which forms part of its type. Depending on the response for the method call, the object switches to a new state. So, an interface for a state-dependent interface `Interface`s consists of a set of external states, a set of methods depending on the state, a set of responses depending on state and methods, and a next state function, which depends on state, methods, and responses.

```
record Interface{s} : Set₁ where
  State⁺ : Set
  Method⁺ : State⁺ → Set
  Result⁺ : (s : State⁺) → Method⁺ s → Set
  next⁺ : (s : State⁺) (m : Method⁺ s) → Result⁺ s m → State⁺
```

More precisely an object `obj` for an external state `s` is an element of the type of objects `(Object⁺ s)` for this interface. A method call is invoked by invoking the field .method `m of `obj`, where `m : Method⁺ s`. When `m` is called it runs an interactive program, which concludes by returning a result `r : Result⁺ s m` and an updated object. The updated object has a new state `s′ = next⁺ s m r` and is therefore an element of `Object⁺ s′`. Therefore `(Object⁺ s)` is a recursive definition. Since an object can be called infinitely many times, this recursive definition is coinductive.

### 3.3 A Data Type for GUls

In this section we briefly describe the representation of GUls. A GUI can be broken into two parts: the frame and widgets that make up the interface, and the data-dependent object that handles events generated by the interface. The data type is defined below, with fields .gui and .obj for accessing the respective parts.

```
record GUI : Set where
  gui : Frame
  obj : FrameObj gui
```

For the purposes of this paper, we treat the first part, `Frame`, as an abstract data type with the following operations for creating an empty frame, adding labeled buttons, adding text labels, and adding text boxes.

```
emptyFrame : Frame
addButton : String → Frame → Frame
addLabel : String → Frame → Frame
addTextbox : Frame → Frame
```

The handler object, `FrameObj`, is a state-dependent object whose state is the `.gui` interface it must handle events for. As described in Section 3.1, the handler method accepts pairs `(m, s)`, where `m` is a finite number indicating which button has been clicked, and `s` is a list of strings representing the text in all of the text boxes in the interface. In response to a method call, an interactive program will be executed. This may perform arbitrary IO actions, such as performing a database query or interacting with the console. In the future, we plan to add additional commands to support interactions with other external devices, such as medical equipment. The result of the interactive program is a new GUI, which replaces the current one.

### 3.4 Implementation Details

GUls are translated into a value of `NativeIO` by the function `compile`.

```
compile : SDLWindow → GUI → NativeIO Unit
```

The required argument `SDLWindow` can be obtained from a native interactive program `createWindowFFI`. The resulting `NativeIO` value can be compiled to and invoked from Haskell, which will create and execute the GUI that we have defined.

On the Haskell side, we use the libraries SDL [66] and Rasterific [62] as a backend. Rasterific is one of the outcomes of the STEP's project [39], which is focused on the concise formulation of declarative GUls based on FRP and objects. At its core is an elegant DSL for vector graphics [18] that was also ported to Haskell [29]. We used this Haskell port (Rasterific), the bindings to the SDL library, and some additional glue code that we contributed ourselves. Our Agda GUI library binds to this Haskell glue code and presents vector-based GUI elements that are rasterized via Rasterific and then presented on the screen via an SDL window.

This approach solves a previous issue when using the library wxHaskell [78], which exhibits a mismatch for functional/declarative programming. In wxHaskell, handler functions are IO programs which only have side effects and no return value, and are implemented via pointers in Haskell. With some effort, we were able to implement state-dependent objects and declarative GUI data types using wxHaskell, but the complexity was substantial. This made verification challenging. Switching to Rasterific solved this problem.
4 Reasoning About GUI Applications

We can 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, the handler IO program is executed. The handler may trigger IO commands that yield IO responses and eventually determines a new state to transition to. Thus, we can reason about the transition graph for a GUI application by reasoning about the different responses each command could yield.

However, a complication is that the IO programs used to implement event handlers are coinductive, meaning they may have infinitely many 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.

