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 ubiquitous in real-world software and a notorious source of bugs that are difficult to catch through software testing. Model checking has been used to prove the absence of certain kinds of bugs, but model checking works on an abstract model of the GUI application, which might be inconsistent with its implementation. We present a library for developing directly verified, state-dependent GUI applications in the dependently typed programming language Agda. In the library, the type of a GUI's controller depends on a specification of the GUI itself, statically enforcing consistency between them. Arbitrary properties can be defined and proved in terms of user interactions and state transitions. Our library connects to a custom-built Haskell back-end for declarative vector-based GUI elements. Compared to an earlier version of our library built on an existing imperative GUI framework, the more declarative back-end supports simpler definitions and proofs.

As a practical application of our library to 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.

CCS Concepts • **Software and its engineering → Formal software verification**; *Software verification*; *Functional languages*; • **Applied computing → Business process modeling**; • **Theory of computation → Type theory**; • **Computing methodologies →** *Model verification and validation*;

Keywords Agda, interactive theorem proving, dependently typed programming, graphical user interfaces, GUI verification, state-dependent GUIs, reachability, dependable software, data-aware business processes, verification of business processes

ACM Reference Format:
Stephan Adelsberger, Anton Setzer, and Eric Walkingshaw. 2018. Declarative GUIs: Simple, Consistent, and Verified. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3236950.3236962

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 [72].

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),[1] 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 [71]. Additionally, other researchers have noted the difficulty in maintaining consistency between GUI specifications, the underlying business logic [72], and the corresponding test suites [28].

[1] https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL

Besides typical maintenance challenges associated with mixing generated and handwritten code [79], GUI applications pose special challenges to ensuring software quality. The most significant is that traditional software testing of GUIs is notoriously difficult [50, 53], and that traditional measures of test quality, such as code coverage, are not useful in the context of GUIs [52]. The core challenge to testing GUIs is that automated tests must simulate user interactions, but 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 [51]. 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 [8]. 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 [49]. 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 [35, 81]. 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 [63]. However, the HIT system itself can introduce a new class of errors related to software bugs. Magrabi et al. [47] 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 [47]. There is evidence that HIT errors are widespread and that many of these errors are specifically related to GUIs. The Pennsylvania Patient Safety Authority [45] reported 889 medication errors attributed to HIT systems within six months. Graber et al. [27] reported 248 malpractice claims attributed to HIT systems. Finally, a review of FDA data in 2011 [58] 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.[2] 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

[2]We use a coinductive rather than inductive data type because we allow potentially infinite sequences of interactions.

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 constructing values of those types. But the utility of dependent types runs deeper in the context of healthcare processes. Healthcare processes are complex and need to consider patient data generated at one step in future steps (e.g. blood test results). Thus, correctness constraints must also take into account data generated at intermediate steps. Dependent types are well-suited to describing such data-dependent specifications and proof conditions, and are more expressive than other verification methods that have been applied to healthcare processes. For example, Montali et al. [56] allow only propositional linear temporal logic (LTL) formulas. Dependent types support defining and verifying conditions over numeric values, such as age or renal value. This corresponds to LTL with natural number values, which is equivalent to first-order logic and therefore undecidable [26] Undecidability is not a problem in our setting since proof objects are provided by the programmer.

1.5 Contributions

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 [6]. For readability, we have hidden some details and show only the crucial parts of the code which is available online at [6].

2 Background

Agda and Sized Types Agda [8] 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 types, which is called Set. In fact, Agda has infinitely many type levels. The next level, Set_1, extends the collection of types by Set itself and types formed from it, while Set_2 is the next type level above Set_1, 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 [33]. 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 [69], 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 [29–31, 70]. The theoretical basis for the IO monad was developed by Moggi [55]. It was pioneered by Peyton-Jones and Wadler [61, 74–77] 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-Synek trees [60], 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.

```
record IOInterface : Set₁ 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 : IOInterface, then p .Command : 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 introduces our library. Section 3.1 describes a representation of state-dependent objects from previous work, which will be used in our library to handle GUI events. In Section 3.2, we introduce the library from a programmer's perspective, developing a simple application that illustrates some unique capabilities of our library. In Section 3.3, we introduce the type of GUIs which consists of a frame together with an object handling events of that frame. In Section 3.4, we describe the translation of applications defined in our library into executable programs via a custom-built backend.

Our library separates an application's *view*—the appearance of its GUIs—from its *controller*—the handlers that process events generated by user interactions, which is similar to current practice with model-view-controller frameworks [43] and graphical GUI-builder tools [72]. However, our approach is both safer and more flexible. Handlers are dependently typed with respect to the GUIs they interface with, which means that GUI specifications can be dynamically modified (see Section 3.2) or programmatically generated (see Section 5) without sacrificing the static guarantee of consistency with its handlers. Such dynamically changing or programmatically generated 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.

3.1 State-dependent Objects

GUI events are handled by objects, which we have defined in our previous work [1], where they are described in detail. As in other object-oriented frameworks, an object receives methods and in response returns a result and updates its internal state. Since Agda is purely functional, state updates are simulated by pairing the method's return value with the update object. More uniquely and requiring dependent types, a *state-dependent object* is an object whose available methods depend on its (externally visible) state. Therefore, an object's state is part of its type, and the interface of an object may change after invoking a method. The following type Interface^s defines the interface of a state-dependent object (^{s} indicates state dependency). It consists of the type of the externally visible state, a set of methods that depends on the value of that state, the corresponding results of those methods, and a function that yields the next state based on the current state, the invoked method, and its result.