In the rest of this section, we introduce infrastructure for reasoning about coinductive programs (Section 4.1), and for specifying properties over the state transition graphs of GUI applications (Section 4.2). In Section 5.2, we use the infrastructure developed in this section to prove properties about the case study described in Section 5.

4.1 Reasoning about Coinductive Programs

To cope with the fact that IO programs and thus GUI applications are coinductive, we do not reason about GUI states directly. Instead we 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, so 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 this 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 argument pr of started captures the remaining IO program still to be executed.

```agda
data MethodStarted (g : GUI) : Set where
  notStarted : MethodStarted g
  started : (m : GUIMethod g)
  (pr : IO consoleI ∞ GUI) → MethodStarted g
```

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

```agda
data State : Set where
  state : (g : GUI) → MethodStarted g → State
```

Using this model, we can simulate the execution of a GUI application. To do this, we define a simulator for state-dependent IO programs. The simulator matches states of the GUI model and either triggers events (e.g. by pressing buttons), provides responses to IO commands, or transitions to subsequent states of the model. The following function defines the types of available actions at each state in the model.

```agda
Cmd : State → Set
Cmd (state g notStarted) = GUIMethod g
Cmd (state g (started m (exec' c f))) = IOResponse c
Cmd (state g (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).4 In this case the GUI is waiting on a response to the IO command c, which the event simulator must provide. The second sub-case 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:

```agda
guiNext : (g : State) → Cmd g → State
```

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.

4.2 Properties Over GUI Application States

Many of the properties we want to express about GUI applications contain data dependencies. That is, we want to state that starting from a given state, if the user provides certain inputs (e.g. text in a text box) and those inputs satisfy certain conditions, then a certain result state is either reached or not. To support writing such data-aware properties, we define the following data type Cmds that formalizes a sequence of user inputs and a function guiNexts that computes the state obtained after a sequence of inputs.

```agda
data Cmds : State → Set where
  nilCmd : (g : State) → Cmds g
  _⇒_ : (g : State) (l : Cmds g) → Cmd (guiNexts g l) → Cmds g
```

4Previously we used do instead of exec, but do has now become a keyword in Agda.
Declarative GUIs: Simple, Consistent, and Verified

Note that the `Cmds` essentially defines a sequence of commands parameterized by a `State` type that statically ensures that each input corresponds to the preceding state. This is an example of an inductive-recursive definition [21, 22].

We can now define a relation between two states `s` and `s'` of a GUI. The following data type formulates that `s'` is reachable from `s` by a sequence of GUI commands.

```plaintext
data _-gui->_ (s : State) : State → Set where
  refl-gui-> : s -gui-> s
  step : {s : State}(c : Cmd s)
    → guiNext s c -gui-> s'
    → s -gui-> s'
```

The first constructor defines that the relation is reflexive, while the second links two GUI states via a command.

Finally, we define a property that, starting from one GUI state `start`, we will eventually reach another state `final`, for all possible user interactions. It holds if `start` and `final` are the same state (constructor `hasReached`) or if we will eventually reach `final` from the next state reached after any input a user provides (constructor `next`).

```plaintext
data _-eventually->_ : (start final : State) → Set where
  hasReached : {s : State}(c : Cmd s)
    → (guiNext start m) -eventually-> s
  next : (start final : State)
    → (fornext : (m : Cmd start)
      → (guiNext start m) -eventually-> final)
    → start -eventually-> final
```

## 5 Case Study: Healthcare Process Models

In this section, we present a healthcare case study that we have developed in cooperation with the Medical University of Vienna. It investigates the prescription of anticoagulants ("blood thinners") to patients admitted to the accident and emergency department of Vienna General Hospital (AKH).

In Section 5.1 we provide a high-level description of our case study by presenting an overview of the domain and its significance, and by presenting a business process model that describes the healthcare process we want to implement.

In Section 5.2 we define Agda data types and functions for specifying business processes and translating them into GUIs. This allows us to build GUI applications directly from the business process descriptions already used in the domain. In Section 5.3 we describe the implementation of the healthcare process for prescribing anticoagulants, and finally, in Section 5.4 we demonstrate how to prove medically relevant properties about the resulting GUI application.

### 5.1 A Process for Prescribing Oral Anticoagulants

Most patients prescribed anticoagulants are treated for atrial fibrillation (AF), which is an abnormal heart rhythm that affects 3% of the population in the EU/US. For these patients, four new oral anticoagulants (called NOACs) have been shown to be equally effective at stroke prevention compared to the older medication warfarin. The NOACs are preferable in many cases because they are faster acting and have shorter half-lives than warfarin.

However, NOACs are also frequently associated with medication errors. Studies have shown an 11–16% error rate in NOAC prescriptions [32, 79], and these errors can lead to serious or fatal events [63, 64]. The problem is that prescribing the correct NOAC is complex and depends on several clinical factors, such as the grade of renal impairment, age, and risks of falls/accidents.

In this case study we consider the process of prescribing anticoagulants to treat AF. Figure 1 specifies the business process using the language BPMN. In BPMN, steps of the process are indicated by rounded rectangles and data artefacts are depicted as notes with three black bars. For example, the step `Patient registration` involves recording the patient’s age and weight, the step `Patient history` involves assessing and recording the patient’s stroke risk (CHA2DS2-VASC-Score) and fall risk, and most importantly, the step `Receive blood test results` involves recording the result of a GFR (glomerular filtration rate) blood test. GFR is an estimation of kidney function, and is also called a `renal value`.

While the upper half of the diagram in Figure 1 is related to the acquisition of data and an ensuing diagnosis of AF, the lower half illustrates the critical steps of the doctor’s decision of which anticoagulant to prescribe. In BPMN, the X-symbol represents exclusive decision path branching and merging. At the `MD Choice` step, the doctor may choose to either prescribe warfarin or a NOAC. If the doctor chooses to prescribe a NOAC, they must choose one of the four specific medications (Med A–D) and a suitable dosage (high or low).

As mentioned, the safety constraints for this decision-making process are complex. A subset of the constraints is summarized in Table 1. While the full clinical specification is more complex, we focus on the constraints presented in the table for brevity. The first non-header line of the table

### Table 1. Indications and contraindications of NOACs and warfarin according to the European Society of Cardiology.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No anticoagulation required</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Contraindicated if GFR &lt; 30</td>
<td>≤ 30</td>
<td>≤ 30</td>
<td>≤ 30</td>
<td>≤ 30</td>
<td>N/A</td>
</tr>
<tr>
<td>Use in hospital if GFR ≥ 30</td>
<td>≥ 30</td>
<td>≥ 25</td>
<td>≥ 30</td>
<td>≥ 30</td>
<td>N/A</td>
</tr>
<tr>
<td>Dose reduction when GFR 30–49</td>
<td>≤ 60kg</td>
<td>15–49</td>
<td>15–49</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

* non-mandatory constraint | † note non-GFR measurement

3https://www.omg.org/spec/BPMN/2.0.2/
captures one such safety constraint: patients with severe kidney damage (GFR below 30) cannot be given dabigatran and patients with kidney failure (GFR below 15) cannot be given any NOAC and so must be treated with warfarin. Consequently, an implementation of the process in Figure 1 must not reach the step \textit{Prescribe NOAC} or any subsequent NOAC step if the GFR is below 15. A stricter version of this property (incorporating hospital-specific constraints, see below) is proved as \textit{theoremWarfarin} in Section 5.3.

In addition to safety constraints from the medication manufacturer, additional constraints can be imposed by the hospital to further reduce risk. In AKH, the hospital of our case study, the second row captures additional AKH-specific constraints. By applying these, we see that if the GFR is between 25 and 30, apixaban is the only NOAC that may be prescribed.

The third row of the table captures constraints related to the dosage of a NOAC. For most NOACs, a smaller dosage must be prescribed when the GFR is in a particular range. However, apixaban dosage is instead constrained by the weight of the patient. An additional constraint not reflected in the table is that patients over the age of 75 should always be given a restricted dosage.