```
record Interfaceˢ : Set₁ where
  Stateˢ     : Set
  Methodˢ : Stateˢ → Set
  Resultˢ   : (s : Stateˢ) → Methodˢ s → Set
  nextˢ      : (s : Stateˢ) (m : Methodˢ s) → Resultˢ s m
               → Stateˢ
```

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

3.2 Example: A GUI with Infinitely Many States

We will now 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 the ith button extends the GUI with i additional buttons. First, we define a function nFrame that constructs a frame with n buttons. Creating an empty frame is trivial. A button is added to a frame by providing its label as a String.

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

Next we define a function infiniteBtns that constructs a GUI application with n buttons, which will be an element of the data type GUI. An element of GUI is a record with two fields: the GUI .gui, and its handler .obj, which is a state dependent object. The GUI application is defined by determining the values for these fields:

```
infiniteBtns : ∀{i} → (n : ℕ) → GUI {i}
infiniteBtns n .gui = nFrame n
infiniteBtns 0 .obj .method ()
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 GUI. This argument is boilerplate needed so that Agda's termination checker accepts the corecursive definition of infiniteBtns. 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. In case when $n = 0$, there are no events to be handled, indicated by the notation (). Otherwise, 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, the program immediately terminates returning 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 NativeIO. 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)
```

3.3 A Data Type for GUIs

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 GUI of GUIs 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 3.2, 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

GUIs 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 [68] and Rasterific [64] as a backend. Rasterific is one of the outcomes of the STEPs project [9], which is focused on the concise formulation of declarative GUIs based on FRP and objects. At its core is an elegant DSL for vector graphics [21] that was also ported to Haskell [32]. 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 [80], 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 view a GUI application as a state transition graph, where each state of the application consists of the current frame and its handler. When an event is triggered, an IO program is executed that should eventually yield the next state in the graph. However, reasoning about this graph is difficult for two reasons. First, handlers may interact with the user via IO commands, so we must reason about the potential responses to each command. Second, both the handlers' IO program and the GUI application itself are coinductive, meaning that they may never terminate. 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, we use the infrastructure developed in this section to prove properties about our case study.

4.1 Reasoning about Coinductive Programs

Because of the difficulties described above, we do not reason about a GUI's state transition graph directly. Since we need to be able to reason about user interactions, we unroll each handler into all of the (potentially infinite) sequences of commands produced by the associated IO program. Each command corresponds to a new intermediate state in the transition graph. The resulting graph is potentially infinite, both because of the handlers and because the GUI application itself may have infinitely many states (see Section 3.2). Therefore, we do not reason about the graph directly, but instead reason about *finite simulations* of this model.

First, we introduce a data type to distinguish the two kinds of states in our model: either a handler method has been invoked and we're in an intermediate handler state (started), or we're at a resting state, waiting for an event (notStarted). The argument *pr* to started captures the remaining IO program to be executed for the handler.

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

Next, a state in the model can be represented by the parent GUI state and the handler method invocation state.

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

With this model, we can simulate the execution of a GUI application. The simulator matches states of the 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 type of actions available to the simulator at each state in the model.

```
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))) = ⊤
```

From a resting (notStarted) state, the event simulator can trigger an event drawn from the methods supported by the GUI interface. From within a handler (started), there are two possibilities. If the handler has not finished, the IO program has the form (exec' c f), where c is the next command to execute and f is a continuation to be applied to the response.[3] In this case, the application is waiting for a response to the IO command c, which the event simulator must provide. Finally, if the handler has returned, then the simulator must take the trivial action (⊤) to advance to the next notStarted state.

The simulator's transition function has the following type, which states that given a model state and an action of the appropriate type, we can transition to the next model state.

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

The implementation of this function optimizes away states with trivial transitions to simplify proofs over the model. For example, when the next state is a returned handler state, it moves directly to the following notStarted state. Similarly, it collapses sequences of trivial IO actions, such as print commands, into a single transition.

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.

```
data Cmds : State → Set where
  nilCmd : {g : State} → Cmds g
  _»>_ : {g : State} (l : Cmds g)→ Cmd (guiNexts g l)
         → Cmds g

guiNexts : (g : State) → Cmds g → State
```

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 [24, 25].

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.

```
data _-gui->_ (s : State) : State → Set where
  refl-gui-> : {s' : State} → s ≡ s' → 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.

[3] Previously we used do instead of exec, but do is now a keyword in Agda.

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).

```
data _-eventually->_ :
     (start final : State) → Set where
  hasReached : {s : State} → s -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 [35, 81], and these errors can lead to serious or fatal events [65, 66]. 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.

	Med D	Med A	Med E	Med C	Med W
Contraindicated if GFR	< 30	< 15	< 15	< 15	N/A
Contraindicated in AKH if GFR	< 30	< 25	< 30	< 30	N/A
Contraindicated if fall risk	no	yes	yes	yes	no
Low Dosage when GFR	30–49*	$\leq 60 kgs$†	15–49	15–49	N/A

* non-mandatory constraint † weight of patient

Table 1. Contraindications of NOACs and warfarin according to the European Society of Cardiology. All numerical units are GFR (glomerular filtration rate) unless otherwise noted. Medications are: D, dabigatran; A, apixaban; E, edoxaban; C, rivaroxaban; and W, warfarin.

In this case study we consider the process of prescribing anticoagulants to treat AF. Figure 1 specifies the business process using the language BPMN.[4] 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 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. As a result, patients must be treated with warfarin. Consequently, an implementation of the process in Figure 1 must not reach the step *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 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

[4]https://www.omg.org/spec/BPMN/2.0.2/

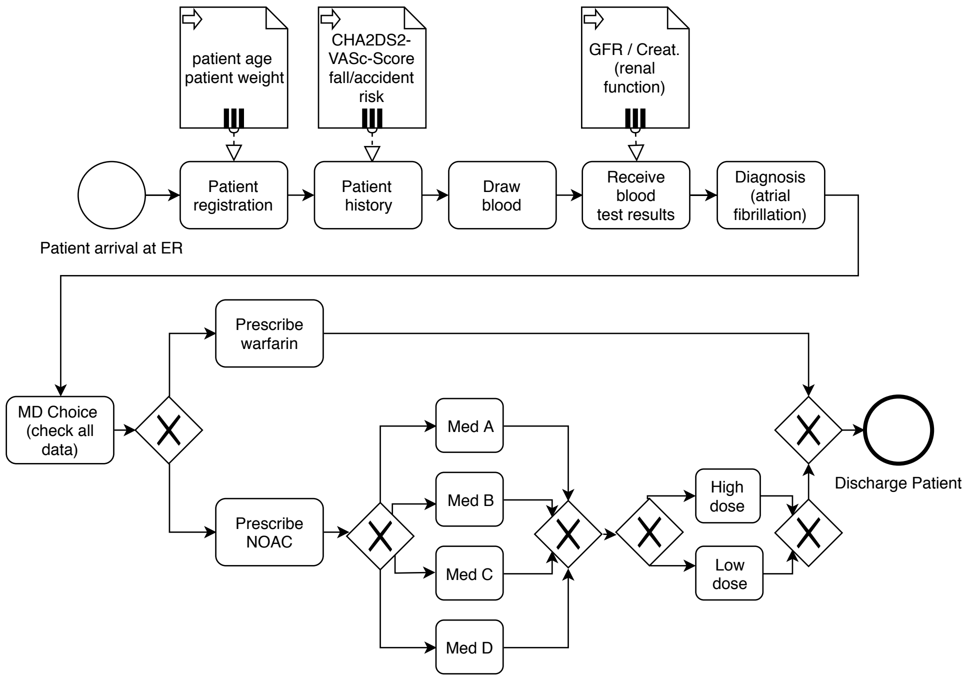

Figure 1. Specification of a healthcare process for prescribing oral anticoagulants to treat atrial fibrillation.

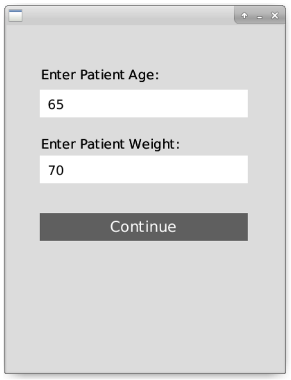

Figure 2. Screenshot from the corresponding GUI application.

study, the second row captures additional AKH-specific constraints. By applying these constraints, we see that if the GFR is between 25 and 30, apixaban is the only NOAC that may be prescribed.

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 with a different NOAC or by warfarin.

The last 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 in the case of dabigatran, patients over the age of 75 should always be given a restricted dosage.

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
  xor       : List (String × BusinessModel)
              → BusinessModel
  input     : {n : ℕ} → Tuple String n
              → (Tuple String n → BusinessModel)
              → BusinessModel
  simple    : String → BusinessModel → BusinessModel
```

A value of type BusinessModel represents a business process as illustrated in Figure 1, augmented with some additional information needed to generate a corresponding GUI application. The terminate constructor represents a final step of the process, whose string argument represents a concluding message. The xor constructor represents a branching step, where the possible next steps are represented by a list of string-labeled 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.[5] 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.

```
businessModel2Gui : BusinessModel → GUI
```

A simple or xor step will be translated into a GUI with string-labeled buttons leading to the corresponding next steps. An input step will be translated into a form with text boxes for each input and a handler that collects these inputs and invokes the provided function to transition to the next step.