Finally, if a patient may need immediate surgery, or in the case of an elderly patient who has an increased risk of falling and sustaining an injury, one should prescribe only NOACs which have an antidote that can reverse the thinning effect of the medication. This is not the case for Med A (apixaban).

Therefore, in case of fall risk, Med A needs to be substituted by a different NOAC or by warfarin.

5.2 Formalization of Business Processes

The healthcare process of our case study is specified as a business process, in BPMN, plus additional safety constraints. We could implement and verify a GUI application for our case study by directly constructing GUIs and handlers as described in Section 3. However, we expect many healthcare processes (and other safety-critical business processes) can be expressed in a similar way as our case study, and so building a GUI application directly would miss out on potential for reuse among implementations of such processes.

Instead, we define an inductive data type for directly representing business processes as Agda values, and then translate these specifications into GUIs that implement each step. This illustrates how our library is suitable not only as a way to build verified GUIs from scratch, but also as infrastructure for defining domain-specific libraries or languages for building GUI applications from higher-level specifications. Crucially, the GUI applications generated in this way are still directly verifiable executable programs.

A simple data type for specifying business processes is given below. The data type includes only the cases needed to realize our case study, but could easily be extended to support more of the BPMN language.

data BusinessModel : Set where
terminate : String → BusinessModel
where the possible next steps are represented by a list of
form. In our example, this can be used to ensure that numeric values are
represents a basic step in the process, which consists of a
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.

A value of type BusinessModel can be translated into a
GUI using the following function.

A simple or xor step will be translated into a GUI with string-
labels of business models. The input constructor corresponds to a step requiring n inputs from the user. The
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.

A value of type BusinessModel can be translated into a
GUI using the following function.

A simple or xor step will be translated into a GUI with string-
labels of business models. The input constructor corresponds to a step requiring n inputs from the user. The
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.

A value of type BusinessModel can be translated into a
GUI using the following function.

A simple or xor step will be translated into a GUI with string-
labels of business models. The input constructor corresponds to a step requiring n inputs from the user. The
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.

A value of type BusinessModel can be translated into a
GUI using the following function.

A simple or xor step will be translated into a GUI with string-
labels of business models. The input constructor corresponds to a step requiring n inputs from the user. The
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.

A value of type BusinessModel can be translated into a
GUI using the following function.

A simple or xor step will be translated into a GUI with string-
labels of business models. The input constructor corresponds to a step requiring n inputs from the user. The
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.

A value of type BusinessModel can be translated into a
GUI using the following function.

A simple or xor step will be translated into a GUI with string-
labels of business models. The input constructor corresponds to a step requiring n inputs from the user. The
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.

A value of type BusinessModel can be translated into a
GUI using the following function.

A simple or xor step will be translated into a GUI with string-
labels of business models. The input constructor corresponds to a step requiring n inputs from the user. The
function argument determines the next step based on the
inputs provided by the user. Finally, the simple constructor represents a basic step in the process, which consists of a
message to display and a business model that is the next step
in the process.
steps. The rest of the business process is constructed similarly to the steps illustrated above. The compiled program is now obtained by applying businessModel2GUI to the initial business process patientRegistration. Figure 2 shows a screenshot of the resulting GUI application.

5.4 Verifying GUI Applications

Now we want to verify medically relevant safety properties about our GUI application. In particular, we want to verify that for various kinds of inputs, our application will recommend the correct coagulant and dosage.

To express and prove such properties, we need a way to automate and abstract over the initial input steps of the process. The following function takes several string values corresponding to each of the inputs requested in the initial steps of the process (age, weight, fall risk, stroke risk score, and blood test result) and returns the state of the GUI application after submitting each of these inputs.