The function associated with an input step may perform arbitrary processing. For example, in the input step associated with collecting the GFR test result (renal value), the string input from the text box is converted into an integer, and then categorized into elements of the following data type, which is used to select the next step in the process.

```
data RenalCat : Set where
    <25 ≥25<30 ≥30<50 ≥50 : RenalCat
```

Finally, the initial state of the GUI application for a business process can be obtained from the following function.

```
businessModel2State : BusinessModel → State
businessModel2State b
  = state (businessModel2GUI b) notStarted
```

5.3 From Business Processes to GUI Applications

Using the BusinessModel data type defined in Section 5.2, we can encode the business process depicted in Figure 1. We show only a representative sample of the full encoding of the process. First, we start from the end and work our way backwards. The following three lines define the final discharge step and two of the simple steps that precede it.

[5]The repository contains a more advanced variant where the input constructor also accepts an input validation function. The resulting GUI application will then validate its inputs; if validation fails, the validation function specifies an error message to display and the application will return to the input form. In our example, this can be used to ensure that numeric values are entered as numbers and fulfil reasonable range conditions. The repository shows as well how to adapt the verification of the health care process given below to a process with validation of inputs.

```
discharge          = terminate "Discharge Patient"
lowdoseSelection   = simple "Low Dose"  discharge
highdoseSelection  = simple "High Dose" discharge
```

Many xor steps in the business process are not points where doctors should be able to make arbitrary decisions. Rather, the next steps are fully determined by safety constraints and by data acquired earlier in the process. Therefore, we model such steps not with xor, which would enable arbitrarily choosing among next steps, but with simple and a next step computed by a function on the incoming data. For example, from the *Med A* step, whether to choose the high dosage or low dosage depends on the patient's weight (see Table 1). This is implemented below by defining *Med A* (NOACSelectionA) to be a simple step whose next step is computed by the doseSelectionA function.

```
doseSelectionA : WghtCat → BusinessModel
doseSelectionA ≤60 = lowdoseSelection
doseSelectionA >60 = highdoseSelection

NOACSelectionA : WghtCat → BusinessModel
NOACSelectionA w = simple "Med A" (doseSelectionA w)
```

For the other three NOAC medications, the transition to the corresponding dosage step is determined by the following function, which implements the remainder of the third row of Table 1, plus the additional constraint that patients over 75 years of age should receive a low dosage.

```
doseSelection¬A : RenalCat≥30 → AgeCat
                  → BusinessModel
doseSelection¬A ≥30<50 <75  = lowdoseSelection
doseSelection¬A ≥50    <75  = highdoseSelection
doseSelection¬A r      ≥75  = lowdoseSelection
```

The first steps of the process model are concerned with collecting data about the patient. The following defines the input step for entering the patient's blood test results.

```
bloodTestRes : FallRisk → AgeCat → WghtCat
               → BusinessModel
bloodTestRes f a w =
  input "Enter Bloodtest Result" λ str →
  diagnosis f (str2RenalCat str ) a w
```

The subsequent state is determined by the diagnosis function, which takes into account the renal value (converted to a categorical value of type RenalCat), and the categorical values for fall risk and age, obtained at previous input 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 :
  (strAge strWght strFallR strScore strBlood : String)
    → str2RenalCat strBlood ≡ <25
    → stateAfterBloodTest strAge strWght strFallR
                                   strScore strBlood
      -eventually-> warfarinState
theoremWarfarin strAge strWght strFallR strScore strBlood =
  theoremWarfarinAux (patientHist2FallRisk strFallR)
  (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
theoremWarfarinAux r .<25 a w refl =
  next (λ _ → next (λ _ → hasReached))
```

The proof expresses that no matter which inputs are provided in the first two steps of the GUI, we will reach the warfarin state if the categorized renal value is <25. The proof constructs a value of the -eventually-> type. Each next constructor corresponds to a GUI state that must be clicked through before reaching the desired warfarin state. Since no further inputs are needed or used, each step ignores its inputs as indicated by _.

The second theorem states that if the weight is > 60, and if we reach the state where apixaban (A) is prescribed, then a low dosage is not prescribed.

```
theoremNoLowDosisWeight>60 :
  (strAge strWght strFallR strScore strBlood : String)
  → str2WghtCat strWght ≡ >60
  → (w' : WghtCat)
  → stateAfterBloodTest strAge strWght strFallR
        strScore strBlood
     -gui-> NOACSelectionAState w'
  → (s : State)
  → NOACSelectionAState w' -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.). We also demonstrate a way to define and prove properties about GUI applications in Sections 4–5. Such properties include data-dependent reachability and unreachability constraints for GUIs generated from declarative business process specifications. In another paper [5], we have adapted the current framework to allow the specification and verification of feature-based composition of workflows implemented as a feature-oriented software product line [10]. The framework allows the modular definition of features and promotes the separation of concerns in workflow definitions.

The library presented in this paper uses a custom-built backend based on SDL and Rasterific, as described in Section 3.4. We have also developed an alternative version of this library based on the standard wxWidgets toolkit via wxHaskell [80], which is described in previous work [7]. There are several tradeoffs between the two versions of the library. The version presented here is much simpler (since wxWidgets is an inherently imperative toolkit with concurrency, interfacing to it from Agda leads to significant complexity), which supports scaling to larger applications and greatly simplifies the definition and proof of properties. However, wxWidgets integrates better with the host operating system and has a larger selection of components that can be integrated with our approach without too much additional effort. The library described in [7] is also more generic in that it supports nested frames and modifying properties without redrawing the entire GUI. An advantage of the simpler version presented here is that it is easier to programmatically generate GUI applications from higher-level declarative specifications, as illustrated by the translation of business processes into GUIs in Sections 5.2 and 5.3. Due to the increased complexity, the application presented in [7] is much simpler than the case study presented here.

Functional Reactive Programming (FRP) FRP is another approach to writing interactive programs in functional programming languages [78]. Sculthorpe and Nilsson [67] provide an implementation of FRP in Agda. Jeffrey [36] define 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 [13].

Krishnaswami and Benton [44] 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 STEPs project [9] 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 for our Haskell GUI backend. A major difference between the STEPs project and regular FRP is that it is based on dynamically-typed languages without static type guarantees.

Formlets [20] are abstract user interfaces that define static forms, which combine several inputs into one output; they do not support sequencing multiple intermediate requests.

Formalizations in Isabelle There has been work on formalizing end-user-facing applications in Isabelle. There are formalizations of distributed applications [15] and conference management systems [37]. 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 [54], which is a declarative modeling language [73] for business processes based on human-readable diagrams. The Declare constraints are often embedded in 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 [19] and implemented as an extension for Declare [56]. A first-order logic approach is presented by [46], which would be sufficiently expressive, however, it is undecidable which is problematic (e.g. most model checkers don't support it).

In the current paper, in contrast to [19, 54, 56] 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 [23] proves in Isabelle a general result that a certain labeling of events in a business logic guarantees orthogonality of events. Montali et al. [57] 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 An alternative approach is to use a co-inductive approach to directly verify imperative programs [59]. [16] presents Dynamic Trace Logic (DTL), which combines dynamic logic with temporal logic, allowing to prove functional and information-flow properties in concurrent programs. Properties of interactive programs are specified using modal operators. We also plan to develop a full modal logic.

Idris and Algebraic Effects. Bauer and Pretnar [14] have introduced the notion of algebraic effects. Brady [17] adapted this approach to represent interactive programs in Idris [18, 34]. In [1, Sec. 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. The focus of Idris is on practical dependently typed programming, but it is also usable as an interactive theorem prover. There are no established GUI libraries for Idris, but GUI programming should be possible using its FFI interface [62].

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 [11], 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. [12] proposed that a key building principle for deep specifications is vertical compositionality, which means that higher specification levels are related provably correct to *n* 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 [38–40] 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 [22, 42, 48].

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 instead 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 fulfils 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 thank Prof. Gottsauner-Wolf for providing the details of the 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 (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 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.

References

[1] Andreas Abel, Stephan Adelsberger, and Anton Setzer. 2017. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming* 27, Article 38 (Jan 2017), 54 pages. https://doi.org/10.1017/S0956796816000319

[2] Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *JFP* 26, Article e2 (2016), 61 pages. https://doi.org/10.1017/S0956796816000022 ICFP 2013 special issue.

[3] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '13)*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2429069.2429075

[4] Samina Raza Abidi, Jafna Cox, Ashraf Abusharekh, Nima Hashemian, and Syed Sibte Raza Abidi. 2016. A Digital Health System to Assist Family Physicians to Safely Prescribe NOAC Medications. *Studies in health technology and informatics* 228 (2016), 519—523. http://europepmc.org/abstract/MED/27577437

[5] Stephan Adelsberger, Bashar Igried, Markus Moser, Vadim Savenkov, and Anton Setzer. 2018. Formal Verification for Feature-based Composition of Workflows. http://www.cs.swan.ac.uk/~csetzer/articles/SERENE/SERENE18/SERENE18AdelsbergerIgriedMoserSavenkoSetzer.pdf To appear in proceedings of http://www.cs.swan.ac.uk/~csetzer/articles/SERENE/SERENE18/SERENE18AdelsbergerIgriedMoserSavenkoSetzer.pdf.