```
stateAfterBloodTest : (strAge strWght strFallR strScore strBlood : String) → State
```

```
stateAfterBloodTest strAge strWght strFallR strScore strBlood =
  guiNexts
  patientRegistrationState
  (nilCmd
    >> textboxInput2 strAge strWght
    >> textboxInput2 strFallR strScore
    >> btnClick
    >> textboxInput strBlood)
```

The body of this function returns the next state of the GUI after executing a sequence of user actions on the initial patient registration state. The sequence of actions follows the sequence of steps at the top of Figure 1: it inputs the age and weight strings in the GUI corresponding to the first step, inputs the fall risk and stroke risk in the next step, clicks to confirm that blood has been drawn, and finally inputs the blood test result in the fourth step.

Our first theorem states that, given a complete set of inputs, if the blood test yields a renal value of less than 25, then we will eventually reach the state where warfarin is prescribed. The theorem is expressed in terms of the -eventually-→ relation defined in Section 4.2.

```
theoremWarfarin : (str2RenalCat strBlood : (str2AgeCat strAge)
  (str2WghtCat strWght)
  The proof relies on an auxiliary theorem where the string-valued inputs have been converted into the corresponding categorized values to support pattern matching.

```

```
theoremWarfarinAux : (f : FallRisk)(r : RenalCat)
  (a : AgeCat)(w : WghtCat)
  → r ≡ <25
  → diagnosisState f r a w
  →-eventually-→ warfarinState
```

The second theorem states that if the blood test yields a renal value of between 25 and 30, and if we reach the state where dabigatran (D) is prescribed, then a low dosage is not prescribed.