[6] Stephan Adelsberger, Anton Setzer, and Eric Walkingshaw. 2017. Declarative GUIs: Simple, Consistent, and Verified. (2017). https://github.com/stephanadelsb/PPDP18 Git respository.

[7] Stephan Adelsberger, Anton Setzer, and Eric Walkingshaw. 2018. Developing GUI Applications in a Verified Setting. In *Symp. on Dependable Software Engineering: Theories, Tools and Applications*. ACM.

[8] Agda Community. 2017. The Agda Wiki. (2017). http://wiki.portal.chalmers.se/agda

[9] Dan Amelang, Bert Freudenberg, Ted Kaehler, Alan Kay, Stephen Murrell, Yoshiki Ohshima, Ian Piumarta, Kim Rose, Scott Wallace, Alessandro Warth, and Takashi Yamamiya. 2011. *STEPS Toward Expressive Programming Systems, 2011 Progress Report.* Technical Report. Viewpoints Research Institute.

[10] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer-Verlag, Berlin/Heidelberg.

[11] Heinrich Apfelmus. 2017. Threepenny-gui. https://wiki.haskell.org/Threepenny-gui

[12] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 1 – 24. https://doi.org/10.1098/rsta.2016.0331

[13] Manuel Bärenz and Sebastian Seufert. 2017. Verifying Functional Reactive Programs with Side Effects. (2017). [41], pp. 47 – 48.

[14] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108 – 123. https://doi.org/10.1016/j.jlamp.2014.02.001

[15] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. 2017. CoSMeDis: A Distributed Social Media Platform with Formally Verified Confidentiality Guarantees. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE, Piscataway, New Jersey, US, 729–748. https://doi.org/10.1109/SP.2017.24

[16] Bernhard Beckert and Daniel Bruns. 2013. Dynamic Logic with Trace Semantics. In *Automated Deduction – CADE-24*, Maria Paola Bonacina (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–329.

[17] Edwin Brady. 2015. Resource-Dependent Algebraic Effects. In *Trends in Functional Programming: 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, Jurriaan Hage and Jay McCarthy (Eds.). Springer International Publishing, Cham, 18–33. https://doi.org/10.1007/978-3-319-14675-1_2

[18] Edwin Brady. 2017. *Type-driven Development with Idris* (1 ed.). Manning Publications, Greenwich, Connecticut.

[19] Carolina Ming Chiao, Vera Künzle, and Manfred Reichert. 2012. Towards Object-aware Process Support in Healthcare Information Systems. In *4th International Conference on eHealth, Telemedicine, and Social Medicine (eTELEMED 2012)*. IARIA, Wilmington, Delaware, US, 227–236. http://dbis.eprints.uni-ulm.de/775/

[20] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2008. The essence of form abstraction. In *Programming Languages and Systems: 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, G. Ramalingam (Ed.). Springer, Berlin, Heidelberg, 205–220.

[21] Dan Amelang. 2018. Gezira Library. https://github.com/damelang/gezira https://github.com/damelang/gezira.

[22] R. Davies. 1996. A temporal-logic approach to binding-time analysis. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, Piscataway, New Jersey, US, 184–195. https://doi.org/10.1109/LICS.1996.561317

[23] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. 2015. Concurrency and Asynchrony in Declarative Workflows. In *Business Process Management: 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 – September 3, 2015, Proceedings*, Hamid Reza Motahari-Nezhad, Jan Recker, and Matthias Weidlich (Eds.). Springer International Publishing, Cham, 72–89. https://doi.org/10.1007/978-3-319-23063-4_5

[24] Peter Dybjer. 2000. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic* 65, 2 (June 2000), 525 – 549. https://doi.org/10.2307/2586554

[25] Peter Dybjer and Anton Setzer. 2003. Induction-Recursion and Initial Algebras. *Annals of Pure and Applied Logic* 124 (2003), 1 – 47. https://doi.org/10.1016/S0168-0072(02)00096-9

[26] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. 1980. On the Temporal Analysis of Fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '80)*. ACM, New York, NY, USA, 163–173. https://doi.org/10.1145/567446.567462

[27] Mark L Graber, Dana Siegal, Heather Riah, Doug Johnston, and Kathy Kenyon. 2015. Electronic health record-related events in medical malpractice claims. *Journal of patient safety* 00, 00 (2015), 1– 9. https://doi.org/10.1097/pts.0000000000000240

[28] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and Evolving GUI-directed Test Scripts. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 408–418. https://doi.org/10.1109/ICSE.2009.5070540

[29] Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *CSL'00 (Lect. Notes in Comput. Sci.)*, Peter Clote and Helmut Schwichtenberg (Eds.), Vol. 1862. Springer, Berlin / Heidelberg, 317–331. https://doi.org/10.1007/3-540-44622-2_21

[30] Peter Hancock and Anton Setzer. 2000. Specifying interactions with dependent types. http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000.

[31] Peter Hancock and Anton Setzer. 2005. Interactive programs and weakly final coalgebras in dependent type theory. In *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics (Oxford Logic Guides)*. Clarendon Press, Oxford, UK, 115 – 136. https://doi.org/10.1093/acprof:oso/9780198566519.003.0007

[32] haskell.org. 2018. SDL. Haskell Library. https://wiki.haskell.org/SDL https://wiki.haskell.org/SDL.

[33] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

[34] Idris Development Team. 2017. Idris. A language with dependent types. https://www.idris-lang.org/

[35] K Ioannidis, I Scarlatinis, A Papachristos, and X Madia. 2018. 4CPS-017 Misuse of novel oral anticoagulants in hospital settings.

[36] Alan Jeffrey. 2012. LTL Types FRP: Linear-time Temporal Logic Propositions As Types, Proofs As Functional Reactive Programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (PLPV '12)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2103776.2103783

[37] Sudeep Kanav, Peter Lammich, and Andrei Popescu. 2014. A conference management system with verified document confidentiality. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 167–183. https://doi.org/10.1007/978-3-319-08867-9_11

[38] Karim Kanso. 2012. *Agda as a Platform for the Development of Verified Railway Interlocking Systems.* Ph.D. Dissertation. Dept. of Computer Science, Swansea University, Swansea, UK. http://www.swan.ac.uk/csetzer/articlesFromOthers/index.html

[39] Karim Kanso. 2017. Agda. https://github.com/kazkansouh/agda Github repository, fork of Agda installation, containing code from PhD thesis Kanso.

[40] Karim Kanso and Anton Setzer. 2014. A light-weight integration of automated and interactive theorem proving. *Mathematical Structures in Computer Science* FirstView (12 November 2014), 1–25. https://doi.org/10.1017/S0960129514000140

[41] Ambrus Kaposi (Ed.). 2017. 23rd International Conference on Types for Proofs and Programs TYPES 2017, Budapest, Hungary, 29 May - 1 June 2017, Abstracts. (May 2017). http://types2017.elte.hu/proc.pdf

[42] Kensuke Kojima and Atsushi Igarashi. 2011. Constructive linear-time temporal logic: Proof systems and Kripke semantics. *Information and Computation* 209, 12 (2011), 1491 – 1503. https://doi.org/10.1016/j.ic.2010.09.008 Intuitionistic Modal Logic and Applications (IMLA 2008).

[43] Glenn E. Krasner and Stephen T. Pope. 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming* 1, 3 (1988), 26–49.

[44] Neelakantan R. Krishnaswami and Nick Benton. 2011. A Semantic Model for Graphical User Interfaces. In *Proceedings of ICFP '11*. ACM, New York, NY, USA, 45–57. https://doi.org/10.1145/2034773.2034782

[45] S Lawes and M Grissinger. 2017. Medication errors attributed to health information technology. *PA-PSRS Patient Saf Advis* 14, 1 (2017), 1–8.

[46] Fabrizio Maria Maggi, Marlon Dumas, Luciano García-Bañuelos, and Marco Montali. 2013. Discovering Data-Aware Declarative Process Models from Event Logs. In *Business Process Management*, Florian Daniel, Jianmin Wang, and Barbara Weber (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 81–96.

[47] Farah Magrabi, Mei-sing Ong, and Enrico Coiera. 2016. An Overview of HIT-Related Errors. In *Safety of Health IT: Clinical Case Studies*, Abha Agrawal (Ed.). Springer, Cham, 11–23. https://doi.org/10.1007/978-3-319-31123-4_2

[48] Patrick Maier. 2004. Intuitionistic LTL and a New Characterization of Safety and Liveness. In *Computer Science Logic: 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004. Proceedings*, Jerzy Marcinkowski and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 295–309. https://doi.org/10.1007/978-3-540-30124-0_24

[49] Martin A Makary and Michael Daniel. 2016. Medical error-the third leading cause of death in the US. *BMJ: British Medical Journal (Online)* 353 (May 03 2016), 1–5.

[50] Atif M. Memon. 2002. GUI Testing: Pitfalls and Process. *Computer* 35, 8 (2002), 87–88.

[51] Atif M. Memon. 2007. An Event-Flow Model of GUI-Based Applications for Testing. *Software Testing, Verification and Reliability* 17, 3 (2007), 137–157.

[52] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 2001. Coverage Criteria for GUI Testing. *ACM SIGSOFT Software Engineering Notes* 26, 5 (2001), 256–267.

[53] Atif M. Memon and Qing Xie. 2005. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. *IEEE Transactions on Software Engineering (TSE)* 31, 10 (2005), 884–896.

[54] Steven Mertens, Frederik Gailly, and Geert Poels. 2015. Enhancing declarative process models with DMN decision logic. In *International Conference on Enterprise, Business-Process and Information Systems Modeling*. Springer, Cham, 151–165.