```
theoremNoLowDosis<30 : (strAge strWght strFallR strScore strBlood : String)
  → str2RenalCat strBlood ≡ ≥25<30
  → (r : RenalCat≥30)
  → (a : AgeCat)
  → stateAfterBloodTest strAge strWght strFallR strScore strBlood
    →-gui-→ NOACSelectionDState r’ a’
    → {s : State}
    → NOACSelectionDState r’ a’-gui-→ s
    → ¬ (s ≡ lowdoseSelectionState)
```

The proof of this theorem is carried out by a case distinction on possible paths. We do this manually, but current work on Agda will make it possible to perform such finite case distinctions automatically in the future.

6 Related Work

In our previous article [1], we introduced an Agda library for developing state-dependent, interactive, object-based programs. We demonstrated its use for the development of basic GUIs. In this paper, we have extended this work by adding a declarative specification of GUIs as a data type that captures all aspects of a GUI application (components, handlers, etc.). Also new to this work is the verification of GUI applications, as demonstrated in Sections 4–5. This includes verifying properties such as data-dependent reachability and...
unreachability constraints for GUIs generated from declarative business process specifications.

**Functional Reactive Programming (FRP)** FRP is another approach to writing interactive programs in functional programming languages [76]. In connection with dependent types, FRP has been studied from the foundational perspective [65] and for verified programming. Jeffrey [33] contributed an embedding of linear temporal logic (LTL) into Martin-Löf type theory. This embedding supports program verification in which specifications are expressed as LTL formulas. An extension of FRP with side effects was introduced in Agda [9].

Krishnaswami and Benton [42] introduced a semantic model of GUIs based on the Cartesian closed category of ultrametric spaces using FRP. However, their work doesn’t address the verification of GUIs.

The STEP’s project [39] led by Alan Kay presented an approach to declarative GUIs based on FRP and Morphic (Smalltalk GUIs). Its vector graphics library is particularly concise and elegant, which is why it can be considered as suitable for use in vector-based GUI backends. We used it in our work for our Haskell GUI backend. A major difference between the STEP’s project and regular FRP is that it is based on dynamically-typed languages without static type guarantees.

**Formlets** Formlets [17] are abstract user interfaces that define static, HTML-based forms that combine several inputs into one output. However, they do not support sequencing multiple intermediate requests. Formlets could conceivably be used to define individual states of a larger state-dependent GUI application within our library.

**Formalizations in Isabelle** There has been work on formalizing end-user-facing applications in Isabelle. There are formalizations of distributed applications [12] and conference management systems [34]. In both cases, the verification provides confidentiality guarantees about the information flow. However, only the core of the server is verified. The user interface and server API are external to the system and therefore not part of the Isabelle formalization and not checked for consistency. Since this aspect is of particular interest for the healthcare domain, we plan to implement similar confidentiality guarantees and other security properties. We can define stronger guarantees since we can define them over the entire running system (including the GUIs), and not just the core information flow.

**Semantic Reasoning in the Healthcare Domain** Abidi et al. [4] present a decision support system for the prescription of NOACs. They use semantic reasoning (OWL) to address the challenge of determining the appropriate medication. However, their work does not involve formal verification. The fact that this system has been developed illustrates the difficulty of the medication process and thus the need for automated systems.

**Process Models in the Healthcare Domain** Healthcare processes have been specified using Declare [52], which is a declarative modeling language [71] for business processes based on human-readable diagrams. The Declare constraints are often embedded in linear temporal logic (LTL) or finite automata theory. A useful extension for healthcare processes is to model patient data in the underlying business models. This was studied from the perspective of databases [16] and implemented as an extension for Declare [54]. A first-order logic approach is presented by [44], which would be sufficiently expressive, however, it is undecidable and which is problematic (e.g., most model checkers don’t support it).

In the current paper, in contrast to [16, 52, 54] and other approaches to process models, we apply formal verification using a theorem prover (Agda) and provide machine-checked proofs as safety guarantees. We have found only two papers using formal specifications: Debois [20] proves in Isabelle a general result that a certain labeling of events in a business logic guarantees orthogonality of events. Montali et al. [55] developed a language DecSerFlow to model properties of business processes via choreographies mapped to LTL and then to abductive logic programming. These properties can be reasoned about with automated theorem provers. However, that limits their use to finite systems. Furthermore, it doesn’t deal with the execution of process models/GUIs.

**Deductive Verification of Imperative Programs** Another avenue of research is to use a co-inductive approach to directly verify imperative programs. In [57], there is a formalization of four coinductive operational semantics for the while language (a small subset of an imperative language). The semantics and their equivalence proofs are formalized in Coq. However, in order to be applicable to IO and GUI programming, the work needs to be extended with IO and higher imperative constructs (e.g., subroutines, objects). Beckert and Bruns [13] present Dynamic Trace Logic (DTL), which combines dynamic logic with temporal logic. This forms a basis of proving functional and information-flow properties in concurrent programs. It allows to specify and prove properties of interactive programs using modal operators. We also plan to develop our specifications into a full modal logic.

**Idris and Algebraic Effects.** Bauer and Prentrat [10, 11] have introduced the notion of algebraic effects. Brady [14] adapted this approach in order to represent interactive programs in Idris [15, 31]. In [1], Sect. 11 we have conducted a detailed comparison of the IO monad used in Agda and the use of algebraic effects in Idris. Additionally, we have detailed how to translate between the two approaches. Idris has been primarily designed to be a language for dependently typed programming while being able to be used as
an interactive theorem prover. It is used for practical dependently typed programming. Regarding GUIs, we could only detect a forum post [60] which shows that GUI programming should be possible using the FFI interface of Idris but has yet to be performed.