[55] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55 – 92. https://doi.org/10.1016/0890-5401(91)90052-4

[56] Marco Montali, Federico Chesani, Paola Mello, and Fabrizio M. Maggi. 2013. Towards Data-aware Constraints in Declare. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, New York, NY, USA, 1391–1396. https://doi.org/10.1145/2480362.2480624

[57] Marco Montali, Maja Pesic, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. 2010. Declarative Specification and Verification of Service Choreographies. *ACM Trans. Web* 4, 1, Article 3 (Jan. 2010), 62 pages.

[58] Risa B Myers, Stephen L Jones, and Dean F Sittig. 2011. Review of reported clinical information system adverse events in US Food and Drug Administration databases. *Applied clinical informatics* 2, 1 (2011), 63.

[59] Keiko Nakata and Tarmo Uustalu. 2009. Trace-based coinductive operational semantics for while. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 375–390.

[60] Kent Petersson and Dan Synek. 1989. A Set Constructor for Inductive Sets in Martin-Löf's Type Theory. In *CTCS'89 (Lect. Notes in Comput. Sci.)*, Vol. 389. Springer-Verlag, London, UK, 128–140.

[61] Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

[62] Keith Pinson. 2015. GUI programming in Idris? https://groups.google.com/forum/#!topic/idris-lang/R_7oixHofUo Google groups posting.

[63] David C Radley, Melanie R Wasserman, Lauren EW Olsho, Sarah J Shoemaker, Mark D Spranca, and Bethany Bradshaw. 2013. Reduction in medication errors in hospitals due to adoption of computerized provider order entry systems. *Journal of the American Medical Informatics Association* 20, 3 (2013), 470–476.

[64] Rasterific. 2018. Github repository. https://github.com/Twinside/Rasterific https://github.com/Twinside/Rasterific.

[65] William B Runciman, Elizabeth E Roughead, Susan J Semple, and Robert J Adams. 2003. Adverse drug events and medication errors in Australia. *International Journal for Quality in Health Care* 15, suppl_1 (2003), i49–i59.

[66] Eva A Saedder, Birgitte Brock, Lars Peter Nielsen, Dorthe K Bonnerup, and Marianne Lisby. 2014. Identifying high-risk medication: a systematic literature review. *European journal of clinical pharmacology* 70, 6 (2014), 637–645.

[67] Neil Sculthorpe and Henrik Nilsson. 2009. Safe Functional Reactive Programming Through Dependent Types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/1596550.1596558

[68] SDL. 2018. Single Directmedia Layer. Library. http://www.libsdl.org/ http://www.libsdl.org/.

[69] Anton Setzer. 2006. Object-oriented programming in dependent type theory. In *Conference Proceedings of TFP 2006*. Intellect Books, Bristol, 1–16. http://www.cs.nott.ac.uk/~psznhn/TFP2006/Papers/26-AntonSetzer-OOPInDependentTypeTheory.pdf

[70] Anton Setzer and Peter Hancock. 2004. Interactive Programs and Weakly Final Coalgebras in Dependent Type Theory (Extended Version). In *Dependently Typed Programming 2004 (Dagstuhl Seminar Proc.s)*, Thorsten Altenkirch, Martin Hofmann, and John Hughes (Eds.), Vol. 04381. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 1 – 30. http://drops.dagstuhl.de/opus/volltexte/2005/176/

[71] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug Characteristics in Open Source Software. *Empirical Software Engineering* 19, 6 (2014), 1665–1705.

[72] Laura A. Valaer and Robert G. Babb. 1997. Choosing a User Interface Development Tool. *IEEE Software* 14, 4 (1997), 29–39.

[73] Wil MP van Der Aalst, Maja Pesic, and Helen Schonenberg. 2009. Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development* 23, 2 (2009), 99–113.

[74] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 61–78. https://doi.org/10.1145/91556.91592

[75] Philip Wadler. 1995. Monads for functional programming. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text*, Johan Jeuring and Erik Meijer (Eds.). Springer, Berlin, Heidelberg, 24–52. https://doi.org/10.1007/3-540-59451-5_2

[76] Philip Wadler. 1997. How to Declare an Imperative. *ACM Comput. Surv.* 29, 3 (September 1997), 240–263. https://doi.org/10.1145/262009.262011

[77] Philip Wadler. 1998. The Marriage of Effects and Monads. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/289423.289429

[78] Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 242–252.

[79] J. Whittle, J. Hutchinson, and M. Rouncefield. 2014. The State of Practice in Model-Driven Engineering. *IEEE Software* 31, 3 (2014), 79–85.

[80] wiki.haskell. 2017. WxHaskell. (Retrieved 9 February 2017). https://wiki.haskell.org/WxHaskell

[81] Xiaoxi Yao, Nilay D Shah, Lindsey R Sangaralingham, Bernard J Gersh, and Peter A Noseworthy. 2017. Non–vitamin K antagonist oral anticoagulant dosing in patients with atrial fibrillation and renal dysfunction. *Journal of the American College of Cardiology* 69, 23 (2017), 2779–2790.