7 Future Work

We plan to evaluate the GUI application developed in our case study by running it on live, anonymized patient data. This is awaiting the approval of the AKH hospital’s ethical board.

A constraint imposed by our framework is that a GUI has one event handler object that handles all events generated by the GUI. We are working on a version that removes this constraint by allowing the composition of more loosely coupled GUI objects, which is desirable from several perspectives, including modularity, security, and performance.

We also plan to investigate combining the formlets approach with our notion of GUIs. This would allow us to create generic GUIs that compute results from user inputs, which can be plugged together. Specifically, it would enable better composition and reuse of high-level GUI components.

Currently, our library generates application programs. We plan to extend this to also support web-based GUIs and mobile apps. Event handlers in our framework are written in Agda and compiled to Haskell. We plan to use the Threepenny-gui framework [7], which is a Haskell API for developing GUIs that run in a web browser via JavaScript. The JavaScript layer for Threepenny-gui is lightweight while most of the heavy lifting happens in Haskell. This makes it ideal as an alternate back-end for our library, allowing us to generate both application and web-based GUIs from the same Agda code.

One important application we have in mind is to use our approach to create a verified mobile app to be used by medical professionals. Trials of this app with real patients could be made in collaboration with our partners in the Medical University of Vienna.

Appel et al. [8] proposed that a key building principle for deep specifications is vertical compositionality, which means that higher specification levels are related provably correct to a lower specification levels. This allows proving properties at the higher specification level that hold at the lower specification levels. For us, it would be useful to be able to translate properties on the level of business processes to properties on the level of GUI applications. Since verification at the level of business processes is easier, this would facilitate easier verification of process-based GUI applications.

Model checking provides a convenient way to verify properties about finite state machines. We could build on the work by Kanso [35–37] on integrating model checkers into Agda. Such a verification could take place on the business process logic and then translate, via vertical compositionality, into verification of the GUI. With integrated automated theorem proving, interactive theorem proving can focus on generic systems and universal statements, such as properties of operators on GUIs, while proofs of properties of concrete systems can be carried out by model checking. This would enable automatically proving the theorems in this paper.

Finally, we plan to support new kinds of properties of GUI applications by extending the language of properties with LTL or related modal logics. To be compatible with Agda, this requires adapting one of several constructive modal logics [19, 40, 46].

8 Conclusion

GUI applications are ubiquitous in real-world systems but are inherently difficult to verify through software testing. This is problematic because many GUI applications are safety-critical, such as in the healthcare domain. In this paper, we presented a library for implementing state-dependent GUI applications in Agda to address this problem through formal verification. In our library, a GUI application consists of a declarative GUI specification and an object that handles its events. The type of the event handler depends on the GUI specification, ensuring that the two are consistent. We demonstrated how to create an application with infinitely many steps in a straightforward way. In order to generate GUIs from a higher-level specification, we formalized business processes and translated them into GUIs. As a case study, we formalized an error-prone example from the medical domain as a business process. We generated a GUI from it and proved correctness properties. The properties expressed that if one starts at the beginning, gives the inputs as required by the GUI, then one reaches a state such that if the inserted value fulfills certain conditions, then a certain state will eventually be reached, and another state will never be reached. This shows that it is possible to formalize GUI applications at this level of complexity and formally prove their correctness.

Acknowledgments

The authors would like to thank Prof. Michael Gottsauner-Wolf, MD, for providing the details of the NOAC medication process. The second author was supported by the projects CORCON (Correctness by Construction, FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638), COMPUTAL (Computational 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 first and second author were supported by CA COST Action CA15123 European research network on types for programming and verification (EUTYPES). The third author was supported by AFRL Contract FA8750-16-C-0044 under the DARPA BRASS program.

Stephan Adelsberger, Anton Setzer, and Eric Walkingshaw
References


[34] Sudeep Kanav, Peter Lammich, and Andrei Popescu. 2014. A con-


[36] Sudeep Kanav, Peter Lammich, and Andrei Popescu. 2014. A con-


[38] Ambrus Kaposi (Ed.). 2017. 23rd International Conference on Types

[39] Antonio Zan, Ioan Panaita, Kim Rose, Dan Ingalls, Daniel Amelang, Ted


[43] S Lawes and M Grissinger. 2017. Medication errors attributed to health

[44] Fabrizio Maria Maggi, Marlon Dumas, Luciano García-Bañuelos, and


[49] Antonio Zan, Ioan Panaita, Kim Rose, Dan Ingalls, Daniel Amelang, Ted


[52] Steven Mertens, Frederik Gailly, and Geert Poels. 2015. Enhancing
decision processes with a decision logic in: International Conference on


[56] Risa B Myers, Stephen L Jones, and Dean F Sittig. 2011. Review of

[57] Keiko Nakata and Tarmo Uustalu. 2009. Trace-based coinductive oper-

groups.google.com/forum/#!topic/idris-lang/R_7oixHofUo

[59] William B Runciman, Elizabeth E Roughead, Susan J Semple, and

[60] David C Radley, Melanie R Wasserman, Lauren EW Olsho, Sarah J

[61] Keiko Nakata and Tarmo Uustalu. 2009. Trace-based coinductive oper-

[62] David C Badley, Melanie R Wasserman, Lauren EW Olsho, Sarah J

[63] Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional

[64] Fabrizio Maria Maggi, Marlon Dumas, Luciano Garcia-Bañuelos, and


[68] Stephen Adelsberger, Anton Setzer, and Eric Walkingshaw